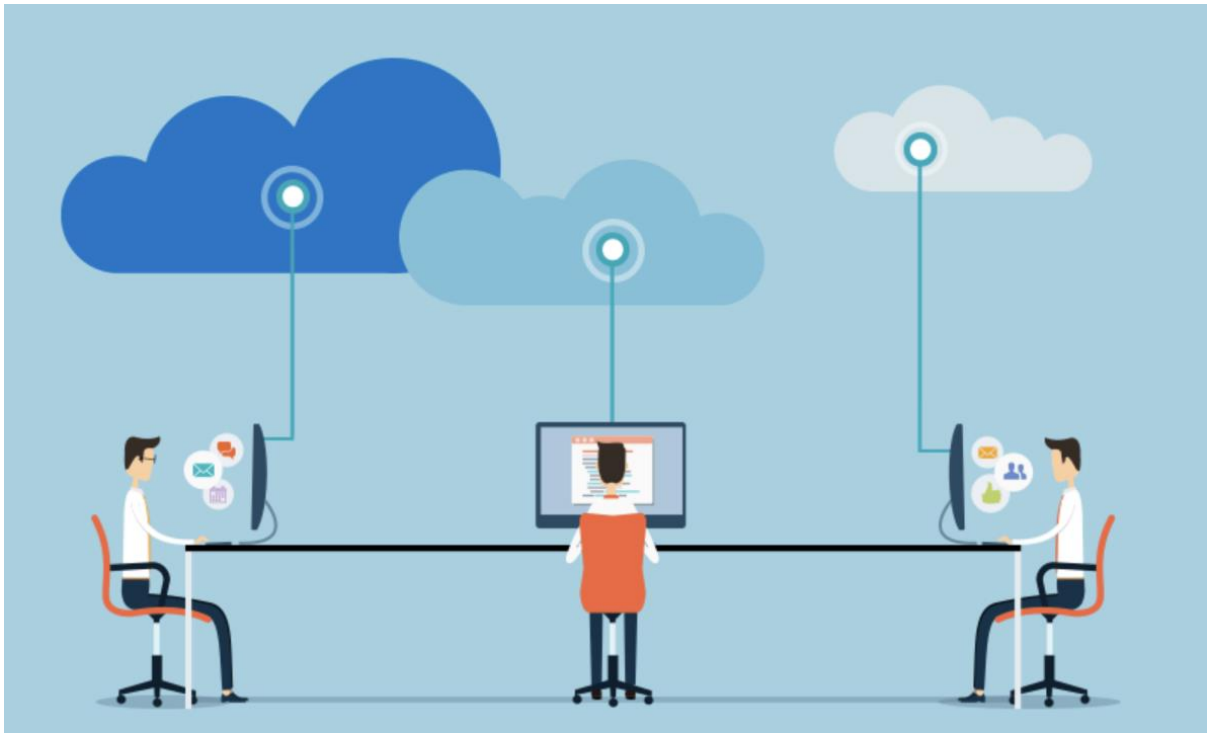
Task 1. Scaling distributed Systems using direct and indirect communication middleware



Noms: Lyubomyr Grygoriv & Raúl Garcia

Professor: Pedro Antonio García López

Laboratori: L3

Data: 20/05/2025

Índex

1 – Objectius

Aquesta pràctica té diversos objectius principals:

1 - Implementar 2 aplicacions escalables usant quatre *middlewares* de comunicació diferents: XMLRPC, Pyro, Redis i RabbitMQ.

Aquestes dues aplicacions són, d'una primera instància, el **InsultService**, el qual ha de rebre insults remotament i poder-los emmagatzemar. També ha de proporcionar un mecanisme per poder realitzar missatges broadcast cada 5 segons, notificant events aleatoris als subscriptors interessats.

Per altra banda, tenim el **InsultFIlter**, el qual és un servei basat en el patró Work queue. Aquest servei implementarà un mecanisme de filtratge d'insults, on substituirà els insults del text per la paraula CENSORED.

2 - Un cop tenim tots els serveis implementats amb els diferents *middlewares*, s'ha de realitzar un *stress test* i una comparació de *speedup*, per poder observar i comparar l'escalabilitat i el potencial de cada un dels *middlewares* que s'han usat.

Pel cas del *speedup*, s'ha de realitzar els *tests* amb un, dos i tres nodes. (En el nostre cas hem hagut de fer modificacions en les arquitectures estàtiques d'un node)

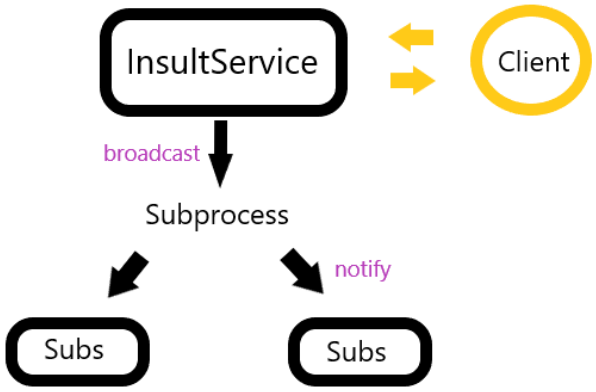
3 - Finalment, es vol implementar un sistema d'escalat dinàmic, on anem escalant nodes respecte al número d'esdeveniments que ens arriben.

2 – Aquitectura

2.1 - XMLRPC

InsultService

Arquitectura 1 Nodes:

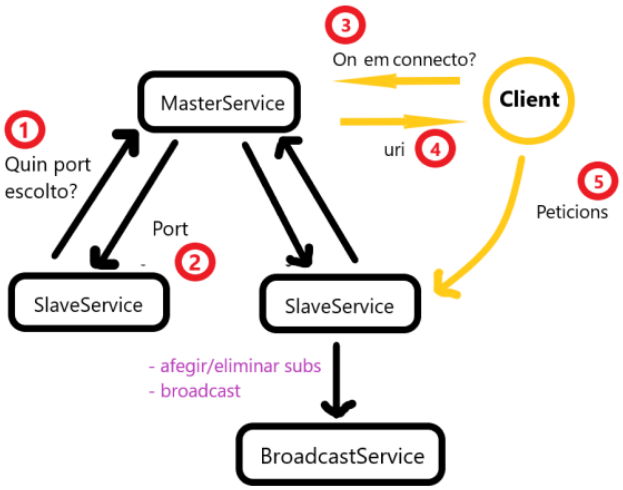


Abstracte 1. Arquitectura InsultService XMLRPC d'un node.

En l'asbtracte 1, podem observar el diagrama que segueix el servei InsultService. L'usuari interactua directament sobre l'instància del servei, on pot afegir i esborrar insults, afegir i esborrar subscriptors (a partir de la seva uri), fer que el servei l'insulti i sobretot, poder **activar i descativar el broadcast**.

Per poder fer el mecanisme del broadcast, el servei crea un procés que notificarà cada 5 segons als subscriptors (per parar el mecanisme, mata el procés). D'aquesta manera, el servei no es queda bloquejat i pot continuar atenent peticions.

Arquitectura n Nodes:



Abstracte 2. Arquitectura InsultService XMLRPC de varius nodes.

En l'asbtracte 2, es mostra l'arquitectura per l'InsultService amb varius nodes. Podem observar que tenim diverses instàncies, a continuació explicarem el que fa cada una i els passos que segueixen per permetre la comunicació de tota l'arquitectura.

El **SlaveService** és el servei que atén les peticions de l'usuari, com ara, afegir insults i retornar un insult. Quan s'inicia un node d'aquesta instància, aquest es connecta al MasterService per saber en quin port ha d'escoltar les peticions. S'ha usat una llista de **Redis** per permetre la sincronització d'informació entre slaves i que tots tinguin la mateixa llista d'insults.

Per altra banda, s'ha decidit separar el broadcast d'aquest servei perquè només hi haguí un procés realitzant el broadcast a la vegada i que tots els serveis slave (diferents clients) puguin parar-lo i engegar-lo.

Pel que fa al **BroadcastService**, és el que controla els subscriptors i el broadcast, ho fa de la mateixa manera que ho feia el InsultService amb un únic node, crea un procés i és aquest qui envia les peticions als subscriptors.

