1 Первая глава

1.1 Протокол

SCGI – это протокол по взаимодействию с веб приложениями. Серверы, настроены по протоколу SCGI, когда видят конец посланного запроса, отправляют ответ клиенту и закрывают соединение.

Пример посылаемого запроса:

```
 "68:"
"CONTENT_LENGTH" <00> "17" <00>
"SCGI" <00> "1"  <00>
"REQUEST_METHOD«00> "GET«00>
"REQUEST_URI«00> "/requesturi«00>
"
"What do you want?"
```

Ответ:

```
        "Status: 200 OK" <0d 0a>
"Content-Type: text/plain" <0d 0a>
" "  <0d 0a>
"Answer"
```

Запрос формируется следующим образом: первой строкой идет десятичное представление длины всей последовательности заголовков (`N:`), после чего идет последовательность пар «заголовок» – «значение», причем первый заголовок должен быть `CONTENT_LENGTH`. Важно отметить, что запрос и ответ являются конкатенацией строк.

1.2 Тестирование

Для тестирования производительности использовалась утилита `wrk`. Данная утилита позволяет создавать значительную нагрузку на тестируемый сервер с выставлением следующих гиперпараметров:

— `t` – количество потоков для тестирования;
— `c` – количество подключений;
— `d` – количество секунд для тестирования.

Пример запуска команды:

```
wrk -t2 -c100 -d10s http://localhost/highloadtesting
```

1.3 Технология для асинхронности

В качестве технологии для реализации асинхронности был выбран API `epoll`. `epoll` позволяет осуществлять мониторинг заданных открытых файловых дескрипторов. В случае данной работы открытыми файловыми дескрипторами являются сокеты подключения клиентов. API отправляет уведомления при наступлении события готовности к очередному вводу/выводу клиента. Считывание уведомлений осуществляется в бесконечном цикле.

API предоставляет следующие вызовы:

— `epoll_create1(`*`int`* `size)`: функция создает структуру данных для всех файловых дескрипторов. Возвращает файловый дескриптор, который в последующем передается во всем остальные вызовы.

— `epoll_ctl(`*`int`* `epfd,` *`int`* `op,` *`int`* `fd,` *`struct`* `epoll_event *event)`: функция управления файловым дескриптором. Возможны следующие операции:

а) *EPOLL_ CTL_ ADD*: добавление файлового дескриптора для наблюдения за ним;

б) *EPOLL_ CTL_ DEL*: удаление файлового дескриптора из мониторинга;

в) *EPOLL_ CTL_ MOD*: изменение опций мониторинга;

г) *EPOLL_ CTL_ DISABLE*: потокобезопасная опция для отключения мониторинга за дескриптором в многопоточном приложении.

— `epol_wait(`*`int`* `epfd,` *`struct`* `epoll_event *events,` *`int`* `maxevents,` *`int`* `timeout)`: функция возвращает количество файловых дескрипторов, у которых поменялось состояние.

Существует 4 вида событий, за которыми можно наблюдать с помощью `epoll`:

а) `EPOLLIN` – событие для чтения данных;

б) `EPOLLOUT` – событие для записи данных;

в) `EPOLLERR` – событие об ошибке;

г) `EPOLLHUP` – событие о закрытии дескриптора.

В данной работе используются все события: в начале при создании подключения ставится мониторинг на все события, а после используются для понимания о том, какое событие на каком подключении произошло.

1.4 Многопоточность приложения

Многопоточность в данном приложение реализуется с помощью библиотеки `thread`.

Для того чтобы создать многопоточный сервер, создается основной сокет приложения подключения к SCGI-серверу. Для каждого базового подключения включается мониторинг подключений. По умолчанию, в файле конфигураций сервера установлен параметр многопоточности на 2.

