full-stack-service-networking

2015104220 컴퓨터공학과 정종윤

2021-2 풀스택서비스네트워킹 프로젝트 결과물입니다. 본문에 첨부된 영상은 리포지터리 링크에 직접 접속하면 별도 다운로드 없이 바로 재생할 수 있습니다.

리포지터리

> - https://github.com/wormwlrm/full-stack-service-networking

선택한 프로그래밍 언어

> - JavaScript(Node.js)

소스코드는 `./HTTP_1.1`, `./ZMQ` 디렉토리 내에서 확인하실 수 있습니다.

HTTP/1.1

HTTP GET & POST server example

https://user-images.githubusercontent.com/26682772/142756683-52000dd2-48ab-4b59-ac60-5de684577293.mov

```
# 서버
npm run http-1

# 클라이언트
npm run http-2
```

> - HTTP GET과 POST 메소드를 이용해 클라이언트에서 전달받은 데이터에 대한 간단한 연산을 수행하고 돌려주는 서버 프로그램을 제작했습니다.
> - Node.js의 기본 http 모듈에서는 서버에서는 메소드 처리 및 라우팅을 비교적 낮은 추상화로 제공하고 있기 때문에, 좀 더 효율적인 코드 작성을 위해 별도 라이브러리(`koa`)를 사용했습니다.
> - GET 또는 POST 로 요청이 오게 되면 상세 내용을 출력(`printHttpRequestDetail()`)하고 공통으로 사용되는 헤더(`sendHttpResponseHeader()`)를 설정합니다.
> - 연산 로직(`simple_calc()`)과 유틸성 로직(`parameter_retrieval()`)도 별도로 분리해 호출합니다.
> - 클라이언트는 비동기 로직 수행이 가능한 즉시 실행 함수(IIFE) 형태로 구현했고, 이 때 HTTP 요청을 추상화하는 라이브러리(`axios`)를 함께 사용했습니다.

json.load() example

https://user-images.githubusercontent.com/26682772/142771425-817a785e-4bc7-4a84-8a02-feaf5261697a.mov

```
npm run http-3
```

- 파일 시스템의 `json` 파일을 불러오고 조회하는 프로그램을 작성했습니다.
- Node.js의 모듈 시스템을 사용하기 위해 `require()`로 파일을 불러옵니다.
- 원본 Python 프로그램에서는 JSON 파일을 딕셔너리 형태로 변형하여 사용했지만, Node.js는 언어 특성 상 별도의 파싱 과정 없이 JSON을 그대로 사용할 수 있습니다.

json.dump() example

https://user-images.githubusercontent.com/26682772/142771435-18eb0c15-2a8b-4ef5-ac27-2c5235e5b058.mov

```
npm run http-4
```

- 메모리에 저장된 JavaScript Object 를 JSON 파일로 저장하는 프로그램을 작성했습니다.
- node.js의 파일 입출력 기능을 사용하기 위해 `fs` 모듈을 사용했습니다.
- 파일 쓰기 메소드(`writeFileSync()`) 파라미터에 JSON을 넘겨주기 위해 `JSON.stringify()` 로 직렬화 과정을 거쳤습니다.

json.dumps() example

https://user-images.githubusercontent.com/26682772/142771440-4a05fa75-e3a4-4205-83ce-debb8d91130b.mov

```
npm run http-5
```

- 메모리에 저장된 JavaScript Object 를 JSON 형태 문자열로 직렬화하는 프로그램을 작성했습니다.
- `JSON.stringify()` 메소드를 사용하여 직렬화 과정을 거쳤습니다.

json.loads() example

https://user-images.githubusercontent.com/26682772/142771456-8ec372b8-588a-4a6d-8194-7c942233bfdf.mov

```
npm run http-6
```

- 직렬화된 JSON 형태의 데이터를 JavaScript Object 형태로 불러오는 프로그램을 작성했습니다.
- `JSON.parse()` 메소드를 사용하여 불러오기 과정을 거쳤습니다.

