



EE547-DISCUSSION I



TOPICS WE ARE COVERING

- Homework I:
 - Question 1 – Distributed Merge of Lists
 - Question 2 – HTTP Client
 - Question 3 – Message Event Log
- Docker
 - What is Docker?
 - Simple start to Docker

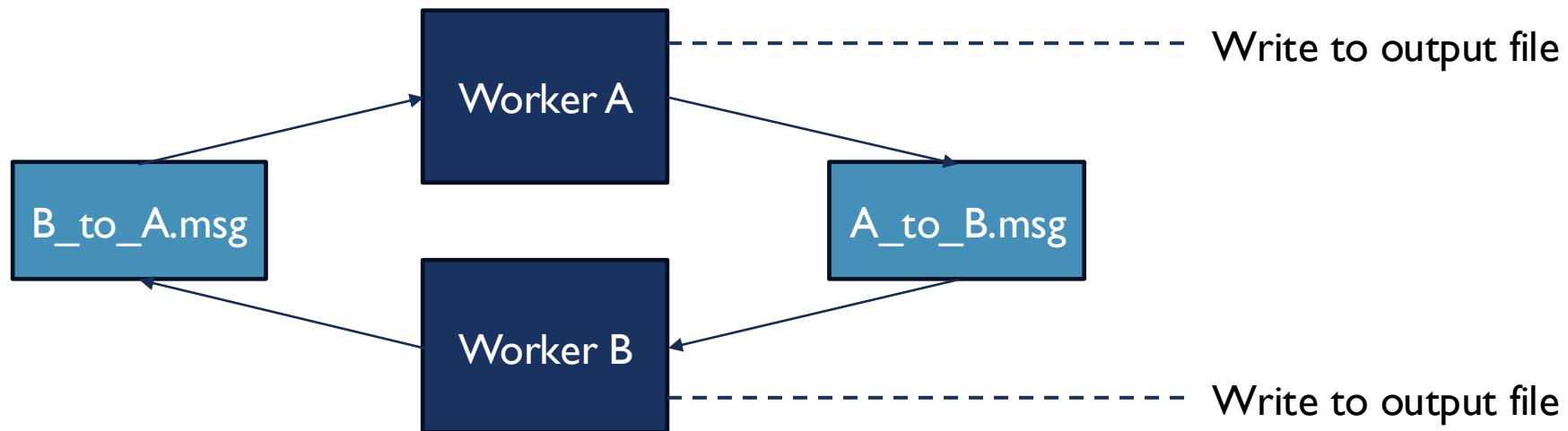


HOMEWORK I



Q1—PROBLEM OVERVIEW

- Fundamental:
 - 2 workers communicating with each other through simple messages
 - Each hold an integer list (unsorted)
 - Workers do not share memory: communicate through message (shared files) only
- Initial Setup:
 - Disjoint values are split between two workers
- Desired Output:
 - Final sorted list with all values



Q1—PROBLEM DETAILS

- Implement MergeWorker:
 - `_load_state`: provided in the starter (loads saved state, or initialises it)
 - `_save_state`: provided in the starter (also syncs stats into the state file)
 - `_initial_state`: creates initial state on first startup. Must include a "stats" entry; the rest is your design
 - `step()`:
 - Read from inbox
 - Update states
 - Write message to outbox
 - Write finalised value to output
 - Save state
 - `get_stats`: what did the worker do this step? Include all information needed from WorkerStats.

QI—CONSTRAINTS

- No shared memory. Workers only communicate via limited messages! (10 integers max, JSON file, message type less than 5 characters)
- At most one message per step!
- Efficiency & work balance affect grade
- Workers are rebuilt every step: only the state file persists
- State file holds at most 5000 integers (enforced by the provided monitor)

QI—HINTS

- Both workers share the same code: ensure symmetric protocols!
- **Sorting the data** at initialisation could help
- Exchanging ranges first lets many values be output without comparisons
- Work should best be balanced:
 - # of comparisons
 - # of messages written
 - # of messages read
 - # of values written

Q2—PROBLEM OVERVIEW

- HTTP Client to handle real-world failures
- Take into consideration:
 - Success
 - Slow response
 - Timeouts
 - Server errors
 - Client errors
 - Connection failures