Так как сервер многопоточный, а операции чтения и записи данных потенциально самые долгие операции, необходимо сделать основное подключение не блокирующим, то есть каждый поток будет независимо от другого производить операции ввода-вывода, исключая атомарность операций. Чтобы это сделать, необходимо добавить флаг `O_NONBLOCK` к основному подключению:

```
1  void set_non_blocking(int fd)
2  {
3    int flags = fcntl(fd, F_GETFL, 0);
4    if (flags == -1)
5      throw std::runtime_error("Can't get flags");
6
7    flags |= O_NONBLOCK;
8
9    if (fcntl(fd, F_SETFL, flags) == -1)
10     throw std::runtime_error("Can't set flags");
11 }
```

1.5 Чтение запроса

В определенном классе `Client` (клиент подключения к SCGI-серверу) определено перечисление из 3-х типов чтения запроса: `READING_HEADER_LENGTH, READING_HEADER, READING_CONTENT`. Эти типы служат для понимания, какую часть запроса процесс считывает в данный момент времени (все типы считываются последовательно, как это определено протоколом) и для подсчета уже прочитанного количества символов. Так как операциями чтения/записи занимается операционная система, процесс может не успеть дописать/прочитать данные, до того, как ОС заморозить процесс, для этого случая предусмотрена переменная `res`, которая хранит в себе текущее количество прочитанных символов (или позицию в буффере данных). Для поиска и вычленения нужной информации из заголовков используется библиотека `string`, которая предоставляет удобный функционал работы со строками и буфферами данных (поиск подстрок, например).

1.6 Отправка ответа на запрос

При успешном получении и считывании всего запроса, из него сервер получает поле URL и генерирует ответ по следующему правилу (согласно протоколу): в

ответ вставляется поле URL, которое пришло из запроса, а в конец дописываются 65536 букв `A` для увеличения размера ответа.

```
_response = std::string("Status: 200 OK""\x0d""\x0a"
    "Content-Type: application/octet-stream""\x0d""\x0a"
    "\x0d""\x0a"
    "You requested URI: ") +
    std::string(_uri) +
    std::string("\x0d""\x0a"
    "Your request content was: ") +
    std::string(_content) +
    std::string("\x0d""\x0a"
    "Garbage to make the response larger""\x0d""\x0a");
_response += std::string((1 << 16), 'A');
```

2 Вторая глава

2.1 Нагрузочное тестирование

Нагрузочное тестирование проводилось, изменяя при этом гиперпараметры сервера (количество потоков и количество соединений за заданный промежуток времени). Для того, чтобы тестировать сервер без использования второй машины (docker контейнер) была взята реализация утилиты `wrk`. Эта утилита позволяет произвести нагрузочное тестирование многопроцессорного сервера на одной машине.

На схеме (2.1) представлено визуально, как проводилось тестирование. Саму работоспособность сервера можно проверить из браузера, но для отправки множества запросов одновременно нужна некоторая система. Схема тестирования следующая: утилита `wrk` генерирует заданное количество соединений в заданном количестве потоков, отправляя запросы в заданном интервале времени на сервер `nginx`, который в свою очередь отправляет запросы на сервер `SCGI`. `SCGI` сервер обрабатывает запросы по мере поступления в одном (нескольких) потоках и при успешной завершении отправляет обратно ответ серверу `nginx`, который отправляет ответ в утилиту `wrk` для просчета статистики.

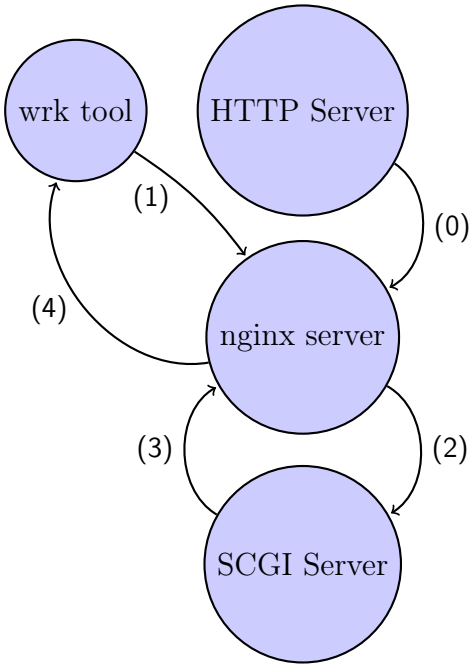

Рисунок 2.1 — Схема тестирования

Для прослушивания сервером `nginx` необходимо установить следующие параметры в конфигурационном файле `/etc/nginx/nginx.conf`:

```
location / {
  include scgi_params;
  scgi_pass localhost:3000;
}
```

