Imprimer en PDF

WEBSERV

Sommaire

1. Concepts Fondamentaux HTTP

Qu'est-ce que HTTP ?

HTTP (HyperText Transfer Protocol) est un protocole de communication client-serveur qui fonctionne sur TCP/IP.

Caractéristiques principales :

- **Sans état (stateless)** : chaque requête est indépendante
- **Basé sur du texte** : les messages sont lisibles
- **Port par défaut** : 80 (HTTP) ou 443 (HTTPS)
- **Méthodes principales** : GET, POST, DELETE

Structure d'une Requête HTTP

```
GET /index.html HTTP/1.1 Host: www.example.com User-Agent: Mozilla/5.0 Accept: text/html Connection: keep-alive [Corps de la requête si POST]
```

Composants :

1. **Request Line** : Méthode + URI + Version HTTP
2. **Headers** : Métadonnées (Host, User-Agent, etc.)
3. **Body** : Données (optionnel, pour POST)

Structure d'une Réponse HTTP

```
HTTP/1.1 200 OK Content-Type: text/html Content-Length: 1234 Connection: keep-alive <html>...</html>
```

Composants :

1. **Status Line** : Version + Code statut + Message
2. **Headers** : Content-Type, Content-Length, etc.
3. **Body** : Contenu de la réponse

Codes de Statut HTTP

Codes Importants à Connaître

Code	Signification	Utilisation
200	OK	Requête réussie
201	Created	Ressource créée (POST)
204	No Content	Succès sans contenu (DELETE)
301	Moved Permanently	Redirection permanente
400	Bad Request	Requête malformée
403	Forbidden	Accès interdit
404	Not Found	Ressource introuvable
405	Method Not Allowed	Méthode non autorisée
413	Payload Too Large	Body trop grand
500	Internal Server Error	Erreur serveur
505	HTTP Version Not Supported	Version HTTP non supportée

⚠ Important pour la Défense

Vous devez pouvoir expliquer :

- Pourquoi vous retournez tel ou tel code
- La différence entre 301 et 302 (redirection permanente vs temporaire)
- Quand utiliser 204 vs 200
- Comment gérer les erreurs 4xx vs 5xx

2. Architecture du Serveur

Vue d'Ensemble

Le serveur webserv suit une architecture non-bloquante avec I/O multiplexing.

Composants Principaux :

1. **Server** : Gère les sockets d'écoute
2. **Client** : Représente une connexion client
3. **Request Parser** : Parse les requêtes HTTP
4. **Response Builder** : Construit les réponses
5. **Config Parser** : Lit le fichier de configuration
6. **CGI Handler** : Exécute les scripts CGI

Flux de Traitement

1. Configuration du serveur ↓ 2. Création des sockets d'écoute ↓ 3. Boucle epoll/select/poll ↓ 4. Acceptation de nouvelles connexions ↓ 5. Lecture des requêtes (non-bloquant) ↓ 6. Parsing de la requête ↓ 7. Traitement (fichier statique ou CGI) ↓ 8. Construction de la réponse ↓ 9. Envoi de la réponse (non-bloquant) ↓ 10. Fermeture ou keep-alive

Sockets et Binding

```
// Création d'un socket int socket_fd = socket(AF_INET, SOCK_STREAM, 0); // Configuration d'adresse sockaddr_in addr; addr.sin_family = AF_INET; addr.sin_port = htons(8080); addr.sin_addr.s_addr = INADDR_ANY; // Binding bind(socket_fd, (struct sockaddr*)&addr, sizeof(addr)); // Listen listen(socket_fd, SOMAXCONN); // Accept (dans la boucle) int client_fd = accept(socket_fd, NULL, NULL);
```

3. ⚡ I/O Multiplexing (epoll/select/poll)

Pourquoi le Multiplexing ?

Permet de gérer **plusieurs connexions simultanées** avec un seul thread, sans bloquer.

