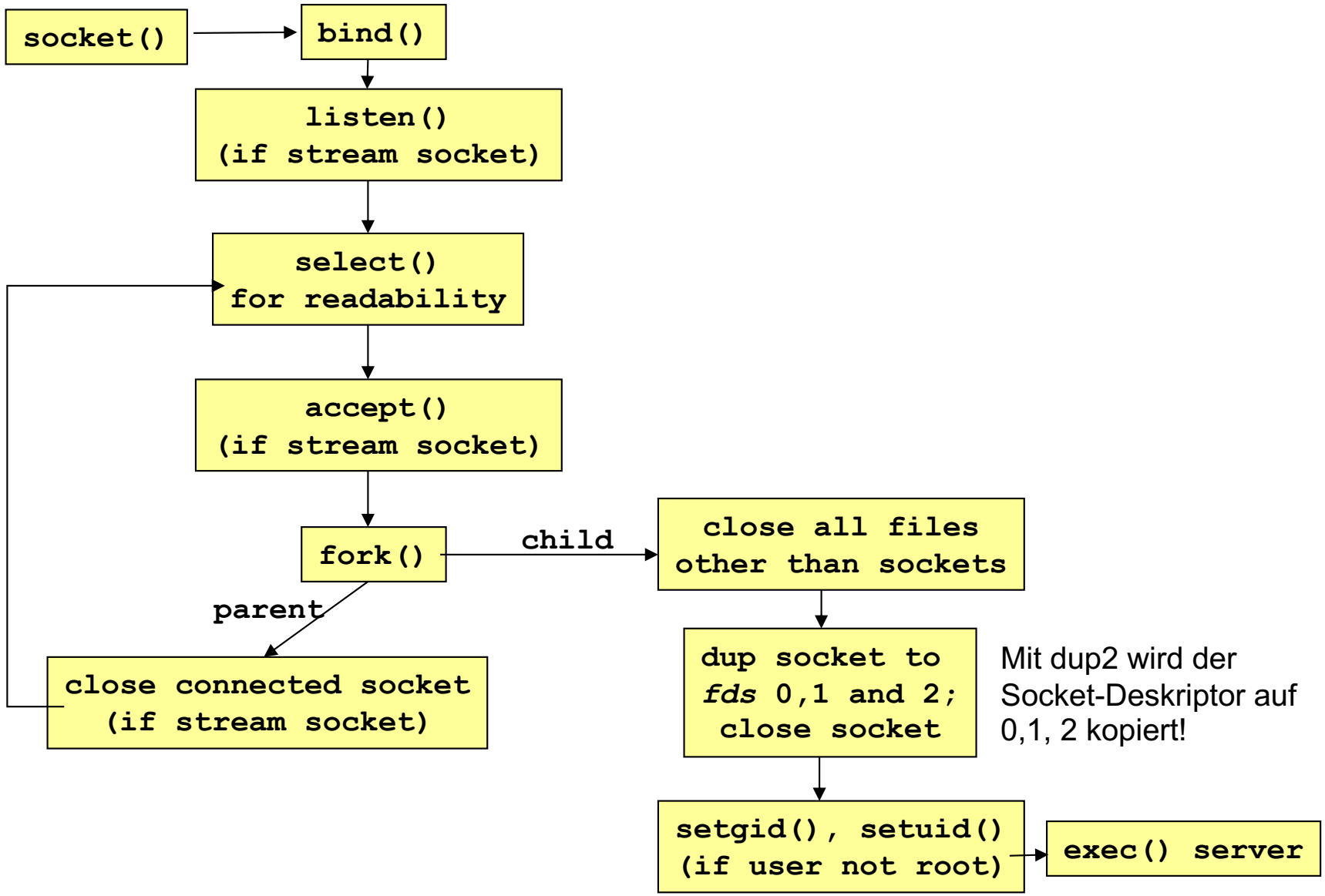
Webserver

Apache, nginx, node.js

Server
inetd: klassische Master-Server-Implementierung



Mit dup2 wird der Socket-Deskriptor auf 0,1, 2 kopiert!

Server
inetd

Ablauf:

- Master Server wartet auf Anforderungen

- Anforderung erhalten
  - > Master Server akzeptiert Verbindung
  - > Master Server erzeugt Child Prozess und übergibt Verbindung
  - > Child Prozess behandelt Anforderung

Ergebnis

- Pro Anforderung wird ein (Child) Server Prozess erzeugt
- Jeder Server muss seine Initialisierung selbst vornehmen

Hinweis

- Heutige Linux-Systeme verwenden den generelleren systemd-Prozess für diese Aufgabe

Weitere Informationen: http://www.fmc-modeling.org/category/projects/apache/amp/4_3Multitasking_server.html

Diskussion

- Diese Multiprocessing-Architektur ist sinnvoll, wenn die individuellen Aufgaben langfristiger Natur sind und der Client einen Status der Verbindung verwaltet

- HTTP ist jedoch zustandslos, wodurch der Client nicht über einen solchen Status verfügt

- Eine Server-Implementierung auf der Basis dieser Architektur wäre somit wenig effektiv
  > Der Master-Server müsste für jede HTTP-Verbindung einen neuen Prozess kreieren, der nur diese eine Verbindung bedient. Aufwand!
  > Das Erzeugen eines Kindprozesses blockiert die Verarbeitung des Master-Servers. Dieser kann währenddessen keine eingehenden Anfragen bearbeiten.

Server
Multi Processing Modules (MPM)

Wie werden Server-Anfragen beim Apache abgearbeitet?

- **M**ulti **P**rocessing **M**odules bestimmen wie es funktioniert
- Apache kommt standardmäßig mit zwei Modellen:

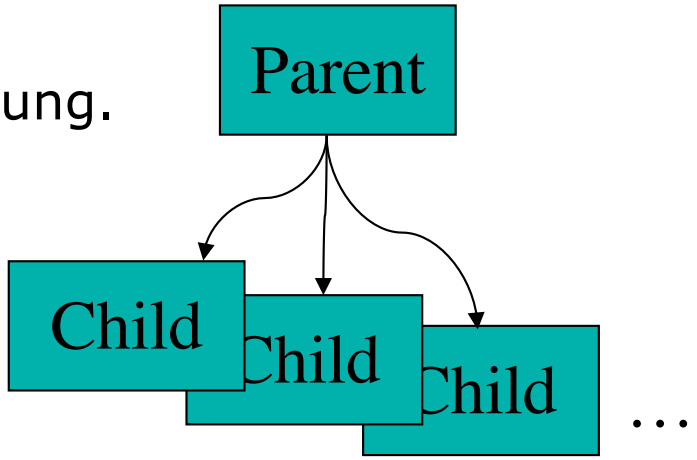
> **Prefork**
  - Pro Anfrage ein Prozess (ähnlich inetd)
  - Saubere Trennung der Anfragen
  - Nachteil:
    - RAM-intensiv

> **Worker**
  - Pro Anfrage ein Thread
  - Nachteile:
    - Verarbeitende Library muss multithreading-fähig sein
    - Außer Kontrolle geratener Thread hat eine Beendigung der kompletten Prozesses zur Folge