Для запуска тестирования был использован следующий `bash` скрипт:

```
#!/bin/bash
```

```
2
3 for i in {1..6}; do
4 wrk -t"$i" -c100 -d10s http://localhost/highloadtesting >> ~/laba2/testing.txt
5 done
6
7 for ((i=100; i<=1000; i+=100)); do
8 wrk -t2 -c"$i" -d10s http://localhost/highloadtesting >> ~/laba2/testing.txt
9 done
```

Данный скрипт собирает информацию, выдаваемую утилитой, в отдельный файл. Тестировалась производительность сервера при изменении количества рабочих потоков от 1 до 6 и при изменении количества соединений от 100 до 1000 за 10-ти секундный интервал. В оценивании участвовали 3 величины: пропускная способность сервера в секунду (в мегабайтах), количество обработанных запросов в секунду и количество ошибок.

При изменении количества рабочих потоков и стабильном уровне соединений можно увидеть, что производительность падает в среднем на 5% (графики 2.2 и 2.3), но ошибок соединений не происходит. При тестировании количества соединений и количестве потоков равным 2, можно заметить ухудшение производительности на 10% и прирост количества ошибок после 300 соединений за 10 секунд (графики 2.4, 2.5 и 2.6).

Важно отметить, что данная реализация сервера выдерживает все нагрузки и после проведенного тестирования продолжает работать.

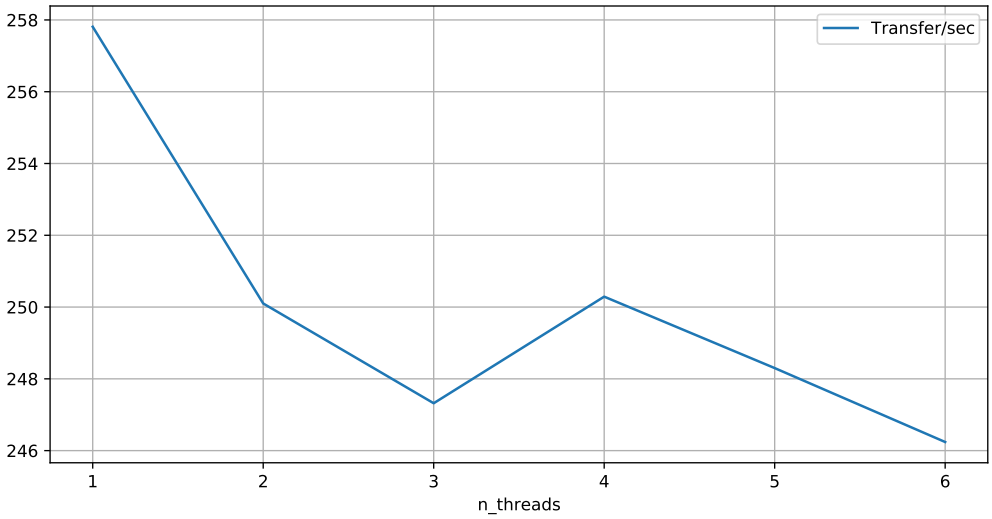

Рисунок 2.2 — Зависимость скорости от количества потоков

Точные статистики, полученные из утилиты можно посмотреть в таблицах 2.1 и 2.2 для тестирования потоков и соединений соответственно.