Problème sans Multiplexing :

- Un thread par client = inefficace et coûteux
- Opérations bloquantes = serveur gelé

Solution avec Multiplexing :

- Un seul thread surveille tous les sockets
- Traite uniquement les sockets prêts
- Non-bloquant et scalable

Comparaison : select vs poll vs epoll

Fonctionnalité	select	poll	epoll
Limite FD	1024 (FD_SETSIZE)	Illimité	Illimité
Performance	O(n)	O(n)	O(1)
Portabilité	Maximum	POSIX	Linux only
Edge-triggered	Non	Non	Oui

Exemple avec epoll

```
// Création de l'instance epoll int epoll_fd = epoll_create1(0); // Ajout d'un socket à surveiller struct epoll_event event; event.events = EPOLLIN; // Lecture disponible event.data.fd = socket_fd; epoll_ctl(epoll_fd, EPOLL_CTL_ADD, socket_fd, &event); // Boucle principale struct epoll_event events[MAX_EVENTS]; while (true) { int n = epoll_wait(epoll_fd, events, MAX_EVENTS, -1); for (int i = 0; i < n; i++) { if (events[i].events & EPOLLIN) { // Données à lire handleRead(events[i].data.fd); } if (events[i].events & EPOLLOUT) { // Prêt à écrire handleWrite(events[i].data.fd); } } }
```

Exemple avec select

```
fd_set read_fds, write_fds; int max_fd = 0; while (true) { FD_ZERO(&read_fds); FD_ZERO(&write_fds); // Ajouter tous les sockets FD_SET(server_fd, &read_fds); max_fd = server_fd; for (auto& client : clients) { FD_SET(client.fd, &read_fds); if (client.has_data_to_send) FD_SET(client.fd, &write_fds); if (client.fd > max_fd) max_fd = client.fd; } // Attendre des événements select(max_fd + 1, &read_fds, &write_fds, NULL, NULL); // Traiter les événements if (FD_ISSET(server_fd, &read_fds)) { // Nouvelle connexion acceptClient(); } for (auto& client : clients) { if (FD_ISSET(client.fd, &read_fds)) { readFromClient(client); } if (FD_ISSET(client.fd, &write_fds)) { writeToClient(client); } } }
```

Pièges Communs

- **Oublier fcntl()** : Les sockets doivent être en mode non-bloquant
- **select() modifie les fd_set** : Il faut les reconstruire à chaque itération
- **EAGAIN/EWOULDBLOCK** : Normal en non-bloquant, ne pas considérer comme erreur
- **Broken pipe** : Le client se déconnecte, gérer proprement

4. Parsing et Configuration

Fichier de Configuration

Similaire à nginx, avec des directives en blocs.

```
server { listen 8080; server_name localhost; client_max_body_size 10M; location / { root /var/www; index index.html; autoindex on; allow_methods GET POST; } location /upload { allow_methods POST DELETE; upload_path /var/www/uploads; } location /cgi-bin { cgi_pass .py /usr/bin/python3; cgi_pass .php /usr/bin/php-cgi; } error_page 404 /404.html; }
```

Directives Importantes

Directive	Description
listen	Port d'écoute du serveur
server_name	Nom du serveur (virtual host)
root	Répertoire racine des fichiers
index	Fichier par défaut
allow_methods	Méthodes HTTP autorisées
client_max_body_size	Taille max du body
autoindex	Listing de répertoire
error_page	Pages d'erreur personnalisées
return	Redirection

5. Gestion des Requêtes HTTP

Étapes du Parsing

1. **Lecture incrémentale** : recv() en boucle jusqu'à avoir la requête complète
2. **Parse Request Line** : Extraire méthode, URI, version
3. **Parse Headers** : Ligne par ligne jusqu'à \r\n\r\n
4. **Parse Body** : Si Content-Length présent (POST)
5. **Validation** : Vérifier conformité HTTP/1.1

Gestion du Body (POST)

Content-Length

Indique la taille exacte du body en octets.

```
Content-Length: 1234 [1234 octets de données]
```

- Lire exactement Content-Length octets
- Si reçu plus → erreur 400
- Si reçu moins → attendre plus de données
- Comparer à client_max_body_size → 413 si dépassé