Server
Prefork-Modell

Grundsätzliche Idee

- Forking findet vor der eigentlichen Anfrage statt („Pre-Forking“)
- Prozesse „warten“ darauf genutzt zu werden
- Jeder Prozess behandelt jeweils eine Verbindung.
  - > Hoher Speicherbedarf
  - > “You’ll run out of memory before CPU”



Leader-Followers-Pattern

- Die Preforking-Architektur basiert auf einem Vorrat von Prozessen, die drei verschiedene Rollen haben:
  - > Listener: Warten auf Anfragen (Leader)
  - > Worker: Verarbeiten von Anfragen (untätige Worker = Follower)
  - > Idle-Worker: Einreihen in die Warteschlange um darauf zu warten, die Rolle des Listeners zu übernehmen

Weitere Informationen: https://httpd.apache.org/docs/2.4/de/mod/prefork.html

Server
Prefork-Modell

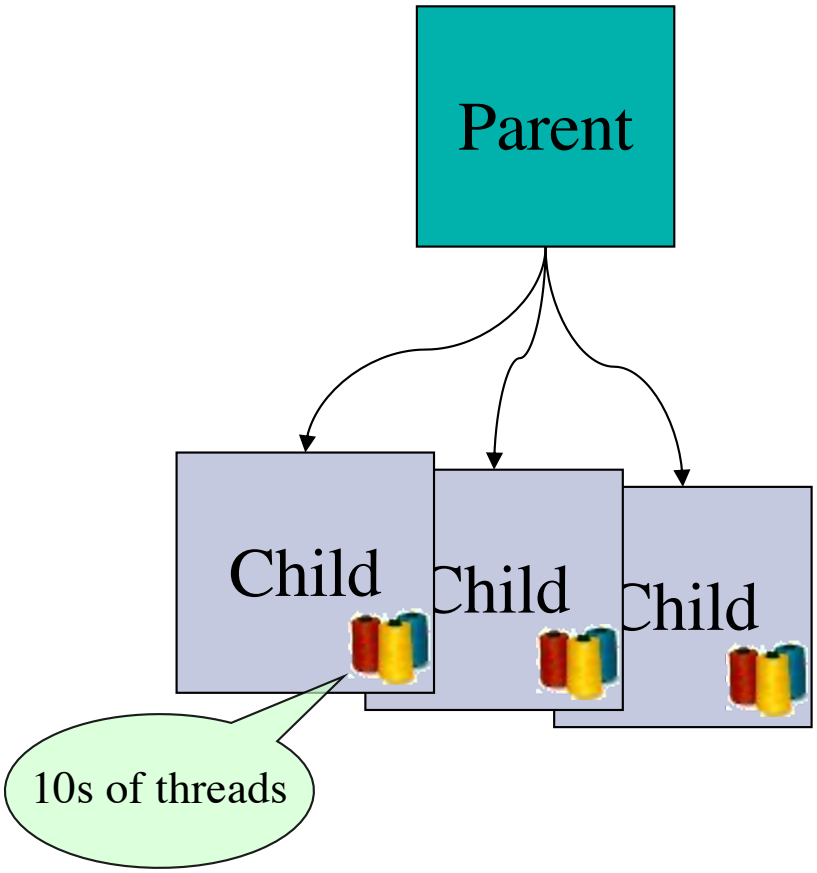
Direktiven

- StartServers
  > Anzahl der zu startenden Serverprozesse
- MinSpareServers
  > Mindestzahl der Serverprozesse, die ohne Last bleiben dürfen
- MaxSpareServers
  > Höchstzahl der Serverprozesse, die ohne Last bleiben können
- MaxClients
  > Höchstzahl der Serverprozesse, die gestartet werden dürfen

Server
(Multithreaded-)**Worker**-Modell

Grundsätzliche Idee

- Nur wenige Child-Prozesse
- Jeder Child-Prozess behandelt viele Verbindungen gleichzeitig
  - > Ein Thread pro Verbindung



Server
(Multithreaded-)**Worker**-Modell

Direktiven

- MinSpareThreads
  > Minimale Anzahl unbeschäftigter Threads
- MaxSpareThreads
  > Maximale Anzahl unbeschäftigter Threads, die zur Bedienung von Anfragespitzen zur Verfügung stehen
- ThreadsPerChild
  > Anzahl der Threads, die mit jedem Kindprozess gestartet werden
- MaxClients
  > Maximale Anzahl der Kindprozesse, die zur Bedienung von Anfragen gestartet wird

Server
Multi Processing Modules (**MPM**)

Prefork vs Worker

- Standard für Apache ist Prefork
- Worker-Modell wäre zu präferieren, da es weniger RAM benötigt
- Nur möglich, wenn das verwendete Modul auch thread-safe ist

Beispiel: mod_php (Apache PHP Module)

- Auszug aus der PHP Installation FAQ:

  Why shouldn't I use Apache2 with a threaded MPM in a production environment?

  […] When you make the underlying framework more complex by not having completely separate execution threads, completely separate memory segments and a strong sandbox for each request to play in, further weaknesses are introduced into PHP's system.

  If you want to use a threaded MPM, look at a FastCGI configuration where PHP is running in its own memory space.

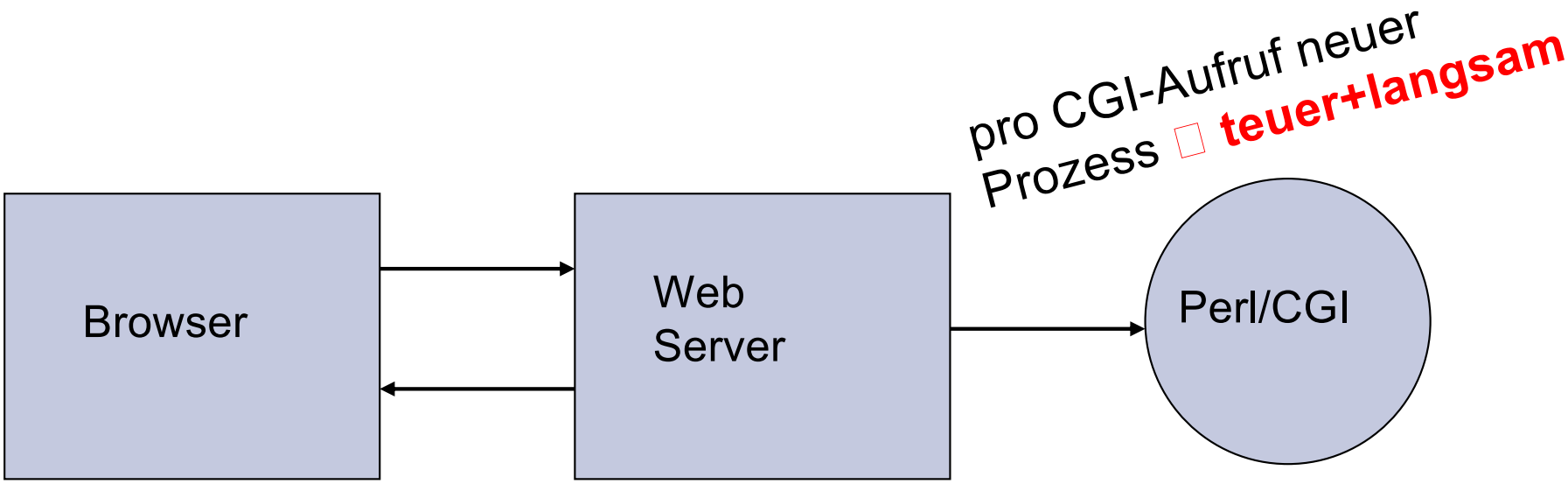
  […]

