دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

عنوان:

گزارش پروژه طراحی Mini NGINX

اعضای گروه

امیرحسن جعفرآبادی

امیرعلی سلیمی
سید علی طیب

نام درس

سیستم‌های عامل

نیم‌سال اول ۱۴۰۲-۱۴۰۳

نام استاد درس

دکتر اسدی

شرح پروژه: در این پروژه قصد پیاده‌سازی یک وب‌سرور مشابه nginx را داریم. وب‌سرور ما علاوه بر proxy کردن درخواست‌های HTTP به پورت‌های مختلف سیستم، وظیفه serve کردن فایل‌های static را نیز خواهد داشت. تمامی این تنظیمات و شخصی‌سازی‌ها تحت یک فایل config به وب‌سرور داده می‌شود و وب‌سرور با توجه همان فایل کانفیگ درخواست‌های HTTP را پاسخ خواهد داد.

۱ مراحل انجام پروژه

این پروژه شامل چند قسمت است، که متشکل از پیاده سازی فایل پیکربندی، رمزگشایی HTTP Request و هدایت درخواست‌ها به پردازه مورد نظر و سپس برگرداندن پاسخ آن، و همچنین پیاده سازی File Server که فایل‌های استاتیک را به کاربر برگرداند.

۱-۱ خواندن و پردازش فایل پیکربندی (config.txt)

پیاده سازی این بخش از پروژه در فایل های config.c و config.h انجام شده است. در هدر فایل config.h یک استراکت Route تعریف کرده‌ایم. این استراکت برای نگه‌داری rule route هاست و دارای شماره port آدرس path و url متناظر می‌باشد. همچنین در همین هدرفایل، پروتوتایپ سه تابع را تعریف کرده‌ایم. همچنین در فایل config.c یک آرایه از Route ها تعریف شده و تعداد Route هایی که در این آرایه قرار دارند را در متغیر number-of-routes ذخیره می‌کنیم. در ادامه به شرح توابع این قسمت خواهیم پرداخت:

- تابع get-route تعریف شده است. این تابع یک url در ورودی می‌گیرد و در خروجی، Route متناظر با آن url را برمی‌گرداند.

- تابع بعدی، تابع read-config-file است که در ابتدای برنامه فراخوانده می‌شود. وظیفه این تابع خواندن محتوای فایل config.txt و ساختن آرایه Routes است.

- آخرین تابع نیز print-all-config است که بیشتر جنبه دیباگ دارد و تمام Route ها را در ترمینال پرینت می‌کند.

۱-۲ پاسخ به درخواست‌های HTTP

در اینجا برای پاسخ به درخواست‌های HTTP کاربران چند حالت وجود دارد:

- در صورتی که درخواست کاربر، آدرس url باشد که در فایل کانفیگ به یک پورت مپ شده باشد، در این صورت باید درخواست HTTP به پردازه‌ای که روی پورت مد نظر در سرور در حال اجرا است هدایت شده و پاسخ آن به کاربر برگردانده شود
- در صورتی که درخواست کاربر، آدرس url باشد که در فایل کانفیگ به یک آدرس از کامپیوتر سرور مپ شده باشد، باید یک File Server فایل‌های پوشه مورد نظر را نمایش دهد و درصورتی که یک فایل بود، آن را به صورت باینری به کاربر جهت دانلود برگرداند.

برای پیاده‌سازی HTTP Server از لایبرری <sys/socket.h> استفاده شده است. این لایبرری جهت Socket Programming در زبان سی استفاده می‌شود، که توابع socket, bind, send, recv, connect و پاره‌ای از توابع دیگر را جهت برقراری ارتباط در لایه TCP پیاده‌سازی کرده است.

در حالت اول، باید وب سرور ما در حکم Proxy Server بازی کند و صرفا درخواست HTTP را به پردازه مورد نظر ارسال کرده و پاسخ را برگرداند. این بخش در فایل main.c پیاده سازی گردیده است.