Transfer-Encoding: chunked

Le body est envoyé par morceaux (chunks).

```
Transfer-Encoding: chunked 5\r\n Hello\r\n 6\r\n World!\r\n 0\r\n \r\n
```

Format : `taille_hexa\r\n données\r\n`
Fin : `0\r\n\r\n`

Méthodes HTTP

GET

- Récupérer une ressource (fichier, listing)

- Pas de body dans la requête
- Retourner 200 + contenu ou 404

POST

- Envoyer des données (upload, formulaire)
- Body dans la requête
- Retourner 201 (Created) ou 200

DELETE

- Supprimer une ressource
- Vérifier les permissions
- Retourner 204 (No Content) ou 200

6. CGI (Common Gateway Interface)

Qu'est-ce que CGI ?

CGI permet au serveur web d'exécuter des scripts externes (Python, PHP, etc.) pour générer du contenu dynamique.

Principe :

1. Le serveur détecte une requête pour un script CGI
2. Il lance le script avec `fork()` + `execve()`
3. Passe les infos via variables d'environnement
4. Envoie le body via stdin du script
5. Récupère la sortie via stdout du script
6. Retourne la sortie au client

Variables d'Environnement CGI

Variable	Description	Exemple
REQUEST_METHOD	Méthode HTTP	GET, POST
QUERY_STRING	Paramètres URL	name=John&age=30
CONTENT_LENGTH	Taille du body	1234
CONTENT_TYPE	Type du body	application/x-www-form-urlencoded
PATH_INFO	Chemin après script	/extra/path
SCRIPT_NAME	Nom du script	/cgi-bin/script.py
SERVER_PROTOCOL	Version HTTP	HTTP/1.1
HTTP_*	Headers HTTP	HTTP_USER_AGENT

Implémentation CGI

```
// 1. Créer des pipes int pipe_in[2], pipe_out[2]; pipe(pipe_in); pipe(pipe_out); // 2. Fork pid_t pid = fork(); if (pid == 0) { // Processus enfant // Rediriger stdin/stdout dup2(pipe_in[0], STDIN_FILENO); dup2(pipe_out[1], STDOUT_FILENO); // Fermer les descripteurs inutiles close(pipe_in[0]); close(pipe_in[1]); close(pipe_out[0]); close(pipe_out[1]); // Configurer l'environnement char *env[] = { "REQUEST_METHOD=POST", "CONTENT_LENGTH=100", NULL }; // Exécuter le script char *argv[] = {"/usr/bin/python3", "script.py", NULL}; execve(argv[0], argv, env); exit(1); // Si execve échoue } // 3. Processus parent close(pipe_in[0]); close(pipe_out[1]); // Envoyer le body au script (via pipe_in[1]) write(pipe_in[1], body.c_str(), body.size()); close(pipe_in[1]); // Lire la sortie du script (via pipe_out[0]) char buffer[4096]; std::string output; ssize_t n; while ((n = read(pipe_out[0], buffer, sizeof(buffer))) > 0) { output.append(buffer, n); } close(pipe_out[0]); // Attendre la fin du script waitpid(pid, &status, 0);
```

⚠ Timeout CGI

Un script CGI peut prendre trop de temps. Il faut implémenter un timeout :

- Utiliser `alarm()` ou un timer
- Si timeout, `kill(pid, SIGKILL)`
- Retourner 500 (Internal Server Error) au client

Format de Sortie CGI

Le script CGI doit retourner des headers HTTP + body :

```
Content-Type: text/html <html> <body>Hello from CGI!</body> </html>
```

Le serveur doit parser cette sortie et construire une réponse HTTP complète.

7. Questions Fréquentes de Défense

Architecture & Design

✓

Expliquez le flux d'une requête dans votre serveur
Configuration → Socket listen → epoll/select → Accept → Read → Parse → Process → Build response → Write → Close/Keep-alive

✓

Pourquoi utilisez-vous epoll/select/poll ?
Pour gérer plusieurs connexions simultanées de manière non-bloquante avec un seul thread, ce qui est efficace et scalable.