Per altra banda, el **client** s'ha de connectar al MasterService perquè aquest li assigni un slave que atendrà les seves peticions.

Finalment, el **MasterService**, que és el que gestiona tot aquest flux d'informació i qui assigna els ports dels slaves i on s'ha de connectar cada client (amb Round Robbin).

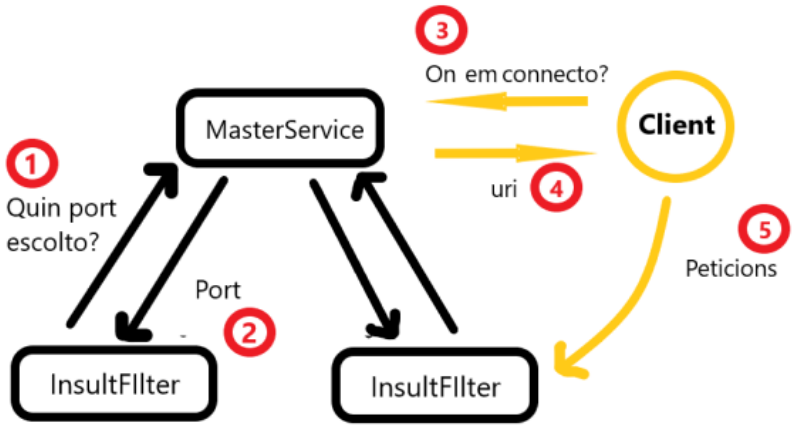
Important: no s'ha tingut en compte el cas en què un node (slave) caigui. En cas que això passi, el master el pot assignar a un client, fent que el client obtingui errors realitzi una petició.

InsultFilter

Arquitectura 1 Nodes:

Per aquest cas és molt simple, el servei del InsultFIlter disposa de 2 funcions principals, una serveix per produir treball (introdueix event a una llista local) i l'altra per consumir (filtra els events de la llista local i guarda el resultat a una altra llista local). Per aquest cas, el servei només podrà realitzar una de les dos coses a la vegada i per petició de l'usuari.

Arquitectura n Nodes:



Abstracte 3. Arquitectura InsultFilter XMLRPC de varius nodes.

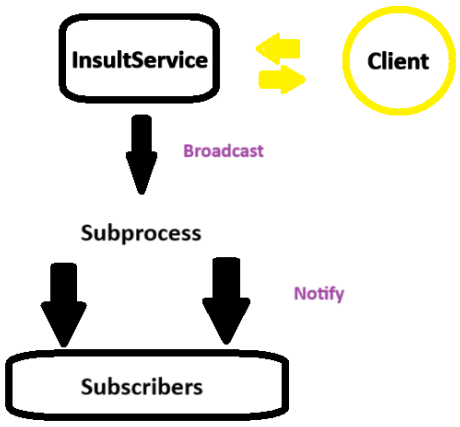
En la figura 3 podem observar l'arquitectura del InsultFIlter amb varius nodes. Aquesta és idèntica a la del InsultService, ja que em reutilitzat el MasterService.

A causa d'aquesta decisió, el flux del diagrama és el mateix. El InsultFilter demana al Master a quin port escoltar i el client li demana a quin servei connectar-se. En aquest cas, s'ha decidit usar Redis per permetre una sincronització de les llistes de treball i de resultats del servei.

2.2 - Pyro4

InsultService

Arquitectura 1 Nodes:

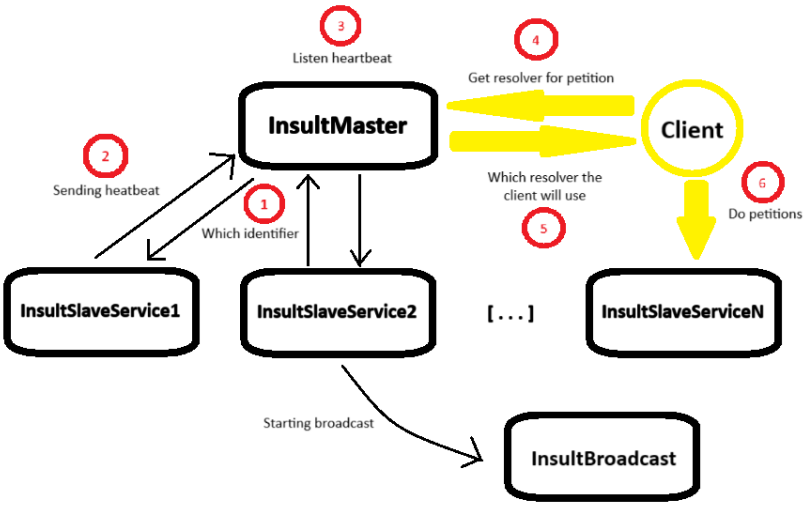


Abstracte 4. Arquitectura InsultService Pyro4 d'un node.

En la figura 4, podem observar l'estructura d'un node utilitzada pel middleware de *Pyro*. Igual que per *XMLRPC* els clients es connectaran directament a una instància del *InsultService* permeten així que aquest pugui afegir i eliminar insults, activar i desactivar el procés de *broadcast* i afegir i eliminar subscriptors a partir de la *uri* del client subscriptor. Cal destacar, que no podem emmagatzemar directament la instància de *Pyro4.Proxy(uri)*, ja que ens donava errors en l'execució del codi.

Com hem explicat anteriorment per fer el procés broadcast des del servei creem un procés que enviarà periòdicament insults a tots i cadascun dels subscriptors d'aquell servidor, d'aquesta manera permetem que el servei pugui seguir atenent peticions de manera concurrent.

Arquitectura n Nodes:



Abstracte 5. Arquitectura InsultService Pyro4 de varius node.

En la figura (5) anterior podem observar l'arquitectura dissenyada per l'estructura de múltiples nodes per a Pyro. En aquest cas, hem decidit realitzar una estructura jeràrquica, més concretament una *Master Slave arquitecture*. A continuació explicarem cadascuna de les instàncies que es mostren en el diagrama i les seves principals funcions que ens permeten obtenir el work flow que indiquem en l'abstracte.

En primer lloc, el *InsultMaster* és la peça fonamental de tota l'arquitectura, ja que s'encarrega d'indicar als clients a quins servidors *slave* i *filter* s'ha de connectar i a més a més monitoritza quins són els servidors esclaus es troben connectats en aquell moment. Les funcions que s'encarreguen de fer aquestes tasques són: *get_resolver_slave()*, *get_resolver_filter()*, *heartbeat_slave()*, *check_slave()* i *next_identifier()*.

Els mètodes *get_resolver_slave()* i *get_resolver_filter()* són les funcions definides en l'api del servei que poden utilitzar els clients per saber a quin de tots els nodes ha de dirigir les seves peticions. A continuació podem observar els fragments de codi implementats per a les següents funcions:

```python
def get_resolver_slave(self):

    # We return the id of the oldest pulse of the slave actived that we recieved
    sorted_by_petitions_time = dict(sorted(self.slave_list.items(), key=lambda item: (item[1]["number_petitions"], item[1]["last_seen"])))
    filtered_by_service = {k: v for k, v in sorted_by_petitions_time.items() if v["is_filter"] is False}
    slave_keys = list(filtered_by_service.keys())
    if slave_keys:

        # Updating the number petitions of the slave
        tmp = self.slave_list[slave_keys[0]]
        tmp["number_petitions"] = tmp["number_petitions"] + 1
        self.slave_list[slave_keys[0]] = tmp

        # We return the url of the slave service
        return "insult" + str(slave_keys[0]) + ".service"
    else:
        return None
```

```python
def get_resolver_filter(self):

    # We return the id of the oldest pulse of the slave actived that we recieved
    sorted_by_petitions_time = dict(sorted(self.slave_list.items(), key=lambda item: (item[1]["number_petitions"], item[1]["last_seen"])))
    filtered_by_service = {k: v for k, v in sorted_by_petitions_time.items() if v["is_filter"] is True}
    slave_keys = list(filtered_by_service.keys())
    if slave_keys:

        # Updating the number petitions of the slave
        tmp = self.slave_list[slave_keys[0]]
        tmp["number_petitions"] = tmp["number_petitions"] + 1
        self.slave_list[slave_keys[0]] = tmp

        # We return the url of the filter service
        return "insult" + str(slave_keys[0]) + ".service"
    else:
        return None
```

Abstracte 6. Funcions get_resolve_slave/filter usades en el InsultService/Filter de varius nodes.