  Quelle: http://php.net/manual/en/faq.installation.php#faq.installation.apache2

- Fazit: PHP sollte nur im Prefork-Modus betrieben werden oder mittels FastCGI

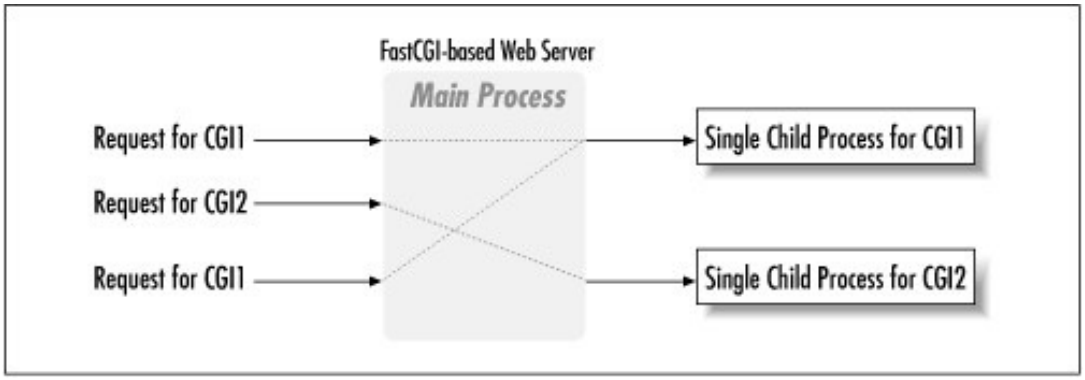
Server
Common Gateway Interface (CGI)

- Datenschnittstelle zum Ausführen beliebiger, externer Programme
- Angabe der Skript-Ressource über die URL
- Ursprünglich zumeist ein CGI/Perl Programm
- Der Interpreter wird in einem externen Prozess gestartet und erzeugt die HTML-Ausgabe



Server
Fast Common Gateway Interface (FastCGI)

- Ähnlich dem Worker-Modell: Es gibt einen Pool an CGI-fähigen Prozessen, die **isoliert vom aktuellen Web-Server** laufen (verbesserte Security)
- kein neuer Prozess pro Aufruf ☐ höhere Performance und statischer Ressourcenverbrauch im Gegensatz zu CGI



Quelle: https://docstore.mik.ua/orelly/java-ent/servlet/ch01_01.htm

- Sprachenunabhängig (häufig C++-Programme)
- Funktioniert nicht mit .htaccess (siehe später)
- Nutzt Socket-basierte Kommunikation über ein binäres Protokoll und kann so Verteilung stützen (Scale Out)

Beispiel: httpd.conf

```
ServerRoot "C:/xampp/apache"
Listen 80

LoadModule alias_module modules/mod_alias.so
LoadModule autoindex_module modules/mod_autoindex.so
LoadModule auth_basic_module modules/mod_auth_basic.so
LoadModule dir_module modules/mod_dir.so
# rewrite, ssl, ...

ServerAdmin postmaster@localhost
ServerName localhost:80

<Directory />
    AllowOverride none      # .htaccess darf das nicht überschreiben
    Require all denied      # Wir sprechen vom Wurzelverzeichnis des
Servers!
</Directory>
```

Beachten Sie bitte, dass der vom Apache voreingestellte Zugriff für **`<Directory />`**
`Allow from All` ist ☐ Apache liefert jede Datei aus, die durch eine URL abgebildet wird

Weitere Informationen: http://httpd.apache.org/docs/2.4/

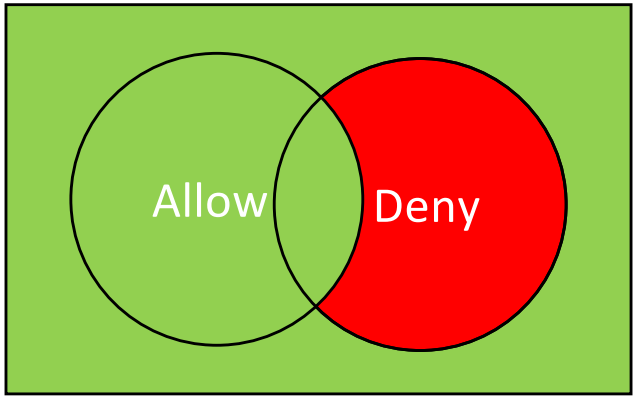
Beispiel: httpd.conf (Fortsetzung)

```
DocumentRoot "C:/xampp/htdocs"
<Directory "C:/xampp/htdocs">
    Options Indexes           # Würde das Verzeichnis ohne index.html ausliefern
    AllowOverride All         # Eigene Regeln über .htaccess
    Require all granted       # Hier dürfen alle zugreifen
</Directory>

<IfModule dir_module>
    DirectoryIndex index.php index.html index.htm
</IfModule>

<Files ".ht*">
    Require all denied        # Die Dateien sollten nicht lesbar sein
</Files>

ErrorLog "logs/error.log"
LogLevel warn
```

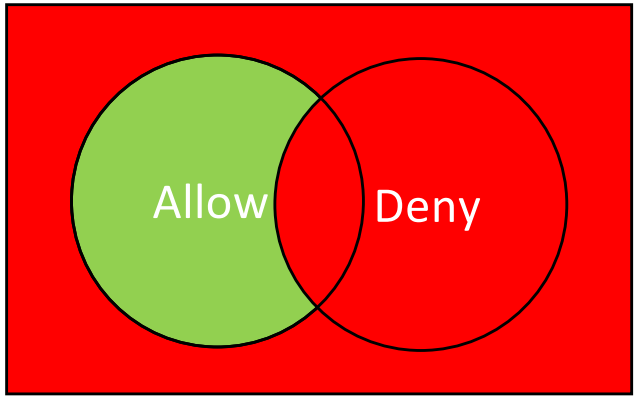
Zugriffskontrolle (Mittels Require einfacher seit Apache 2.4)

- Order Deny, Allow
  - > First, all *Deny* directives are evaluated; if any match, the request is denied **unless** it also matches an *Allow* directive. Any requests which do not match any *Allow* or *Deny* directives are permitted.
- Order Allow, Deny
  - > First, all *Allow* directives are evaluated; at least one must match, or the request is rejected. Next, all *Deny* directives are evaluated. If any matches, the request is rejected. Last, any requests which do not match an *Allow* or a *Deny* directive are denied by default.

