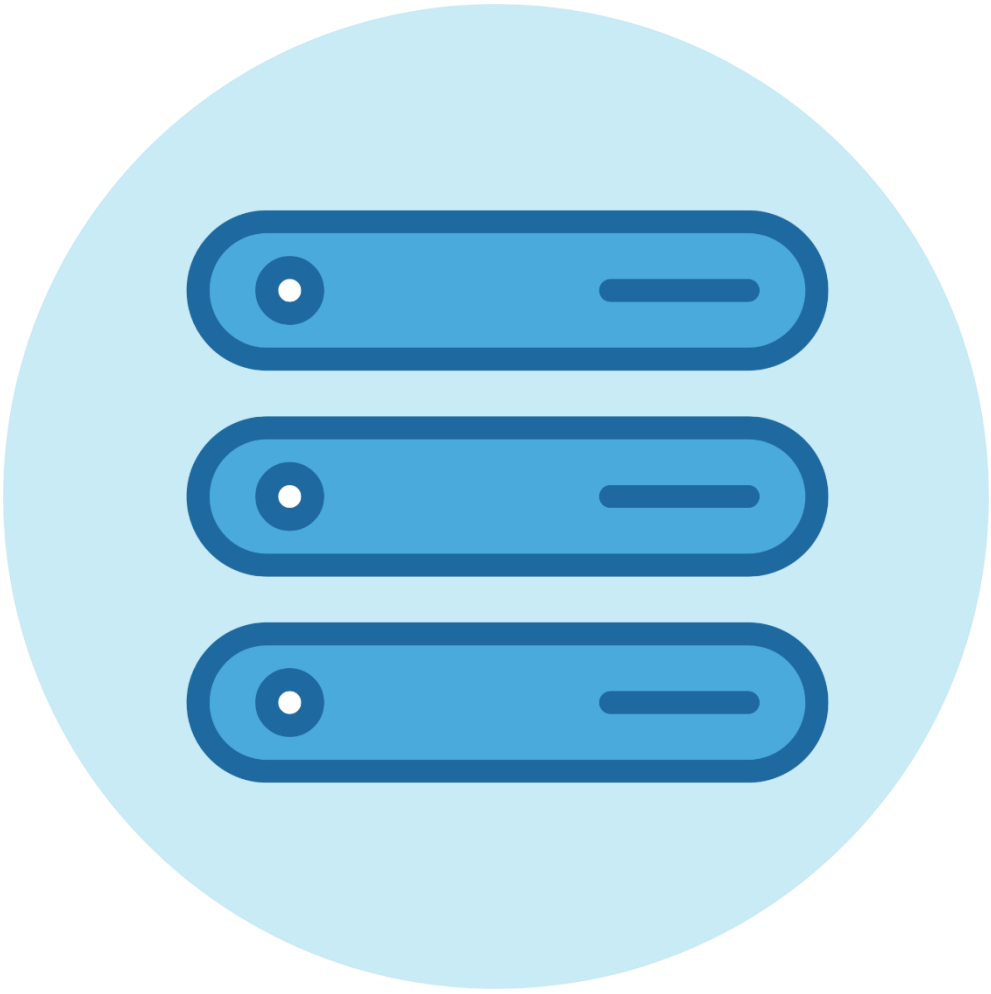
ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

Sergi Izquierdo Segarra
48280130W
2024-2025

SD P1:
Scaling distributed Systems using direct and indirect communication middleware



Sergi Izquierdo Segarra

48280130W

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

Sergi Izquierdo Segarra
48280130W
2024-2025

1- Index

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

Sergi Izquierdo Segarra
48280130W
2024-2025

Sergi Izquierdo Segarra
48280130W
2024-2025

2- XMLRPC

2.1- InsultService

2.1.1- Overview and Design

The XMLRPC-based InsultService uses a client-server architecture:

1. **InsultServer (insult_server_xmlrpc.py):** An XMLRPC server that exposes methods for managing insults and subscribers. It hosts the main logic.
2. **InsultClient (insult_client_xmlrpc.py):** An XMLRPC client application for adding new insults to the server and retrieving the current list of all insults.
3. **InsultSubscriber (insult_subscriber_xmlrpc.py):** An application that acts as both an XMLRPC client (to register with the InsultServer) and an XMLRPC server (to receive insult notifications).

2.1.2- Architecture and Features

- **InsultServer:**
  - Built using Python's xmlrpc.server.SimpleXMLRPCServer.
  - An instance of an InsultService class is registered with the server, making its public methods remotely callable (e.g., via http://127.0.0.1:8000/RPC2).
  - **Exposed Methods:**
    - add_insult(insult_string): Adds a new unique insult to an internal Python set.
    - get_insults(): Returns a list of all stored insults.
    - register_subscriber(subscriber_url): Stores the URL of an InsultSubscriber's XMLRPC server.
    - unregister_subscriber(subscriber_url): Removes a subscriber's URL.
  - **Broadcasting:** A background daemon thread runs every 5 seconds. It selects a random insult from its internal set. For each registered subscriber_url, it creates an xmlrpc.client.ServerProxy and calls a predefined method (like receive_insult_notification) on the subscriber's server.
- **InsultClient:**
  - Uses xmlrpc.client.ServerProxy to connect to the InsultServer.
  - Calls the server's add_insult and get_insults methods.
- **InsultSubscriber:**
  - Server Role: Starts its own SimpleXMLRPCServer on a unique port (e.g., 9001). This server exposes a method like

receive_insult_notification(insult_string) that the main InsultServer calls.
    - Client Role: After starting its server, it uses xmlrpc.client.ServerProxy to connect to the main InsultServer and calls register_subscriber, passing its own server's URL.
- Communication: Configured to use 127.0.0.1 for local communication to potentially reduce latency, as suggested in course materials.

2.1.3- Code Structure

- **insult_server_xmlrpc.py**: Contains the InsultService class (with core logic and broadcasting thread) and the main function to run the XMLRPC server.
- **insult_client_xmlrpc.py**: Implements the client logic for adding and retrieving insults.
- **insult_subscriber_xmlrpc.py**: Contains the InsultNotificationHandler class (for receiving broadcasts) and logic to run its own XMLRPC server and register with the main server.

All scripts use Python's built-in xmlrpc.server and xmlrpc.client modules.

2.1.4- How to Run

All commands assume the virtual environment is activated and the user is in the xmlrpc_insult_service directory. You can use the run script located in the same folder, or do the steps manually:

1. Start the InsultServer (Terminal 1): `python insult_server_xmlrpc.py`
2. Start one or more InsultSubscribers (Terminal 2 [or more, but manually modifying each port for example 9001, 9002… for each Subscriber]): `python insult_subscriber_xmlrpc.py`
3. Run the InsultClient (Terminal 3): `python insult_client_xmlrpc.py`

2.1.5- Observed Behavior

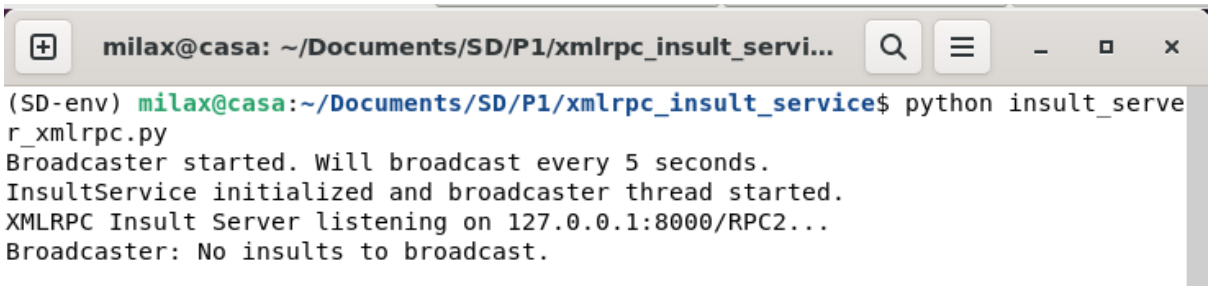
The system functions as intended:

- The InsultServer starts and listens for requests.
- InsultSubscriber(s) start their own servers, then register their listening URLs with the main InsultServer.
- The InsultClient successfully adds insults (uniqueness is maintained by the server's internal set) and retrieves the list of insults.
- Registered InsultSubscriber(s) receive periodic broadcasts of random insults from the InsultServer via XMLRPC calls to their receive_insult_notification method.

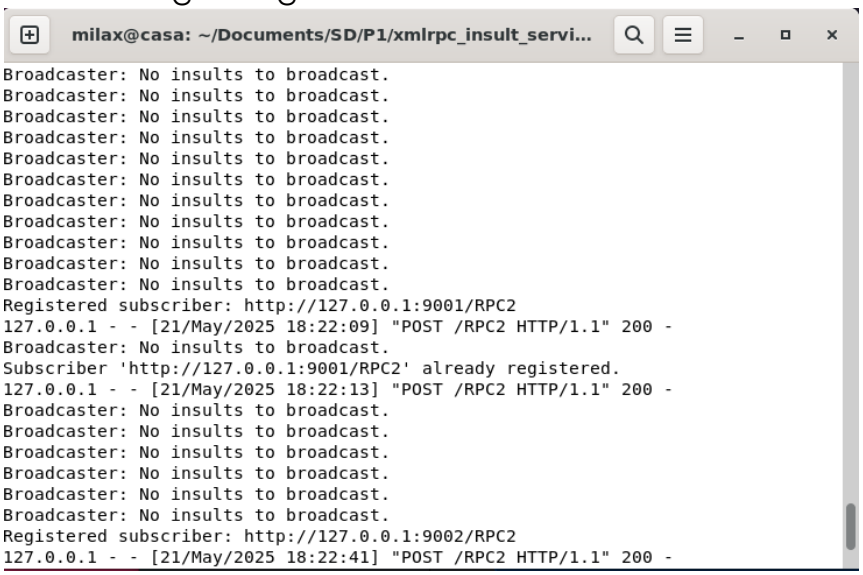
2.1.6- Screenshots

Server:

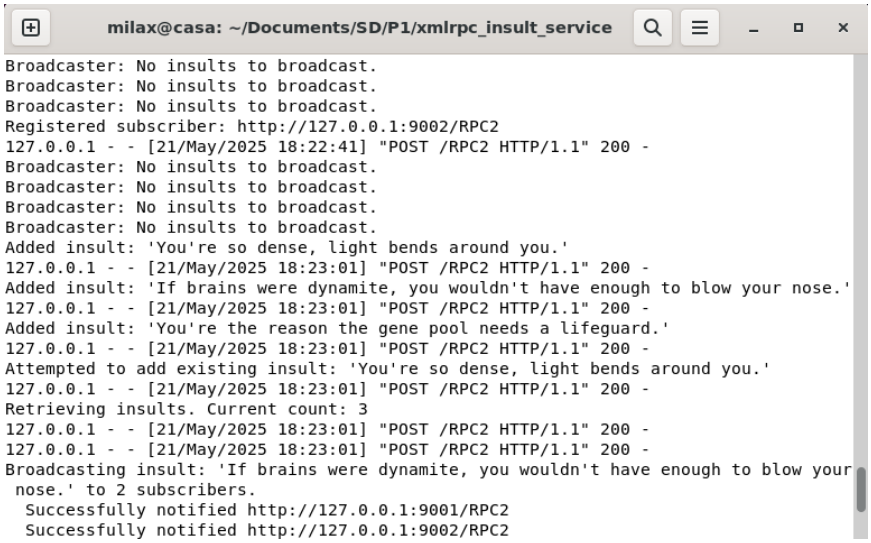
No insults or subscribers:



Subscriber gets registered:



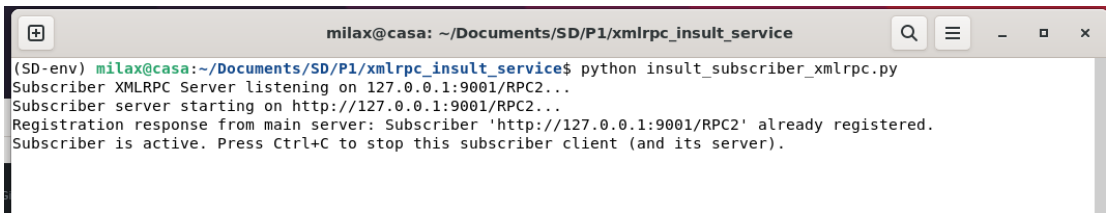
Insults added by client:



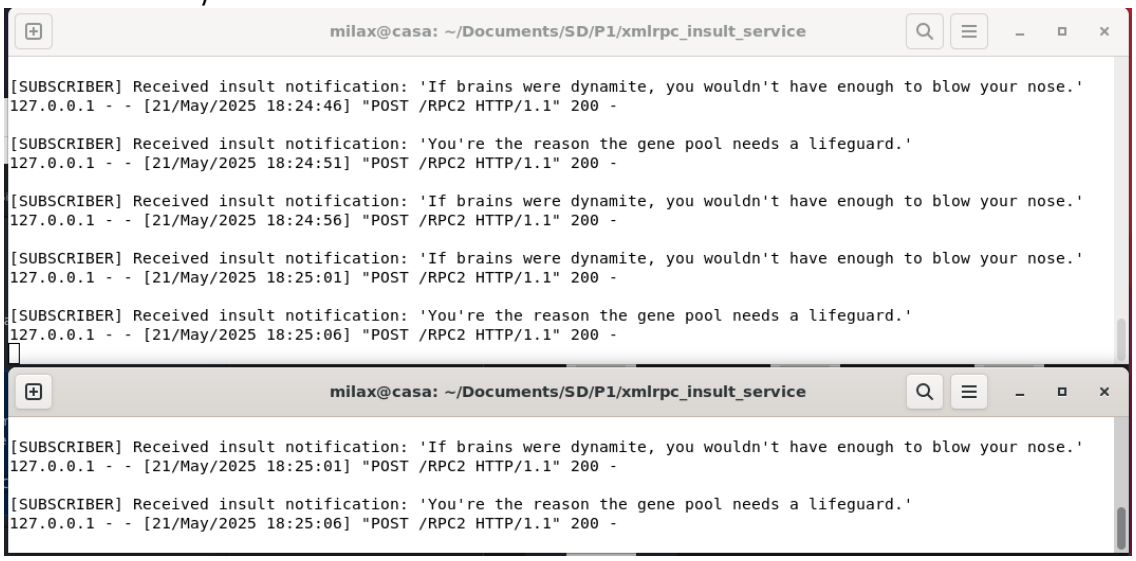
ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

Sergi Izquierdo Segarra
48280130W
2024-2025

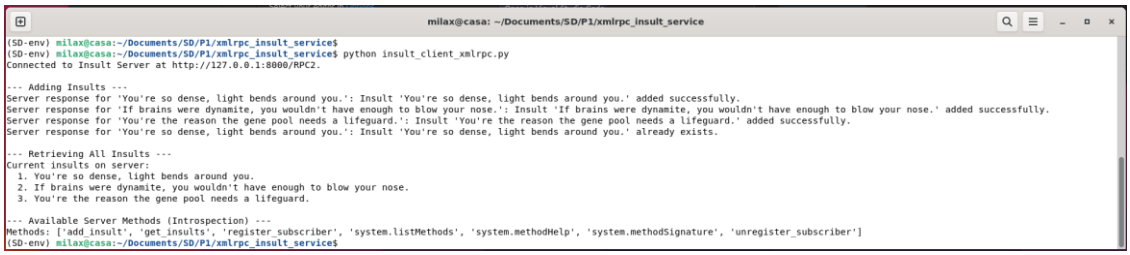
Subscriber:



Subscribers getting insults (There's two clients, one in port 9001, then other in 9002):



Client:



2.2- InsultFilter

2.2.1- Overview

The service uses a client-server model via XMLRPC:

- **FilterServer (filter_server_xmlrpc.py):** An XMLRPC server that receives texts, queues them internally for processing, filters them using a predefined list of insults, and stores the results.
- **FilterClient (filter_client_xmlrpc.py):** An XMLRPC client that submits texts to the server and can retrieve the list of filtered results.

2.2.2- Architecture

- Server (FilterService class):
  - Exposes methods via XMLRPC (e.g., on http://127.0.0.1:8001/RPC2).
  - submit_text_for_filtering(text): Client sends text here. Text is added to an internal Python queue.Queue.
  - get_filtered_results(): Client calls this to get a list of all processed (original, filtered, timestamp) texts.
  - An internal worker thread continuously takes texts from the queue, applies a basic case-insensitive filter to replace known insults with "CENSORED", and stores results in a list.
- Client:
  - Uses xmlrpc.client.ServerProxy to connect and call server methods.

2.2.3- Code Structure

- **filter_server_xmlrpc.py:** Contains the FilterService class (with the internal queue, worker thread, filtering logic) and runs the SimpleXMLRPCServer.
- **filter_client_xmlrpc.py:** Submits texts and retrieves results using XMLRPC.

2.2.4- How to Run

Assumes virtual environment is active and in the xmlrpc_filter_service directory. You can use the run script located in the same folder, or do the steps manually:

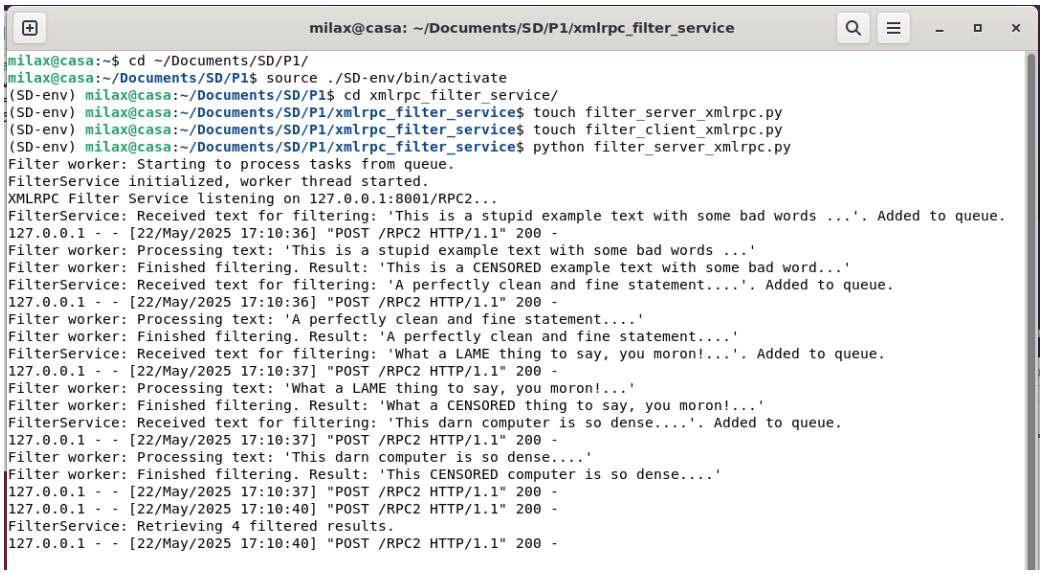
1. Start Server (Terminal 1): `python filter_server_xmlrpc.py`
2. Run Client (Terminal 2): `python filter_client_xmlrpc.py`

2.2.5- Observed Behavior

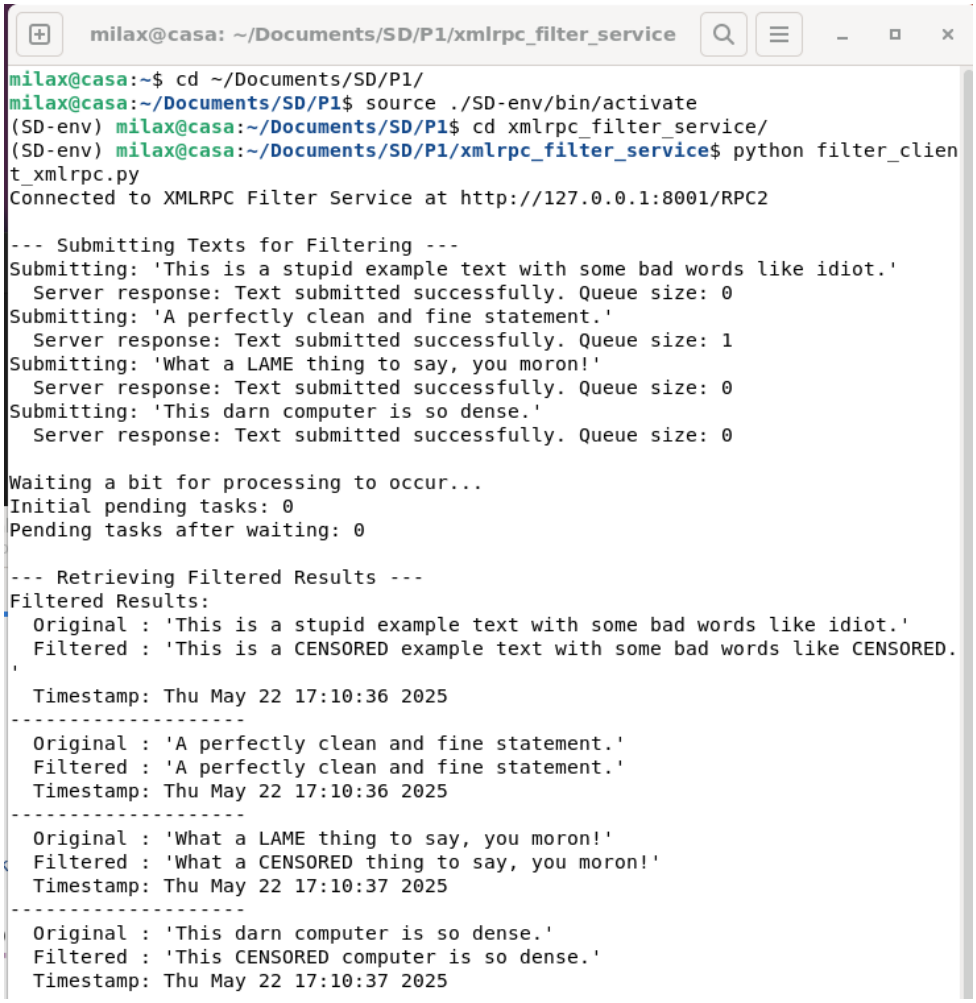
Clients submit texts to the server. The server processes these texts in its worker thread, replacing known insults. Clients can then retrieve the list of original and filtered texts.

2.2.6- Screenshots

Server:



Client:



3- Pyro4

3.1- InsultService

3.1.1- Overview and Design

The Pyro4 InsultService consists of three main components:

1. **InsultServer (insult_server_pyro.py):** A central Pyro4 object that manages the collection of insults, registers subscribers, and broadcasts random insults periodically.
2. **InsultClient (insult_client_pyro.py):** A client application that interacts with the 'InsultServer' to add new insults and retrieve the current list of insults.
3. **InsultSubscriber (insult_subscriber_pyro.py):** An application that can receive insult broadcasts from the 'InsultServer'. It hosts its own minimal Pyro4 object, 'NotificationReceiver' for this reason.

This design is inspired from the available Pyro example files in campusvirtual: *pyroserver.py, pyroclient.py, subject.py and observer.py* principally.

For the components to work, a Pyro4 Name Server must be running in the background (Executing this command line in a separate cmd: 'python -m Pyro4.naming) for the components to locate each other.

3.1.2- Architecture and Features

- **InsultServer:**
  - This class is exposed as a Pyro4 object using '@Pyro4.expose' and '@Pyro4.behavior(instance_mode="single") to ensure a single, shared instance.
  - It registers itself with the Pyro4 Name Server under the logical name "example.insult.service".
  - **Exposed Methods:**
    - add_insult(insult_string): Adds a new unique insult.
    - get_insults(): Returns a list of all current insults.
    - register_subscriber(subscriber_uri_str): Stores the string URI of a subscriber's 'NotificationReceiver' object.
    - Unregister_subscriber(subscriber_uri_str): Removes a subscriber's URI.
  - **Broadcasting:** A background daeomn thread runs every 5 seconds. If insults and registered subscriber URIs exist, it selects a random insult. For each URI, it creates a 'Pyro4.Proxy(sub_uri)' and calls the 'receive_insult' method on the subscriber's remote 'NotificationReceiver' object.

- **InsultClient:**
  - Looks up the 'InsultServer' from the Name Server.
  - Calls 'add_insult' and 'get_insults' methods on the server proxy.

- **InsultSubscriber:**
  - Defines a 'NotificationReceiver' class with an '@Pyro4.expose' method **receive_insult(insult_message)**. This class acts as the "observer" part. [Like *observer.py* from the examples]
  - The 'InsultSubscriber' script starts its own Pyro4 daemon and registers an instance of 'NotificationReceiver' with this local daemon, obtaining a **unique Pyro URI** for it, this way we can just open several terminals without having to modify any code and be able to connect to the server.
  - It then acts as a client to the main 'InsultServer' and calls the function '**register_subscriber**', passing the string representation of its URI.
  - Finally, it runs its local daemon's 'requestLoop()' to listen for incoming 'receive_insult' calls from the 'InsultServer'

3.1.3- Code Structure

- **insult_server_pyro.py**: Contains the InsultServer class and the start_server() function to initialize and run the server, including registration with the Name Server.
- **insult_client_pyro.py**: Implements the client logic for interacting with the InsultServer's add_insult and get_insults methods.
- **insult_subscriber_pyro.py**: Contains the NotificationReceiver class and the main logic for the subscriber to start its own daemon, register its receiver, and then register itself with the main InsultServer.