La funció de *heartbeat_slave()* és una funció interna que utilitzen els serveis auxiliars de l'arquitectura per poder enviar informació al *MasterService*. Aquests mètodes són llançats per un procés per cada *InsulSlaveService* o *InsultSlaveFilter* que s'instanciï, d'aquesta manera permetem que tots els nodes auxiliars de l'arquitectura permetin rebre informació de les peticions dels usuaris de manera concurrent mentre s'envia informació al servidor principal. Seguidament, mostrem el codi utilitzat pel registre d'informació dels nodes auxiliars en l'arquitectura:

```python
def heartbeat_slave(self, raw_slave_data):

    # We recieve the heartbeat of each insult slave server
    slave_data = json.loads(raw_slave_data)
    slave_id = int(slave_data["id"])
    slave_pulse = int(slave_data["pulse"])
    is_filter = bool(slave_data["is_filter"])

    # We check if the slave is new, if that's the case we set number_petition variable to zero
    if slave_id not in self.slave_list:
        self.slave_list[slave_id] = {
            "last_seen": slave_pulse,
            "is_filter": is_filter,
            "number_petitions" : 0
        }

    # We only update the last_seen variable if the slave is still active
    else:
        tmp = self.slave_list[slave_id]
        tmp["last_seen"] = slave_pulse
        self.slave_list[slave_id] = tmp

    # We check if the pulse comes from a filter service
    if is_filter:
        # If that's the case we register it to the filter list
        self.filter_list[slave_id] = self.slave_list[slave_id]
```

Abstracte 7. Funció que envia un puls al master.

El mètode de *check_slave()* permet al MasterService comprovar quin del servidors, filter i slave es troben actius en aquell instant. A continuació ensenyem quina ha estat la implementació realitzada:

```python
def check_slave(self, timeout=10):
    while True:
        time.sleep(1)
        now = time.time()

        # For each slave registered
        for slave in list(self.slave_list.keys()):
            # Check if the last pulse was 10 seconds ago
            if now - self.slave_list[slave]["last_seen"] > timeout:
                self.slave_list.pop(slave)
```

Abstracte 8. Funció del master que comprova que els slaves estiguin actius.

La funció de *next_identifier()* és la funció que s'utilitza en el moment d'instanciació d'un servei auxiliar en l'arquitectura, per garantir que l'identificador de cada node sigui únic.

```python
def next_identifier(self):
    # We update the id counter for the slaves
    self.next_id = self.next_id + 1
    return self.next_id
```

Abstracte 9. Funció per general el següent identificdor dels slaves.

En segon lloc, tenim el *InsultSlaveService* són aquells nodes de l'estructura que s'encarreguen de resoldre les peticions dels usuaris com: afegir o eliminar insults, registrar o eliminar un subscriptor o bé iniciar o parar el procés de broadcast. En el nostre cas hem realitzat petites adaptacions per maximitzar les capacitats de sincronisme i replicació, utilitzant cues de *Redis* pels insults i pels subscriptors.

Endemés, consta totes les instàncies consten d'un mètode per poder enviar periòdicament al MasterService les seves dades. Aquest és el següent:

```python
def send_info(self):

    # We establish the connection with the master server
    ns = Pyro4.locateNS(host='localhost', port=9090)
    uri = ns.lookup('master.service')
    server = Pyro4.Proxy(uri)

    print("Sending information to master...")
    # We proceed to do the hearbeat, to notify to the master server that the slave is still connected
    while True:
        raw_slave_data = {
            "id" : self.id,
            "pulse" : time.time(),
            "is_filter" : self.is_filter
        }
        server.heartbeat_slave(json.dumps(raw_slave_data))
        time.sleep(3)
```

Abstracte 10. Funció per enviar informació al master.

Una altra característica a destacar de la nostra arquitectura és que alguna de les instàncies de *InsultSlaveService* és l'encarregada d'aixecar el InsultBroadcast, que envia periòdicament informació als clients subscriptors. En aquesta arquitectura vam decidir realitzar un nou servei que només tingués aquella funcionalitat, per facilitar la llegibilitat en el codi i el rendiment de l'arquitectura. A continuació adjuntem la implementació d'aquest servei:

```python
class InsultBroadcast:

    def __init__(self):
        self.client = redis.Redis(host = 'localhost', port = 6379, db = 0, decode_responses = True)
        self.insult_list = "insult_list"
        self.subscriber_list = "subscriber_list"

    # We define a broadcast function where we send one random insult to each master subscriber
    def broadcast_function(self):
        while True:

            # We retrieve subscriber list and insult list
            subscriber_list = self.client.lrange(self.subscriber_list, 0, -1)
            insult_list = self.client.lrange(self.insult_list, 0, -1)
            for uri in subscriber_list:
                subscriber = base64.urlsafe_b64decode(uri).decode('utf-8')
                subscriber = Pyro4.Proxy(subscriber)
                subscriber.notify(random.choice(insult_list))
            time.sleep(5)
```

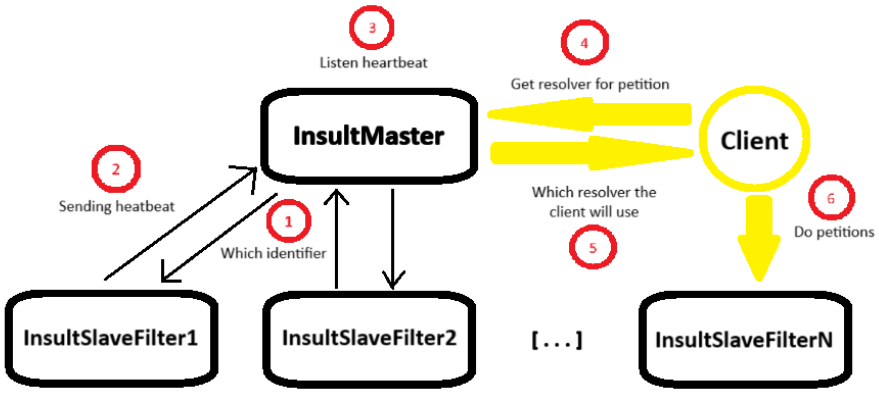
Abstracte 11. Codi del InsultBroadcaster.

InsultFilter

Arquitectura 1 Nodes:

Igual que per l'arquitectura de *XMLRPC* l'estructura seguida per *Pyro* consta de 2 mètodes principals, un que serveix per registrar nous treballs, en una llista local del servei, i un segon mètode que resol les peticions prèviament registrades, aquest mètode substitueix els insults emmagatzemats en el servei principal per *CENSORED*.

Arquitectura n Nodes:



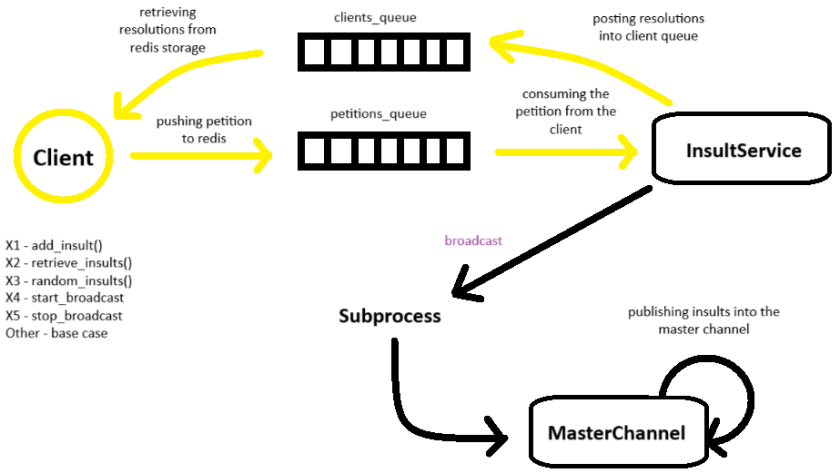
Abstracte 12. Arquitectura InsultService Pyro4 de varius nodes.