Order Deny, Allow



Order Allow, Deny



Weitere Informationen (& Quelle der Texte): https://httpd.apache.org/docs/2.4/mod/mod_access_compat.html

Zugriffskontrolle neu (ab 2.4.):

```
Require host apache.org


Require all granted
Require not ip 18.20.20.12

Require not sander.fh-aachen.de

Require not de
```

Apache
Konfiguration

.htaccess

- Erlaubt es Direktiven innerhalb von Verzeichnissen zu überschreiben, ohne das hierfür der Server neugestartet werden muss
- Häufig wird dies genutzt um z.B. den Zugriff zu beschränken (z.B. auf das Intranet oder bestimmte IP-Adressen)
- Es muss also nicht jegliche Konfiguration in Konfigurationsdateien geschrieben werden, sondern dies kann auch über .htaccess passieren

- Diese Datei bildet auch die Basis für eine HTTP-basierte Authentifikation

HTTP bietet kein eigenes Konzept zur Datensicherheit

- Absicherung der Übertragung der Daten zwischen Client und Server durch kryptographische Verschlüsselung
  - > HTTP über TLS/SSL (HTTPS)
  - > Dazu im Sicherheitsteil mehr (Zertifikate, Private/Public-Keys, …)

- Es werden allerdings Verfahren zur Einschränkung des Zugriffs durch Authentifizierung des Klienten bereitgestellt:
  - > Basic Authentication
  - > Digest Authentication

Server
Zugriffskontrolle

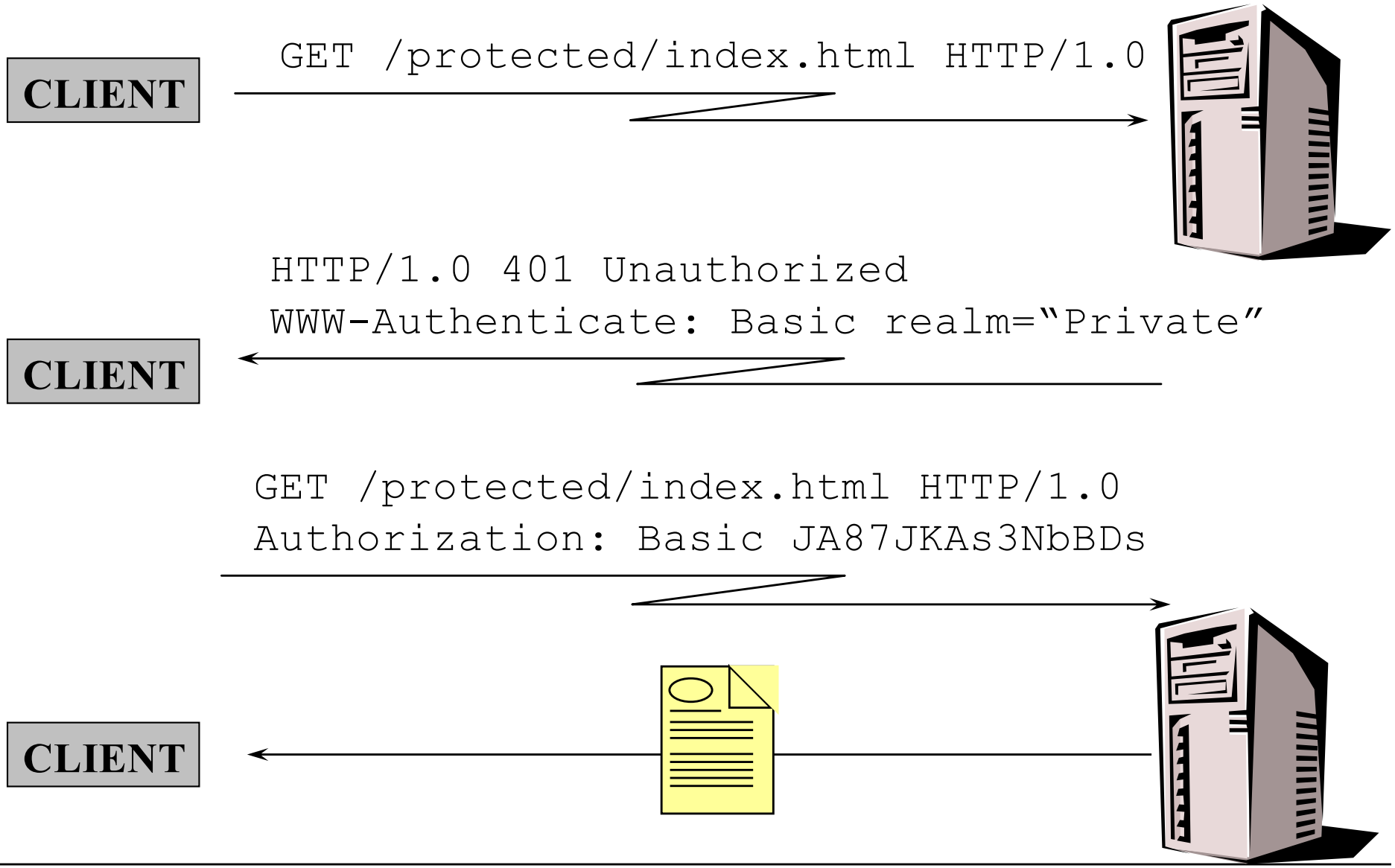
Basic Authentication Scheme

- Authentifizierungsmethode seit HTTP 1.0
- Übertragung von Benutzerkennung und Passwort
- Passwort kann sogar unverschlüsselt sein
- Autorisierung basiert auf passwd-ähnlicher Datei (crypt)

Digest Authentication Scheme

- Verbesserung in HTTP 1.1
- Server überträgt eine zufällig ausgewählte Zeichenkette (Challenge)
- Benutzer verwendet sein Kennwort um hierzu eine Signatur (Digest) zu erzeugen
- Server kennt das Passwort und prüft ob das verwendete Kennwort korrekt war

Server
Basic Authentication



Server
Basic Authentication

Beispiel

- (Annahme: AllowOverride-Direktive erlaubt den Einsatz von .htaccess)
- Anlegen einer .htaccess Datei im zu schützenden Verzeichnis

```
AuthUserFile c:\xampp\htdocs\protected\.htpasswd
AuthType Basic
AuthName "Der Zugriff auf die Kursseiten ist geschützt"
require valid-user
```

- Einfügen der Nutzer mit dem htpasswd-Kommando

```
htpasswd -d -c c:\xampp\htdocs\protected\.htpasswd user
```

Password und Benutzername wird mittels Base64-Kodierung übertragen!