RESTful Server example

https://user-images.githubusercontent.com/26682772/142771466-477aaa89-7fa9-47eb-a62b-2308b841d548.mov

```
# 서버
npm run http-7

# 클라이언트
npm run http-8
```

- 멤버십에 대한 CRUD 기능을 제공하는 RESTful API를 구현하는 프로그램을 작성했습니다.
- `MembershipHandler` 클래스에서 CRUD 요청에 따라 데이터베이스에 접근하는 메소드를 미리 구현했고, `koa` 라우터에서 해당 핸들러를 호출하는 방식으로 구현했습니다.
- 클라이언트는 비동기 로직 수행이 가능한 즉시 실행 함수(IIFE) 형태로 구현했고, 이 때 HTTP 요청을 추상화하는 라이브러리(`axios`)를 함께 사용했습니다.

ZMQ

Request-Reply Pattern

https://user-images.githubusercontent.com/26682772/143219295-51511ebe-b594-4e31-a7f5-b85ef4d854f9.mov

```
# 서버
npm run zmq-1

# 클라이언트
npm run zmq-2
```

- ZMQ의 Request-Reply 패턴을 구현한 프로그램을 작성했습니다.
- ZMQ 구현 예제에서 비동기 이터레이션 핸들러(`for await of`) 구현 방법을 참조하였습니다.
- Node.js에서는 `sleep` 함수를 제공하지 않기 때문에 `Promise`와 `setTimeout`을 이용해 해당 메소드를 직접 구현했습니다.
- ZMQ에서 `receive` 로 받은 결과물이 `Buffer` 이기 때문에, 사람이 읽기 위해서는 `.toString()` 메소드를 사용하여 문자열로 변환하는 과정을 거쳐야 합니다.
- 서버 입장에서는 소켓에 `bind`, 클라이언트 입장에서는 소켓에 `connect` 하는 방식으로 구현해야 합니다. 두 방식 모두 비동기로 연결하지만, `bind` 는 `Promise` 를 리턴하기 때문에 `await` 문법과 함께 쓰면 소켓 연결 여부를 보장할 수 있다는 장점이 있습니다.

Publish-Subscribe Pattern

https://user-images.githubusercontent.com/26682772/143219370-28213242-0c3c-47ed-9520-08ab82cc91a0.mov

```
# 서버
npm run zmq-3

# 클라이언트
npm run zmq-4
```

- 서버가 `Publisher`, 클라이언트가 `Subscriber`인 Pub-Sub 패턴을 구현한 프로그램을 작성했습니다.
- 서버는 비동기 이터레이션 핸들러 대신 `while (true)` 반복문을 이용해 작성해보았습니다.
- 서버는 데이터를 무작위 난수로 생성하여 발행(publish)하고, 클라이언트는 특정 관심사에 대해서만 구독(publish)합니다. 난수 범위가 너무 커서, 예제 코드보다는 범위를 좁게 조정한 부분이 있습니다.
- 위에서 언급한대로 `Publisher`는 소켓에 `bind` 하고, `Subscriber`는 소켓에 `connect` 하는 방식으로 구현했습니다.
- 관심사 구독은 `subscribe` 메소드를 사용했고, 파라미터에 전달된 값이 prefix로 오는 것만 수신합니다.

Publish-Subscribe with Pipeline Pattern

v1

https://user-images.githubusercontent.com/26682772/143219426-5eedc705-62f0-4b62-b694-960a57dd085d.mov

```
# 서버
npm run zmq-5

# 클라이언트
npm run zmq-6
```

- Pub-Sub 패턴과 Push-Pull 패턴을 혼용한 프로그램을 구현했습니다.
- Node.js는 쓰레드 기반이 아닌 이벤트 루프 기반이기 때문에, 클라이언트에서 `setInterval()` 메소드를 이용하는 방식으로 비동기 Push를 수행합니다. 서버 역시 응답을 폴링하는 방식으로 듣기보다는 `Promise`를 resolve 하는 방식으로 구현합니다.
- 서버에서는 Push에 대한 `receive`가 resolve 되면 Publish를 수행합니다.
- 클라이언트에서도 반복문 내에서 `receive`의 resolve 상태에 따라 메시지를 출력합니다.
- `subscribe()`를 빈 파라미터로 호출해야 모든 관심사에 대한 정보를 구독합니다.

v2

https://user-images.githubusercontent.com/26682772/143219492-038e8502-911c-4858-b6bb-846935da30bd.mov

```
# 서버
npm run zmq-7

# 클라이언트
npm run zmq-8 [클라이언트 이름]
```

- 위 프로그램에 대한 개선 버전입니다.