En l'anterior diagrama observem l'estructura utilitzada per al *InsultFilter*. Com podem visualitzar aquest abstracte mostra pràcticament el mateix *work flow* que per l'estructura del *InsultService*, obviant el component del *InsultBroadcast*.

2.3 - Redis

InsultService

Arquitectura 1 Nodes:



Abstracte 13. Arquitectura InsultService Redis d'un node.

En la darrera figura podem observar l'estructura implementada per al InsultService pel Middleware de Redis. Donat que Redis utilitza un sistema de llistes, cues i conjunts vam decidir fer que cada client indiqués de manera individual quina seria la cua sobre la qual s'emmagatzemarien les seves resolucions i/o peticions. Per tal d'implementar aquesta funcionalitat vam concretar que utilitzaríem *json instances* per enviar quina petició vol realitzar el client i contingut addicional de la petició, les dades. Per indicar la petició que realitza el nostre *customer* utilitzem un conjunto de marques (aquestes són les que es troben en la llegenda del diagrama). Seguidament, mostrem quins són els arguments necessaris per cada una de les peticions:

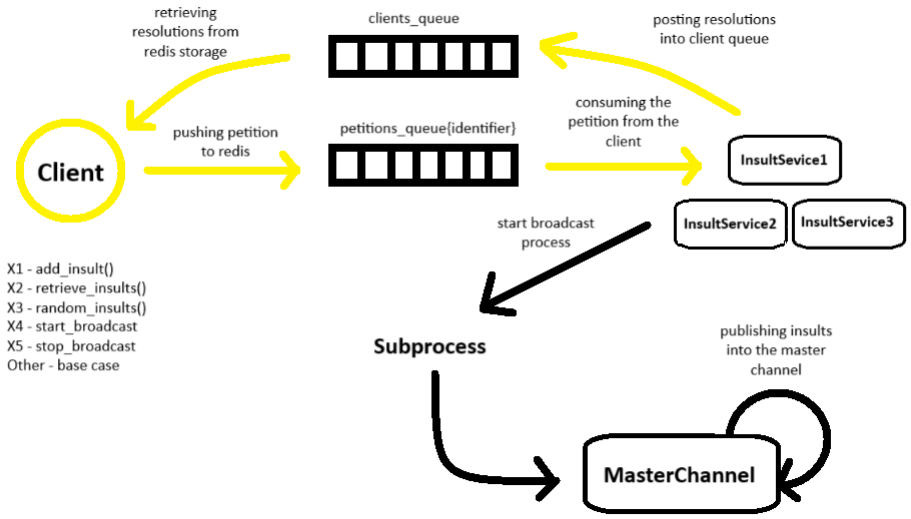
- X1, aquesta petició ens permet afegir un insult al *storage* de Redis, per poder fer-ho indiquem com a paràmetre addicional o *data* aquell insult que es vol afegir.
- X2, aquesta petició ens permet obtenir tots i cadascun dels insults que tenim emmagatzemats a la nostra cua de redis del servidor. Per tal que el client pugui accedir al llistat d'insults indicarà la llista on es pujarà el contingut de la llista.
- X3, aquesta petició ens permet obtenir un únic insult del redis storage. A l'igual que l'anterior petició el client hauria d'indicar quina és la llista on s'hauria de guardar l'insult escollit.
- X4, aquesta petició inicia el procés de broadcast perqùe clients escoltin els insults publicats en el canal.
- X5, aquesta petició finalitza el procés de broadcast.

El format de *json* que enviem és el següent:

$$petition = \{\text{operation} : \backslash, \text{data} : \backslash\}$$

Seguint els anteriors models creem un procés quan iniciem el broadcast per no bloquejar el servei i que pugui seguir rebent peticions de manera concurrent.

Arquitectura n Nodes:

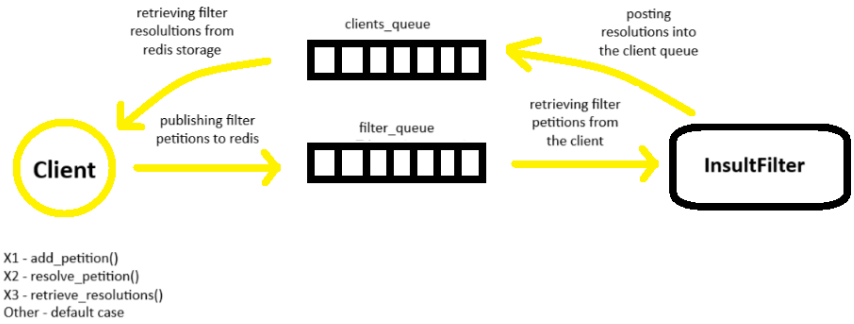


Abstracte 14. Arquitectura InsultService Redis de varius nodes.

En l'anterior abstracte podem observar quin ha estat el disseny seleccionat per la implementació de múltiples nodes amb el Middleware de Redis. En el nostre cas pràcticament hem reutilitzat tota l'arquitectura utilitzada en la implementació d'un únic node. L'única diferència destacable és que utilitzem un comptador per assignar un identificador a la cua de peticions pertinent del servei, d'aquesta manera distribuïm les peticions en múltiples llistes diferents.

InsultFilter

Arquitectura 1 Nodes:



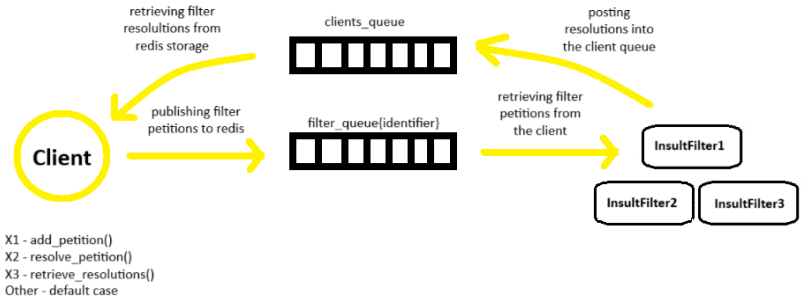
Abstracte 15. Arquitectura InsultFilter Redis d'un node.

La darrera imatge mostra l'arquitectura utilitzada per al InsultFilter. Com podem observar aquesta estructura és molt senzilla, ja que el client tan sols ha d'enviar les peticions cap a la cua de filtratge del servei i el servidor ja realitzarà la resolució d'aquestes peticions. Com en l'anterior arquitectura

(InsultService), també hem definit un conjunt d'identificadors d'operacions per indicar quina és la petició que realitza el client. Aquestes són:

- X1, aquesta petició ens permet afegir una petició de filtratge al *storage* de Redis, per poder fer-ho indiquem com a paràmetre addicional o *data* aquella petició que es vol afegir.
- X2, aquesta petició ens permet resoldre la petició més antiga que ha estat emmagatzemada. Per tal que el client pugui accedir al llistat d'insults indicarà la llista on es pujarà el contingut de la llista.
- X3, aquesta petició ens permet obtenir llista de resolucions que estan emmagatzemades al redis storage. A l'igual que l'anterior petició el client hauria d'indicar quina és la llista on s'hauria de guardar la llista de resolucions.

Arquitectura n Nodes:



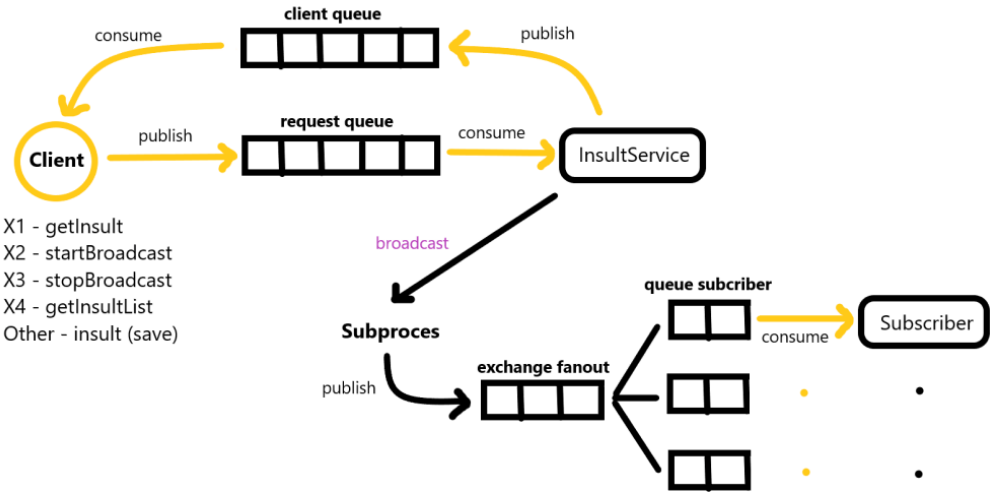
Abstracte 16. Arquitectura InsultFilter Redis de varius nodes.

