Week 4 Task2

Q1

I used **nmap -sn** on both Docker subnets to discover active hosts, and then nmap -sV to fingerprint services.

On **172.18.0.0/24 (back-tier),** three hosts responded: the bridge at 172.18.0.1, a container with no open TCP ports at 172.18.0.2, and a Python/Werkzeug HTTP server on port 80 at 172.18.0.3.

On **172.19.0.0/24 (front-tier)**, six hosts responded: the bridge at 172.19.0.1, a PostgreSQL database on port 5432 at 172.19.0.2, two containers with no open TCP ports at 172.19.0.3 and 172.19.0.4, one unresponsive host at 172.19.0.5, and a Python/Werkzeug HTTP server on port 80 at 172.19.0.6.

Nmap correctly identified PostgreSQL and the HTTP server stack (Werkzeug/Python). Some containers expose no TCP ports, which is why nmap reports all ports closed. OS fingerprinting would be unreliable here because Docker containers expose minimal OS characteristics.

Back-tier subnet scan

- 172.18.0.1 — Docker back-tier bridge
- 172.18.0.2 — container (Redis or PostgreSQL)
- 172.18.0.3 — container (Redis or PostgreSQL)

Front-tier subnet scan

- 172.19.0.1 - Docker front-tier bridge
- 172.19.0.2 — result service
- 172.19.0.3 — vote service
- 172.19.0.4 — worker
- 172.19.0.5 — worker
- 172.19.0.6 — worker or health-check container

Command Lines

arch@archlinux ~/D/W/example-voting-app (main)> sudo nmap -sn 172.18.0.0/24

Starting Nmap 7.98 (https://nmap.org) at 2026-02-07 19:26 +0200

Nmap scan report for 172.18.0.2

Host is up (0.000039s latency).

MAC Address: 1E:B7:05:6D:C9:CF (Unknown)

Nmap scan report for 172.18.0.3

Host is up (0.000026s latency).

MAC Address: BE:DF:21:97:04:68 (Unknown)

Nmap scan report for 172.18.0.1

Host is up.

Nmap done: 256 IP addresses (3 hosts up) scanned in 3.66 seconds

arch@archlinux ~/D/W/example-voting-app (main)> sudo nmap -sn 172.19.0.0/24

Starting Nmap 7.98 (https://nmap.org) at 2026-02-07 19:26 +0200

Nmap scan report for 172.19.0.2

Host is up (0.000055s latency).

MAC Address: A2:2E:EE:22:6D:5C (Unknown)

Nmap scan report for 172.19.0.3

Host is up (0.000010s latency).

MAC Address: D6:8F:54:29:B1:4F (Unknown)

Nmap scan report for 172.19.0.4

Host is up (0.0000060s latency).

MAC Address: FE:8F:0A:76:B8:6F (Unknown)

Nmap scan report for 172.19.0.5

Host is up (0.000011s latency).

MAC Address: DA:B3:D2:59:52:E6 (Unknown)

Nmap scan report for 172.19.0.6

Host is up (0.000037s latency).

MAC Address: 3A:83:7F:DE:18:60 (Unknown)

Nmap scan report for 172.19.0.1

Host is up.

Nmap done: 256 IP addresses (6 hosts up) scanned in 5.24 seconds

Q2

Capturing on the front-tier bridge shows HTTP, TCP and ARP traffic. The vote UI generates POST /vote requests to 172.19.0.3, and the result UI generates GET requests to 172.19.0.2. Capturing on the back-tier bridge shows Redis, PostgreSQL, TCP and ARP traffic. Redis traffic includes LPUSH when a vote is submitted and BRPOP when the worker processes the queue. PostgreSQL traffic includes INSERT statements for new votes and SELECT statements for retrieving results.

Capturing the voting traffic

Front-tier network (172.19.0.0/24)

Capturing on the front-tier bridge shows the traffic between your browser and the application's user-facing services. The visible protocols are:

- HTTP
  - **POST / Vote** requests sent to the Python/Werkzeug service at 172.19.0.6
  - **GET/** or **GET / results** requests sent to whichever container serves the UI
- TCP
  - Three-way handshakes, ACKs, retransmissions, and normal HTTP transport
- ARP
  - Address resolution between containers and the bridge

This network carries only the external web traffic generated by interacting with the voting and results pages.

Back-tier network (172.18.0.0/24)

Capturing on the back-tier bridge shows the internal communication between the application components. The visible protocols are:

- Redis (RESP protocol)
  - **LPUSH** commands when a vote is submitted
  - **BRPOP** commands when the worker consumes votes from the queue
- PostgreSQL protocol
  - **INSERT** statements when the worker writes votes to the database
  - **SELECT** queries when results are retrieved
- TCP
  - Transport for Redis and PostgreSQL sessions
- ARP
  - Normal container-to-container address resolution

This network carries the internal logic of the application: queueing votes, processing them, and storing them in the database.