۱-۲-۱ تکنیک‌های مورد استفاده

Multi Threading جهت تسریع در پاسخگویی به درخواست‌های کاربران، از لایبرری <pthread.h> استفاده کرده‌ایم تا هر درخواست کاربر را یک پردازه به طور مجزا پاسخ دهد. اگر این عملیات پیاده سازی نمی‌شد، دو اشکال در برنامه به وجود می‌آمد:

- هر درخواست کاربر باید معطل می‌ماند تا درخواست‌های کاربران قبلی پاسخ داده شود و بعد از آن سرور به درخواست کاربر کنونی بپردازد.
- اجرای درخواست‌های کاربران به علت اجرای تک هسته‌ای کندتر انجام می‌شد.

بنابراین با استفاده از تابع pthread_create برای پاسخ به درخواست HTTP یک پردازه جدید ایجاد می‌کنیم و آن پردازه را برای اجرای تابع handle_req که وظیفه پاسخگویی به HTTP Request را دارد، کال می‌کنیم.

Regular Expression در بخش پردازش URL از لایبرری <regex.h> زبان سی استفاده شده است که عملیات پردازش HTTP Request و استخراج پترن از آن را سریعتر و راحت‌تر انجام می‌دهد. این لایبرری استراکت regex_t را در اختیار ما قرار می‌دهد که با استفاده از آن و تابع regexec می‌توان به راحتی هر پترنی را از درخواست HTTP ارسال شده استخراج نمود.

۱-۳ پیاده‌سازی File Server

پس از خوانده شدن درخواست و پیداکردن Route مربوط به آن در صورتی که Route دارای قسمت static_path باشد تابع handle_static صدا زده می‌شود. این تابع ابتدا با ترکیب کردن url و static_path آدرس فایل یا دایرکتوری مورد درخواست را پیدا می‌کند. سپس بررسی می‌کند که در آدرس فایلی وجود دارد یا خیر و در صورت عدم وجود فایل ارور ۴۰۴ برمی‌گرداند.

در صورت وجود فایل با استفاده از fstat نوع آن را بررسی می‌کند و برای فایل‌های معمولی از تابع get_file و برای دایرکتوری‌ها تابع get_directory را صدا می‌زند.

تابع get_file فایل را می‌خواند و آن را به صورت باینری در بدنه‌ی یک HTTP Response با Content-Type از نوع application/octet-stream قرار می‌دهد.

تابع get_directory با استفاده از توابع opendir و readdir محتویات دایرکتوری را می‌خواند و با استفاده از stat مشخصاتی مانند modified_time و سایز آن‌ها را به دست می‌آورد. سپس یک صفحه‌ی HTML با ساختاری شبیه به خروجی NGINX می‌سازد و برمی‌گرداند.

۱-۳-۱ تکنیک‌های مورد استفاده

Local Caching جهت تسریع در پاسخگویی به درخواست‌های دریافت فایل کاربران، از روش etag با استفاده از هدرهای Etag و Last-Modified در Http-Response استفاده کردیم که موجب می‌شود هرگاه مرورگر درخوستش را با Http-Request با هدر If-None-Match که باید با Etag مقایسه شود و یا هدر If-Modified-Since که باید با هدر Last-Modified مقایسه شود؛ ارسال کند، آنگاه پاسخ کد ۳۰۴ دریافت خواهد کرد که مرورگر از کش لوکال برای سرو کردن فایل استفاده می‌کند. هدر Etag باید برای هر فایل یکتا باشد و Last-Modified نیز بر اساس آخرین زمان تغییر فایل ست می‌شود که برای آن از تابع st_mtime کمک گرفتیم. همچنین برای تولید Etag نیز از آخرین زمان تغییر فایل و حجم فایل به بایت استفاده شده است.

