Project AstroEdge DIP

Team members:-

Abhay Prajapati [SC22B083]

Rishi Gupta [SC22B140]

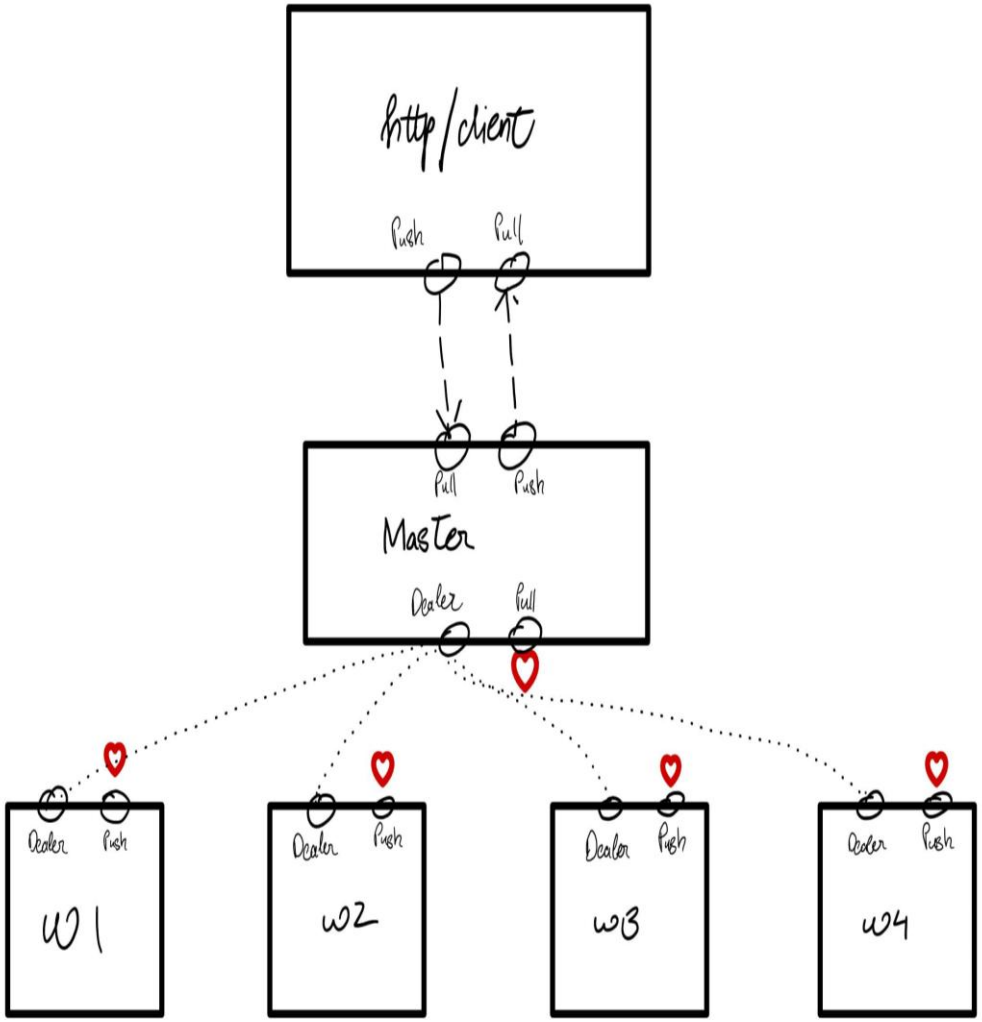
Anand Choubey [SC22B124]

Aditya Sharma [SC22B084]

Distributed edge computing system design

Understanding Patterns, Use Cases, and Internal Implementation

System Achitecture



Message format at each socket connection:

- Client →Master (via client_sender PUSH socket)

  Format: JSON request

  {
  "task_id": "<unique_task_id>",
  "task": "grayscale" | "edge",
  "image": "<base64_encoded_image_data>"
  }

  task_id : Unique ID generated by client

  task : type of processing requested

  image: Base64-encoded JPG image

Message format at each socket connection:

- Master →Client (via client_responder PUSH socket)

  Format: JSON request

  {

  "task_id": "<unique_task_id>",

  "task": "grayscale" | "edge",

  "image": "<base64_encoded_processed_image_data>"

  }

  task_id : Unique ID generated by client

  task : type of processing requested

  image: Processed image in Base64-encoded JPG format

Message format at each socket connection:

- Master →Worker (via worker_sender DEALER socket)

  Format: JSON request

  ```
  {
    "task_id": "<unique_task_id>",
    "task": "grayscale" | "edge",
    "image": "<base64_encoded_processed_image_data>"
  }
  ```

  Master simply forwards the request from client to any available worker.

Message format at each socket connection:

- Worker → Master (via worker_socket Dealer socket)

  Format: JSON request

  ```
  {
    "task_id": "<unique_task_id>",
    "task": "grayscale" | "edge",
    "image": "<base64_encoded_processed_image_data>"
  }
  ```

  - Worker sends back the result to the master.
  - Master forwards this directly to the client.

Message format at each socket connection:

- Worker → Master (heartbeat via PULL socket)

  Format: JSON request

  {

  “worker_id": "<worker-xyz>",

  “timestamp": current_unix_timestamp

  }

  - Sent every 2 seconds to let master know worker is alive
  - Master monitors last-seen times to detect timeouts.

ZeroMQ Messaging Patterns in Python

Understanding Patterns, Use Cases, and Internal Implementation

What is ZeroMQ?

- High-performance asynchronous messaging library

- Abstracts socket-level details into messaging patterns

- Enables scalable, distributed systems

- Used via pyzmq in Python

Messaging Patterns Overview

• REQ / REP – Request/Reply

• PUB / SUB – Publish/Subscribe

• PUSH / PULL – Pipeline pattern

• ROUTER / DEALER – Advanced async messaging

• PAIR, STREAM, XPUB/XSUB – Other patterns

REQ / REP Pattern

- Description: Synchronous client-server request-reply
- Use Case: RPC, simple question-answer systems
- Backend: Strict send-receive alternation, ordered responses

PUB / SUB Pattern

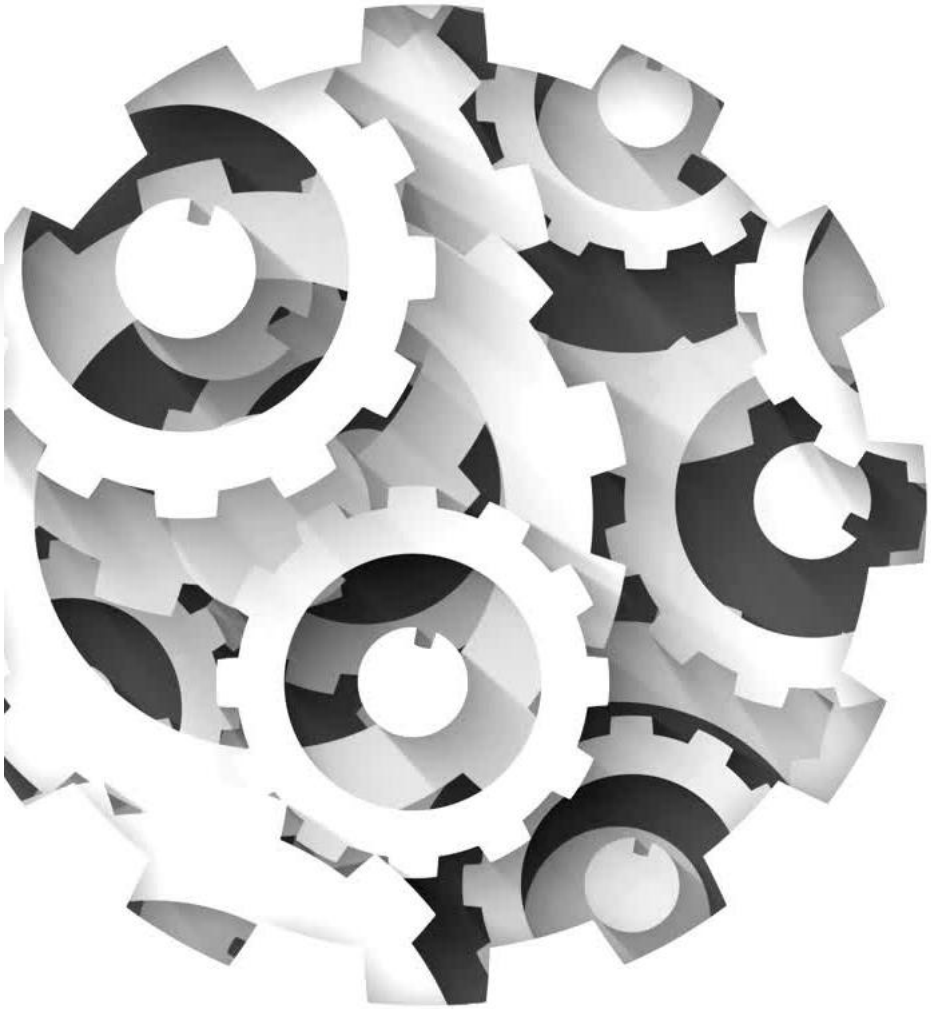
- Description: Asynchronous publish-subscribe based on topics
- Use Case: Real-time feeds, sensor broadcasting, IoT
- Backend: Topic filtering, publisher unaware of subscribers