Aquest diagrama mostra l'arquitectura utilitzada per al InsultFilter amb múltiples nodes de resolució. Com podem observar manté la mateixa estructura base que el InsultFilter estàtic d'un únic node, però ara fem que cada una de les llistes dels filter tingui un identificador, com en l'estructura de múltiples nodes del InsultService.

2.4 - RabbitMQ

InsultService

Arquitectura 1 Nodes:



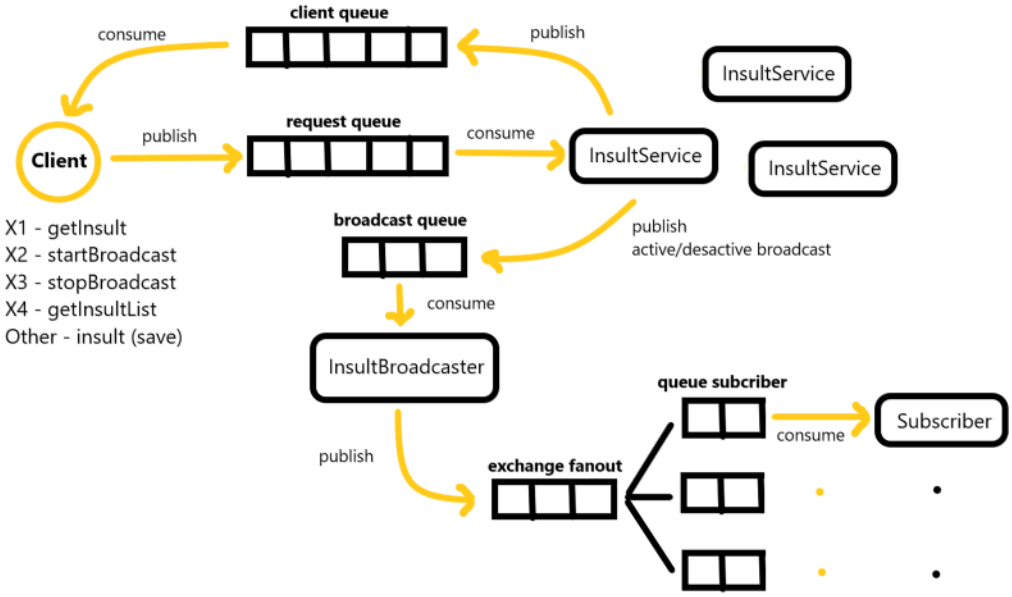
Abstracte 17. Arquitectura InsultService RabbitMQ d'un node.

En l'anterior figura observem com funciona tot el mecanisme de cues del InsultService. En aquest cas, el client publica events amb un identificador (X1,X2,...) que representen una funció específica que ha de realitzar el servidor i en cas de no tenir identificador, representa un insult que ha de guardar.

Per permetre la comunicació de servei a client (respostes), es realitza a partir d'una altra cua, la qual és creada de forma exclusiva pel client i passada juntament amb l'event quan sigui necessari (properties).

Per poder fer el broadcast i notificar els subscriptors, creem un subproces que està contínuament publicant events a una cua de tipus exchange fanout. Aquest tipus de cues permeten fer "forward" a les cues que estan subscrites a aquesta. D'aquesta manera, els subscriptors poden llegir el contingut de les seves pròpies cues (subscrites a la de tipues exchange fanout).

Arquitectura n Nodes:

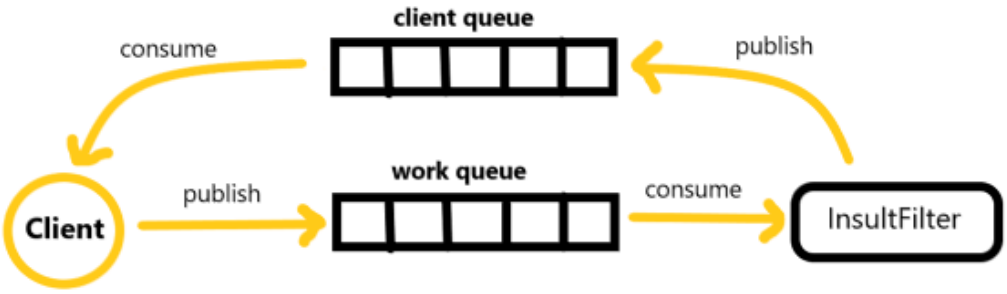


Abstracte 18. Arquitectura InsultService RabbitMQ de varius nodes.

En aquest cas, l'arquitectura és mont semblant a l'anterior, però per poder realitzar el broadcast, s'ha realitzat un nou servei el qual consumeix d'una cua. Totes les instàncies del InsultService, consumeixen d'una mateixa cua (balanceig de càrrega automàtica amb Round Robbin) i en cas que els hi arribi event relacionat amb el broadcast, el passen al InsultBroadcaster a partir de la seva cua.

InsultFilter

Arquitectura 1/n Nodes:



Abstracte 19. Arquitectura InsultFilter RabbitMQ d'un node.

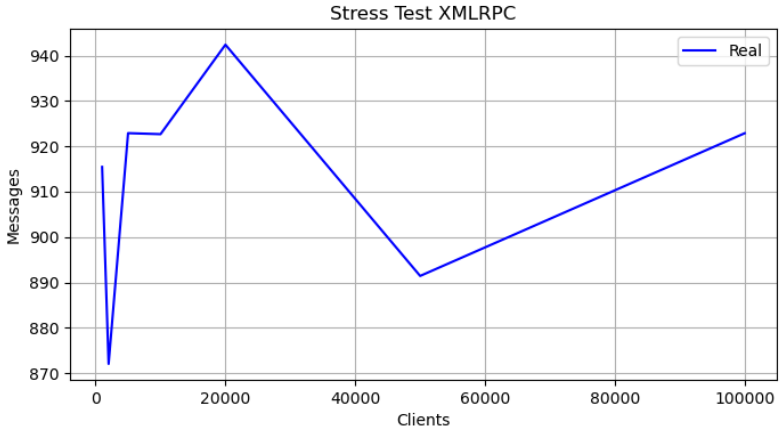
Aquest cas és molt més simple que els anteriors, el client publica el text amb insults o sense, el InsultFilter agafa els events i fa la seva feina, un cop acabada ho publica a la cua de resposta, la qual el client ha creat anteriorment i la passa per paràmetres.

A diferència dels altres casos, aquest no necessita una adaptació per l'arquitectura de múltiples nodes ja que la seva única feina es filtrar el text i retornar-lo a l'usuari. Tampoc necessitem gestionar l'ús de diversos consumidors a la cua, ja que Rabbit ja fa un load balancing amb Round Robbin.

3 – Stress testing

Per cada una de les proves de *stress testing* hem definit tant el nombre màxim de processos a executar com la quantitat de peticions que realitzarà cadascun dels processos llançats en l'execució. A continuació mostrarem les gràfiques que ens relacionen el nombre de peticions realitzades en comparació el nombre de consumicions.