۲ اجرای پروژه

برای build و اجرای پروژه در فایل README.md توضیحات و راهنمایی‌های لازم را قرار داده‌ایم. ولی به طور کلی ما برای راحتی کار یک Makefile تعریف کرده‌ایم که به کمک آن می‌توانید سرویس‌های مختلفی را اجرا کنید. دستورات Makefile را در تصویر زیر مشاهده می‌کنید:

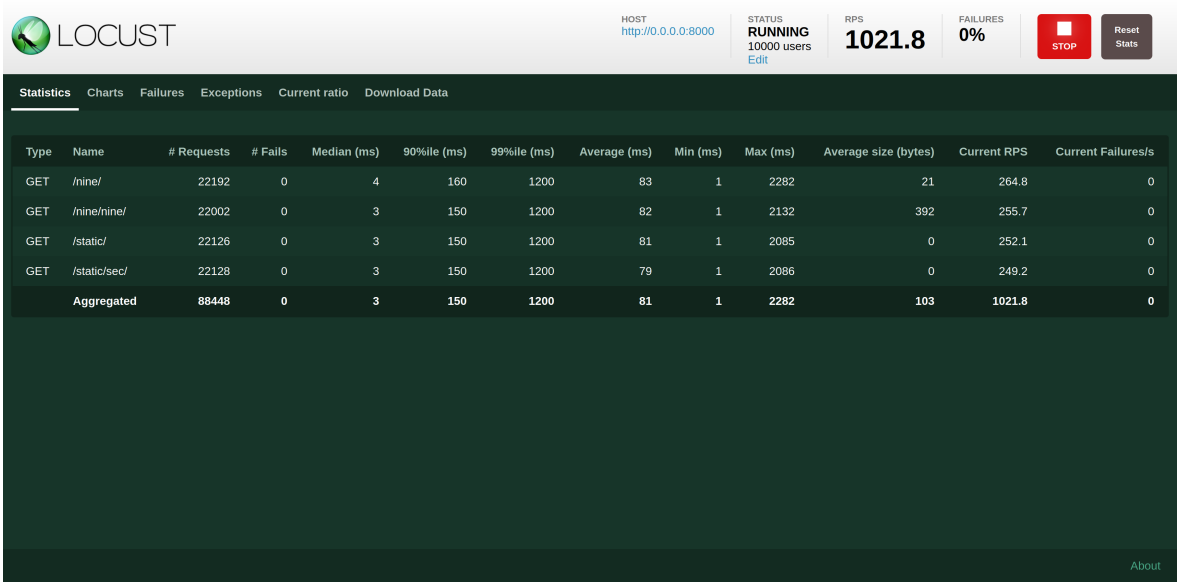
```
build:
    gcc src/config.c src/main.c src/static.c -g -o main
run:
    gcc src/config.c src/main.c src/static.c -g -o main && ./main

clean:
    rm main

django-init:
    python -m venv test/basic-django-app/.venv && \
    . test/basic-django-app/.venv/bin/activate && \
    pip install -r test/basic-django-app/requirements.txt && \
    test/basic-django-app/.venv/bin/python test/basic-django-app/manage.py migrate && \
    deactivate

django-run:
    test/basic-django-app/.venv/bin/python test/basic-django-app/manage.py runserver 9000

http-server-run:
    python -m http.server 9090

load-test:
    (echo Note that you sholud run django first)
    locust -f test/locustfile.py
```

شکل ۱: فایل Makefile که به کمک آن سرویس‌های مختلف پروژه قابل اجراست

۳ تست بار با Locust

در root پروژه یک پوشه به نام test ساخته‌ایم و در آن تمام مواردی که برای تست پروژه نیاز می‌باشد را قرار داده‌ایم. از جمله یک فایل تست بار locusfile.py و یک پروژه جنگوی کوچک به نام basic-django-app. شیوه استفاده و تست از آنها را به تفصیل در فایل README.md پروژه توضیح داده‌ایم. لکن در اینجا نیز اشاره خواهیم کرد.