PUSH / PULL Pattern

- Description: One-way pipeline for load-balanced distribution
- Use Case: Task queues, load-balanced workers
- Backend: Round-robin scheduling, separate worker queues

Round-Robin Fashion

- • Round-robin is a method to distribute tasks in a circular order
- • Each worker receives messages one-by-one in sequence
- • Used in PUSH/PULL to balance load across multiple PULL sockets
- • Example: Worker1 → Worker2 → Worker3 → Worker1 → ...



ROUTER / DEALER Pattern

- Description: Advanced async many-to-many messaging
- Use Case: Custom brokers, proxies, advanced routers
- Backend: ROUTER tracks identities, DEALER is async and non-blocking

How ZeroMQ Works Internally

- Socket abstraction over OS-level sockets
- Queues per connection
- IO threads manage messaging
- Fair queueing, load balancing
- Automatic reconnects, retries

Queue Size in ZeroMQ

- • Each socket in ZeroMQ maintains an internal message queue
- • Default high water mark (queue limit) is 1000 messages per socket
- • Can be configured using socket options:
- - ZMQ_SNDHWM: Send queue size
- - ZMQ_RCVHWM: Receive queue size
- • If the queue is full, ZeroMQ blocks or drops messages (depending on config)

Summary of Patterns

- • REQ/REP: Bidirectional, synchronous (Client-server)
- • PUB/SUB: One-to-many, async (Broadcasting)
- • PUSH/PULL: One-to-many, async (Task queues)
- • ROUTER/DEALER: Many-to-many, async (Proxies, routers)