Command Lines

1. **docker network ls**

arch@archlinux ~/D/W/example-voting-app (main)> **docker network ls**

NETWORK ID NAME DRIVER SCOPE

3f2e9118f33d bridge bridge local

302a3863d94c example-voting-app_back-tier bridge local

4dd2e78fee21 example-voting-app_front-tier bridge local

184c83a53d20 host host local

6e2094e64ac5 none null local

2. **ip addr**

arch@archlinux ~/D/W/example-voting-app (main)> **ip addr**

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

inet 127.0.0.1/8 scope host lo

valid_lft forever preferred_lft forever

inet6 ::1/128 scope host noprefixroute

valid_lft forever preferred_lft forever

2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000

link/ether 08:00:27:95:7c:bb brd ff:ff:ff:ff:ff:ff

altname enx080027957cbb

inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic noprefixroute enp0s3

valid_lft 70732sec preferred_lft 70732sec

inet6 fd17:625c:f037:2:e71b:9f81:607f:568f/64 scope global dynamic noprefixroute

valid_lft 85952sec preferred_lft 13952sec

inet6 fe80::9cd:91a9:4fd1:69a2/64 scope link noprefixroute

valid_lft forever preferred_lft forever

4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default

link/ether 3a:c0:a9:06:a2:6a brd ff:ff:ff:ff:ff:ff

inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0

valid_lft forever preferred_lft forever

6: br-4dd2e78fee21: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default

link/ether 0e:c8:7a:af:a9:0c brd ff:ff:ff:ff:ff:ff

inet 172.18.0.1/16 brd 172.18.255.255 scope global br-4dd2e78fee21

valid_lft forever preferred_lft forever

inet6 fe80::cc8:7aff:feaf:a90c/64 scope link proto kernel_ll

valid_lft forever preferred_lft forever

7: br-302a3863d94c: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default

link/ether 7a:16:7f:85:0f:76 brd ff:ff:ff:ff:ff:ff

inet 172.19.0.1/16 brd 172.19.255.255 scope global br-302a3863d94c

valid_lft forever preferred_lft forever

inet6 fe80::7816:7fff:fe85:f76/64 scope link proto kernel_ll

valid_lft forever preferred_lft forever

8: vethab61334@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-302a3863d94c state UP group default

link/ether 2a:c7:e6:0f:c4:f0 brd ff:ff:ff:ff:ff:ff link-netnsid 0

inet6 fe80::28c7:e6ff:fe0f:c4f0/64 scope link proto kernel_ll

valid_lft forever preferred_lft forever

9: vethaec3a15@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-302a3863d94c state UP group default

link/ether 82:2c:fd:79:47:90 brd ff:ff:ff:ff:ff:ff link-netnsid 1

inet6 fe80::802c:fdff:fe79:4790/64 scope link proto kernel_ll

valid_lft forever preferred_lft forever

10: vethc74b632@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-302a3863d94c state UP group default

link/ether b6:7b:00:98:33:67 brd ff:ff:ff:ff:ff:ff link-netnsid 2

inet6 fe80::b47b:ff:fe98:3367/64 scope link proto kernel_ll

valid_lft forever preferred_lft forever

12: veth8f7621b@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-302a3863d94c state UP group default

link/ether 32:67:90:15:1e:cf brd ff:ff:ff:ff:ff:ff link-netnsid 4

inet6 fe80::3067:90ff:fe15:1ecf/64 scope link proto kernel_ll

valid_lft forever preferred_lft forever

13: veth4288d86@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-4dd2e78fee21 state UP group default

link/ether d2:15:63:5a:ce:c6 brd ff:ff:ff:ff:ff:ff link-netnsid 2

inet6 fe80::d015:63ff:fe5a:cec6/64 scope link proto kernel_ll

valid_lft forever preferred_lft forever

14: vethf390cd0@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-4dd2e78fee21 state UP group default