ابتدا یک http.server پایتونی و یک پروژه django را اجرا می‌کنیم. همچنین دو پوشه می‌سازیم تا محتوای static ما را نگه‌داری کنند. سپس همه اطلاعات این چهار مورد را در فایل config.txt وارد کرده و فایل تست بار (locustfile.py) را اجرا می‌کنیم. سپس در محیط گرافیکی آن مقدار host را روی آدرس و پورتی که Mini-Nginx در حال اجرا است، تنظیم می‌کنیم. با شروع تست می‌توانیم تعداد درخواست در ثانیه و میانگین زمان ریسپانس و همچنین درصد failure را مشاهده کنیم. تصویر زیر حین تست گرفته شده است و در آن پارامترهای گفته شده معلوم است:



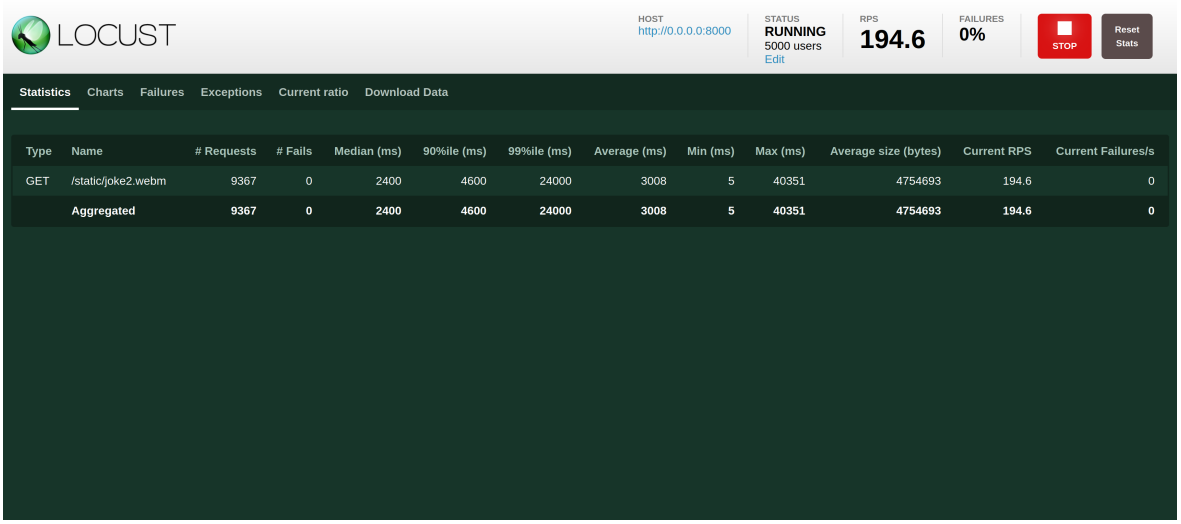
شکل ۲: تست بار با استفاده از Locust

۴ مطالعه فرسایشی روی تکنیک‌های مورد استفاده

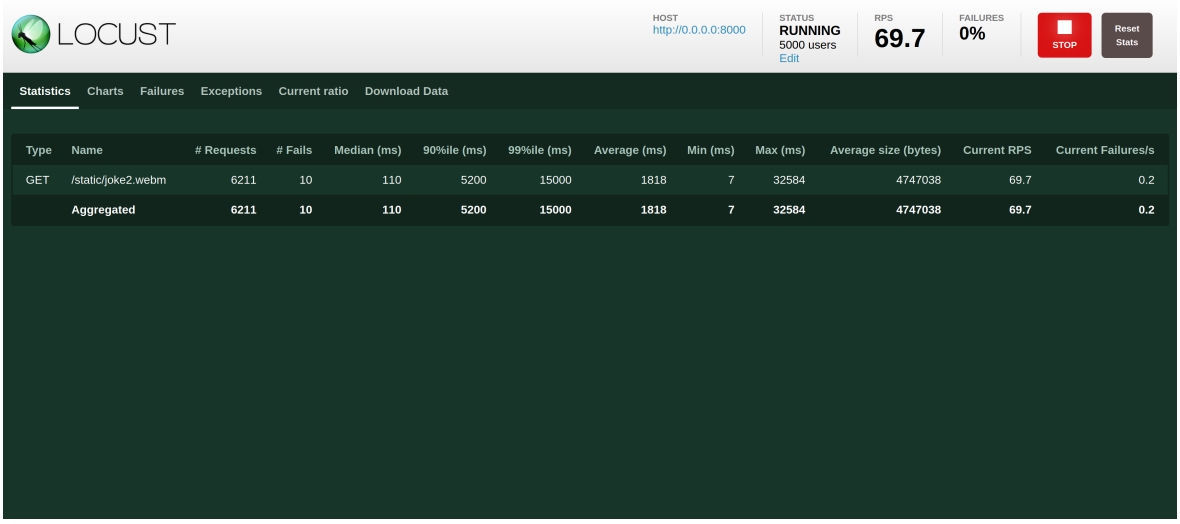
۴-۱ بهره‌گیری از قابلیت Multi-threading

اگر تابع main را مشاهده کنید. می‌بینید که در حلقه اصلی برنامه، به ازای هر درخواست جدید که می‌آید، یک ترد می‌سازیم و پردازش آن درخواست را به آن ترد واگذار می‌کنیم. با این کار ترد اصلی برنامه مشغول نمی‌ماند و دوباره می‌تواند مشغول listen کردن درخواست‌های جدید شود.

برای بررسی فرسایشی بهبود احتمالی این کار، یک متغیر عمومی به نام USE-THREAD در فایل defs.h تعریف کردیم. زمانی که این متغیر صفر شود، برنامه هیچ thread ای نمی‌سازد و تمامی درخواست‌ها را به صورت سریال و سنکرون انجام می‌دهد. سپس با تغییر فایل تست بار (locustfile.py) یک درخواست یک فایل نسبتا حجیم (برای مثال یک ویدئوی ۵ مگابایتی) را تعریف می‌کنیم. با اجرای تست بار می‌توان مشاهده کرد که در حالت Multi-threaded تعداد درخواست بر ثانیه (یا همان per request second) بیشتر از حالت سریال است. تصاویر زیر از نتایج تست بار گرفته شده‌اند:



شکل ۳: تست بار درخواست فایل نسبتا حجیم در حالت multi-threaded



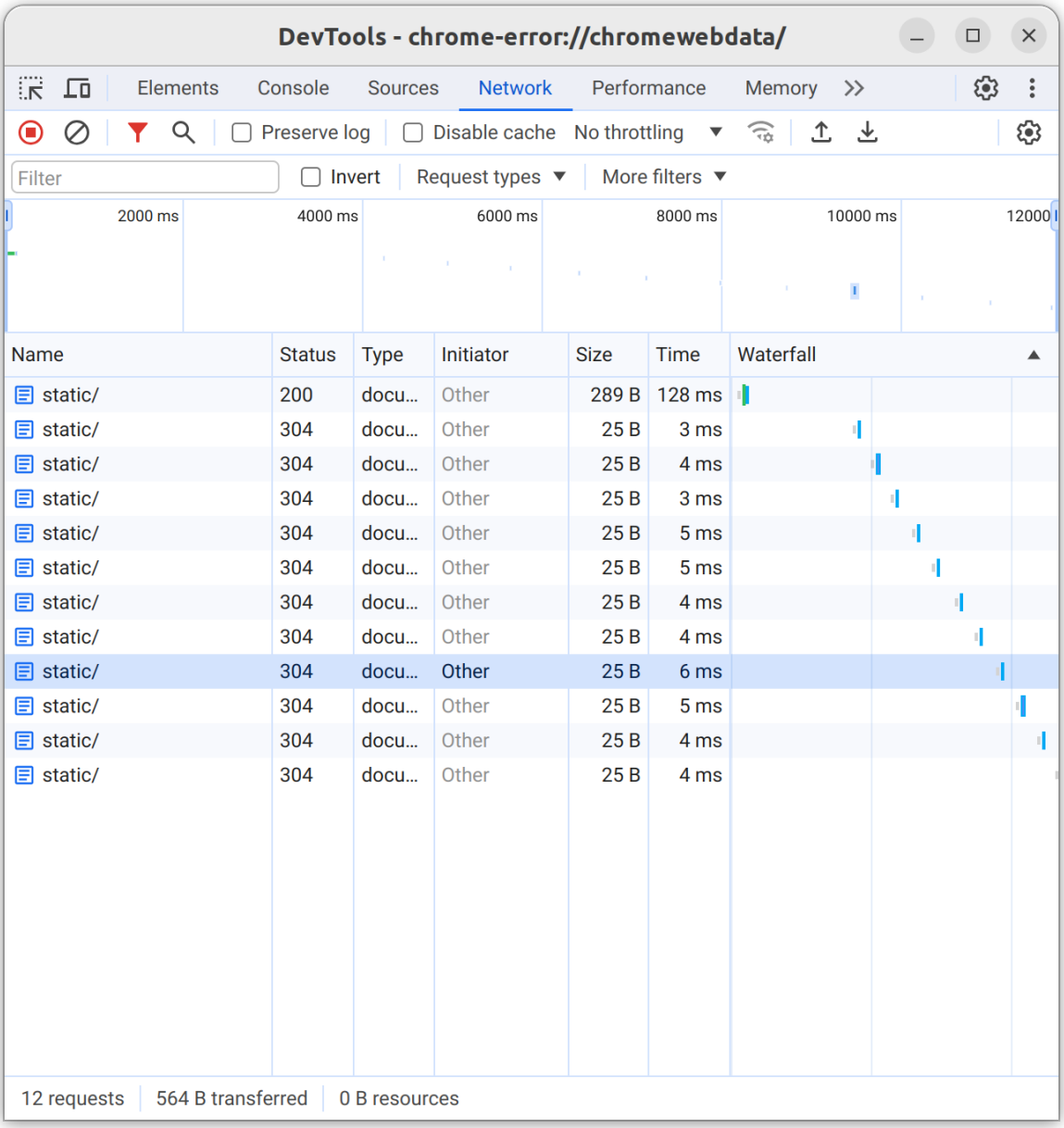
شکل ۴: تست بار درخواست فایل نسبتا حجیم در حالت single-threaded