Efficiency and Fault Tolerance

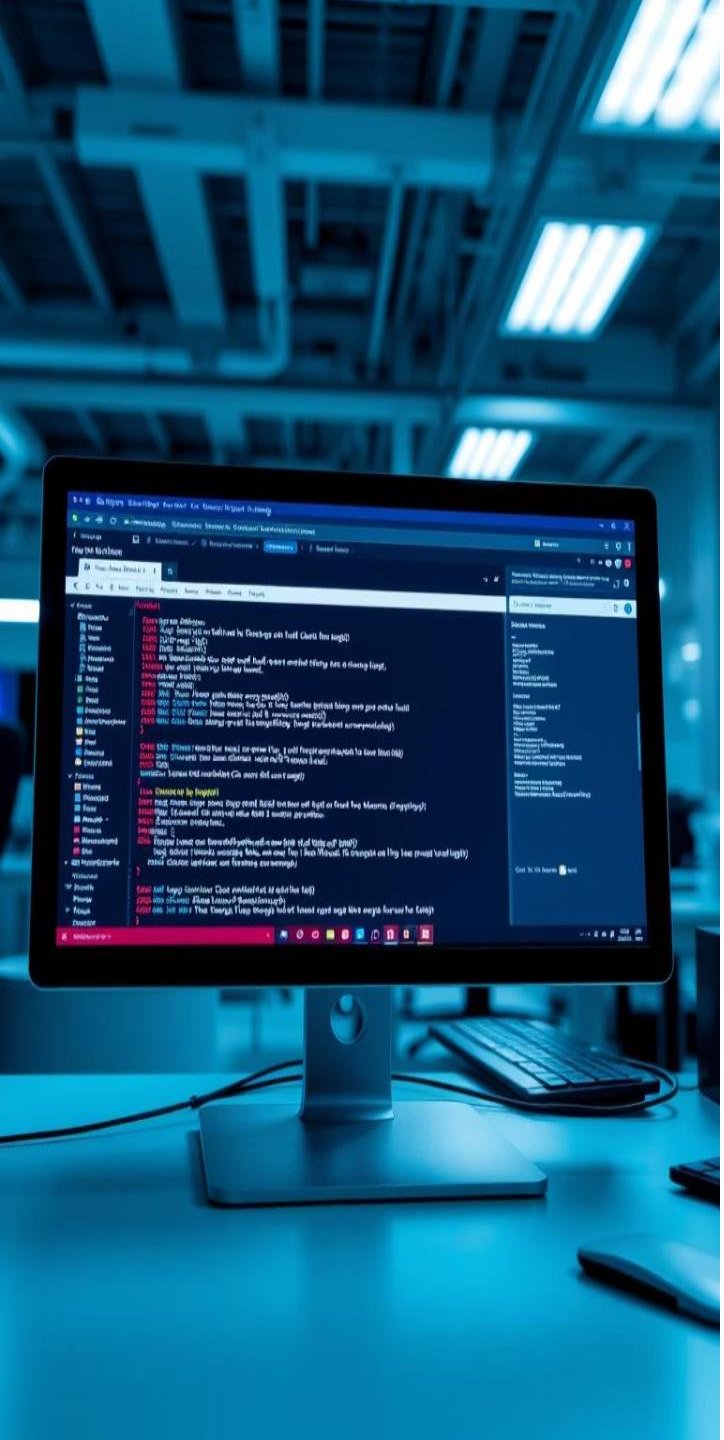
- - Master removes unresponsive workers automatically
- - Clients wait asynchronously (no blocking)
- - Threaded master handles task, result, heartbeat separately
- - Scalable: add more workers/clients easily

Project Highlights

- - Web-based interface for multiple users
- - Distributed processing with ZeroMQ
- - Heartbeat monitoring for fault detection
- - Asynchronous design using threading and events
- - Real-time image processing using OpenCV

Final Review Objectives:

- To fix an obvious bug where the request data is not being handled properly.
- To Add support for broadly used Image processing tasks and sequential processing.
- To add Client side authentication.
- Caching and user history
- To demonstrate distributed nature using RasPy/Esp/microcontrollers.



Reliable Task Queue with Redis

Problem

ZeroMQ caused task overwrite and loss due to async messaging.

Solution

Implemented Redis LPUSH and RPOP for FIFO queue management.