3.1.4- How to Run

All commands shown here assume the virtual environment is activated and the user is in the 'pyro_insult_service' directory. You can use the run script located in the same folder, or do the steps manually:

1. Start Pyro4 Name Server (Terminal 1): `python -m Pyro4.naming`
2. Start the InsultServer (Terminal 2): `python insult_server_pyro.py`
3. Start one (or more) InsultSubscribers (Terminal 3 [or more]): `python insult_subscriber_pyro.py`
4. Run the InsultClient (Terminal 4): `python insult_client_pyro.py`

3.1.5- Observed Behavior

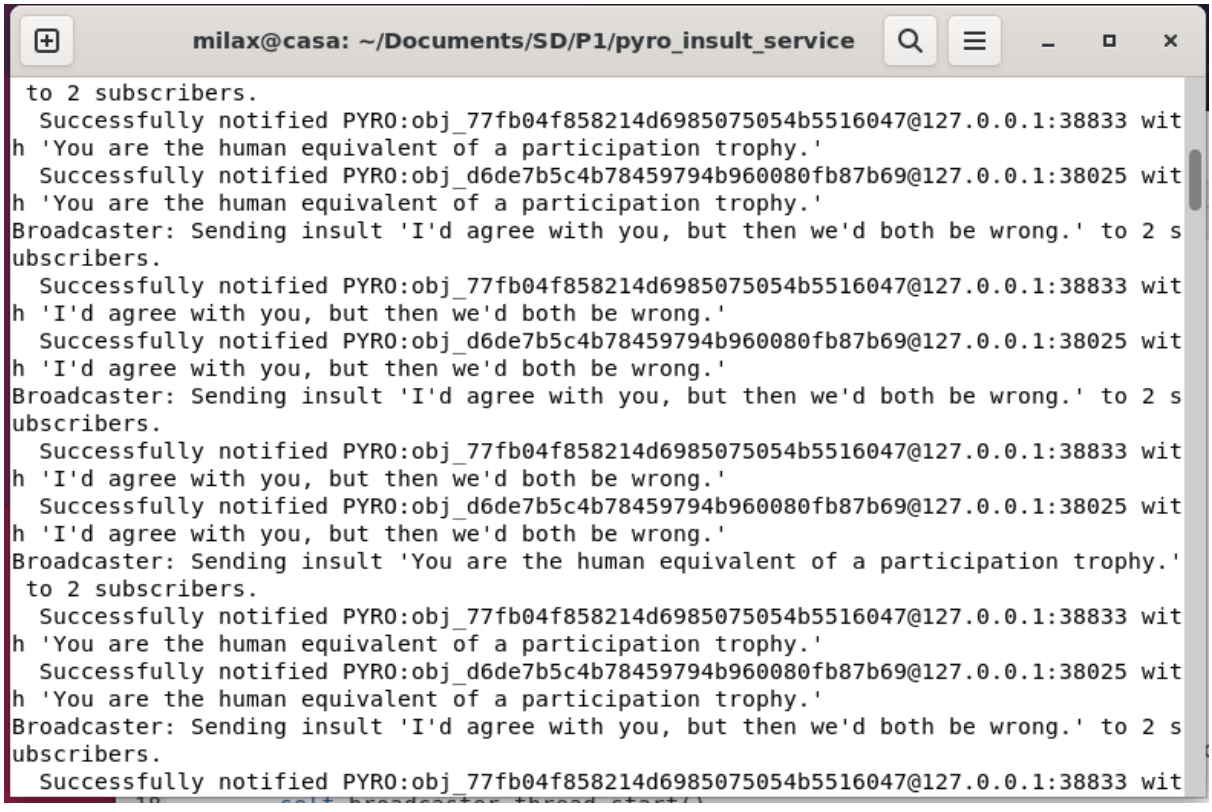
The system functions as expected:

- The **InsultServer** starts and registers with the Name Server.
- **InsultSubscriber**(s) start, register their notification endpoints with their local daemons, and then successfully register their URIs with the InsultServer.
- The **InsultClient** can add insults to the server (duplicates are handled) and retrieve the current list.
- Registered InsultSubscriber(s) receive periodic broadcasts of random insults from the InsultServer.
- Upon shutdown (Ctrl+C), components attempt to clean up their registrations from the Name Server and shut down their daemons.

3.1.6- Screenshots

Server:

```
(SD-env) milax@casa:~/Documents/SD/P1/pyro_insult_service$ python insult_server_pyro.
py
Broadcaster: Starting periodic broadcasts every 5 seconds.
InsultServer initialized and broadcaster thread started.
InsultServer ready. URI: PYRO:obj_9358743f6c3745b5aaa1b34b17d458cb@127.0.0.1:38491
Registered as 'example.insult.service' in the Name Server.
Subscriber URI PYRO:obj_77fb04f858214d6985075054b5516047@127.0.0.1:38833 registered.
Subscriber URI PYRO:obj_d6de7b5c4b78459794b960080fb87b69@127.0.0.1:38025 registered.
```



ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

Sergi Izquierdo Segarra
48280130W
2024-2025

Subscriber:

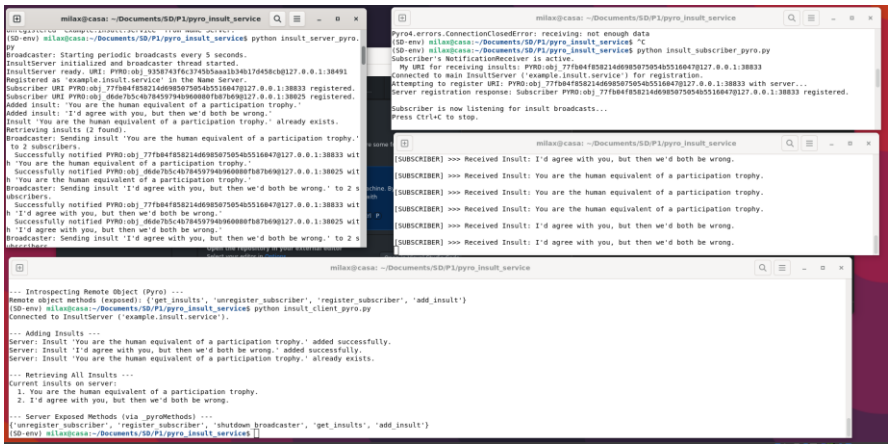
```
milax@casa: ~/Documents/SD/P1/pyro_insult_service
Pyro4.errors.ConnectionClosedError: receiving: not enough data
(SD-env) milax@casa:~/Documents/SD/P1/pyro_insult_service$ ^C
(SD-env) milax@casa:~/Documents/SD/P1/pyro_insult_service$ python insult_subscriber_pyro.py
Subscriber's NotificationReceiver is active.
  My URI for receiving insults: PYRO:obj_77fb04f858214d6985075054b5516047@127.0.0.1:38833
Connected to main InsultServer ('example.insult.service') for registration.
Attempting to register URI: PYRO:obj_77fb04f858214d6985075054b5516047@127.0.0.1:38833 with server...
Server registration response: Subscriber PYRO:obj_77fb04f858214d6985075054b5516047@127.0.0.1:38833 registered.

Subscriber is now listening for insult broadcasts...
Press Ctrl+C to stop.
```

```
milax@casa: ~/Documents/SD/P1/pyro_insult_service
[SUBSCRIBER] >>> Received Insult: You are the human equivalent of a participation trophy.

[SUBSCRIBER] >>> Received Insult: I'd agree with you, but then we'd both be wrong.

[SUBSCRIBER] >>> Received Insult: I'd agree with you, but then we'd both be wrong.

[SUBSCRIBER] >>> Received Insult: I'd agree with you, but then we'd both be wrong.

[SUBSCRIBER] >>> Received Insult: I'd agree with you, but then we'd both be wrong.

[SUBSCRIBER] >>> Received Insult: I'd agree with you, but then we'd both be wrong.
```

Client:

```
milax@casa: ~/Documents/SD/P1/pyro_insult_service

--- Introspecting Remote Object (Pyro) ---
Remote object methods (exposed): {'get_insults', 'unregister_subscriber', 'register_subscriber', 'add_insult'}
(SD-env) milax@casa:~/Documents/SD/P1/pyro_insult_service$ python insult_client_pyro.py
Connected to InsultServer ('example.insult.service').

--- Adding Insults ---
Server: Insult 'You are the human equivalent of a participation trophy.' added successfully.
Server: Insult 'I'd agree with you, but then we'd both be wrong.' added successfully.
Server: Insult 'You are the human equivalent of a participation trophy.' already exists.

--- Retrieving All Insults ---
Current insults on server:
  1. You are the human equivalent of a participation trophy.
  2. I'd agree with you, but then we'd both be wrong.

--- Server Exposed Methods (via _pyroMethods) ---
{'unregister_subscriber', 'register_subscriber', 'shutdown_broadcaster', 'get_insults', 'add_insult'}
(SD-env) milax@casa:~/Documents/SD/P1/pyro_insult_service$
```

Everything together:



3.2- InsultFilter

3.2.1- Overview

The Pyro4-based InsultFilter service involves three main components interacting via a Pyro4 Name Server:

1. **FilterDispatcher (filter_dispatcher_pyro.py):** A central Pyro4 server that receives texts from clients. It distributes these texts to registered workers for actual filtering and centrally stores the results.
2. **FilterWorker (filter_worker_pyro.py):** A Pyro4 server that performs the text filtering. Multiple instances can run. Each worker registers with the FilterDispatcher upon startup.
3. **FilterClient (filter_client_pyro.py):** A client application that submits texts to the FilterDispatcher and can retrieve the processed results from it.

The design is influenced by standard Pyro4 client-server patterns, similar to concepts in the provided Pyro4 examples (e.g., pyroserver.py, observer.py for registration ideas).

3.2.2- Architecture

- **Pyro4 Name Server:** Must be running for components to locate each other.
- **FilterDispatcher (filter_dispatcher_pyro.py):**
    - Exposed as a Pyro4 object (e.g., PYRONAME:example.filter.dispatcher).
    - register_worker(worker_uri_str): Method for FilterWorker instances to register their URI.
    - submit_text_for_filtering(original_text): Receives text from a client. It selects a registered worker, creates a proxy to that worker and calls a method on the worker, like process_this_text(text, known_insults), to perform the filtering. The worker returns the filtered text to the dispatcher.
    - The dispatcher stores the original text, the filtered text returned by the worker, the worker's identity, and a timestamp.
    - get_filtered_results(): Returns the list of all centrally stored filtered results.

- **FilterWorker (filter_worker_pyro.py):**
    - Each instance is an exposed Pyro4 object.
    - On startup, it registers its own Pyro URI with the FilterDispatcher.
    - Exposes process_this_text(original_text, known_insults_list): This method receives the text and the list of insults from the dispatcher, performs the case-insensitive replacement of insults with "CENSORED", and returns the filtered string to the dispatcher.
- **FilterClient (filter_client_pyro.py):**
    - Looks up the FilterDispatcher via PYRONAME.
    - Calls submit_text_for_filtering() on the dispatcher.
    - Can call get_filtered_results() on the dispatcher.

3.2.3- Code Structure

- **filter_dispatcher_pyro.py:** Contains the FilterDispatcher class, managing worker registration, task delegation, and central result storage.
- **filter_worker_pyro.py:** Contains the FilterWorker class, which has the actual text filtering logic.
- **filter_client_pyro.py:** Implements the client logic to submit texts and retrieve results via the dispatcher.

3.2.4- How to Run

Assumes virtual environment active, Name Server running, and in pyro_filter_service directory. You can use the run script located in the same folder, or do the steps manually:

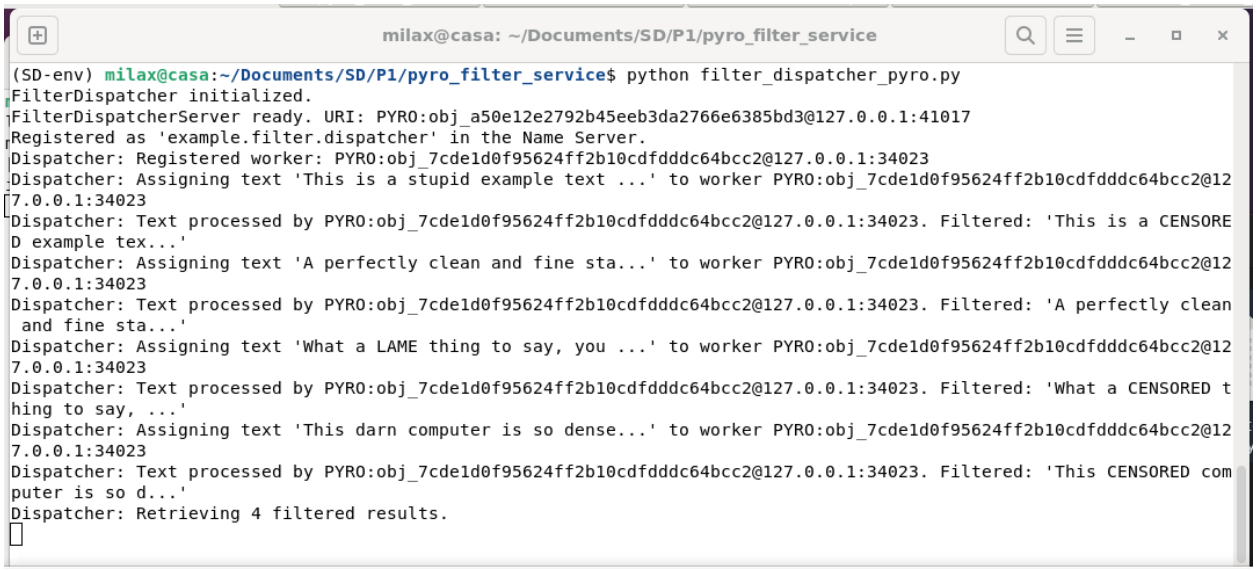
1. Start Pyro4 Name Server (Terminal 1): `python -m Pyro4.naming`
2. Start the Dispatcher (Terminal 2): `python filter_dispatcher_pyro.py`
3. Start one (or more) Workers (Terminal 3 [or more]): `python filter_worker_pyro.py`
4. Run the Filter Client (Terminal 4): `python filter_client_pyro.py`