✓

Quelle est la différence entre epoll et select ?
epoll est O(1) et sans limite de FD, select est O(n) et limité à 1024 FD. epoll supporte edge-triggered, select non.

✓

Comment gérez-vous les connexions keep-alive ?
On garde la connexion ouverte après la réponse si le header Connection: keep-alive est présent, et on attend une nouvelle requête sur le même socket.

HTTP & Parsing

✓

Comment parsez-vous une requête HTTP ?
1. Request line (méthode, URI, version) → 2. Headers ligne par ligne → 3. Body si Content-Length présent → 4. Validation

✓

Comment savez-vous qu'une requête est complète ?
Pour les headers : double CRLF (\r\n\r\n). Pour le body : Content-Length octets reçus, ou dernier chunk (0\r\n\r\n) en chunked.

✓

Comment gérez-vous les requêtes malformées ?
Validation stricte du parsing : méthode invalide, URI malformée, headers incorrects → retourner 400 Bad Request.

✓

Quelle est la différence entre 301 et 302 ?
301 = redirection permanente (le client peut mettre en cache). 302 = redirection temporaire (le client ne doit pas mettre en cache).

CGI

✓

Expliquez le fonctionnement de CGI
Le serveur fork() un processus enfant, configure l'environnement, redirige stdin/stdout via pipes, exécute le script avec execve(), lit la sortie et retourne au client.

✓

Comment passez-vous les données au script CGI ?
Variables d'environnement (REQUEST_METHOD, QUERY_STRING, etc.) pour les métadonnées. Body via stdin pour les données POST.

✓

Comment gérez-vous un script CGI qui ne se termine pas ?
Timeout avec alarm() ou timer. Si dépassé, kill(pid, SIGKILL) et retourner 500 au client.

✓

Pourquoi fork() et pas system() ?
fork() + execve() permet un contrôle total : pipes, environnement, timeout. system() est bloquant et moins sécurisé.

Configuration & Features

✓

Comment gérez-vous plusieurs serveurs virtuels ?
Via le header Host dans la requête. On compare Host avec server_name dans la config pour router vers le bon serveur.

✓

Comment fonctionne l'autoindex ?
Si aucun fichier index n'existe et autoindex est on, on liste le contenu du répertoire et génère une page HTML avec les liens.

✓

Comment gérez-vous les uploads (POST) ?
Parser le body (multipart/form-data ou application/octet-stream), extraire le fichier, sauvegarder dans upload_path avec vérification de taille (client_max_body_size).

✓

Comment gérez-vous DELETE ?
Vérifier que la méthode est autorisée, vérifier les permissions du fichier, utiliser unlink() pour supprimer, retourner 204 ou 200.

Sécurité & Gestion d'Erreurs

✓

Comment empêchez-vous le directory traversal ?

Valider l'URI, résoudre le chemin canonique, vérifier qu'il reste dans le root autorisé. Bloquer ../ et chemins absolus.

✓

Comment gérez-vous les erreurs système (EAGAIN, EPIPE) ?
EAGAIN/EWOULDBLOCK : normal en non-bloquant, réessayer plus tard. EPIPE : client déconnecté, fermer proprement la connexion.

✓

Que faire si recv() retourne 0 ?
Le client a fermé la connexion. Nettoyer les ressources et fermer le socket côté serveur.

8. Checklist de Vérification

Fonctionnalités Obligatoires

- ✓ HTTP/1.1 compliant (méthodes GET, POST, DELETE)
- ✓ Configuration via fichier (ports, routes, methods, etc.)
- ✓ Serveurs virtuels (multi-host via server_name)
- ✓ Pages d'erreur par défaut et personnalisables
- ✓ Limitation de taille du body (client_max_body_size)
- ✓ Routes configurables (location blocks)
- ✓ Support des fichiers statiques
- ✓ Méthodes HTTP autorisées par route
- ✓ Redirections HTTP (301, 302)
- ✓ Directory listing (autoindex on/off)
- ✓ Fichier index par défaut
- ✓ Upload de fichiers (POST)
- ✓ Exécution CGI (scripts externes)
- ✓ Fonctionnement avec navigateur
- ✓ Non-bloquant (epoll/select/poll)
- ✓ I/O multiplexing efficace
- ✓ Gestion propre des erreurs
- ✓ Pas de leaks mémoire
- ✓ Résilience (pas de crash)

