ICN Term Project Report

June 2025

The name of our team members, student ID and workload are as follows:

Student ID/Members	Workload
B11901022丁睿濂	proxy server code
B11901055袁紹翔	proxy server code
B11901013俞柏安	debug/demo/report writing

The topic we're assigned is **Load Balancing Proxy Server**. We've done all three additional features.

1 Basic Functionality

(a) **Brief Explanation**

The port for two backend servers and proxy server are 8001, 8002 and 8888 respectively. The proxy server can parse, forward incoming request and relay response as usual. The iterator `ROUND_ROBIN` and function `round_robin_backend` are declared to implement RR algorithm, and the function is called only when there's no sticky backend and response file isn't cached. The variable `used_rr` indicates whether RR is used or not. The related part of codes are as follows:

```
# backend_server1.py
HOST, PORT = "127.0.0.1", 8001
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind((HOST, PORT))
serverSocket.listen(0)

# backend_server2.py
HOST, PORT = "127.0.0.1", 8002
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serverSocket.bind((HOST, PORT))
serverSocket.listen(0)

# load_balancer.py
BACKEND_SERVERS = [("localhost", 8001), ("localhost", 8002)]
LISTEN_PORT     = 8888
...
ROUND_ROBIN     = itertools.cycle(BACKEND_SERVERS)
rr_lock         = threading.Lock()
...
def round_robin_backend():
    return next(ROUND_ROBIN)
...
def handle(client, client_address):
    try:
        # 2.0 HTTP request parsing
        request = client.recv(65536)
        if not request:
            return
        print("==================================")
        print(f"[REQUEST] {request}")
        header_block = request.split(b"\r\n\r\n", 1)[0]
```

```
        headers      = header_block.decode(errors="ignore").split("\r\n")
        req_line     = headers[0]
        method, path, _ = req_line.split()
        print(f"[INFO] {method} {path} from {client_address[0]}:{client_address[1]}")
        ...
        # 2.2 Sticky-session cookie check
        ...
        used_rr   = False
        if backend is None:
            backend  = round_robin_backend()
            used_rr  = True
        ...
        # 2.3 Forward to backend
        with socket.create_connection(backend, timeout = TIMEOUT) as bsock:
            bsock.sendall(request)
            response = b""
            while True:
                chunk = bsock.recv(65536)
                if not chunk: break
                response += chunk
            bsock.close()
    ...
    client.sendall(response)
    ...
def main():
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind(("", LISTEN_PORT))
        s.listen()
        print(f"[+] Load balancer listening on :{LISTEN_PORT}")
```

(b) **Demonstration**

step (1)(2): Use terminal to run `backend_server1.py`, `backend_server2.py` and `load_balancer.py` simultaneously.

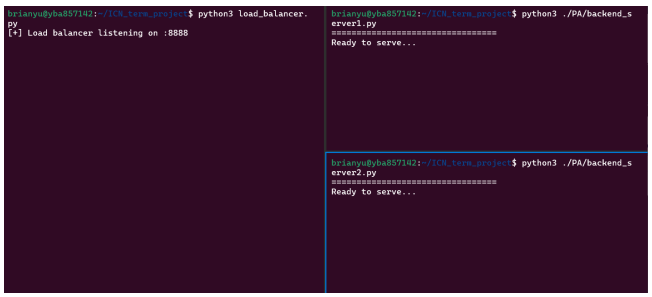

Figure 1: Running two backend servers and a proxy server initially

step (3)(4): First, we access `helloworld.html` through `localhost:8888`, and the server with port 8001 is first hit. Then, we access `index.html` through `127.0.0.1:8888`, and the server with port 8002 is then hit. Because we implemented sticky cookie and cache, it's required to remove the cookie in browser and the cached file in terminal, so there's an additional request after opening sidebar of the browser. We eventually access `helloworld.html` through `localhost:8888` again, and the server with port 8001 is hit. It can be seen that the consecutive requests are sent to different backend servers decided by Round Robin algorithm.