3.2.5- Observed Behavior

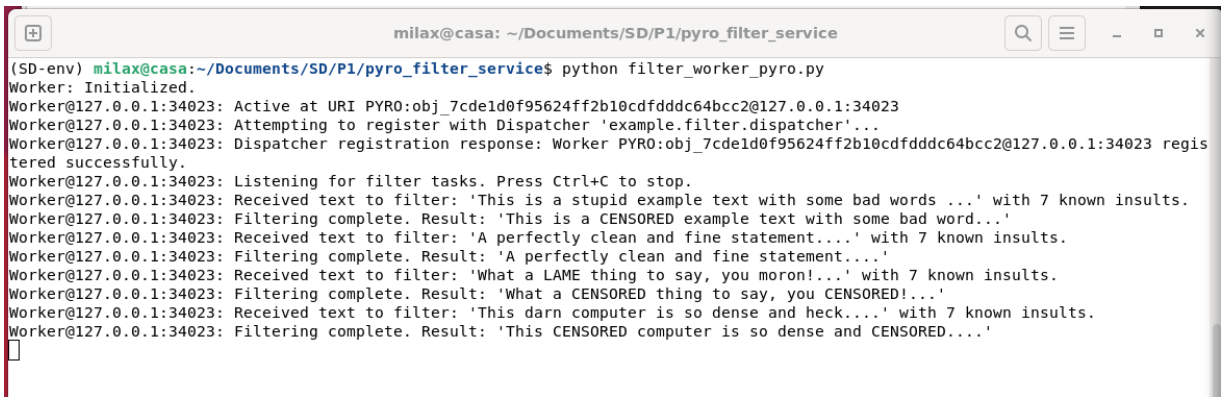
The client submits texts to the dispatcher. The dispatcher assigns these tasks to available workers. Workers perform the filtering and return results to the dispatcher. The client can then retrieve the aggregated, centrally stored results from the dispatcher. Multiple workers share the processing load.

3.2.6- Screenshots

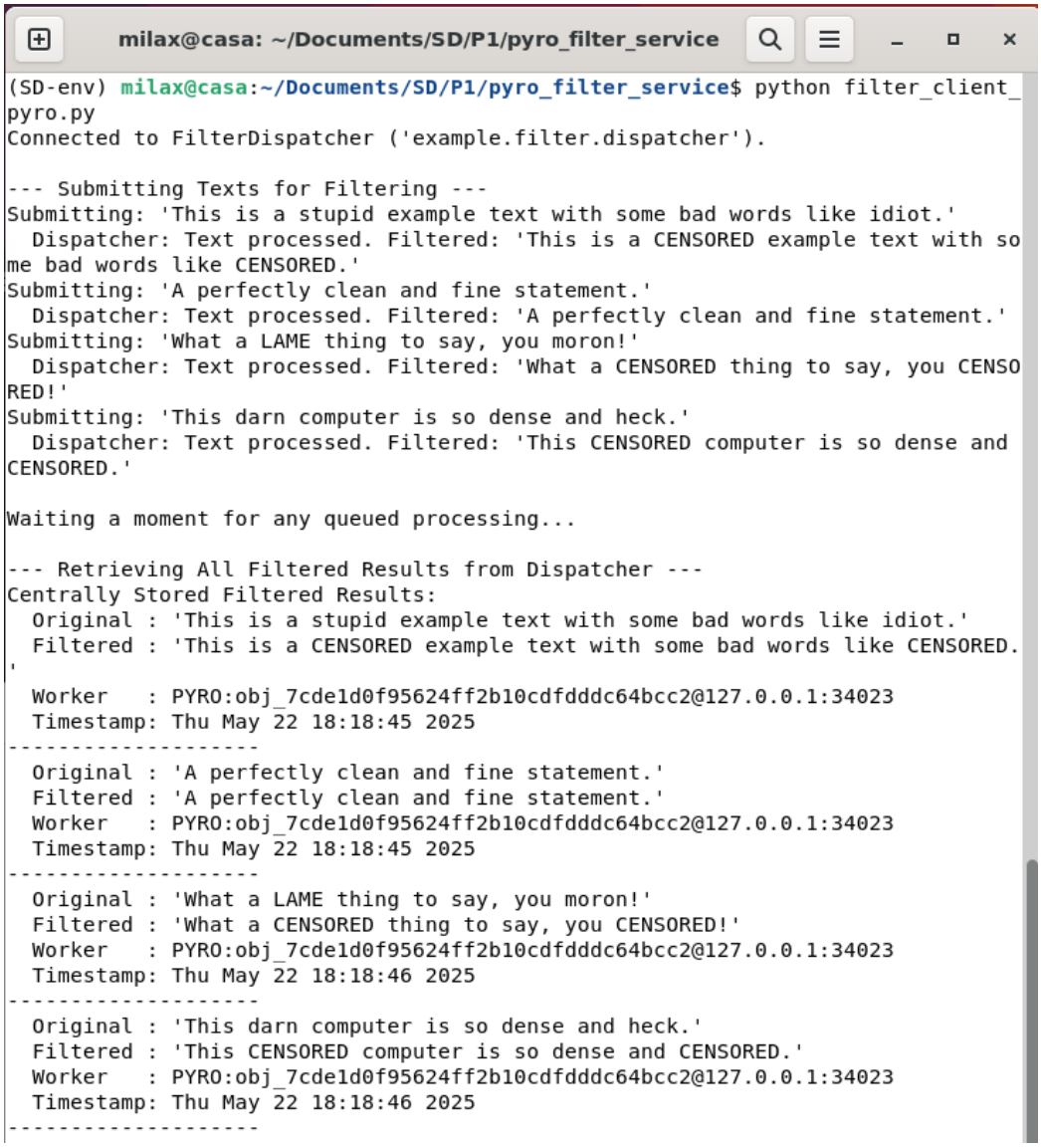
Dispatcher:



Worker:



Client:


milax@casa: ~/Documents/SD/P1/pyro_filter_service
(SD-env) milax@casa:~/Documents/SD/P1/pyro_filter_service$ python filter_client_
pyro.py
Connected to FilterDispatcher ('example.filter.dispatcher').
--- Submitting Texts for Filtering ---
Submitting: 'This is a stupid example text with some bad words like idiot.'
Dispatcher: Text processed. Filtered: 'This is a CENSORED example text with so
me bad words like CENSORED.'
Submitting: 'A perfectly clean and fine statement.'
Dispatcher: Text processed. Filtered: 'A perfectly clean and fine statement.'
Submitting: 'What a LAME thing to say, you moron!'
Dispatcher: Text processed. Filtered: 'What a CENSORED thing to say, you CENSO
RED!'
Submitting: 'This darn computer is so dense and heck.'
Dispatcher: Text processed. Filtered: 'This CENSORED computer is so dense and
CENSORED.'
Waiting a moment for any queued processing...
--- Retrieving All Filtered Results from Dispatcher ---
Centrally Stored Filtered Results:
Original : 'This is a stupid example text with some bad words like idiot.'
Filtered : 'This is a CENSORED example text with some bad words like CENSORED.
'
Worker : PYRO:obj_7cde1d0f95624ff2b10cdfdddc64bcc2@127.0.0.1:34023
Timestamp: Thu May 22 18:18:45 2025
Original : 'A perfectly clean and fine statement.'
Filtered : 'A perfectly clean and fine statement.'
Worker : PYRO:obj_7cde1d0f95624ff2b10cdfdddc64bcc2@127.0.0.1:34023
Timestamp: Thu May 22 18:18:45 2025
Original : 'What a LAME thing to say, you moron!'
Filtered : 'What a CENSORED thing to say, you CENSORED!'
Worker : PYRO:obj_7cde1d0f95624ff2b10cdfdddc64bcc2@127.0.0.1:34023
Timestamp: Thu May 22 18:18:46 2025
Original : 'This darn computer is so dense and heck.'
Filtered : 'This CENSORED computer is so dense and CENSORED.'
Worker : PYRO:obj_7cde1d0f95624ff2b10cdfdddc64bcc2@127.0.0.1:34023
Timestamp: Thu May 22 18:18:46 2025

4- Redis

4.1. InsultService

This section outlines the InsultService implementation using Redis, leveraging its in-memory data structures (Sets) for storing unique insults and its Publish/Subscribe (Pub/Sub) mechanism for broadcasting.

4.1.1- Overview and Design

The Redis-based InsultService is composed of several distinct Python scripts that interact with a running Redis server:

1. **insult_adder_redis.py**: A client script to add new insults to a Redis Set, ensuring uniqueness.
2. **insult_getter_redis.py**: A client script to retrieve and display all stored insults from the Redis Set.
3. **insult_broadcaster_redis.py**: A daemon script that periodically fetches a random insult from the Redis Set and publishes it to a specific Redis Pub/Sub channel.
4. **insult_subscriber_redis.py**: A script that subscribes to the Redis Pub/Sub channel to receive and display broadcasted insults.

This design utilizes Redis as the central hub for data storage (Set for insults) and message passing (Pub/Sub for broadcasts), inspired by the campus Redis example scripts (*redis1.py, redis2.py, publisher.py, subscriber.py*).

4.1.2- Architecture and Features

- **Redis Server:** A prerequisite, expected to be running and accessible (e.g., via Docker on localhost:6379).
- **Data Storage (Insults):**
  - Insults are stored in a Redis Set (default key: insults_set). The SADD command used by insult_adder_redis.py inherently handles uniqueness.
  - insult_getter_redis.py uses SMEMBERS to retrieve all insults.
- **Broadcasting Mechanism:**
  - insult_broadcaster_redis.py acts as a daemon.
  - Periodically (every 5 seconds), it uses SRANDMEMBER to select a random insult from the insults_set.
  - It then uses the PUBLISH command to send this insult to a designated Redis channel (default: insult_broadcast_channel).
- **Subscription Mechanism:**
  - insult_subscriber_redis.py connects to Redis and uses the Pub/Sub mechanism (inspired by subscriber.py example).
  - It SUBSCRIBEs to the **insult_broadcast_channel**.

- It then enters a loop (pubsub.listen()) to receive messages published to that channel.

4.1.3- Code Structure

- **insult_adder_redis.py**: Connects to Redis and uses SADD to add insults to the insults_set.
- **insult_getter_redis.py**: Connects to Redis and uses SMEMBERS to retrieve insults from insults_set.
- **insult_broadcaster_redis.py**: Continuously runs, reading a random insult from insults_set using SRANDMEMBER and publishing it to insult_broadcast_channel using PUBLISH. Includes signal handling for graceful shutdown.
- **insult_subscriber_redis.py**: Subscribes to insult_broadcast_channel and prints received messages. Includes signal handling for graceful shutdown.

All of the scripts use the redis-py library.

4.1.4- How to Run

All commands shown here assume the virtual environment is activated and the user is in the 'redis_insult_service' directory. You can use the run script located in the same folder, or do the steps manually:

1. Start Redis Server (Terminal 1):
   `docker run --name my-redis -d -p 6379:6379 redis`
   or if it's already created: `docker start my-redis`

2. Start the Insult Broadcaster (Terminal 2):
   `python insult_broadcaster_redis.py`

3. Start one or more Insult Subscribers (Terminal 3 [or more]):
   `python insult_subscriber_redis.py`

4. Use the Insult Adder (Terminal 4):
   `python insult_adder_redis.py`
   or add the insult at the end:
   `python insult_adder_redis.py "A new insult"`

5. Use the Insult Getter (Terminal 5):
   `python insult_getter_redis.py`

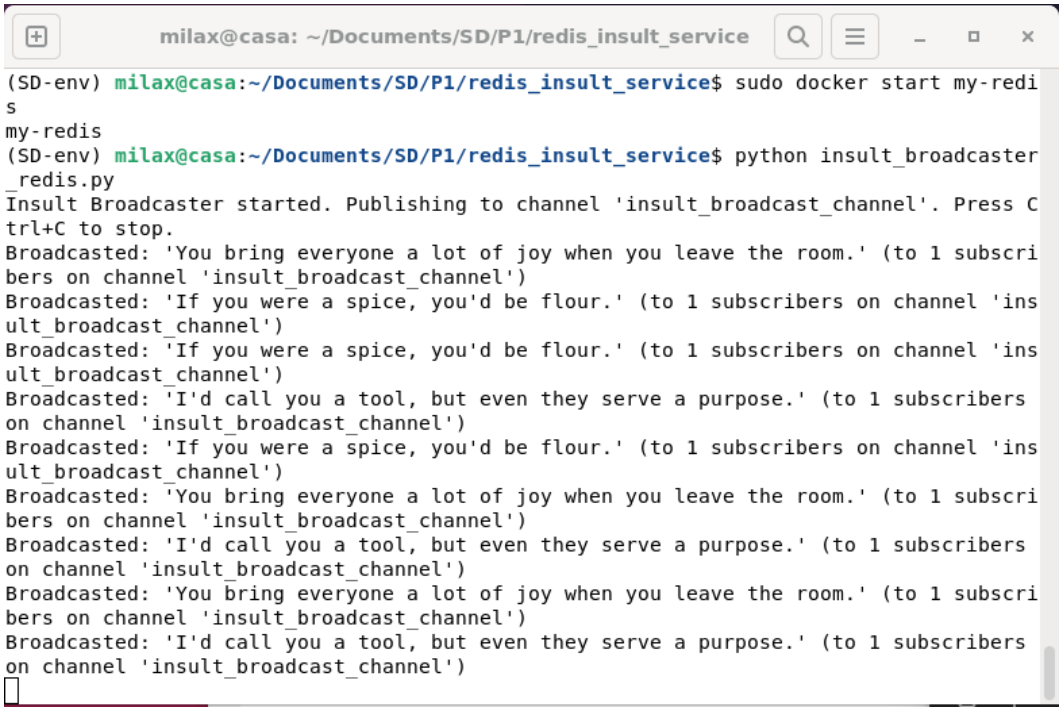
4.1.5- Observed Behavior

The system operates as expected:

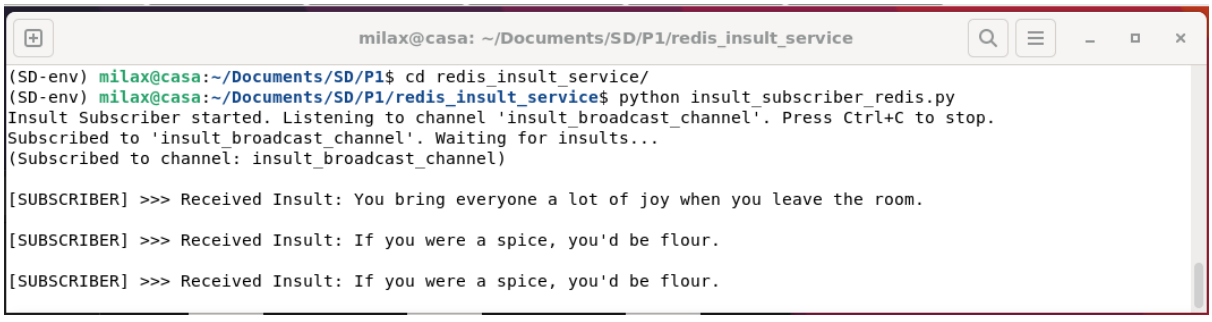
- Insults added via insult_adder_redis.py are stored uniquely in the Redis Set.
- insult_getter_redis.py correctly retrieves and displays the stored insults.
- The insult_broadcaster_redis.py script periodically publishes random insults from the set to the designated Pub/Sub channel.
- Running instances of insult_subscriber_redis.py successfully subscribe to the channel and display the broadcasted insults in real-time.

4.1.6- Screenshots

Broadcast:



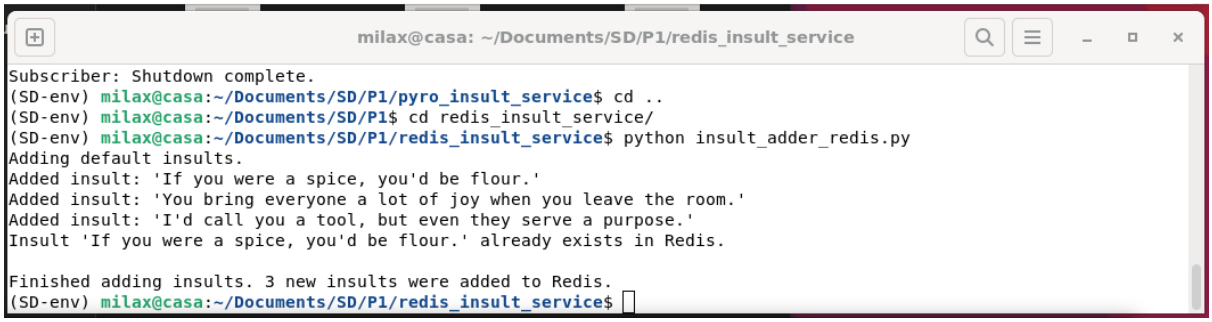
Subscriber:



ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

Sergi Izquierdo Segarra
48280130W
2024-2025

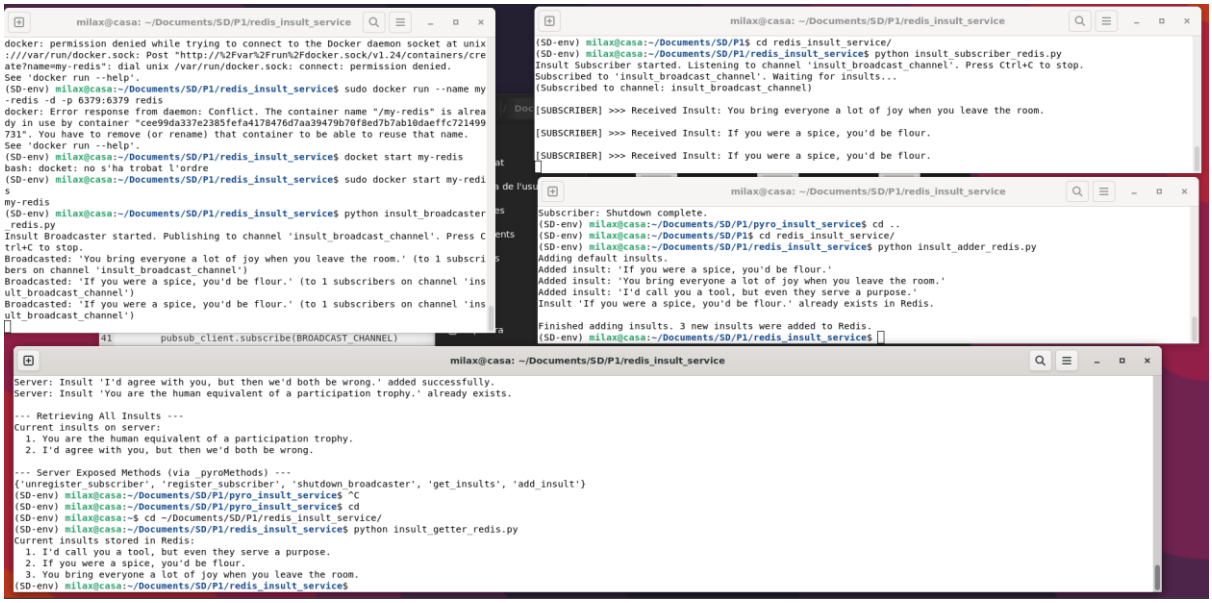
Adder:



Getter:

```
(SD-env) milax@casa:~/Documents/SD/P1/redis_insult_service$ python insult_getter_redis.py
Current insults stored in Redis:
  1. I'd call you a tool, but even they serve a purpose.
  2. If you were a spice, you'd be flour.
  3. You bring everyone a lot of joy when you leave the room.
```

All Together:



4.2- InsultFilter

4.2.1- Overview

The service employs a work queue pattern using Redis:

- **FilterProducer (filter_producer_redis.py):** A client script that sends text strings (tasks) to a Redis List acting as a work queue.
- **FilterWorker (filter_worker_redis.py):** One or more worker scripts that retrieve texts from the work queue, filter out predefined insults (replacing them with "CENSORED"), and store the filtered results in a separate Redis List.
- **FilterResultsRetriever (filter_results_retriever_redis.py):** An optional client script to view the contents of the results list.

This design leverages Redis's list operations (RPUSH, BLPOP, LRANGE) and is based on the campus Redis examples, particularly *producer.py* and *consumer.py* for the queue mechanism.

4.2.2- Architecture

- **Redis Server:** A running Redis instance is necessary.
- **Work Queue (filter_work_queue - Redis List):**
  - filter_producer_redis.py uses RPUSH to add texts to this list.
  - filter_worker_redis.py instances use BLPOP (blocking list pop) to atomically retrieve and remove texts from this queue, enabling multiple workers to share the workload.
- **Filtering Logic:** Performed by each FilterWorker. A predefined set of "bad words" is used for case-insensitive replacement.
- **Results Storage (filtered_texts_results - Redis List):**
  - After filtering, each FilterWorker RPUSHes a JSON string (containing original text, filtered text, worker ID, and timestamp) to this list.
- **Result Retrieval:**
  - filter_results_retriever_redis.py uses LRANGE to fetch all items from the results list and displays them after parsing the JSON.

4.2.3- Code Structure

- **filter_producer_redis.py:** Connects to Redis and pushes text tasks into filter_work_queue.
- **filter_worker_redis.py:** Connects to Redis, continuously fetches tasks from filter_work_queue using BLPOP, filters insults, and stores results in filtered_texts_results. Each worker has a unique Process ID.
- **filter_results_retriever_redis.py:** Connects to Redis and displays items from filtered_texts_results.

All scripts use the 'redis-py' library.

4.2.4- How to Run

Assumes virtual environment active, Redis server running, and in redis_filter_service directory. You can use the run script located in the same folder, or do the steps manually:

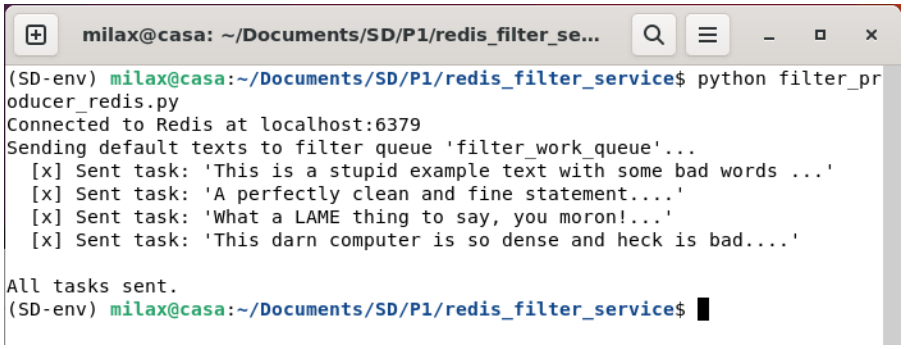
1. Start one or more Filter Workers (Terminal 1 [or more]):
   `python filter_worker_redis.py`
2. Run Filter Producer (Terminal 2):
   `python filter_producer_redis.py`
   or add the insult at the end:
   `python filter_producer_redis.py "Some text with stupid words."`
3. Run Results Retriever (Terminal 3):
   `python filter_results_retriever_redis.py`

4.2.5- Observed Behavior

Texts sent by the producer are added to the Redis work queue. Worker instances pick up these tasks, filter them, and store the structured results in a separate Redis list. The work is distributed among available workers. The results retriever can display the processed texts.

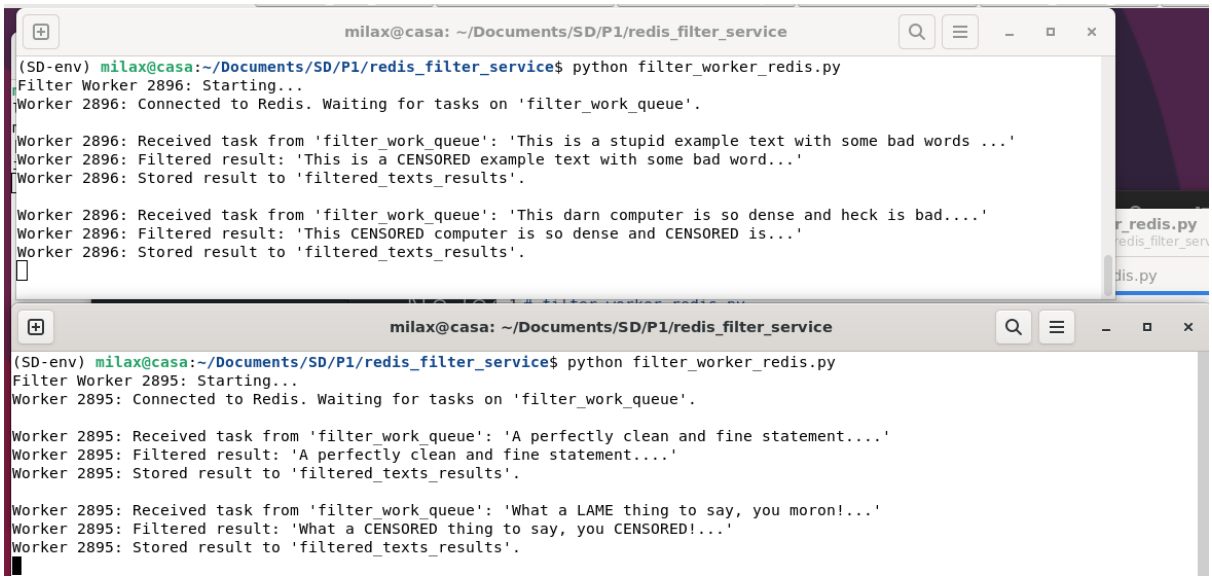
4.2.6- Screenshots

Producer:

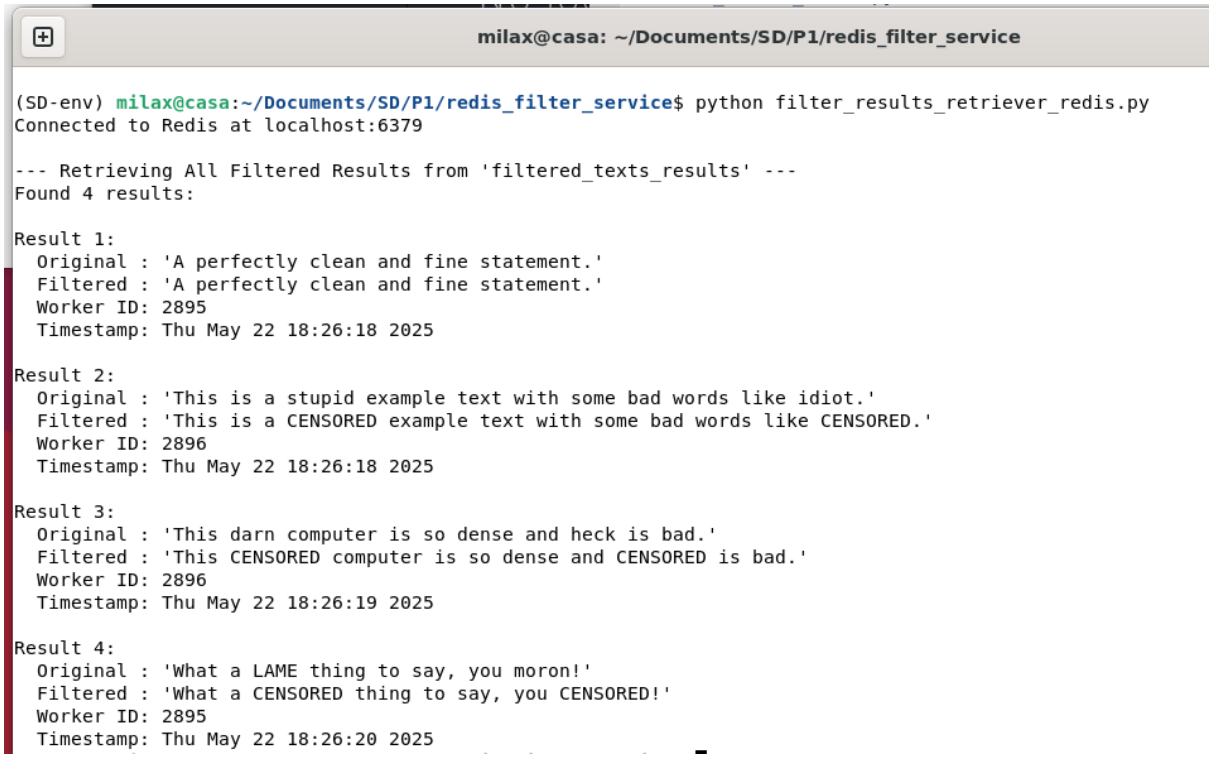


Sergi Izquierdo Segarra
48280130W
2024-2025

Worker(s):



Results:



Sergi Izquierdo Segarra
48280130W
2024-2025

5- RabbitMQ

5.1- InsultService

5.1.1- Overview and Design

The RabbitMQ-based InsultService separates responsibilities across different scripts, interacting via a running RabbitMQ server:

1. **insult_processor_rabbit.py:** This core script acts as both a consumer and a periodic publisher.
    - **Consumer Role:** It listens to a dedicated RabbitMQ queue (add_insult_queue) for new insult strings. Upon receiving an insult, it adds it to an internal, in-memory Python set to maintain uniqueness.
    - **Publisher Role:** A background thread within this script periodically selects a random insult from the in-memory set and publishes it to a RabbitMQ fanout exchange (insult_broadcast_exchange).
2. **insult_adder_client_rabbit.py:** A client script that sends new insult strings as messages to the add_insult_queue, to be picked up by the insult_processor_rabbit.py.
3. **insult_subscriber_rabbit.py:** A script that subscribes to insult broadcasts. It creates a temporary, exclusive queue and binds it to the insult_broadcast_exchange to receive and display broadcasted insults.

5.1.2- Architecture and Features

- **RabbitMQ Server:** A running instance is required (For example via Docker on localhost:5672).
- **Insult Submission (Queueing):**
    - insult_adder_client_rabbit.py publishes insult strings to a durable queue named add_insult_queue on the default exchange. Messages are marked persistent.
    - insult_processor_rabbit.py consumes messages from this queue, decodes the insult, adds it to its internal set, and acknowledges the message (basic_ack).
- **Insult Storage:** The unique list of insults is maintained in an in-memory Python set within the insult_processor_rabbit.py script.
- **Broadcasting (Fanout Exchange):**
    - insult_processor_rabbit.py declares a fanout exchange named insult_broadcast_exchange.

  - Its broadcaster thread periodically publishes a random insult from its internal set to this exchange. The routing_key is empty as it's a fanout. Messages are marked persistent.
- **Subscription:**
  - insult_subscriber_rabbit.py declares the same insult_broadcast_exchange.
  - It creates a new, exclusive, anonymous queue.
  - This queue is bound to the insult_broadcast_exchange.
  - It consumes messages from its unique queue (with auto_ack=True as messages are transient to this subscriber).

5.1.3- Code Structure

- **insult_processor_rabbit.py:**
  - add_insult_callback(): Handles incoming messages from add_insult_queue.
  - start_consuming_new_insults(): Sets up and starts the consumer loop for adding insults (blocking).
  - periodic_broadcaster(): Runs in a daemon thread, publishing to the fanout exchange.
  - Includes signal handling for graceful shutdown.
- **insult_adder_client_rabbit.py:** Connects to RabbitMQ, declares the target queue, and publishes insult messages to it.
- **insult_subscriber_rabbit.py:** Connects to RabbitMQ, sets up an exchange, an exclusive queue, binds them, and consumes/displays broadcast messages. Includes signal handling.

All scripts use the pika library for Python-RabbitMQ communication.

5.1.4- How to Run

All commands assume the virtual environment is activated and the user is in the rabbitmq_insult_service directory. You can use the run script located in the same folder, or do the steps manually:

1. Start RabbitMQ Server (Terminal 1):
   `docker run -d --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:management`
   or if it's already created: `docker start rabbitmq`
2. Start the Insult Processor (Terminal 2):
   `python insult_processor_rabbit.py`
3. Start one or more Insult Subscribers (Terminal 3 [or more]):
   `python insult_subscriber_rabbit.py`
4. Use the Insult Adder Client (Terminal 4):
   `python insult_adder_client_rabbit.py`

or add the insult at the end:

```
python insult_adder_client_rabbit.py "A new insult."
```

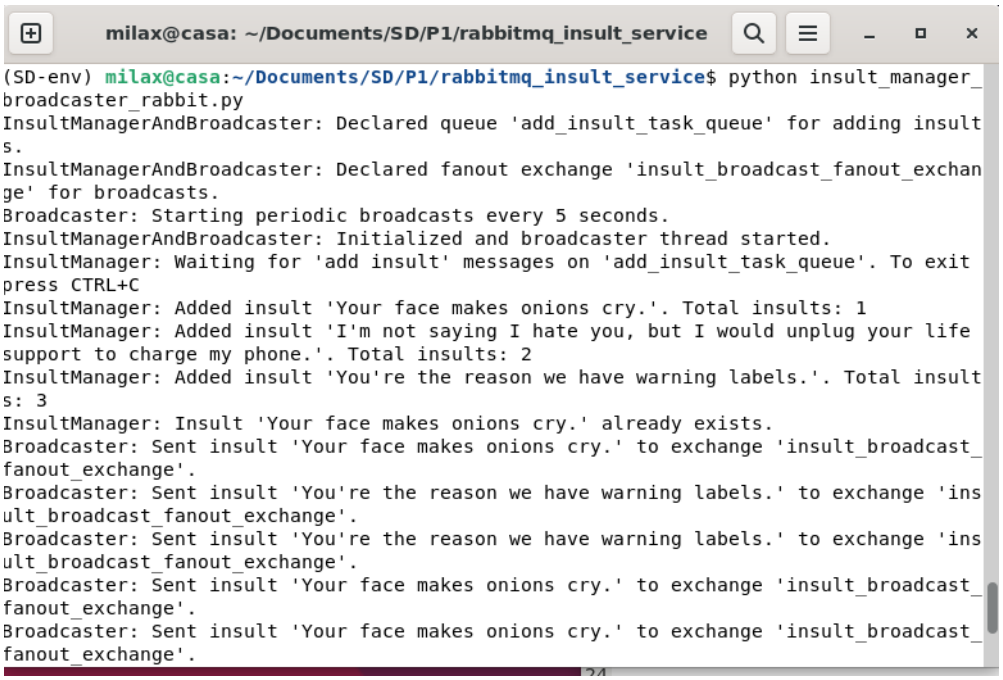
5.1.5- Observed Behavior

The system functions correctly:

- The insult_processor_rabbit.py starts, ready to consume insults and broadcast.
- Insults sent by insult_adder_client_rabbit.py are received by the processor, added to its internal unique set, and acknowledged.
- The processor's broadcaster thread periodically publishes random insults from its set to the fanout exchange.
- Instances of insult_subscriber_rabbit.py receive and display these broadcasted insults.
- The RabbitMQ management UI (http://localhost:15672) can be used to observe queues, exchanges, and message flow.

5.1.6- Screenshots

Processor:



ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

Sergi Izquierdo Segarra
48280130W
2024-2025

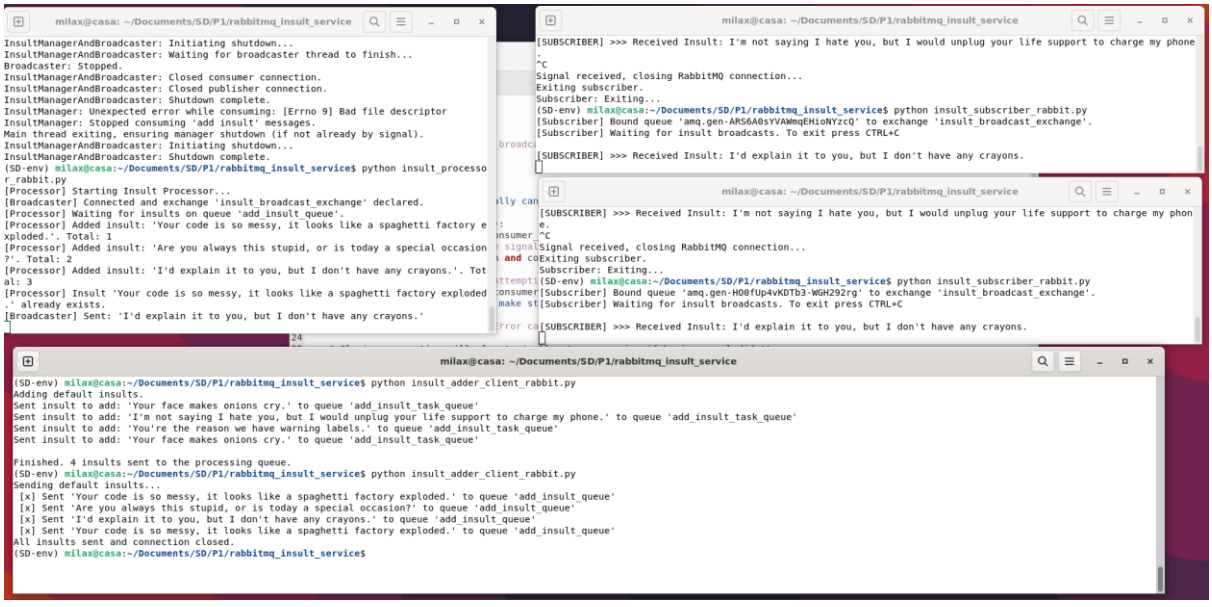
Subscriber:

```
milax@casa: ~/Documents/SD/P1/rabbitmq_insult_service
(SD-env) milax@casa:~/Documents/SD/P1/rabbitmq_insult_service$ python insult_subscriber_rabbit.py
[Subscriber] Bound queue 'amq.gen-ARS6A0sYVAWmqEHioNYzcQ' to exchange 'insult_broadcast_exchange'.
[Subscriber] Waiting for insult broadcasts. To exit press CTRL+C

[SUBSCRIBER] >>> Received Insult: I'd explain it to you, but I don't have any crayons.

[SUBSCRIBER] >>> Received Insult: I'd explain it to you, but I don't have any crayons.

[SUBSCRIBER] >>> Received Insult: Are you always this stupid, or is today a special occasion?

[SUBSCRIBER] >>> Received Insult: Your code is so messy, it looks like a spaghetti factory exploded.
```

Adder Client:

```
(SD-env) milax@casa:~/Documents/SD/P1/rabbitmq_insult_service$ python insult_adder_client_rabbit.py
Sending default insults...
 [x] Sent 'Your code is so messy, it looks like a spaghetti factory exploded.' to queue 'add_insult_queue'
 [x] Sent 'Are you always this stupid, or is today a special occasion?' to queue 'add_insult_queue'
 [x] Sent 'I'd explain it to you, but I don't have any crayons.' to queue 'add_insult_queue'
 [x] Sent 'Your code is so messy, it looks like a spaghetti factory exploded.' to queue 'add_insult_queue'
All insults sent and connection closed.
(SD-env) milax@casa:~/Documents/SD/P1/rabbitmq_insult_service$
```

All Together:



5.1- InsultFilter

5.2.1- Overview

The RabbitMQ-based InsultFilter service processes texts submitted by clients, filters out predefined insults, and makes the results available. It utilizes:

1. **FilterProducer (filter_producer_rabbit.py):** A client script that sends raw text strings (as tasks) to a designated RabbitMQ work queue for filtering.
2. **FilterWorker (filter_worker_rabbit.py):** One or more worker scripts that consume tasks from the work queue. Each worker filters insults from the received text (replacing them with "CENSORED") and then publishes the structured result (original text, filtered text, worker ID, timestamp) as a JSON string to a separate "results" queue.
3. **FilterResultsCollector (filter_results_collector_rabbit.py):** A script that consumes from the "results" queue, parses the JSON data, and displays the filtered outcomes.