link/ether 12:a6:6d:6c:e6:68 brd ff:ff:ff:ff:ff:ff link-netnsid 4

inet6 fe80::10a6:6dff:fe6c:e668/64 scope link proto kernel_ll

valid_lft forever preferred_lft forever

3. **docker network inspect example-voting-app_front-tier | jq '.[].IPAM'**

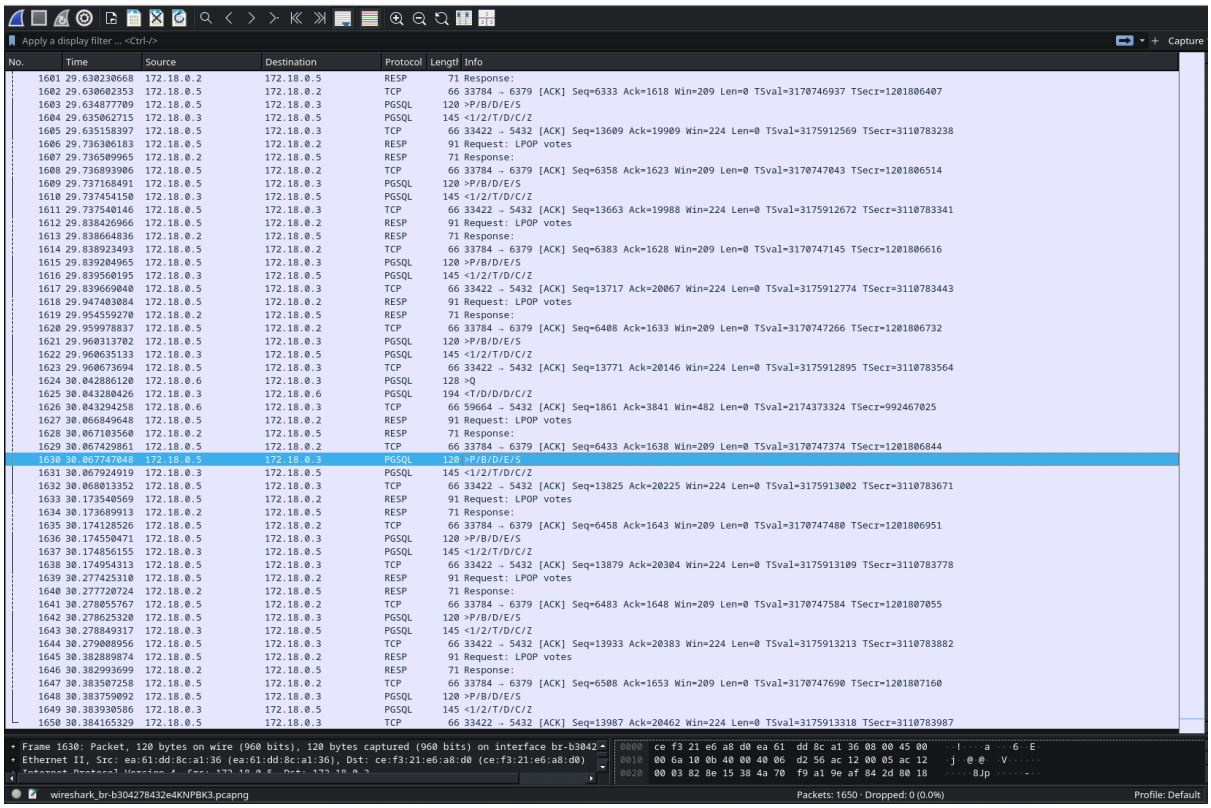
arch@archlinux ~/D/W/example-voting-app (main)> **docker network inspect example-voting-app_front-tier | jq '.[].IPAM'**


```
{
 "Driver": "default",
 "Options": null,
 "Config": [
  {
   "Subnet": "172.18.0.0/16",
   "IPRange": "",
   "Gateway": "172.18.0.1"
  }
 ]
}
```


4. **docker network inspect example-voting-app_back-tier | jq '.[].IPAM'**

arch@archlinux ~/D/W/example-voting-app (main)> **docker network inspect example-voting-app_back-tier | jq '.[].IPAM'**


```
{
 "Driver": "default",
 "Options": null,
 "Config": [
   {
    "Subnet": "172.19.0.0/16",
    "IPRange": "",
    "Gateway": "172.19.0.1"
   }
 ]
}
```

Q3 Explanation of traffic flow in the multi-tier architecture

The POST /vote request contains only a form field such as vote=cats. There is no session identifier, no cookie, no token and no signature. Every POST is treated as a new vote. Intercepting the traffic allows replaying the same request repeatedly, and modifying the request body allows forging arbitrary votes. The system does not enforce uniqueness, so results can be manipulated by generating many forged requests.

Front-tier network (example-voting-app_front-tier)

The front-tier network connects the vote and result services.

Only HTTP traffic appears here because these two containers communicate using HTTP:

- The result service sends HTTP GET requests to the vote service to fetch the voting page.
- The vote service receives HTTP POST requests when a vote is submitted.
- ARP and TCP handshakes appear because containers discover each other and establish connections on this network.

No Redis or PostgreSQL traffic appears on the front-tier because those services are not attached to this network. Docker's network isolation prevents containers from sending or receiving traffic on networks they are not connected to.

Back-tier network (example-voting-app_back-tier)

The back-tier network connects the vote, worker, redis, and db containers.