Probleme von Basic Authentication

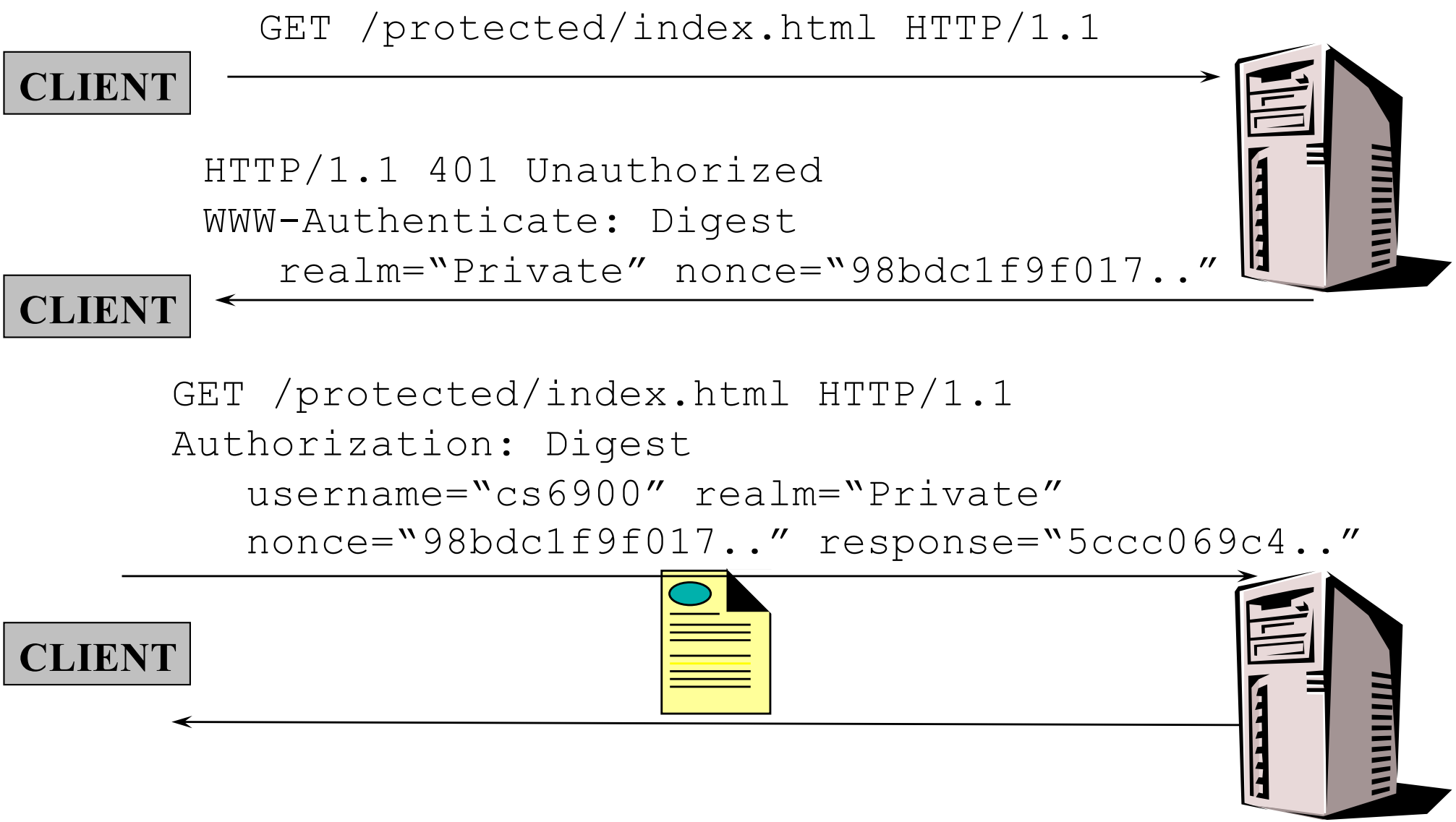
- Passwörter werden im Klartext (Benutzername:Password in Base64-Kodierung)übermittelt
  - > Leicht für Mittelsmänner abzufangen
  - > TSL-Verschlüsselung hilft hier
- Keinerlei Authentisierung des Servers
  - > Offen für Spoofing Attacken
  - > TLS-hilft auch hier
- Keine Absicherung der Nachrichten
  - > Man-in-the-Middle Attacken
  - > Erneut hilft TLS

Probleme:

- Art der Abspeicherung des Passwords beim Server
- BASIC-Auth cached username und password im Browser
- Vergleichbar mit Password-Cookie

Mit HTTP 1.1 eine weitere Alternative: Digest Authentication

Server
Digest Authentication



Server
Digest Authentication

Challenge (“nonce”): Frei wählbare Zeichenkette

- `MD5(IP address:timestamp:server secret)`

Response: Challenge signiert mit Benutzername & Passwort

- `MD5(MD5(name:realm:password):nonce:MD5(request))`

Server-spezifische Optionen

- Einmalige nonces
- Zeitstempel basierte nonces

URL + Realm definieren den geschützten Bereich

- Innerhalb dieses Bereichs kann mit den gleichen Credentials gearbeitet werden, solange sie gültig sind

Server
Digest Authentication

Nutzung

- Httpd.conf:

```
LoadModule auth_digest_module modules/mod_auth_digest.so
```

- Anlegen einer .htaccess Datei im zu schützenden Verzeichnis
  > (Annahme: AllowOverride-Direktive erlaubt den Einsatz von .htaccess)

```
AuthDigestFile c:\apachefriends\xampp\htdocs\.htpasswd.di
AuthType Digest
AuthName "Vorlesung"
require valid-user
```

- Einfügen der Nutzer mit dem htdigest-Kommando

```
htdigest -c c:\apachefriends\xampp\htdocs\.htpasswd.di
 Vorlesung user
```

Server
Digest Authentication

Vorteile gegenüber Basic Auth.

- Keine Passwörter im Klartext
- Keine Abspeicherung von Klartextpasswörtern beim Server
- Server wird authentisiert
- Ansonsten wird nicht viel gewonnen
  - > Man-in-the-middle-Attacken
  - > Sniffing-Attacken

nginx („engine-ex“)

Der von Igor Sysoev ursprünglich für eine russische Suchmaschine entwickelte NGINX-Webserver besitzt besondere Eigenschaften:

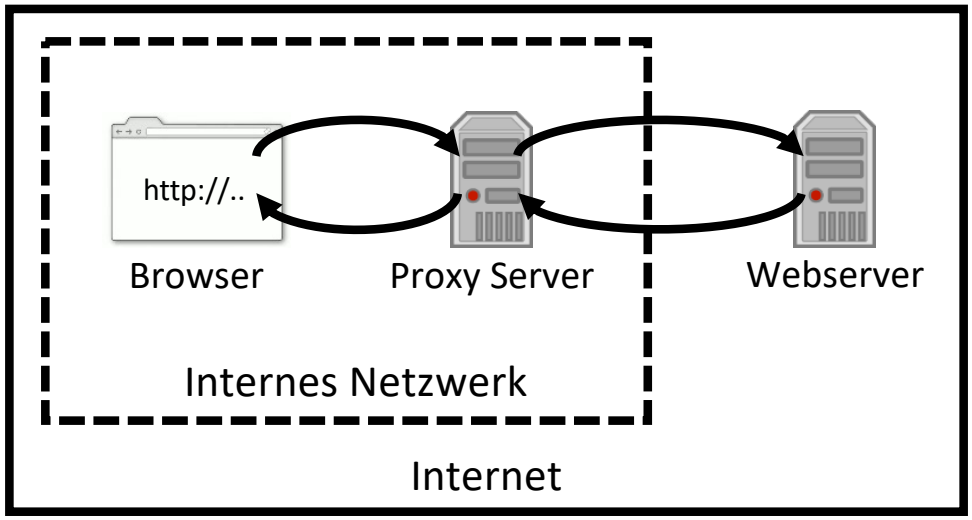
- Hohe Performance
- Geringer Memory-Footprint
- Reverse Proxy
- E-Mail Proxy
- Erweiterbar durch Module

Für einfache Webserver, die nur selten Anpassungen erfordern und für Infrastruktur mit eingeschränkter Hardware (embedded Systeme) bietet NGINX große Vorteile.