3.1 - XMLRPC

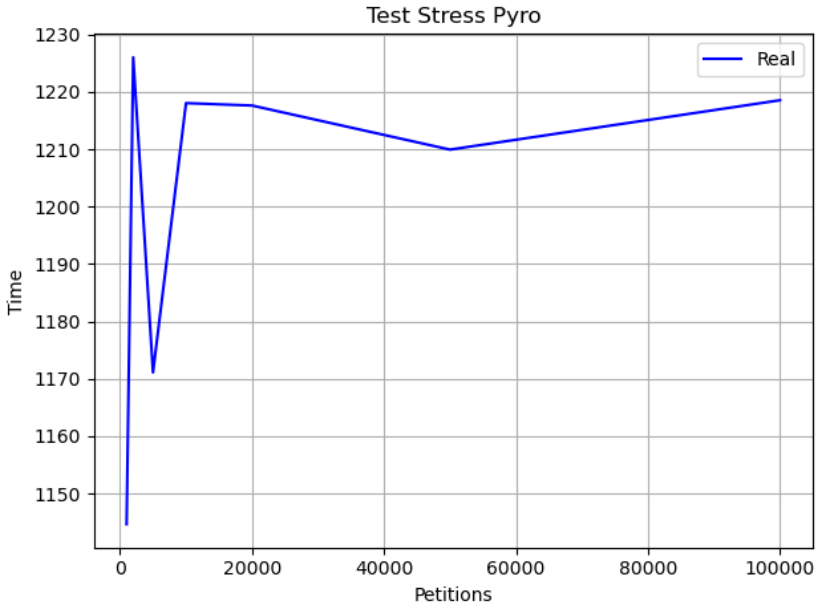


Abstracte 20. Resultat Stress test XMLRPC.

En l'anterior gràfica, podem observar els resultats obtinguts en la realització del *stress testing* a un únic node sobre el *middleware* de **XMLRPC** (InsultService). Observant el *plot* determinen que la mitja de peticions que pot realitzar aquest tipus de *framework* és d'unes **900 peticions per cada segon**. Trobem que els resultats obtinguts són correctes, ja que a l'ésser una **tecnologia bloquejant** fa que el client hagi d'esperar la resposta del servei per poder enviar altres peticions.

Endemés, **XMLRPC** és una tecnologia **acoblada en el temps i en l'espai**, això significa que el client ha de saber on se situa el servidor i han d'estar els dos (tan servidor com client) connectats simultàniament, perquè el client pugui efectuar la connexió amb el servidor i conseqüentment el servidor enviï la confirmació de la petició. Tot i que **XMLRPC** consti de les propietats anteriors (acoblada en el temps i en l'espai) no creiem que afectin directament en el rendiment, però sí que en l'escalabilitat del sistema en cas de tindre múltiples nodes.

3.2 - Pyro4



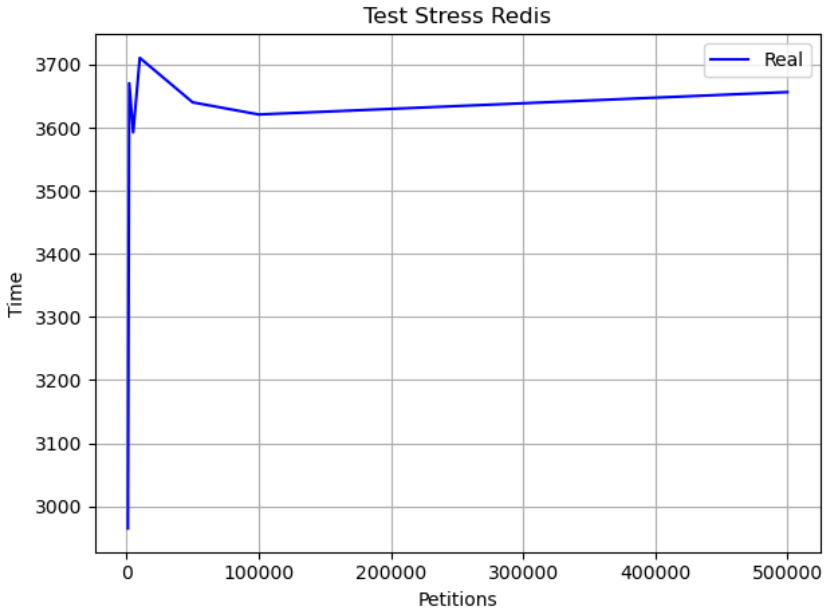
Abstracte 21. Resultat Stress test Pyro4.

En l'anterior *plot*, podem observar els resultats obtinguts en el *stress test* per la tecnologia de Pyro. Observant la gràfica podem determinar que la mitjana de peticions per segon és de 1200. Trobem que els resultats obtinguts són verídics pel fet que Pyro segueix una estructura **bloquejant** (síncrona), en què el client espera la resposta del servidor quan es realitza una petició.

Endemés, cal destacar que aquesta tecnologia igual que XMLRPC és **acoblat tant en el temps com l'espai**, així i tot podem observar com el rendiment de Pyro **és superior al mostrat per XMLRPC**. Les causes per la qual podem justificar aquests resultats:

- **XMLRPC** obre una **nova connexió per cada petició** enviada pel client, en canvi, Pyro pot mantenir una sessió.
- **XMLRPC** realitza un *parsing* de la informació enviada pel client en **format text XML**, mentre que **Pyro** efectua una **serialització** del contingut del paquet fent que sigui més **lleuger** i més **eficient** en l'intercanvi de dades.

3.3 - Redis



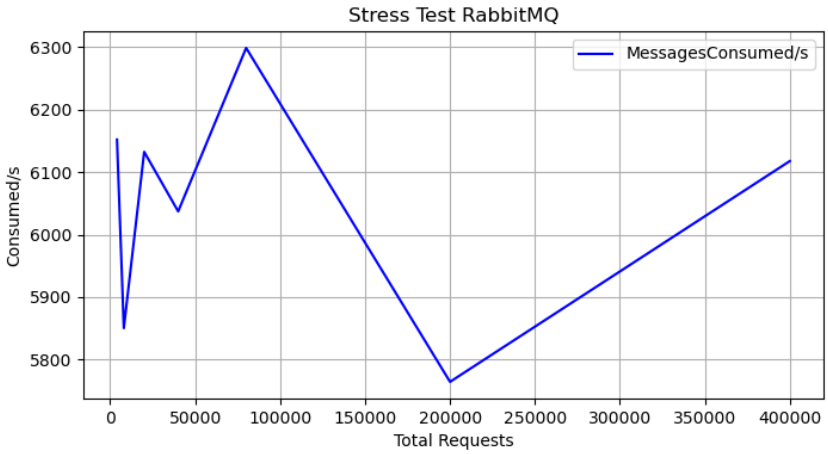
Abstracte 22. Resultat Stress test Redis.

En l'anterior abstracte podem observar els resultats obtinguts durant el stress test realitzat sobre un únic node de **Redis**. Observant el *plot* obtingut en la prova determinen que la mitjana de peticions que pot realitzar **Redis** és de 3600.

Les principals propietats que té aquesta tecnologia són: **arquitectura asíncrona** en l'àmbit de missatges (no bloquejant, permet a l'usuari encuar tantes peticions com ell cregui necessàries sense rebre resposta del servidor), **desacoblat en el temps** (permetem encuar peticions de clients sense que el servidor estigui actiu en aquell instant) i **acoblat en l'espai** (per tal que els clients indiquin a quina cua volen enviar el treball en qüestió d'un servidor Redis concret).

Degut a que és no bloquejant, dona millors resultats que els dos anteriors *middlewares* (XMLRPC i Pyro).

3.4 - RabbitMQ



Abstracte 23. Resultat Stress test RabbitMQ.

En l'anterior gràfica, observem els resultats obtinguts del *stress test* a un únic node de **RabbitMQ** (InsultFilter). Es pot visualitzar que tenim una mitjana de **6000 peticions per cada segon**. Trobem que els resultats obtinguts s'ajusten a la realitat ja que **RabbitMQ** és una tecnologia de comunicació **no bloquejant (asíncrona)**, fent que el client pugui enviar tantes peticions com desitgi sense la necessitat de què hagi d'esperar activament una resposta.

Endemés, cal destacar que aquest middleware és **desacoblat en el temps, però no en l'espai**, així permeten als seus usuaris que puguin enviar les peticions sense saber l'estat (si es troba actiu o no en aquell instant) del servidor i necessàriament saber la ubicació del servidor **RabbitMQ** (*host:port*) i el nom de la cua on s'indicarà la petició del client.