This design uses the work queue pattern (producer/consumer for tasks) and simple queueing for results, based on the provided RabbitMQ Pika examples (producer.py, consumer.py).

5.2.2- Architecture

- **RabbitMQ Server:** Acts as the central message broker.
- **Task Submission (Work Queue -** filter_task_work_queue**):**
  - filter_producer_rabbit.py publishes text messages to this durable queue using the default exchange (routing key is the queue name). Messages are marked persistent.
  - filter_worker_rabbit.py instances consume from this queue. basic_qos(prefetch_count=1) ensures fair dispatch, and workers manually acknowledge (basic_ack) messages only after successful processing and result submission.
- **Filtering Logic:** Performed by each FilterWorker using a predefined, case-insensitive list of insults.
- **Result Aggregation (Results Queue -** filter_results_data_queue**):**
  - After filtering, each FilterWorker publishes a JSON string containing the processing details to this durable queue.

- filter_results_collector_rabbit.py consumes from this queue to display results.

5.2.3- Code Structure

- **filter_producer_rabbit.py:** Connects to RabbitMQ and sends text tasks to filter_task_work_queue.
- **filter_worker_rabbit.py:** Consumes tasks from filter_task_work_queue, performs filtering, and publishes JSON results to filter_results_data_queue. Implements manual message acknowledgment.
- **filter_results_collector_rabbit.py:** Consumes and displays JSON results from filter_results_data_queue.

All scripts use the 'pika' library.

5.2.4- How to Run

Assumes virtual environment active, RabbitMQ server running, and in rabbitmq_filter_service directory. You can use the run script located in the same folder, or do the steps manually:

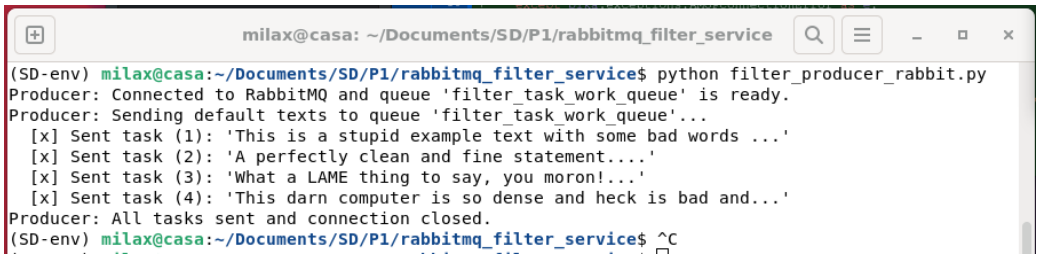
1. Start Filter Results Collector (Terminal 1):
   `python filter_results_collector_rabbit.pyl`
2. Start one or more Filter Workers (Terminal 2 [or more]):
   `python filter_worker_rabbit.py`
3. Run Filter Producer (Terminal 3):
   `python filter_producer_rabbit.py`
   or add an insult to the end:
   `python filter_producer_rabbit.py "Text with a stupid insult"`

5.2.5- Observed Behavior

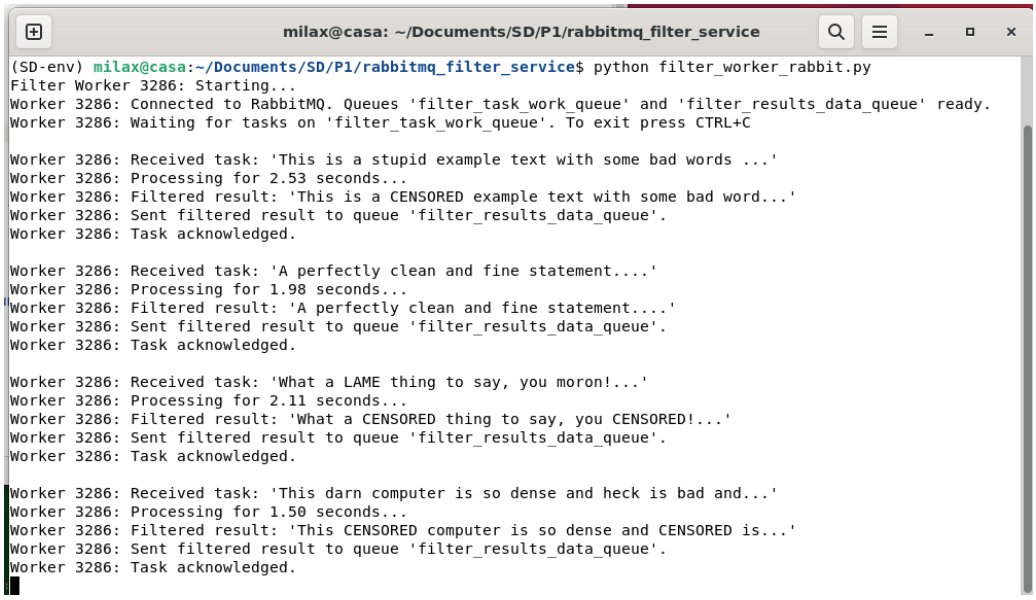
Texts sent by the producer are routed to the work queue. Available workers pick up these tasks one by one, filter them, and send the structured results to a separate results queue. The results collector displays these processed texts. Work is distributed across multiple worker instances if run.

5.2.6- Screenshots

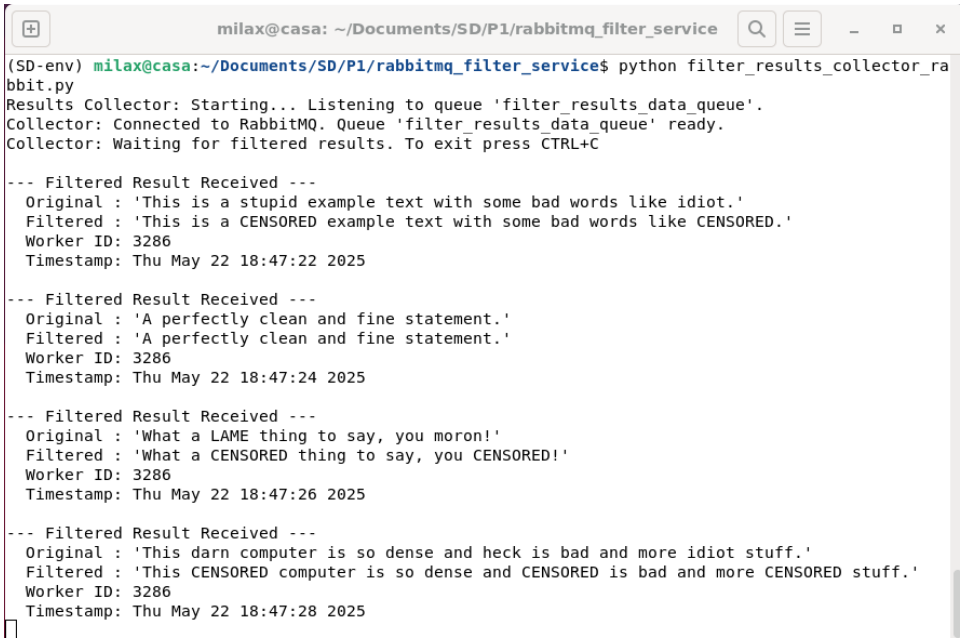
Producer:



Worker:



Results:



6- Performance Tests

This section contains the performance analysis done on the InsultService and InsultFilter implementations across the four middleware. The analysis covers single-node performance, multi-node statis scaling, and multi-node dynamic scaling.

All tests were performed within a VirtualBox VM configured with 8 vCPUs and 4GB of RAM. The Python version used was 3.11 via the sub environment SD-env. Redis and RabbitMQ were run via Docker.

6.1- Single-Node Performance Analysis

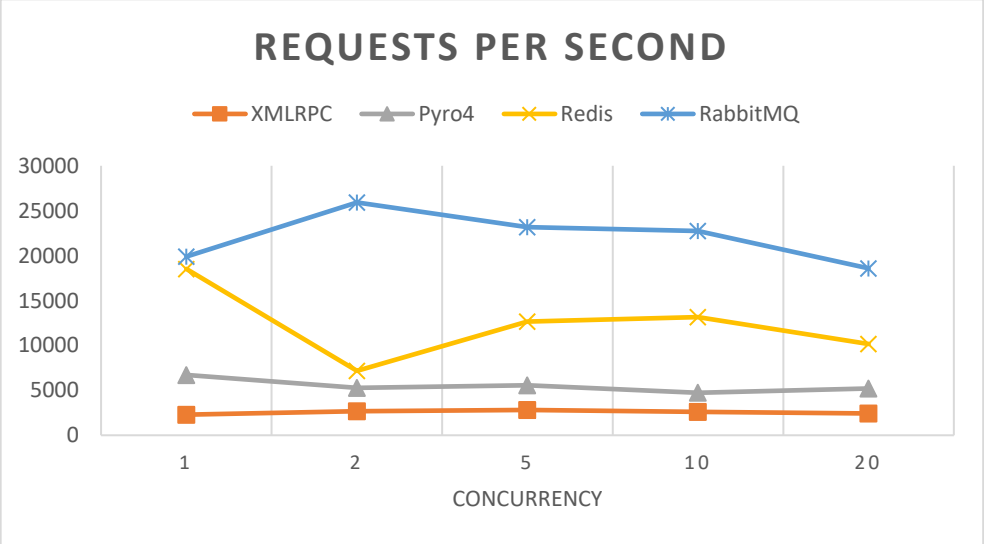
We measured the throughtput (requests per second) of each service on a single no, with varying client load. TOTAL_REQUESTS was set to 10.000 for each of these tests.

6.1.1- InsultService (add_insult operation)

- **Methodology:** Client processes concurrently sent `add_insult` requests. For XMLRPC / Pyro4, this was to a single server instance. For Redis, clients directly performed the `SADD` operations. For RabbitMQ, clients sent messages to a queue consumed by a single `InsultProcessor` instance, updating its internal set.

- **Results and Plots:**

Concurrency	XMLRPC	Pyro4	Redis	RabbitMQ
1	2295.49	6716.76	18518.78	19888.32
2	2690.71	5285.52	7171.86	25917.21
5	2818.12	5569.61	12646.11	23162.48
10	2602.07	4729.67	13161.68	22731.76
20	2442.83	5206.86	10149.75	18584.32



- **Analysis:**

  The graph illustrates the add_insult RPS for each middleware. **RabbitMQ** had the highest throughput, peaking at approximately 26000 RPS. **Redis** also performed well, achieving around 18500 RPS at maximum. This high performance is due to their design: Redis uses direct, low-latency in-memory set additions, while RabbitMQ queues incoming requests for the processor.

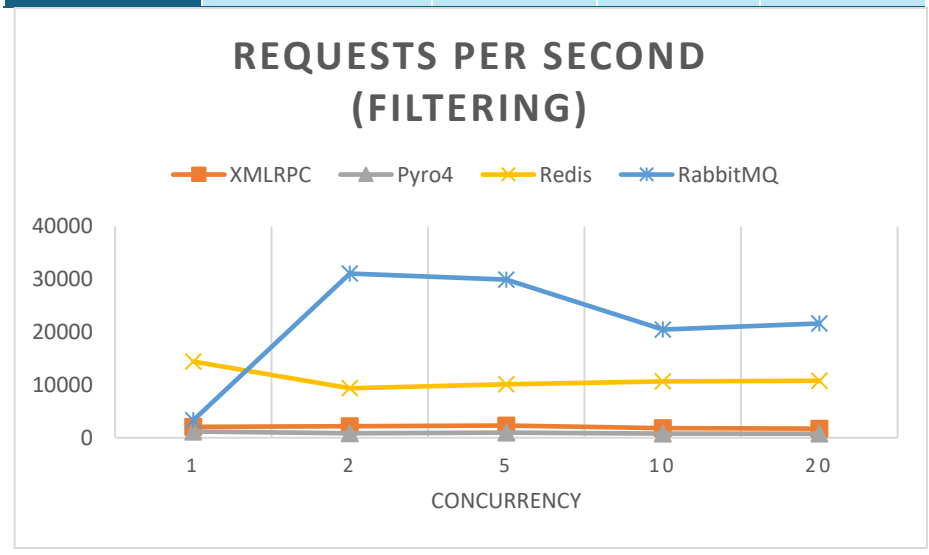
  Pyro4 achieved a maximum of 6700 RPS, with moderate performance. **XMLRPC** had the lowest peak throughput across all middlewares at a max of 2800 RPS. Both RPC-based systems performed noticeably worse than the two before, due to their serialization mechanisms and the network communication being handled by a single server process.