Zusammen mit dem Memcached-Key-Value-Store-Module ergibt sich ein extrem leistungsfähiges System.

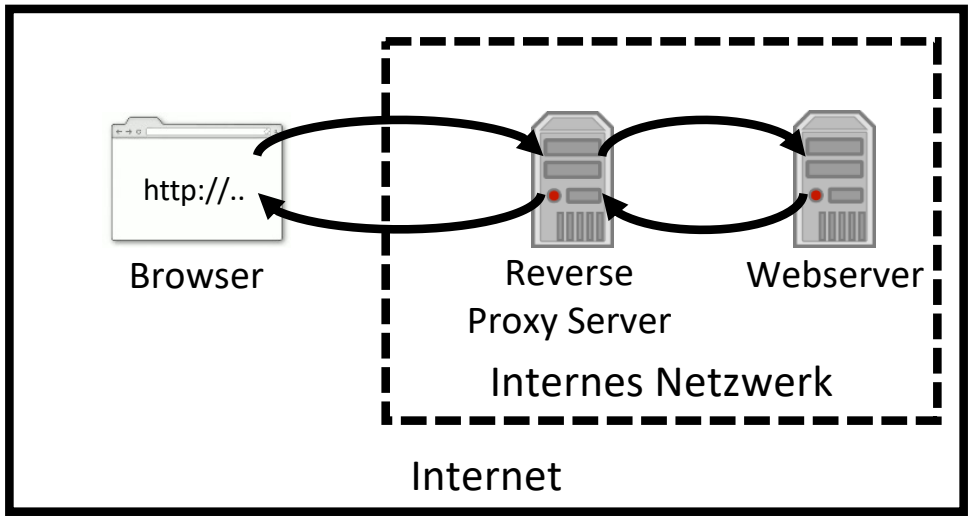
Exkurs: Normaler (Forward) Proxy

- Server, hinter dem sich der Client „versteckt“
- Aus Sicht des Webservers, ist der Proxy der Client
- Anwendungen:
  - > Anonymisierung
  - > Firewall
  - > TLS-Terminierung (keine Verschlüsselung im internet Netzwerk)
  - > SSL Visibility Appliance
  - > Werbefilter
  - > Caching



Reverse Proxy

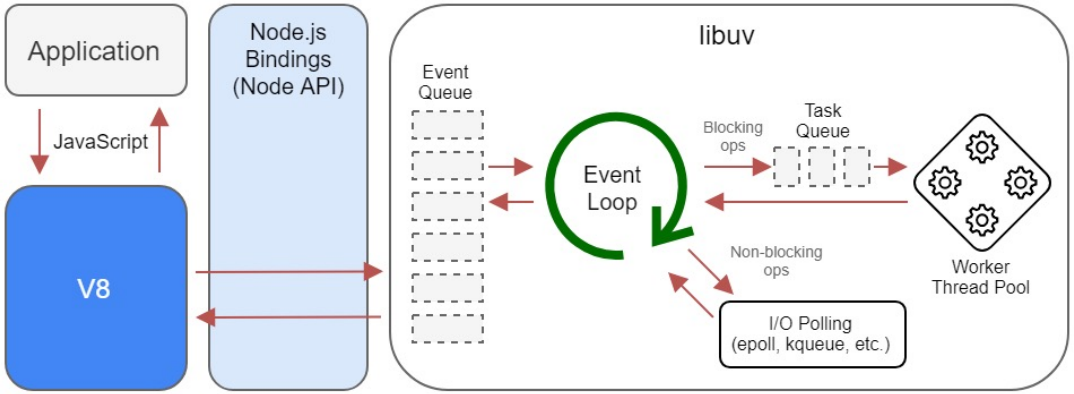
- Server, hinter dem sich ein anderer Server versteckt
- Der Client sieht nur den Reverse Proxy Server
- Anwendungen:
  - > Sicherheit/Firewall
  - > Verschlüsselung über den Reverse Proxy
  - > Load Balancing (hinter dem Proxy können mehrere andere Server liegen)
  - > Caching / Content Delivery
  - > Authentifizierung



Node.js
Wirklich Single Threaded?

Tatsächlich ist node.js keinesfalls Single-Threaded

- Die asynchron durchzuführende Arbeit wird über die libuv (die u.a. die Event-Loop realisiert) an einen Worker-Thread-Pool weiter geleitet
- Die JavaScript-Laufzeitumgebung (V8_engine) ist single-threaded und nutzt die Event-Queue (Loop) u.a. für Low-Level-Aufgaben
  - > Basis hierfür ist „Event Poll" (epoll-Linux), bzw. kqueue (BSD)
  - > Do not Block the Event-Loop gilt!
- Einige Aufgaben (I/O-, CPU-intensive) werden eigentlich in libuv realisiert und können jedoch auf eine andere Thread ausgelagert werden, so wie es auch Threaded-Funktionen gibt (z.B. aus dem Package crypto)

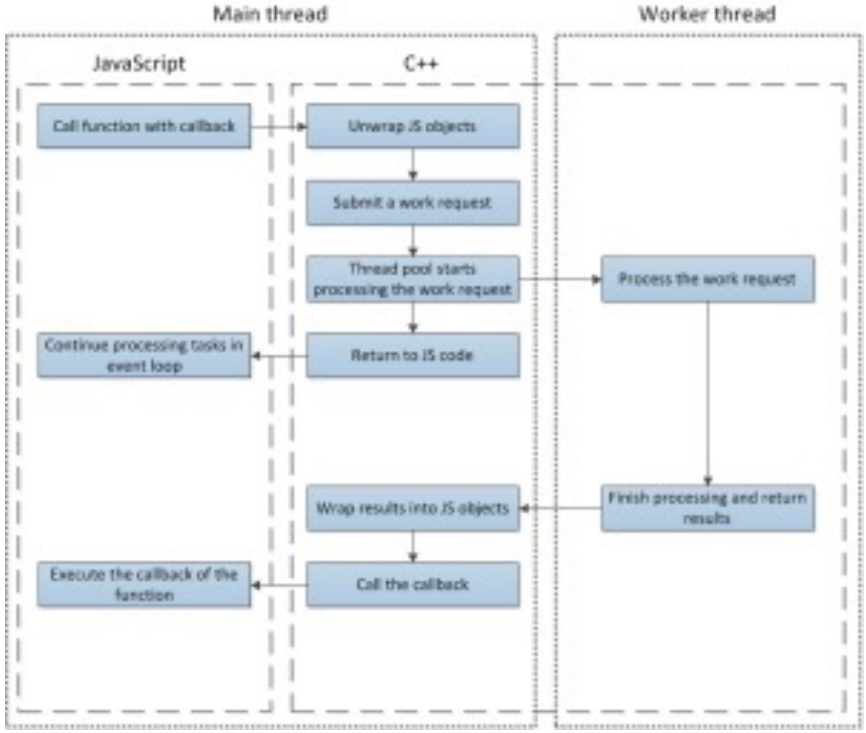


https://medium.com/preezma/node-js-event-loop-architecture-go-deeper-node-core-c96b4cec7aa4