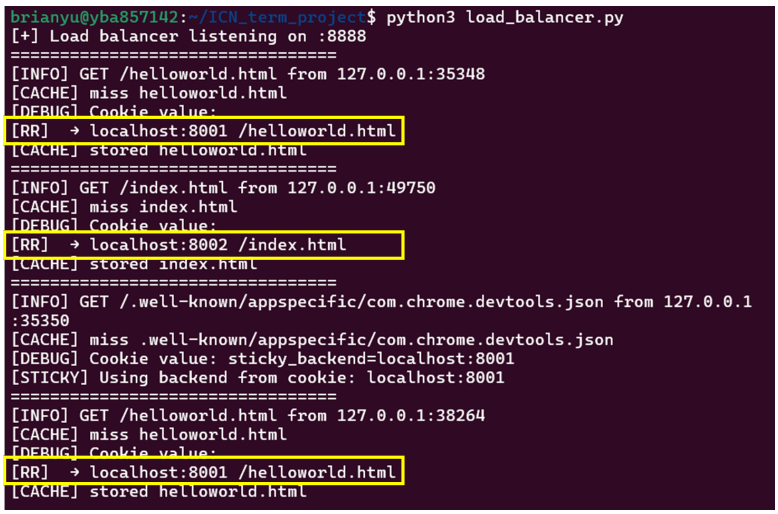

Figure 2: The effect of Round Robin algorithm

2 Additional Features

(a) **Proxy Error Handling**

(i) **Brief Explanation**

We use the function `build_http_error` to generate response for `502 Bad Gateway` and `504 Gateway Timeout` error. The connection time limit `TIMEOUT` is 5 sec, and a 10 sec delay is added to server with port number 8001 exclusively for this case. The related part of codes are as follows:

```
# backend_server1.py
...
while True:
    connectionSocket, address = serverSocket.accept()
    time.sleep(10000)

# load_balancer.py
TIMEOUT         = 5
...
def build_http_error(code, phrase):
    body  = f"<html><body><h1>{code} {phrase}</h1></body></html>".encode()
    hdr   = (f"HTTP/1.1 {code} {phrase}\r\n"
             f"Content-Length: {len(body)}\r\n"
             f"Content-Type: text/html; charset=UTF-8\r\n"
             "Connection: close\r\n\r\n").encode()
    return hdr + body
def handle(client, client_address):
    try:
        ...
        # 2.3 Forward to backend
        with socket.create_connection(backend, timeout = TIMEOUT) as bsock:
        ...
    except ConnectionRefusedError:
        client.sendall(build_http_error(502, "Bad Gateway"))
    except socket.timeout:
        client.sendall(build_http_error(504, "Gateway Timeout"))
```

(ii) **Demonstration**

step(1): same as the figure shown before

step(2): The server with port 8002 is closed (lower right window). We access `helloworld.html` through `localhost:8888` first, and the server with port 8001 is hit. Then, we access `index.html` through `127.0.0.1:8888`, and the server with port 8002 is selected with RR algorithm. However, since 8002 is not working, a `502 Bad Gateway` response is send back to the client.