6.1.2- InsultFilter (submit_text_for_filtering operation)

- **Methodology:** Client processes concurrently submitted texts for filtering. For XMLRPC, this was to the server with its internal worker thread. For Pyro4, to the dispatcher managing one worker process. For both Redis and RabbitMQ, clients pushed tasks to a work queue consumed by one worker process.

- **Results and Plots:**

Concurrency	XMLRPC	Pyro4	Redis	RabbitMQ
1	2090.5	1181.45	14428.18	3320.67
2	2203.71	837.37	9369.72	31079.09
5	2316.88	997.37	10107.03	29943.39
10	1838.41	818.53	10638.77	20474.69
20	1733.6	773.83	10801.2	21613.65



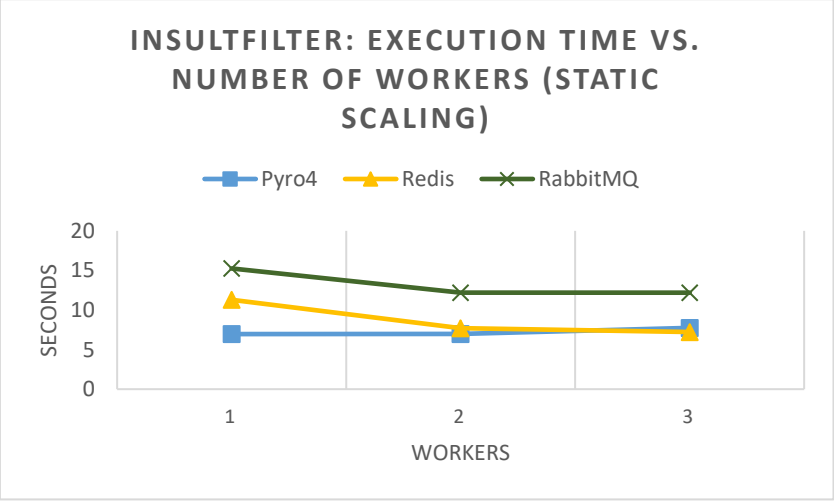
- **Analysis:**
  For task submission to the `InsultFilter` service, **Redis** and **RabbitMQ** showed superior throughput again, achieving approximately 14400 RPS and 31080 RPS respectively. This is because task submission is basically an enqueue operation, which the message brokers handle efficiently and asynchronously from the filtering work.

  Pyro4, with its dispatcher sending tasks to a single worker, had the lowest submission throughput, achieving a maximum of 1180 RPS. **XMLRPC** achieved around 2300 RPS at a maximum. In all cases, the single worker processing the actual filtering tasks in the background would limit overall processing if submissions outpaced its capacity.

6.2- Multiple-Nodes Static Performance Analysis (InsultFilter Service)

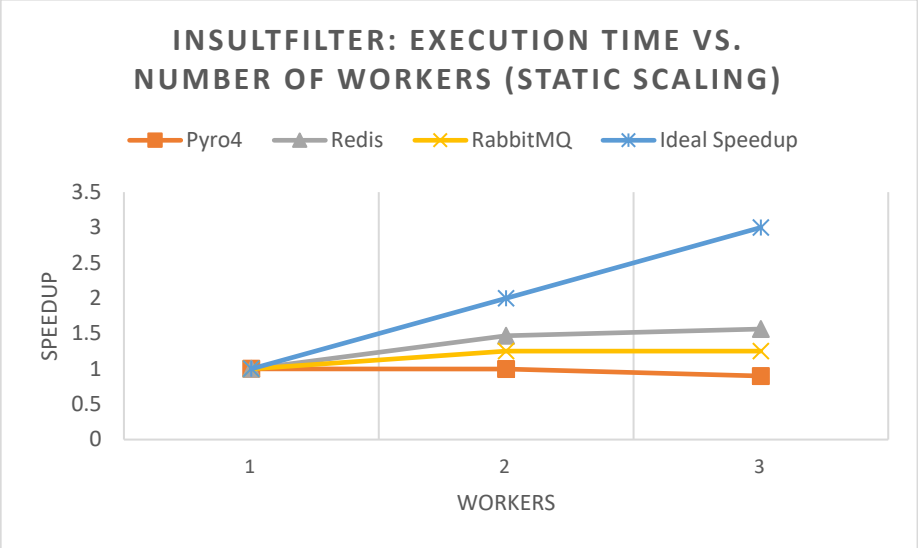
- **Objective:** Evaluate how the InsultFilter service's processing capability scales with 1, 2 and 3 worker processes for a workload of 10.000 texts.
- **Methodology:** Tests used the `test_static_scaling_filter_*.py` scripts. The XMLRPC architecture was not tested for multi-worker scaling due to its limitations in sharing work from a single queue among multiple independent server processes without an external load balancer.
- **Results and Plots:**
  - **Time to complete task (seconds)**

Workers	Pyro4	Redis	RabbitMQ
1	6.96	11.29	15.25
2	6.98	7.69	12.21
3	7.74	7.22	12.19



- **Speedup**

Workers	Pyro4	Redis	RabbitMQ	Ideal Speedup
1	1	1	1	1
2	0.99713467	1.46814	1.248976249	2
3	0.899224806	1.563712	1.251025431	3



- **Analysis:**

  The execution time graph shows how total processing time for 10.000 tasks generally decreased as workers were added. The speedup graph compares this against the ideal linear scaling.

  - **Redis:** Achieved a speedup of **1.47x** for 2 workers and **1.56x** for 3 workers. It shows good scalability, demonstrating that multiple workers could efficiently pull tasks from its list-based work queue and process them in parallel.
  - **RabbitMQ:** Achieved a **1.25x** speedup for 2 workers, but failed to improve with 3 workers, staying at the same speedup achieved with 2 workers. This may be caused by some kind of bottleneck.
  - **Pyro4:** Showed similar performance for 2 workers, albeit slightly worse, but degraded to a **0.90x** speedup for 3 workers, making it slower. The `FilterDispatcher` acts as a central, single-process bottleneck for assigning tasks via RPC to workers. As more were added, the overhead with this dispatcher probably worsened the performance that could be achieved by parallel processing.

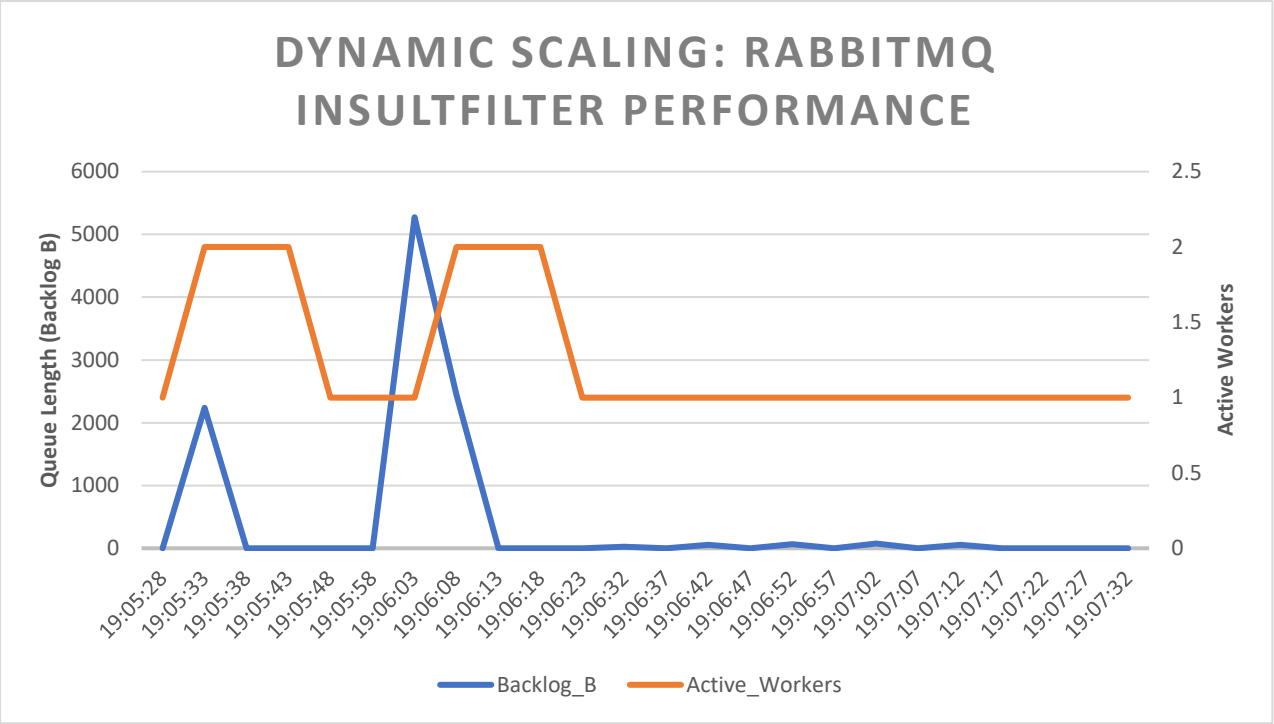
6.3- Multiple-Nodes Dynamic Performance Analysis (RabbitMQ InsultFilter)

- **Objective:** Demonstrate a dynamic scaling mechanism where the number of RabbitMQ workers adjust based on the length of the task queue.
- **Methodology:** The dynamic_scaler_rabbit.py script monitored the filter_task_work_queue length (B). It used the formula `N_desired = ceil((B + λ*Tr)/C)` to calculate the desired number of workers. Parameters:
    - MIN_WORKERS = 1
    - MAX_WORKERS = 3
    - POLL_INTERVAL = 2s
    - SCALE_COOLDOWN_PERIOD = 5s
    - C_WORKER_CAPACITY = 656 [1/0.001525]
    - LAMBDA_ESTIMATED_ARRIVAL_RATE = 150
    - Tr_TARGET_RESPONSE_TIME = 2s
- **Results and Plots:**

Timestamp	Backlog_B	Active_Workers	N_Desired_Formula	Action_Taken
19:05:28	0	1	1	Maintain
19:05:33	2240	2	3	Attempt_ScaleUP_to_3
19:05:38	0	2	1	InCooldown (10s left)
19:05:43	0	2	1	InCooldown (5s left)
19:05:48	0	1	1	Attempt_ScaleDOWN_to_1
19:05:58	0	1	1	InCooldown (10s left)
19:06:03	5274	1	3	InCooldown (5s left)
19:06:08	2442	2	3	Attempt_ScaleUP_to_3
19:06:13	0	2	1	InCooldown (10s left)
19:06:18	0	2	1	InCooldown (5s left)
19:06:23	0	1	1	Attempt_ScaleDOWN_to_1
19:06:32	24	1	1	InCooldown (10s left)
19:06:37	0	1	1	InCooldown (5s left)
19:06:42	55	1	1	Maintain
19:06:47	0	1	1	Maintain
19:06:52	66	1	1	Maintain
19:06:57	0	1	1	Maintain
19:07:02	77	1	1	Maintain
19:07:07	0	1	1	Maintain
19:07:12	56	1	1	Maintain
19:07:17	0	1	1	Maintain
19:07:22	0	1	1	Maintain

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA
Universitat Rovira i Virgili

Sergi Izquierdo Segarra
48280130W
2024-2025



- **Analysis:**

  The graph shows the dynamic scaler's responsiveness. Initially, with no load, one worker was active.

  - When the first burst arrived (19:05:33) the scaler detected the increased queue length. The formula calculated N_desired=3 and increased the number of active workers to 2.
  - As the workers processed the burst, the queue length (Backlog B) decreased.
  - During the pause before the second burst, the backlog dropped to zero. After the cooldown period, the scaler, seeing low backlog and N_desired=1, **reduced the active workers** back to one (15:05:48).
  - This pattern of scaling up in response to an increase in backlog can be shown again in the second burst (19:05:58), demonstrating how it **scales dynamically** depending on the backlog queue length.

6.4- Comparison and Conclusions

- **Performance:**
    - **Single-Node:** For simple, direct operations like add_insult in **InsultService**, Redis and RabbitMQ offered higher throughput than XMLRPC and Pyro4.
      For **InsultFilter**, both Redis and RabbitMQ still outperformed both XMLRPC and Pyro4 due to their asynchronous submission.
    - **Static Scaling:** Redis and RabbitMQ showed positive speedup when adding more worker processes, indicating their suitability for distributing processing tasks. Pyro4's dispatcher model, however, became a bottleneck, showing negative speedup.
    - **Dynamic Scaling:** RabbitMQ was the most suitable middleware due to its queuing, message durability options and easy to work with queue metrics (like the backlog length), using the formula `N = ceil((B + λ*Tr)/C)` for adjusting worker counts dynamically.
- **Conclusions:**
  Having worked with these different middlewares, it's clear each one impacts performance and scalability differently. Middleware like RabbitMQ and Redis were great at asynchronous communication and distributing workloads, while RPC Systems like XMLRPC and Pyro4 are simpler for direct client-server interactions but introduced bottlenecks in complex coordination cases.