Benefits

- Reliable and persistent task handling
- Scalable concurrent task processing
- Improved monitoring and debugging

Raspberry Pi Workers for Edge Compute

Deployment

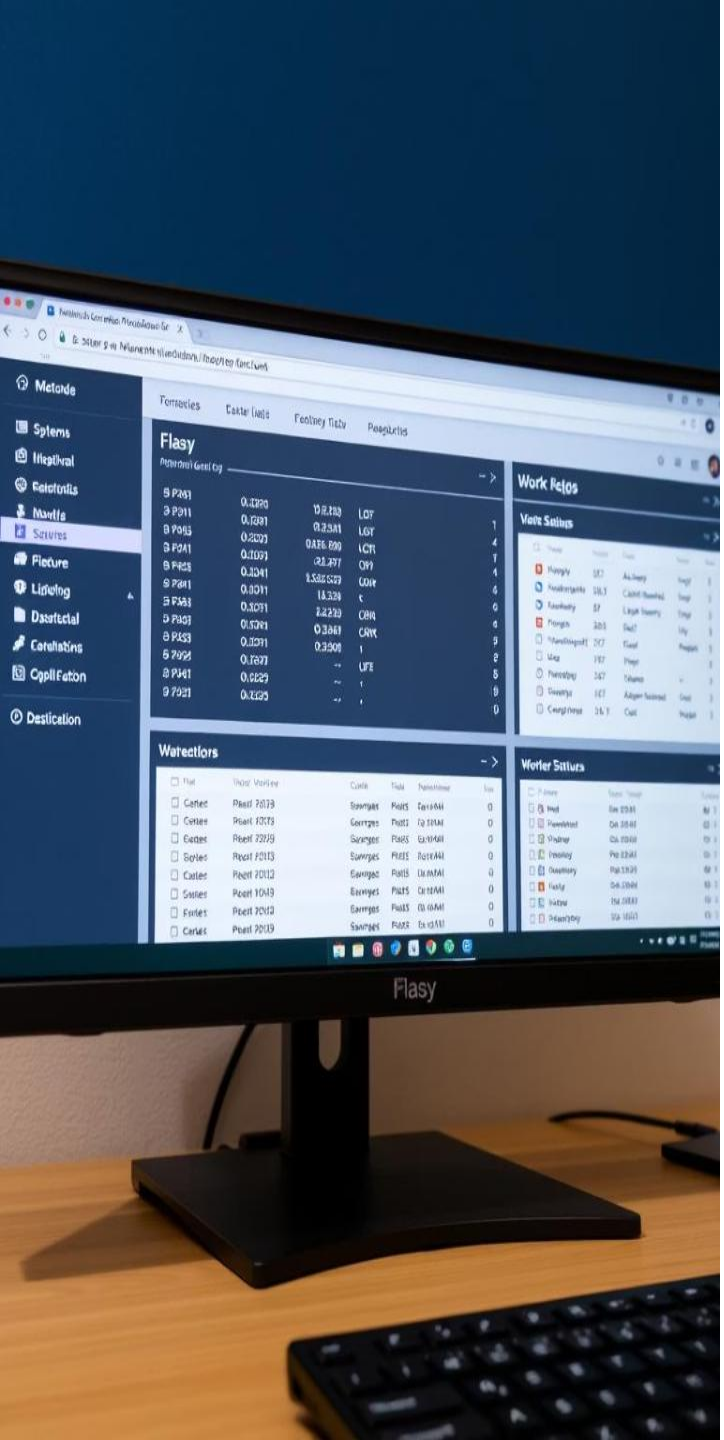
Workers run on Raspberry Pi nodes, enabling distributed processing.

Communication

ZeroMQ DEALER sockets with heartbeat signals track node health.

Edge Benefits

- Low latency processing
- Local data handling reduces bandwidth
- Lightens central system load



Real-Time Monitoring UI

1

Features

- Task requests and timestamps
- Live worker status and last-seen times
- System logs and events

2

Backend

Uses Redis for fast in-memory data access.

3

Value

Enhances observability and debugging capabilities.