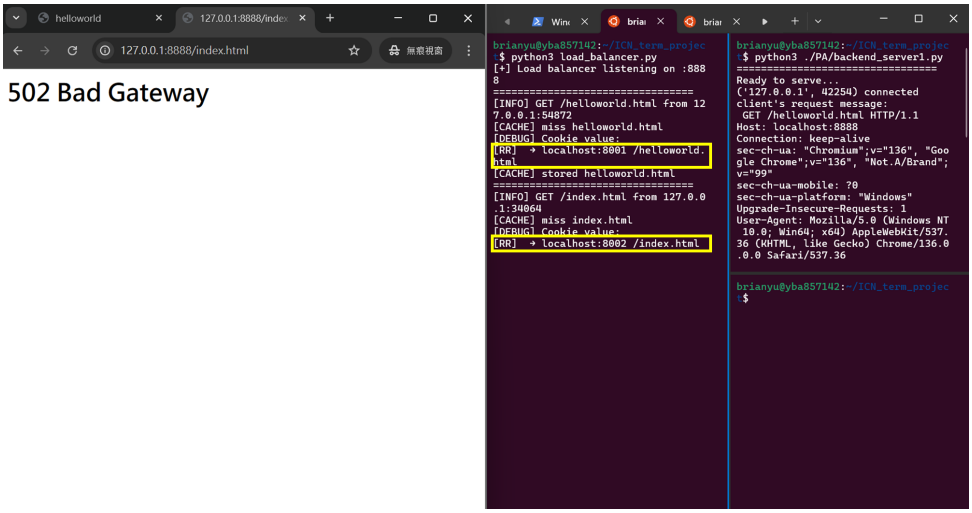

Figure 3: `502 Bad Gateway` response

step(3): Before the program starts, the code `time.sleep(10000)` is added in the server 8001, and `TIMEOUT` is set to be 5(sec). We then try to access `index.html` through `127.0.0.1:8888`, and server 8001 is chosen by RR. Since the sleep time is longer than `TIMEOUT`, a `504 Gateway Timeout` response is send back to the client. The output log of server remains empty since the connection breaks before the sleep time ends.

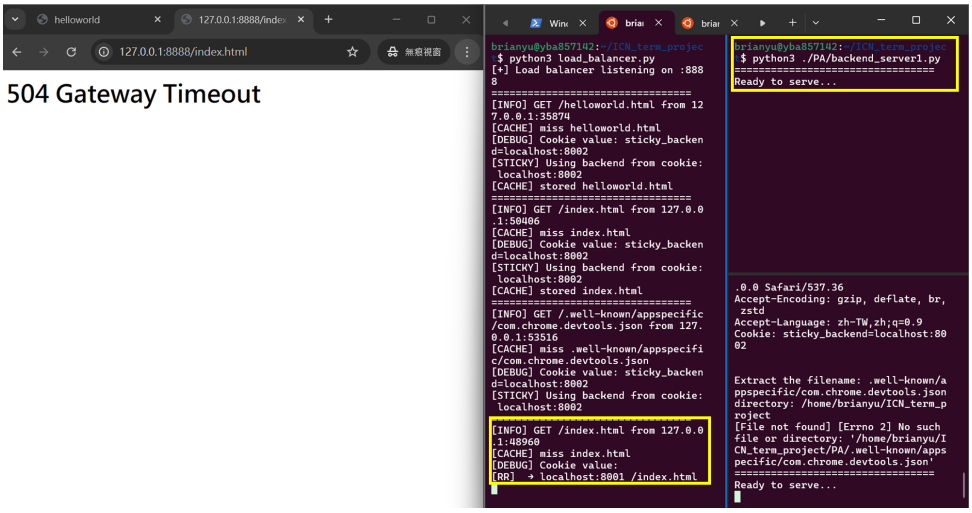

Figure 4: `504 Gateway Timeout` response

(b) **Simple Caching Integration**

(i) **Brief Explanation**

First, we define `CACHE_DIR` as the directory (under `cache` folder) where we save our cached files. When receiving a request, we check if the file exists. If not, the request will be forwarded to a server, and the response containing header will be saved in the file with whole directory `cache_file`. The related part of codes are as follows:

```
# load_balancer.py
CACHE_DIR       = os.path.join(os.getcwd(), "cache")
```

```
os.makedirs(CACHE_DIR, exist_ok=True)
...
CACHE_LOCK      = threading.Lock()
...
def handle(client, client_address):
    try:
        # 2.0 HTTP request parsing
        ...
        clean_path = path.split("?", 1)[0].lstrip("/") or "index.html"
        cache_file = os.path.join(CACHE_DIR, clean_path + ".cache")
        # 2.1 Cache lookup
        with CACHE_LOCK:
            if os.path.isfile(cache_file):
                print(f"[CACHE] hit  {clean_path}")
                with open(cache_file, "rb") as f:
                    temp = f.read()
                    client.sendall(temp)
                    print(f"[Response] {temp}")
                return
            else:
                print(f"[CACHE] miss {clean_path}")
        ...
        # 2.4 Cache store if 200 OK
        if response.startswith(b"HTTP/1.1 200"):
            with CACHE_LOCK:
                os.makedirs(os.path.dirname(cache_file), exist_ok=True)
                with open(cache_file, "wb") as f:
                    f.write(response)
                print(f"[CACHE] stored {clean_path}")
```