Q2—ERROR CODES

Expected Behavior Matrix

Condition	Callbacks	Retry?
200 OK, fast	<code>on_success</code>	No
200 OK, slow (>500ms)	<code>on_success</code> , <code>on_slow_response</code>	No
200 OK, body contains “error”	<code>on_success</code> , <code>on_body_match("error")</code>	No
400 Bad Request	<code>on_client_error</code>	No
404 Not Found	<code>on_client_error</code>	No
500 Internal Server Error	<code>on_server_error</code> , then <code>on_retry</code> , ...	Yes
503 Service Unavailable	<code>on_server_error</code> , then <code>on_retry</code> , ...	Yes
Timeout	<code>on_timeout</code> , then <code>on_retry</code> , ...	Yes
Connection refused	<code>on_connection_error</code> , then <code>on_retry</code> , ...	Yes
Max retries hit	<code>on_max_retries</code>	N/A

- Retry: repeat the request until success/fail for a set number of times
- Remember to log summary file according to instructions

Q2—FUNCTIONS/CLASSES

- **Calculate_backoff:**
 - Compute a retry delay
- **ResponseHandler:**
 - Increment status counts (total/successes/failure/retries etc.)
 - Record latency
 - Etc. (details on website)
 - Log successful requests/failures etc. in the requested format
 - Stats can live in the handler/client. This is up to you.

Q2—FUNCTIONS/CLASSES

- **RobustHTTPClient:**

- Fetch (fetching a single URL):
 - Retrying until success/failure/max_retries reached
 - Call handler on different cases
 - Return True if eventually successful. False if failed
- Fetch all:
 - Keep fetching URLs from the provider until none left
 - Log according to summary template provided

Q2—TIPS AND CLUES (SAMPLE)

- For logs in each case, refer to “Output Format” section strictly
- Summary statistics come from both successes and failures
- **Don’t retry 4xx** — treat them as permanent failures
- **Do retry 5xx**, timeouts, and connection errors

Q3—PROBLEM OVERVIEW

- Maintain a **durable append-only log** from an unreliable message stream
- Messages may be:
 - **out of order** (reordering), **duplicated**, **corrupted**, **lost**, **delivered late**
- Logger must handle:
 - coverage (write as many messages as possible)
 - ordering (minimise inversions)
 - durability (crash at any time → written log must remain valid)
- Fundamental trade-off:
 - **Write now** → safe but out of order
 - **Buffer and wait** → better order but larger risk of losing buffered data on termination

Q3—BUFFER

- The buffer **holds packets before writing**: you can reorder and keep packets here before writing
- What is written cannot be changed, but the buffer can change
- Buffer size = 1: Write every message that arrives immediately
- Buffer size = N: Keep N messages without writing. These may be reordered, or wait for missing messages before writing.
- If termination happens and the buffer is not flushed yet, we lose values in the buffer!
- Abrupt termination runs no cleanup: do not catch `SystemExit` or use `atexit`

Q3—PROBLEM DETAILS

Handling Different Conditions

Condition	Detection	Recommended Response
Reordering	<code>packet.sequence < expected</code>	Buffer and sort, or write with <code>LATE</code> status
Duplicate	<code>sequence in seen_sequences</code>	Discard silently
Corruption	<code>verify_checksum() == False</code>	Call <code>request_retransmit()</code>
Gap	Missing sequence after timeout	Call <code>request_retransmit()</code> or accept loss
Termination	<code>receive() returns None</code>	Flush buffer immediately



DOCKER



WHAT IS DOCKER?

- Docker is a platform for building, shipping, and running applications
- Allows developers to package code + dependencies into **containers**
- Eliminates “works on my machine” issues
- Containers run reliably across laptops, servers, cloud, CI/CD
- Lightweight compared to virtual machines

VMS VS. CONTAINER

Virtual Machine	Container
Virtualizes hardware	Virtualizes OS user space
Includes full OS kernel	Shares host kernel
Heavy (GBs)	Lightweight (MBs)
Slow startup	Fast startup
Strong isolation	Process-level isolation