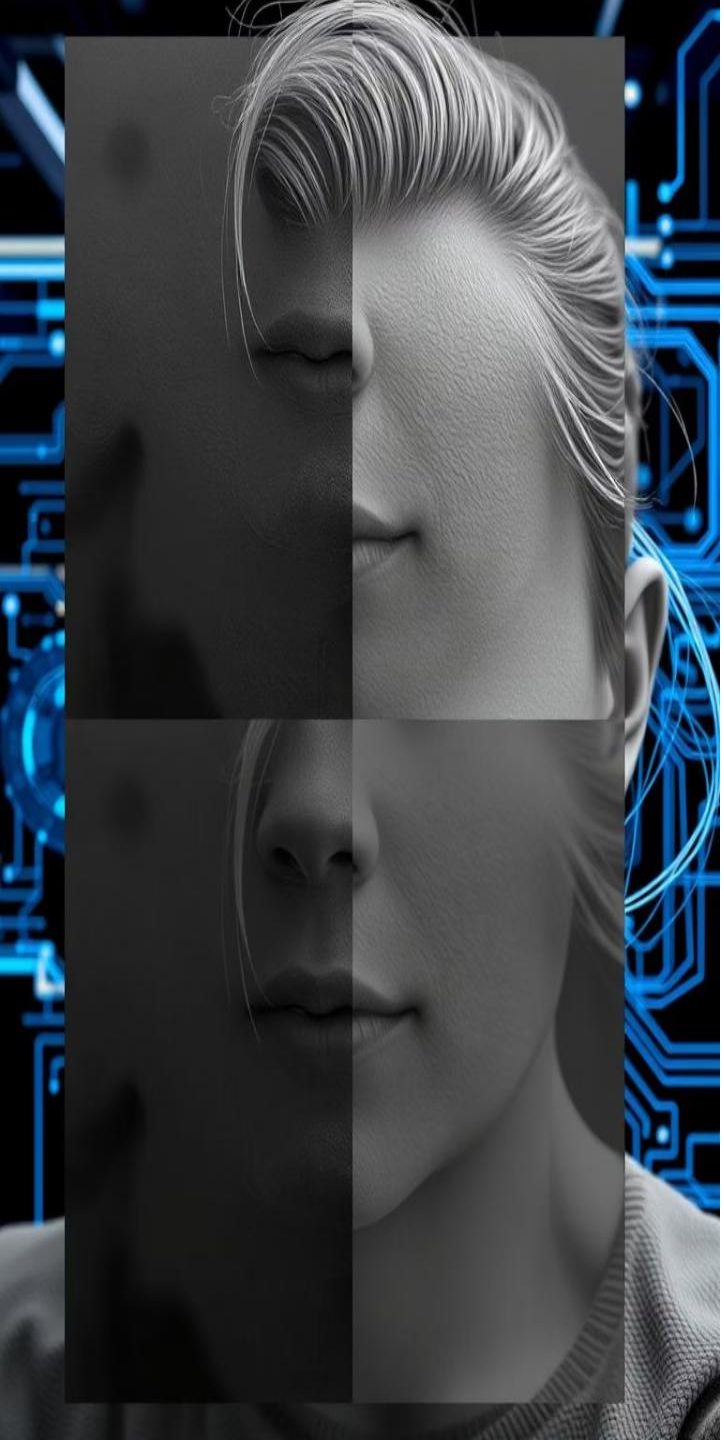
What is Redis?

Redis (short for REmote DIctionary Server) is an open-source, in-memory data structure store used as:

- a **database** (often key-value),
- a **cache**, and message broker

Redis is not a Python library, but a **standalone server software**, written in **C**.

Redis is an open-source, in-memory data structure store used as a database, cache, and message broker. It supports various data types such as strings, hashes, lists, sets, and sorted sets with range queries. Known for its high performance, Redis is often used to speed up applications by enabling fast access to data.



Expanded Image Processing Features

Current Capabilities

- Grayscale conversion
- Canny edge detection
- Threshold-based segmentation

Upcoming

Blur, sharpen, and contour detection planned.

Thank you