(ii) **Demonstration**

step(1): same as the figure shown before

step(2)(3):We access `index.html` through `localhost:8888`, and server 8001 is chosen by RR. In the first request(upper left window), backend server is contacted and sends a response starting with '200 OK'. In the second request(lower left window), no backend server was contacted since proxy server can send the cached file directly to the client. In addition, the terminal (lower right window) shows that after the first request, the new cached file is stored in the folder.

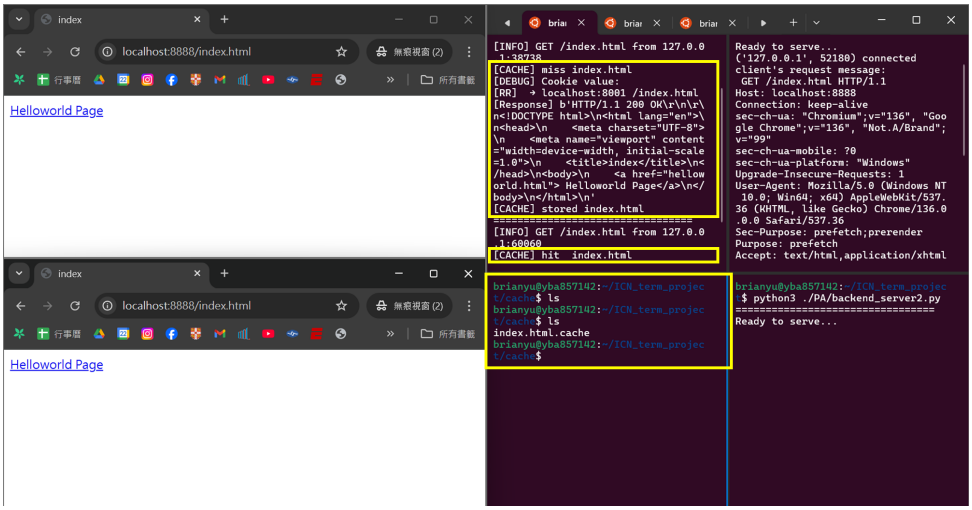

Figure 5: Cache miss and hit

(c) **Session Persistence via Cookies**

(i) **Brief Explanation**

In this part, we declare the function `choose_backend_from_cookie` to parse the header in request. We check if `sticky_backend` is in the header, and we connect to the backend server indicated by the cookie value to examine if it's available. If the server isn't available(`backend is None`), RR will decide the server number, and `used_rr` will br changed to `True`. When proxy finally receives response from backend server, a `Set-Cookie` header will be inserted into the response relayed to the client.

```
# load_balancer.py
def choose_backend_from_cookie(cookie_header: str):
    c = SimpleCookie()
    c.load(cookie_header)
    if "sticky_backend" not in c:
        return None
    host, _, port = c["sticky_backend"].value.partition(":")
    try:
        port = int(port)
    except ValueError:
        return None
    candidate = (host, port)
    try:
        with socket.create_connection(candidate, timeout=1):
            return candidate
    except OSError:
        return None
...
def handle(client, client_address):
    try:
        ...
        # 2.2 Sticky-session cookie check
        cookie_line = next((h for h in headers if h.lower().startswith("cookie:")), "")
        cookie_value = cookie_line.partition(":")[2].strip()
        print(f"[DEBUG] Cookie value: {cookie_value}")
        backend = choose_backend_from_cookie(cookie_value)

        used_rr   = False
        if backend is None:
            backend  = round_robin_backend()
            used_rr  = True

        host, port = backend
        if not used_rr:
            print(f"[STICKY] Using backend from cookie: {host}:{port}")
        else:
            print(f"[RR]  → {host}:{port} /{clean_path}")
        ...
        # 2.5 Inject Set-Cookie header if backend chosen by RR
        if used_rr:
            head, body = response.split(b"\r\n\r\n", 1)
            response   = b"\r\n".join([
                         head,
                         f"Set-Cookie: sticky_backend={host}:{port}; Path=/".encode(),
                         b"", body])
```

(ii) **Demonstration**

step(1): same as the figure shown before

step(2): We access `index.html` through `127.0.0.1:8888`. The response file isn't cached, and the domain name doesn't have cookie either. Hence, a backend server is determined by RR, and the response from it is cached. In addition, the `Set-Cookie:` header was added to the response for the browser to receive and save.