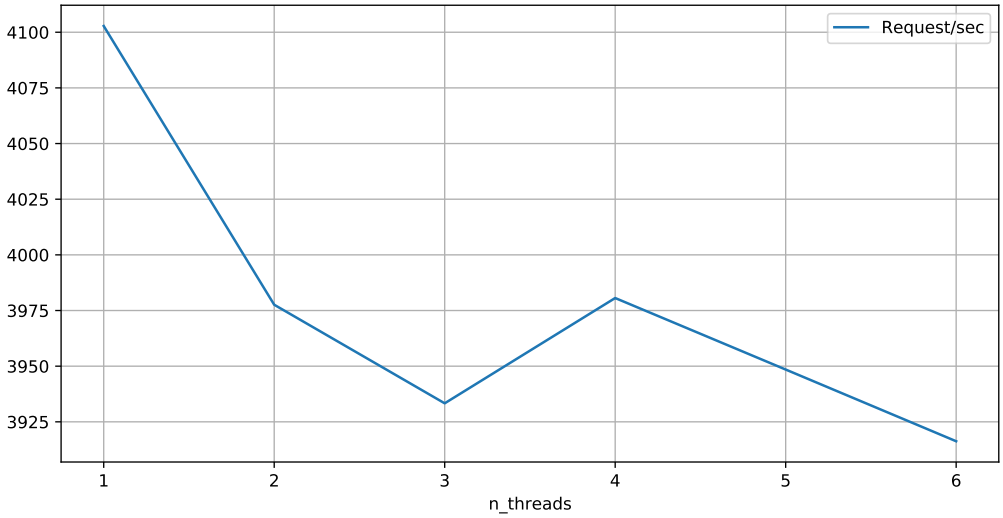

Рисунок 2.3 — Зависимость обработки запросов от количества потоков

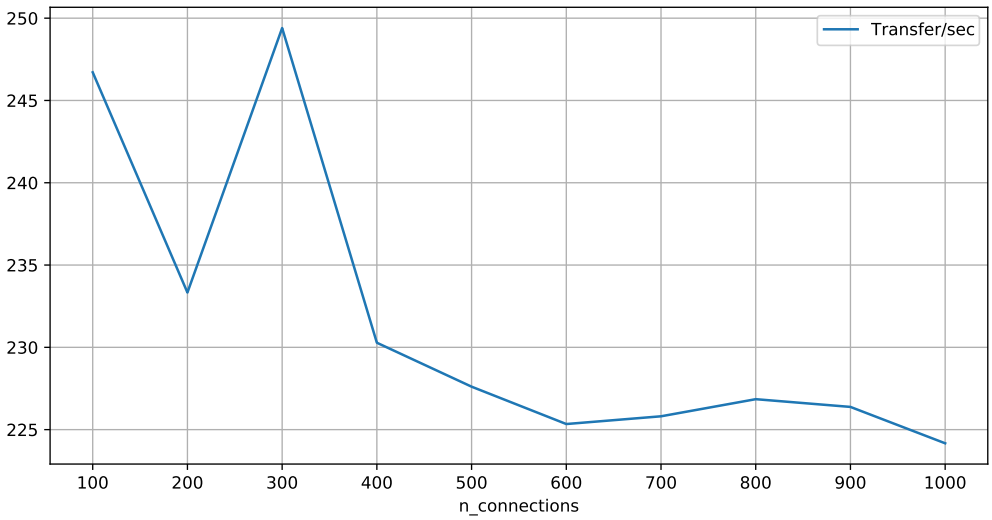

Рисунок 2.4 — Зависимость скорости от количества подключений

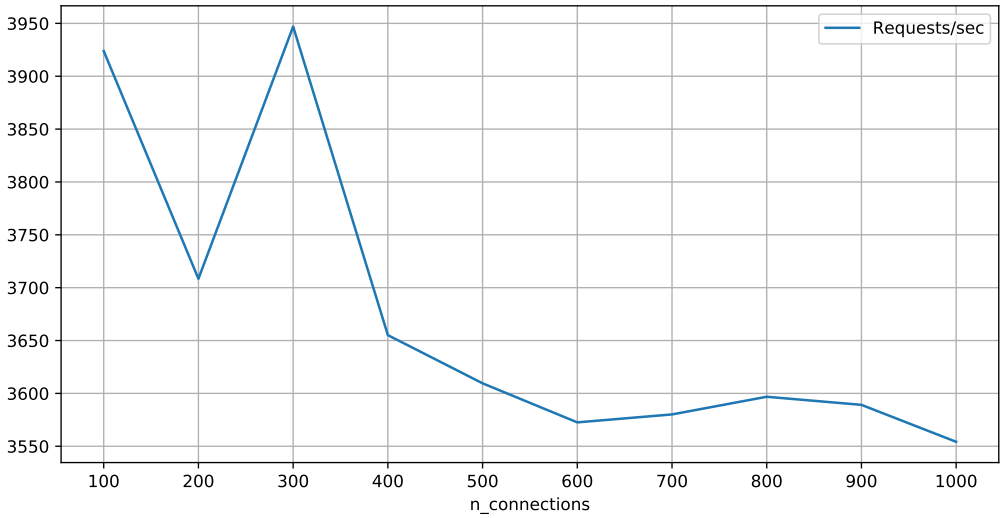

Рисунок 2.5 — Зависимость обработки запросов от количества подключений

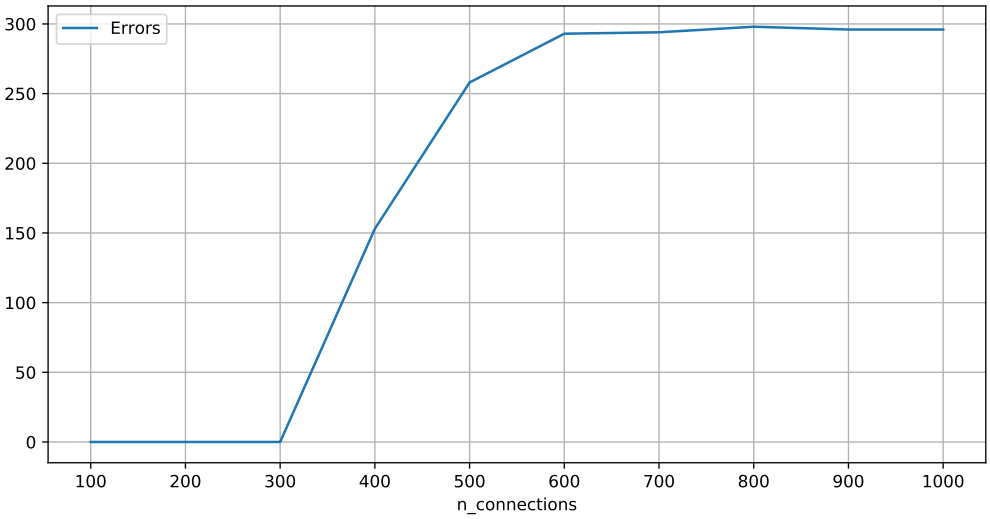

Рисунок 2.6 — Зависимость ошибок от количества подключений

n_threads	Request/sec	Transfer/sec	Errors
1	4102.8	257.81	0
2	3977.6	250.1	0
3	3933.32	247.32	0
4	3980.61	250.29	0
5	3948.49	248.3	0
6	3916.28	246.24	0

Таблица 2.1 — Талица статистики тестирования потоков

2.2 Сборка проекта

Проект сервера использует систему сборки `CMake`. Чтобы собрать проект, необходимо из папки с исходными кодами выполнить следующие команды:

```
mkdir build
cd build
cmake -D CMAKE_BUILD_TYPE=Release ..
cmake --build . --target all
```

n_connections	Requests/sec	Transfer/sec	Errors
100	3923.84	246.71	0
200	3708.44	233.33	0
300	3946.98	249.4	0
400	3655.11	230.28	153
500	3609.53	227.61	258
600	3572.59	225.34	293
700	3580.11	225.81	294
800	3596.82	226.85	298
900	3589.16	226.38	296
1000	3554.21	224.17	296

Таблица 2.2 — Талица статистики тестирования соединений