Tests à Effectuer Avant la Défense

Tests Basiques :

- Accéder à une page HTML statique
- 404 sur fichier inexistant
- Directory listing avec autoindex
- Redirection fonctionnelle
- Page d'erreur personnalisée

Tests POST :

- Upload d'un fichier (formulaire HTML)
- Body trop grand → 413
- Fichier uploadé accessible après

Tests DELETE :

- Supprimer un fichier existant → 204/200
- Supprimer un fichier inexistant → 404
- DELETE non autorisé → 405

Tests CGI :

- Script Python/PHP simple
- GET avec query string
- POST avec body au script
- Timeout CGI géré

Tests Stress :

- Siege : `siege -c 100 -t 30s http://localhost:8080/`
- Multiple connexions simultanées
- Pas de crash, pas de leak

Conseils pour la Soutenance

Préparation

- **Connaître votre code** : Soyez capable d'expliquer chaque partie
- **Tester avant** : Vérifiez que tout fonctionne, y compris les edge cases
- **Préparer des exemples** : Ayez des fichiers de test (HTML, scripts CGI, config)
- **Vérifier les leaks** : Valgrind doit être clean
- **Maîtriser votre config** : Savoir expliquer chaque directive

Pendant la Défense

- **Restez calme** : Si vous ne savez pas, admettez-le honnêtement
- **Expliquez votre raisonnement** : Pourquoi ces choix de design ?
- **Montrez votre compréhension** : Concepts HTTP, I/O multiplexing, CGI
- **Soyez précis** : Évitez les généralités, donnez des détails techniques
- **Défendez vos choix** : epoll vs select ? Pourquoi ?

Points Clés à Maîtriser

1. **Le protocole HTTP** : Structure, codes, méthodes
2. **L'I/O multiplexing** : Pourquoi, comment, différences
3. **Le parsing** : Comment détecter une requête complète
4. **CGI** : Fork, pipes, environnement, timeout
5. **La configuration** : Directives, virtual hosts, locations

⚠ Pièges à Éviter

- Ne pas savoir expliquer pourquoi vous avez choisi epoll/select/poll
- Confondre les codes HTTP (301 vs 302, 400 vs 500)
- Ne pas gérer correctement le mode non-bloquant (EAGAIN)
- Oublier le timeout CGI
- Leaks mémoire sur les cas d'erreur
- Directory traversal non sécurisé

Ressources Utiles

Documentation Officielle

- **RFC 2616** : HTTP/1.1 (remplacée par RFC 7230-7235)
- **RFC 3875** : CGI specification
- **man pages** : socket, bind, listen, accept, epoll, select, fork, execve

Outils de Test

- **curl** : Tester les requêtes HTTP depuis la ligne de commande
- **Postman** : Interface graphique pour tester les API
- **siege** : Load testing et benchmark
- **netcat (nc)** : Envoyer des requêtes brutes
- **telnet** : Connexion manuelle au serveur
- **valgrind** : Détection de leaks mémoire

Commandes Utiles

```
# Tester avec curl curl -X GET http://localhost:8080/ curl -X POST -d "data" http://localhost:8080/upload curl -X DELETE http://localhost:8080/file.txt # Envoyer une requête brute echo -e "GET / HTTP/1.1\r\nHost: localhost\r\n\r\n" | nc localhost 8080 # Load testing siege -c 100 -t 30s http://localhost:8080/ # Vérifier les leaks valgrind --leak-check=full ./webserv config.conf # Monitorer les connexions netstat -an | grep 8080 lsof -i :8080
```

Bonne Chance pour la Défense !

Vous avez tout ce qu'il faut pour réussir !