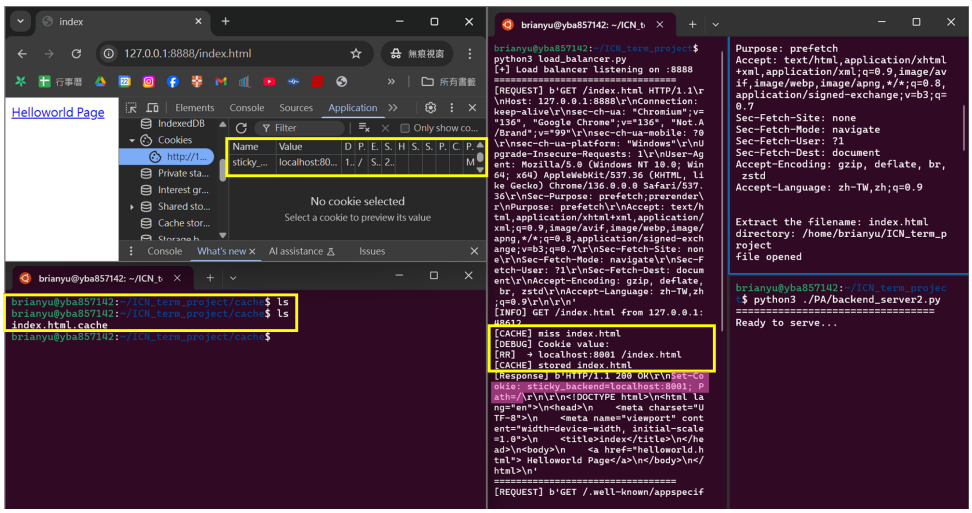

Figure 6: Initial request & cookie set

step(3): We then access `helloworld.html` through `127.0.0.1:8888`. There's cookie in the browser this time. Hence, `Cookie:` header exists in request from browser instead of response to browser.

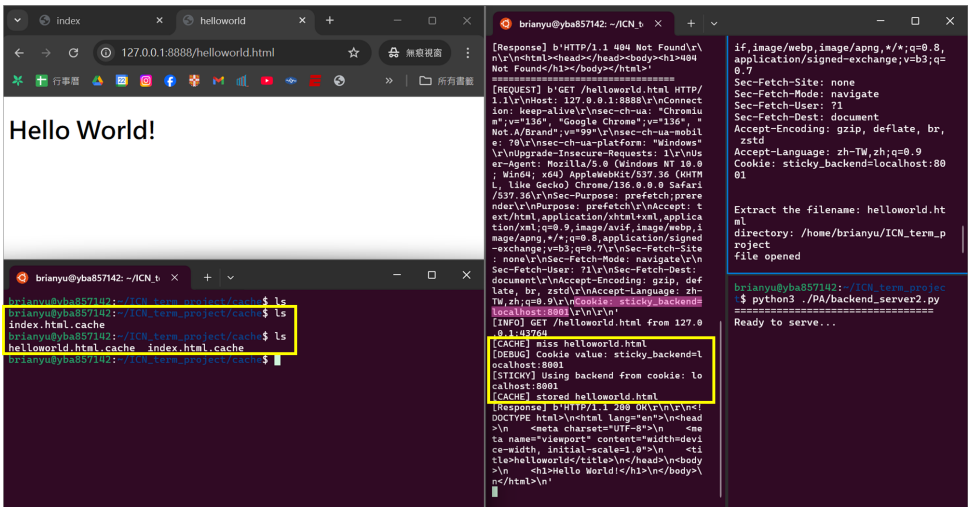

Figure 7: Sticky backend & cache miss

step(4): We access `index.html` through `127.0.0.1:8888` this time. Since we've done it at step(2), the response file has been cached, no backend server should be contacted. In addition, the `Set-Cookie:` header isn't in the cache file, so it won't be in the response either.