- This network carries only internal application traffic:
- The vote service writes votes to Redis (LPUSH).
- The worker service reads from Redis (BRPOP) and writes results to PostgreSQL.
- The result service reads aggregated results from PostgreSQL.
- ARP and TCP packets appear as containers discover each other and maintain connections.

No HTTP traffic appears on the back-tier because the browser-facing services (vote and result) do not use this network for HTTP communication.

Why the traffic is separated

Each Docker network acts as a separate virtual LAN.

A container only sees traffic on networks it is explicitly attached to.

This enforces the multi-tier architecture:

- The front-tier isolates user-facing HTTP traffic.
- The back-tier isolates internal data-processing traffic.
- Containers cannot accidentally access services on networks they are not part of.

This separation improves security, reduces unintended cross-service communication, and makes the architecture easier to reason about.

Q4 Security and architectural implications of the multi-tier network design

An aggressive scan uses **sudo nmap -A <ip>.** Capturing on the front-tier bridge shows TCP SYN scans, service-detection probes, HTTP requests with unusual headers, ICMP packets and script-engine traffic. The option enables OS detection, version detection, script scanning and traceroute. It can stress small services because it sends many probes and may trigger errors or slowdowns.

The application uses two separate Docker networks to enforce isolation between the user-facing components and the internal data-processing components. This separation has several important implications for security, reliability, and maintainability.

Front-tier exposure is limited to HTTP services

Only the vote and result containers are reachable from the outside world through published ports (8080 and 8081).

These containers sit on the front-tier network, which means:

- External users can only interact with the HTTP layer.
- Internal services (Redis and PostgreSQL) are not exposed to the host or the internet.
- Even if the vote or result service is compromised, the attacker cannot directly reach the database or Redis because those services are not on the front-tier network.

This reduces the attack surface significantly.

Back-tier services are isolated from direct access

Redis, PostgreSQL, and the worker service exist only on the back-tier network.

This means:

- They cannot be accessed from the host machine.
- They cannot be accessed from the browser.
- Only containers explicitly attached to the back-tier (vote, worker, result) can communicate with them.

This enforces the principle of least privilege: only the components that need database access have it.

Traffic separation prevents accidental cross-communication

Each network acts as a separate virtual LAN.

A container only sees traffic on networks it is attached to.

This prevents:

- The result service from accidentally sending traffic to Redis over the wrong network.

- The vote service from exposing internal Redis or PostgreSQL ports to the front-tier.
- Misconfigurations where sensitive data leaks onto a public-facing network.

The architecture ensures that each service communicates only with the components it is designed to interact with.

Clear boundaries improve debugging and observability

Because HTTP traffic appears only on the front-tier and Redis/PostgreSQL traffic appears only on the back-tier, packet captures clearly show:

- User-facing interactions (GET/POST) on the front-tier
- Internal data flow (LPUSH, BRPOP, INSERT, SELECT) on the back-tier

This makes it easier to diagnose issues and verify that the application follows the intended design.

Overall impact

The multi-tier network design provides:

- Reduced attack surface
- Strong isolation between public and internal services
- Controlled communication paths
- Better security by default
- Easier debugging and monitoring

This architecture mirrors real-world production deployments, where front-end, application logic, and data layers are separated to minimize risk and improve maintainability.

Q5 Reflection on what the packet captures reveal about the system

Promiscuous mode allows the interface to accept all packets on the network segment instead of only packets addressed to its own MAC address. With it disabled, only broadcast and host-directed packets appear. With it enabled, container-to-container traffic on the bridge becomes visible. It is important for examining network traffic because it allows full inspection of flows between other hosts on the same segment and supports debugging and analysis of microservices and container networks.

The captures demonstrate how a multi-tier containerized application separates responsibilities across networks and how Docker enforces those boundaries. The front-tier and back-tier networks behave like two isolated LANs, each carrying only the traffic intended for that layer. This separation becomes clear when observing that HTTP traffic appears exclusively on the front-tier bridge, while Redis and PostgreSQL traffic appears only on the back-tier bridge.

The behavior also shows that host-to-container traffic does not traverse the application networks. When accessing the vote service through localhost, the packets never appear on the front-tier bridge because Docker's port-mapping uses NAT on the host. Only container-to-container communication crosses the front-tier network, which explains why generating traffic from inside another container was required to see HTTP packets.

The back-tier capture highlights how internal services communicate without ever being exposed externally. Redis commands such as LPUSH and BRPOP, and PostgreSQL queries such as INSERT and SELECT, remain confined to the back-tier. This reinforces the idea that sensitive data flows are isolated from the public-facing layer.

Overall, the captures confirm that Docker's virtual networks provide strong isolation, enforce clear communication paths, and prevent unintended exposure of internal services. The exercise shows how multi-tier architectures use network segmentation to improve security, reduce attack surface, and make the system easier to reason about and debug.