IMAGES

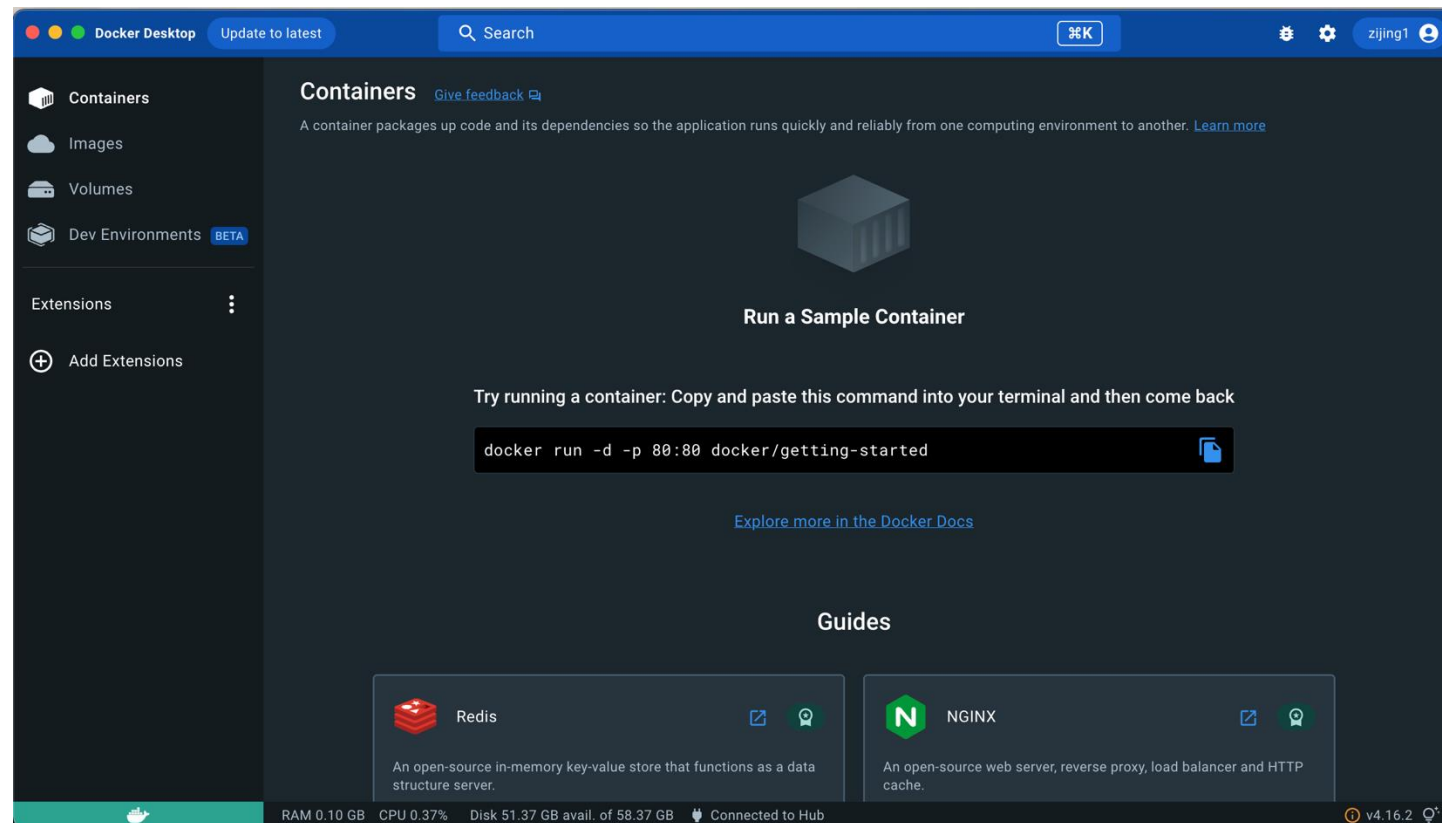
- Read-only template for a container
- Includes:
 - application code
 - dependencies
 - libs & runtimes
 - environment settings
- Example: `python:3.11` image contains Python 3.11

CONTAINERS

- Running instance of an image
- For understanding, compare:
 - **Image = class**
 - **Container = object**
- Multiple containers can run from one image

DOCKER DESKTOP

- Download Docker Desktop: <https://www.docker.com/products/docker-desktop/>



DOCKER:VERIFY INSTALLATION

- Verifying installation:
 - `docker --version`
 - `docker run hello-world`

RUNNING DOCKER

- Busybox: used for this tutorial, a small Linux toolkit to show how Docker is used. We will move on to more advanced examples in the future.
- Commands:
 - `docker pull busybox`
 - `docker images` (or simply check images in Docker Desktop)
 - `docker run busybox` (nothing appears as we did not give it a command)
 - `docker run busybox echo "hello"`
 - `docker run -it busybox sh`
- To remove a container after you don't want it anymore:
 - `docker rm <container ID>`

DOCKER STOPPING/RUNNING CONTAINERS

- Check all running containers:
`docker ps`
- Check all containers:
`docker ps -a`
- If we want to attach the container to console again:
`docker start -ai <container_id or name>`