Com podem observar els resultats obtinguts per RabbitMQ són superiors a Redis, tot i que constin de les mateixes propietats (a nivell de (des)acoblament i sincronisme). Les causes principals en les quals ens basem són:

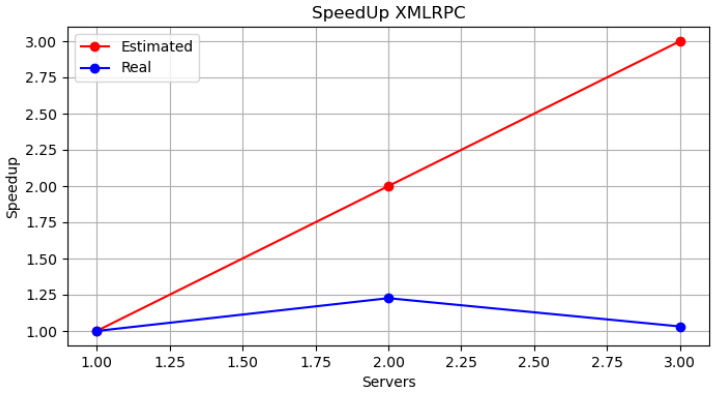
- **RabbitMQ** és una tecnologia dissenyada **l'intercanvi de missatges a partir de cues** (com per exemple el patró de disseny work queue).
- **Redis** està pensat per a funcionar com una **memòria cachè** (moltes lectures però poques escriptures).

En el nostre context, en provar les dues tecnologies seguint un model de cues **obtenim un resultat notòriament superior** per **RabbitMQ** que per **Redis**.

4 – Speedup de les arquitectures

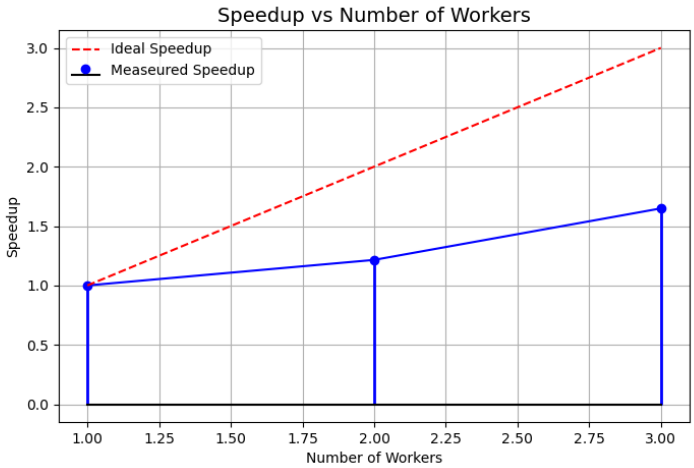
A continuació, mostrarem cadascuna de les gràfiques de *speed-up* obtingudes durant les proves realitzades per cadascun dels *middlewares*. Tots i cadascun dels plots generats aniran acompanyats de la seva explicació pertinent.

4.1 - XMLRPC



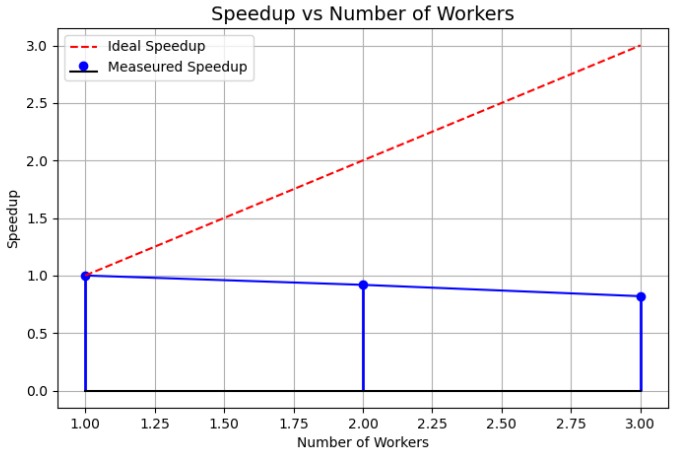
Abstracte 24. Resultat Speedup XMLRPC.

4.2 - Pyro4



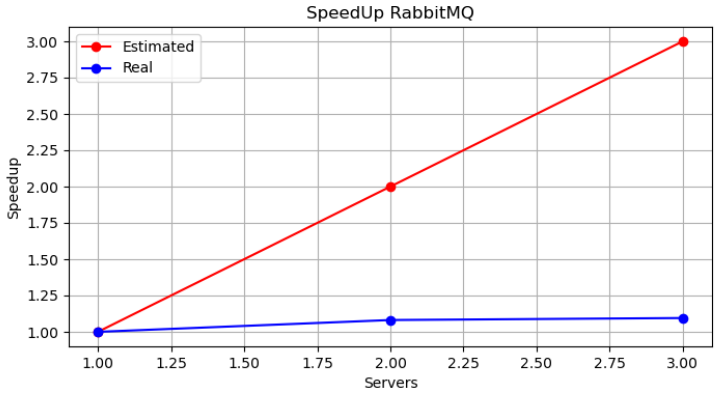
Abstracte 25. Resultat Speedup Pyro4.

4.3 - Redis



Abstracte 26. Resultat Speedup Redis.

4.4 - RabbitMQ



Abstracte 27. Resultat Speedup RabbitMQ.

5 – Comparació de les arquitectures

Cal destacar que els resultats obtinguts durant les proves de speedup no són del prou fiables, ja que hem obtingut resultats molt diferents dels esperats, degut o bé en la manera que realitzem les proves o bé per la implementació escollida. Així i tot, ens agradaria mencionar les arquitectures segons el seu rendiment de manera ascendent.

XMLRPC -> Pyro4 -> Redis -> RabbitMQ

Respecte a l'escalabilitat, deduïm segons els nostres coneixements i l'aprés a classe (no a partir dels resultats), que RabbitMQ és el middleware amb major escalabilitat i **XMLRPC** el que menys escalabilitat té.

RabbitMQ és el més **escalable** d'entre totes les tecnologies provades, ja que ens ofereix facilitat a l'hora d'efectuar escalabilitat horitzontal, ja que introdueix un **balanceig de càrrega de manera automàtica** per totes les peticions introduïdes en una mateixa cua.

Per altra banda, XMLRPC és el menys escalable d'entre tots els middlewares, ja que al seu baix rendiment, la seva sincronització client servidor i el seu acoblament (a nivell d'**espai** i temps). Necessitem saber on està situat cada node, en canvi, amb **Redis** i **RabbitMQ** únicament la cua que usen els serveis.

6 – Dynamic Scaling

6.1 - Decisions de disseny

Per poder realitzar el dynamic scaling, hem decidit utilitzar el framework **RabbitMQ** perquè hem vist que té un rendiment i una escalabilitat bastant alta. També, perquè hem observat que disposa d'un load balancing entre tots els consumidors d'una cua (amb Round Robbin) fent que no ens hàgim que preocupar per la sincronització entre els diferents workers.

Per altra banda, hem decidit que la cua on es publiquen els event sigui de tipus *exchange fanout* fent que faci forward de tots els event que li arriben a dos cues diferents:

- Work_queue. Cua d'on els workers consumiran els events.
- Count_queue. Cua que servirà com a auxiliar per calcular el número de workers necessaris.

Tot aquest mecanisme no seria necessari, ja que podriem utilitzar l'API de RabbitMQ per obtenir les dades, però aquesta té una freqüència d'actualització de 5 segons. En el nostre cas, volíem que el master realitzes la comprovació del número de nodes necessaris cada 2 segons per tenir major precisió i aquesta va ser la solució més senzilla que vam trobar.

Per poder realitzar el càlcul del número de workers necessaris hem usat la següent fórmula:

$$N = \left\lceil \frac{B + (\lambda \times T_r)}{C} \right\rceil$$

Abstracte 28. Fórmula per calcular el nombre de workers necessaris.

B = número d'event a la cua.

Lambda = número d'event/s per segon que arriba a la cua (rate).

Tr = temps amb el que es vol buidar la cua.

C = capacitat d'un worker (events/s que consumeix).

Un cop sabem quins són els paràmetres necessaris, expliquem d'on hem tret cadascuna:

- B -> en aquest cas únicament hem de llegir quants event hi ha a la cua Work_queue.
- Lambda -> calculem el número de events que hi ha abans i després de la cua Count_queue.
- Tr -> vam decidir posar aquest valor a **2**, per buidar la cua per cada iteració del master.
- C -> vam veure oportú introduir el valor de **1000**, el qual és estàtic. Pot ser un valor baix respecte els resultats, però vàlid per veure el correcte funcionament de l'arquitectura proposada.