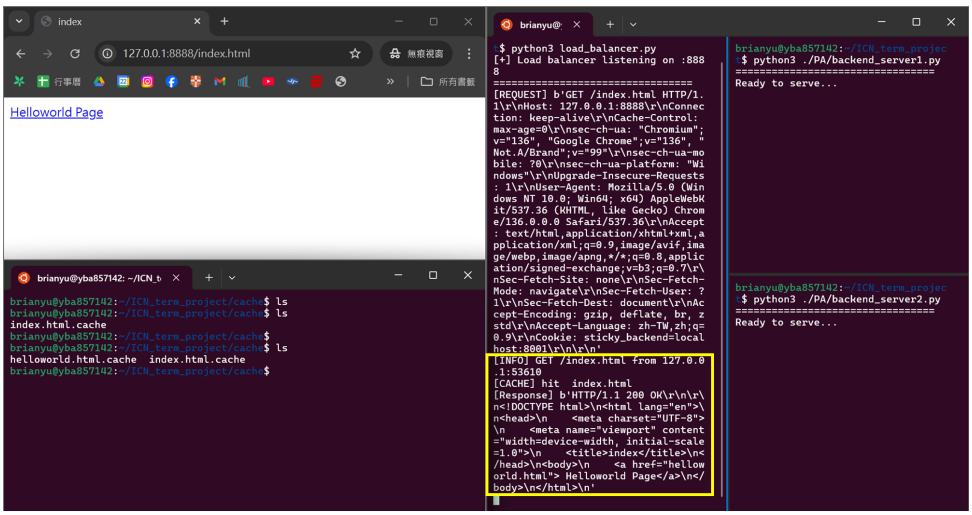

Figure 8: Cache hit request

step(5): We access `index.html` through `localhost:8888`, whose domain name is different from before so that there's no cookie inside. The response file isn't cached either. Hence, a backend server is determined by RR, and the response from it is cached. In addition, the `Set-Cookie:` header was added to the response for the browser to save the cookie.

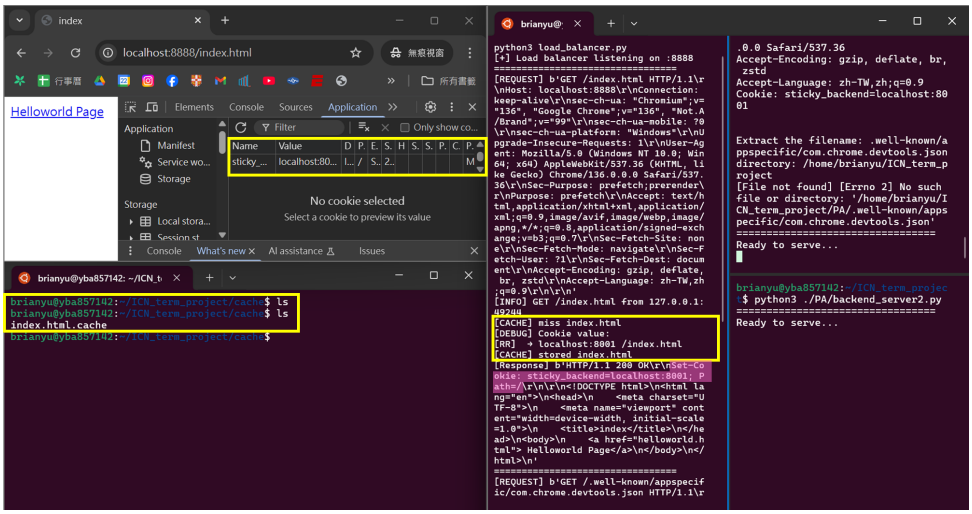

Figure 9: Different session & cache miss