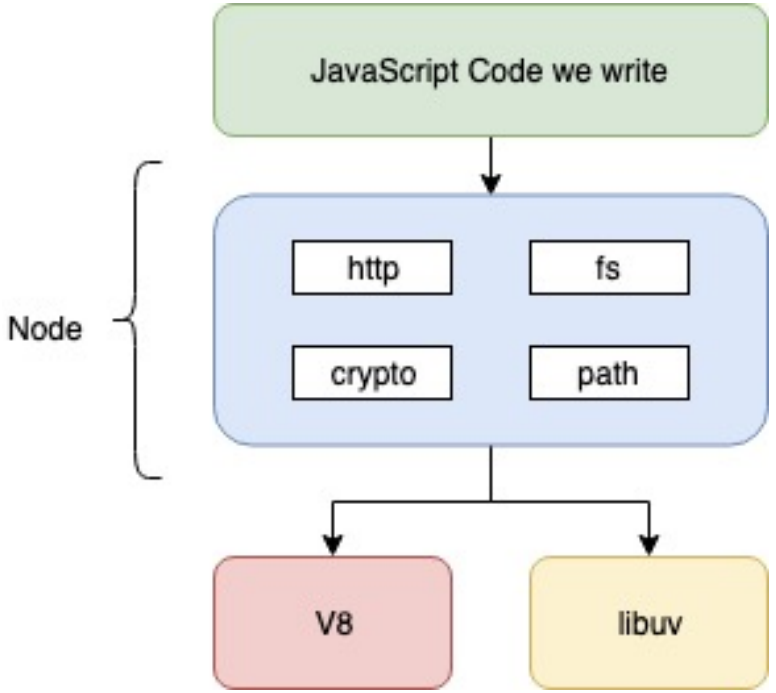
Node.js
libuv ist die Basis

Tatsächlich ist node.js keinesfalls Single-Threaded

- Die eigentliche Idee von libuv war, über Event-Loops den I/O asynchron durchführen zu können
- TCP, UDP und DNS gehört dazu
- Libuv hat hierzu einen Thread-Pool zur Verfügung



https://kariera.future-processing.pl/blog/on-problems-with-threads-in-node-js/

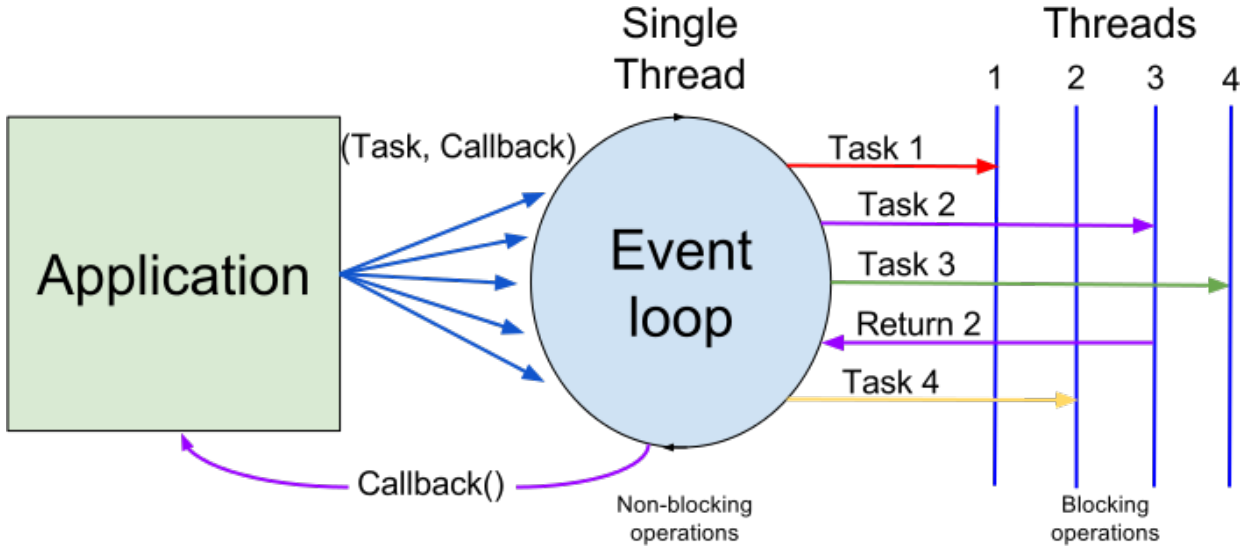


https://miro.medium.com/max/582/0*eJuFOUQRj34f_-In

Node.js
Wirklich Single Threaded?

Tatsächlich nutzt Node.js per Default einen Pool aus 4 Threads!

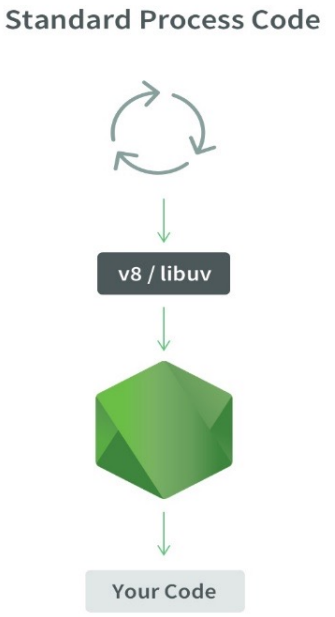
- Asynchrone Aufgaben können so an die Threads übergeben werden (an die Diener)
- UV_THREADPOOL_SIZE=100 && node index.js ->100 Threads!

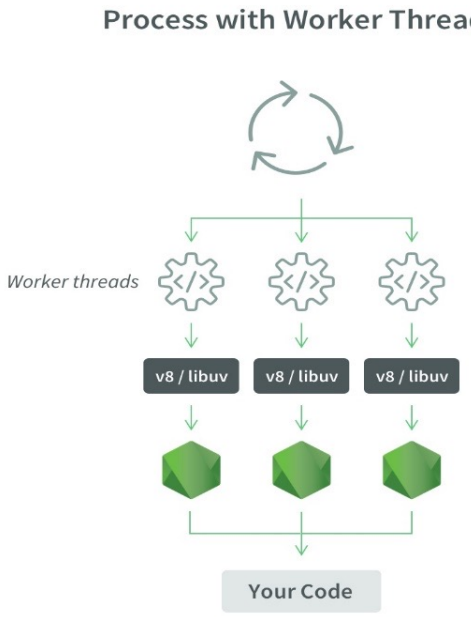


https://miro.medium.com/max/1400/1*fBEbMgk6--QtIUUed3KcMQ.png

Seit Version 10.5 gibt es ein worker_thread-Modul

- JavaScript selber wird nicht Threaded werden
- Wir duplizieren die V8-Engine und die Event-Loop einfach für viele Threads und haben so multiple node-Instanzen in einem Prozess





https://nodesource.com/blog/worker-threads-nodejs

Node.js
worker-thread-Modul

```
const { Worker, isMainThread, parentPort } = require('worker_threads');

if(isMainThread) {
    console.log('main thread start...');
    const worker = new Worker(__filename);
    worker.on('message', (msg) => {
        console.log(`Worker: ${msg}`);
      });
    console.log("doing some random work in main thread..!!");
}else{
    parentPort.postMessage('hello from worker thread');
    cpuIntensiveTask(1000);
    parentPort.postMessage('i am working');
    cpuIntensiveTask(1000);
    parentPort.postMessage('task is done..!!');
}

function cpuIntensiveTask(timeInSecond) {
    const end = Date.now() + timeInSecond;
    while (Date.now() < end) { }
  }
```