۲-۴ تنظیم هدر برای cache کردن

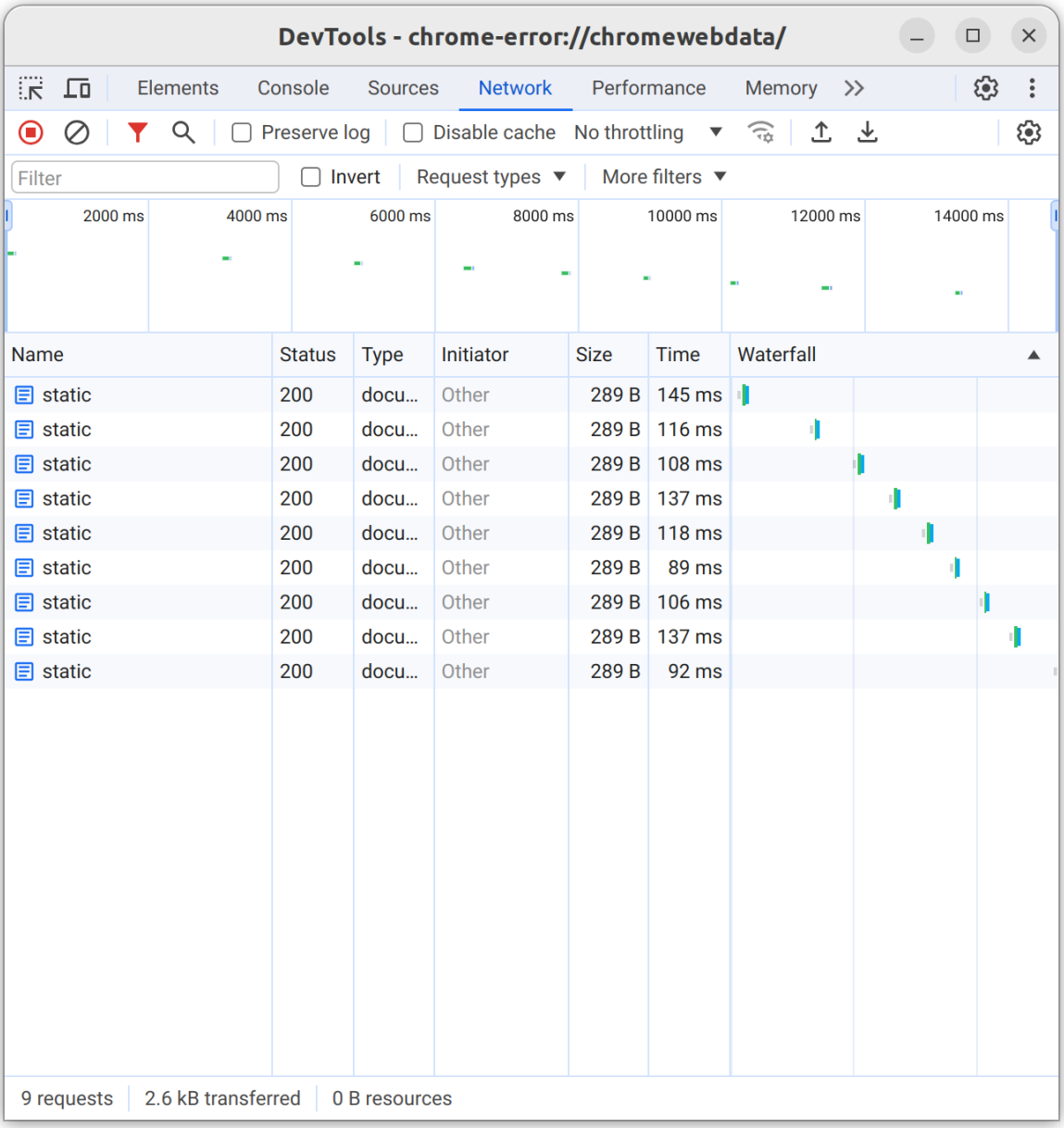
برای فعال یا غیرفعال کردن سیستم کش کردن فایل‌های استاتیک، متغیر USE_STATIC_CACHE در فایل defs.h تعریف شده است که اگر برابر با ۱ تعریف شده باشد، سیستم کش فعال بوده و درصورتی که برابر ۰ تعریف شود، فایل‌ها کش نمی‌شوند. برای فایل‌های با حجم بالا، استفاده از کش کردن فایل‌ها می‌تواند تاثیر هنگفتی در سرعت اجرای برنامه داشته باشد.

همانطور که در شکل ملاحظه می‌کنید، پس از اولین درخواست که با HTTP-Response کد ۲۰۰ مواجه شده است، و حدود ms۱۳۰ نیز زمان می‌برد، باقی درخواست‌ها با کد ۳۰۴ مواجه شده و کمتر از ms۱۰ زمان برده‌اند.

در حالتی که کش غیرفعال باشد، تمام درخواست‌ها با کد ۲۰۰ مواجه شده و همواره تمام فایل برای دانلود توسط سرور سرو می‌شود.



شکل ۵: تست درخواست فایل نسبتا حجیم در حالت استفاده از Local-Caching در مرورگر کروم



شکل ۶: تست درخواست فایل نسبتا حجیم در حالت عدم استفاده از Local-Caching در مرورگر کروم