Per acabar, hem decidit que els workers siguin instàncies del InsultFilter. Per llançar nous nodes, s'ha decidit iniciar nous processos.

6.2 - Codi

El master es basa en una única classe, la qual, disposa de diferents funcions. La que permet engegar el mateix servei és:

```python
def start_managing(self, num_max):
    print(" [*] Start working :)")
    while True:
        workers_need = self.obtain_num_workers()
        print(f'Workers need: {workers_need} - Working: {self.num_workers}')
        # Balance num of workers
        if workers_need != self.num_workers:
            self.up_down_workers(workers_need, num_max)
```

Abstracte 29. Funció principal de manegament del dynamic scaling.

A continuació mostrem la implementació de la funció que comprova contínuament quants workers són necessaris i llença de nous o mata segons la quantitat de peticions que arriben.

```python
def obtain_num_workers(self):
    # Obtain publish rate
    q1 = self.channel.queue_declare(queue=self.count, passive=True)
    time.sleep(2)
    q2 = self.channel.queue_declare(queue=self.count, passive=True)
    rate = q2.method.message_count - q1.method.message_count

    # Obtain the number of events in the queue
    q1 = self.channel.queue_declare(queue=self.work, passive=True)
    count = q1.method.message_count
    print(f'\n*Rate: {rate}\t*Queue: {count}\t*Consumer : {self.C}')
    num_workers = math.ceil((count+(rate*self.T))/self.C)

    self.result_rate.append(rate)
    self.result_workers.append(num_workers)
    return num_workers
```

Abstracte 30. Càlcul del número de workers.

La funció obtain_num_workers(), el que fa és calcular el nombre de nodes que necessiten usant la fórmula i els paràmetres que s'han comentat a l'anterior apartat.

```python
def up_down_workers(self, workers_need, num_max):
    # Up workers
    if workers_need > self.num_workers:
        print(" [!] Up Workers")
        for _ in range(workers_need-self.num_workers):
            # Start new worker
            if self.num_workers <= num_max:
                proc = subprocess.Popen(["python", self.path_worker],
                                        stdout = subprocess.DEVNULL)
                self.id_workers.append(proc)
                self.num_workers = self.num_workers + 1
                print(" -> + worker")
            else:
                print(f" [!] Limit workers: {num_max}")
                break
    # Down workers
    else:
        print(" [!] Down Workers")
        for _ in range(self.num_workers-workers_need):
            # Kill first worker we started
            proc = self.id_workers.pop(0)
            proc.terminate()
            proc.wait()
            self.num_workers = self.num_workers - 1
            print(" -> - worker")
```

Abstracte 31. Up i down dels workers.

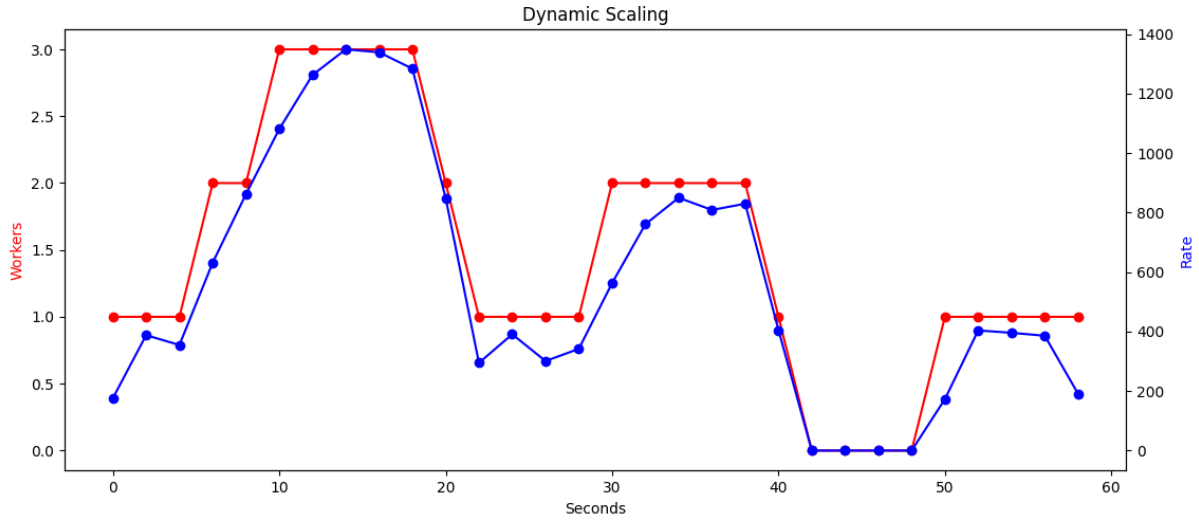
Acabem amb l'última funció i la que llença o mata workers. En cas de necessitar més, primer comprova si no hem arribat al límit (passat per paràmetre quan creem la classe) i en cas que tot estigui bé crea un nou procés amb el codi del worker.

Per altra banda, quan hem de matar un worker, seleccionem el que porta més estona actiu (el primer de la llista).

6.3 - Resultats

Per comprovar el seu correcte funcionament, disposem de 2 fitxers python, Main.py i TestMaster.py. El primer (Main.py), inicialitza la funció start_managing del Master. L'usuari (nosaltres) haurà d'executar altres processos (terminals) els quals publiquin a la cua, per veure així, com escala el sistema (mostra text per terminal).

Per altra banda, el TestMaster.py, és el fitxer que hem usat per generar una gràfica i poder observar com escala el sistema. Aquest codi, el que fa és enviar ràfegues d'events amb diferents freqüències a la cua. Aquest és el resultat:



Abstracte 32. Resultat escalat dinàmic.

A la gràfica és mostra com el sistema escala el número de nodes respecte a la freqüència d'arribada a la cua.

7 – Conclusions

Per acabar, pensem que ha estat una pràctica molt entretinguda i endemés ens ha servit per conèixer les diferents tecnologies de comunicació i poder fer una comparació, observant així, quina d'aquestes és la millor, tant en rendiment com en escalabilitat.

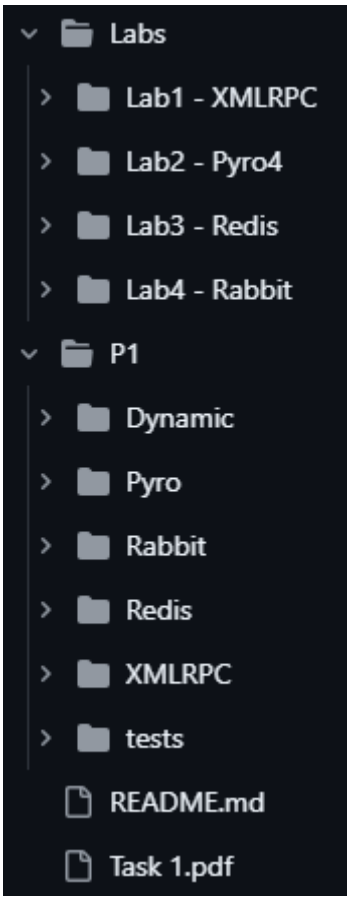
Altres aspectes a destacar, serien: l'aprenentatge de com realitzar diferents proves, com d'estrès i **speedup** i també acabar d'assolir el patró de disseny de **work queue**.

8 – Github

Enllaç repositori:

https://github.com/Lyuuubo/Distributed-Systems.git

Estructura:



Abstracte 33. Estructura del repositori Git.

Observem que disposem de 2 carpetes principals, la primera, **Labs**, conté tots els exemples i exercicis que hem realitzat durant les classes de laboratori.

Per altra banda, tenim la carpeta **P1**, on està tota la feina que hem realitzat. Aquesta està subdividida pel codi dels diferents middleware: **Pyro4, RabbitMQ, Redis i XMLRPC**. També podem observar que tenim un altre fitxer anomenat **Dynamic** el qual conté tot el codi relacionat amb el dynamic scaling.

Finalment, disposem de la carpeta **tests**, on estan els tests de funcionalitat, d'estràs i el codi necessàri per realitzar la comparació d'speedup.