Code für eigene Thread

Kommunikationskanal vom Main thread()

Variable, die auf die eigene Datei zeigt

Kommunikationskanal vom Worker

```
$ node thread1.js
main thread start...
doing some random work in main thread..!!
Worker: hello from worker thread
Worker: i am working
Worker: task is done..!!
```

https://medium.com/khojchakra/worker-threads-in-node-js-9021be83e3d1

Mittels des Cluster-Moduls kann man sogar adressraumtechnisch getrennte Instanzen starten. Die Kommunikation geschieht dann über Interprozesskommunikation

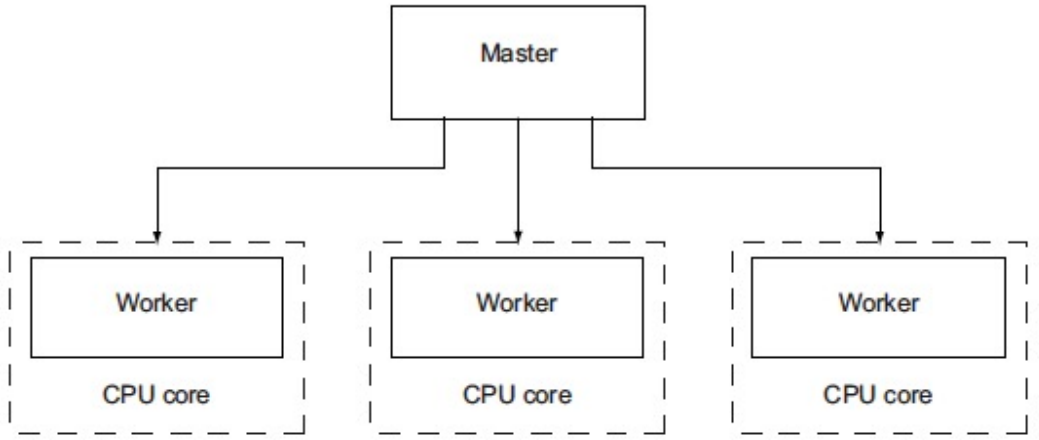


Figure 12.4 A master spawning three workers on a four-core processor

Quelle: Node.js in Action, Manning

Nodejs
Cluster Module

```
var cluster = require('cluster‘);
var express = require('express‘);
var numCPUs = require('os').cpus().length;
if (cluster.isMaster) {
    for (var i = 0; i < numCPUs; i++) {
        // Create a worker
        cluster.fork();
    }
}
else {
    // Workers share the TCP connection in this server
    var app = express();
  app.get('/', function (req, res) {
      res.send('Hello World!‘);
  });
  // All workers use this port
  app.listen(8080); }
```

Quelle: https://stackabuse.com/setting-up-a-node-js-cluster/

Nodejs
Cluster Module

But how are requests divided up between the workers? Obviously they can't (and shouldn't) all be listening and responding to every single request that we get. To handle this, there is actually an embedded load-balancer within the `cluster` module that handles distributing requests between the different workers. On Linux and OSX (but not Windows) the round-robin (`cluster.SCHED_RR`) policy is in effect by default. The only other scheduling option available is to leave it up to the operating system (`cluster.SCHED_NONE`), which is default on Windows.

The scheduling policy can be set either in `cluster.schedulingPolicy` or by setting it on the environment variable `NODE_CLUSTER_SCHED_POLICY` (with values of either 'rr' or 'none').

You might also be wondering how different processes can be sharing a single port. The difficult part about running so many processes that handle network requests is that traditionally only one can have a port open at once. The big benefit of `cluster` is that it handles the port-sharing for you, so any ports you have open, like for a web-server, will be accessible for all children. This is done via IPC, which means the master just sends the port handle to each worker.

Quelle: https://stackabuse.com/setting-up-a-node-js-cluster/

Nodejs und NGINX
Win-Win: Statisches via NGINX

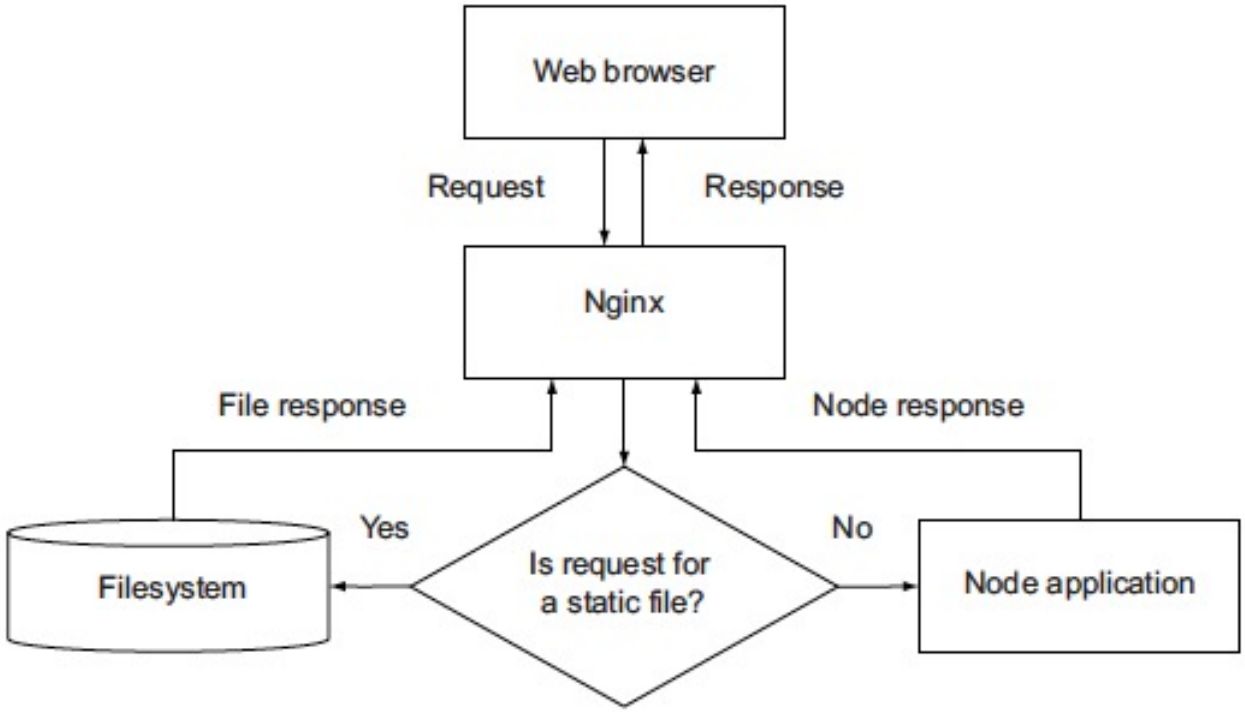


Figure 12.5 You can use Nginx as a proxy to relay static assets quickly back to web clients.

Quelle: Node.js in Action, Manning