Dealer-Router Pattern

Single Thread

https://user-images.githubusercontent.com/26682772/143219543-f6924102-3249-4579-b8aa-95a0c6138309.mov

> - Dealer-Router 패턴으로 구현한 싱글 쓰레드 워커 프로그램입니다.
> - Node.js 특성을 이용해 쓰레드 기반이 아닌 이벤트 루프 기반으로 구현했습니다.
> - 서버에서는 Dealer와 Router를 만든 후 Proxy로 연결합니다.
> - 서버에서는 `argv`로 입력받는 파라미터에 따라 워커의 갯수를 조정합니다.

Four Threads

https://user-images.githubusercontent.com/26682772/143219600-9bc9b56e-578c-4d27-b361-45f99d6917fb.mov

> - 위 프로그램에서 4개의 워커를 사용하는 경우입니다.

```
# 서버
npm run zmq-9

# 클라이언트
npm run zmq-10 [클라이언트 이름]
# or
npm run zmq-11 [클라이언트 이름]
```

Dirty P2P Example

https://user-images.githubusercontent.com/26682772/143219668-129e84d6-8efb-467c-a744-6b77204fae0a.mov

```
npm run zmq-12 [클라이언트 이름]
```

> - ZMQ를 이용해 직접 P2P를 구현한 프로그램입니다.
> - 네트워크 관련 정보를 얻기 위해서 별도 라이브러리를 활용했습니다.
> - 클라이언트에서 처음 실행되는 `search_nameserver()`에서는 이미 서버가 등록되어 있는지 찾는 과정에서 무한한 시간동안 찾지 않도록 타임아웃을 두게 처리하였습니다.
> - `beacon_nameserver()`에서는 서버가 켜져있다면 시그널을 계속 보내도록 `setInterval()` 를 이용해 구현했습니다.
> - `user_manager_nameserver()` 는 서버에서 사용자를 추가하거나 삭제하는 것을 처리합니다.
> - `relay_server_nameserver()` 는 Push-Pull 패턴을 이용해 사용자가 보낸 정보를 다른 사용자에게 Publish 하는 것을 처리합니다.
> - 그 후 각 클라이언트는 Subscriber이면서 Push가 가능하도록 구현합니다.
> - 위 영상에서 클라이언트 수신 화살표 방향이 잘못 보여지고 있는데 이는 현재 수정한 상태입니다.

결론

HTTP/1.1 파트와 ZMQ 파트를 Node.js를 이용해 구현하는 프로젝트를 마무리했습니다. 예제 코드가 있어서 처음에는 오래 걸리지 않을 것이라 생각했는데, 생각보다는 시간이 많이 걸렸습니다. 대략 하루에 3시간 정도 작업했을 때 4일 정도 걸렸습니

다.

HTTP/1.1 파트는 웹 개발 경험을 바탕으로 어렵지 않게 구현할 수 있었습니다. JSON 파트의 경우에는 Node.js의 덕을 많이 봤는데, 애초에 JSON이 JavaScript 기반의 데이터 표현 방식이기 때문입니다. 그래서 언어의 특성에 따른 간편함도 느낄 수 있었습니다. 다만 RESTful API를 지원하는 프레임워크 자체에 대한 이해가 부족한 부분이 있어서, 이 부분은 모르는 부분을 찾아 검색해가며 구현했습니다.

ZMQ는 사실 여태껏 써본 적 없는 기술이었기 때문에 초반에는 어려움을 겪었습니다. API 자체에 대한 이해도가 낮기도 했고, 예제 코드도 오래된 것이 많았기 때문입니다. 가장 어려웠던 부분은 언어 별 특성에 맞추어 코드를 전환하는 과정이었습니다. 예를 들자면 Python의 경우에는 동기적으로 코드가 동작하고, 멀티 쓰레드 기반의 프로그래밍을 지원하는 것에 반해 Node.js는 비동기 처리를 Promise 문법으로 변환해야 하고, 쓰레드 대신 이벤트 루프를 활용하게 로직을 수정하는 부분이 있을 것 같습니다. ZMQ에서 자주 사용하는 패턴을 라이브러리에서 제공해주는 것은 좋지만, 이를 100% 활용하기 위해서는 언어 자체에 대한 높은 이해도가 필수적이라는 생각이 들었습니다.