sk2httpd
Serwer HTTP

3 kwietnia 2006 roku

1 Wstęp

Poniższy dokument zawiera opis koncepcji i architektury serwera protokołu HTTP pisanego w ramach laboratoriów z Sieci Komputerowych 2. Skoncentrowano się w nim na realizacji wydajnej obsługi sieciowej i szczególnie pod tym kątem będzie on dalej omawiany. W ramach realizacji przeprowadzone zostały także testy wydajności różnych wariantów konfiguracyjnych. Na końcu został natomiast opisany sposób konfiguracji programu.

1.1 Cel projektu

Produktem końcowym projektu jest serwer zgodny z protokołem HTTP. Współbieżnie obsługujący wielu klientów oraz zapewniający niezależność między nimi. Serwer ten musi swobodnie obsługiwać popularne przeglądarki internetowe, oddawać pełny dostęp do danych ściśle zdefiniowanych przez administratora, korzystając jednocześnie z ograniczonego zbioru zasobów. Kluczowym elementem projektu jest możliwość obsługi bardzo dużej liczby zapytań, co wynika m.in. z bezpołączeniowej koncepcji protokołu HTTP oraz jego dużej popularności w Internecie.

1.2 Zakres projektu

Z uwagi na ograniczony czas i liczność zespołu, zaimplementowano tylko wybrane elementy protokołu HTTP, oraz podstawową grupę parametrów konfiguracji serwera. Założono że serwer działa na jednej maszynie, udostępnia zasoby jednej domeny (brak tzw. virtual hosting), ma uproszczone mechanizmy cache. Jednocześnie, zaimplementowane elementy tworzą z sk2httpd w pełni działający serwer WWW. Pozwalają śledzić ruch na serwerze, system dynamicznie dobiera ilość zasobów systemowych, umożliwia tzw. stałe połączenia (persistent), pozwala na wykonanie aplikacji CGI i jest bardzo konfigurowalny.

2 Architektura serwera

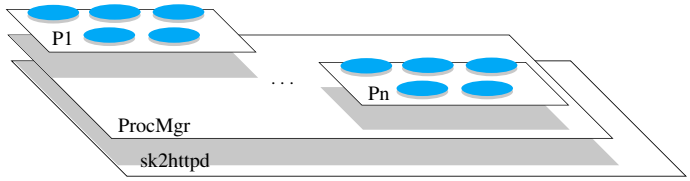
Idea współbieżnego serwera wymaga, aby odpowiedzi na zapytania były generowane jednocześnie. Jednak same zapytania nadchodzą sekwencyjnie. Powstają więc pytania, jak podzielić zadania, kto ma je wykonywać i kiedy. Najlepszym obecnie stosowanym w "dużych serwerach" rozwiązaniem jest tzw. worker pool.

2.1 Worker pool

Koncepcja polega na tym, że przyjmowane zgłoszenia trafiają do bufora, z którego następnie są pobierane przez agentów cały czas gotowych i oczekujących na nie. Jest to odwrót od starszego rozwiązania, które zakładało tworzenie dla każdego zgłoszenia nowego agenta. Powodem jest długi czas tworzenia agenta (implementowanego jako proces lub wątek) w stosunku do całego czasu jego życia. Stąd widać, że istotnym parametrem wydajności serwera będzie rozmiar kolejki oraz liczba agentów oczekujących.
Z uwagi na oszczędność zasobów, założyliśmy, że liczba agentów będzie się zmieniać w zależności od intensywności ruchu, w ramach pewnych granic ustawionych przez administratora serwera.
Z drugiej strony, by jak najpełniej korzystać z czasu procesora, zdecydowaliśmy że do roli agentów obsługi zgłoszeń będą powoływane wątki (threads), pogrupowane w większe ilości w osobnych procesach.

2.2 Implementacja worker pool

By spełnić taką koncepcję Serwer został zaprojektowany ze ścisłym podziałem obowiązków. Pierwszą rolę pełni główny proces sk2httpd, który po załadowaniu konfiguracji i utworzeniu struktur współdzielonych, rozpoczyna nasłuchiwanie zgłoszeń. W osobnym wątku uruchamiany jest manager procesów (ProcMgr) obsługi zapytań. To on na podstawie oceny intensywności ruchu, decyduje o uruchomieniu lub zatrzymaniu dodatkowych procesów i wątków obsługi zapytań. Poniższy rysunek oddaje ten podział.



Rysunek 1: Podział obowiązków między procesami

2.3 Bufor

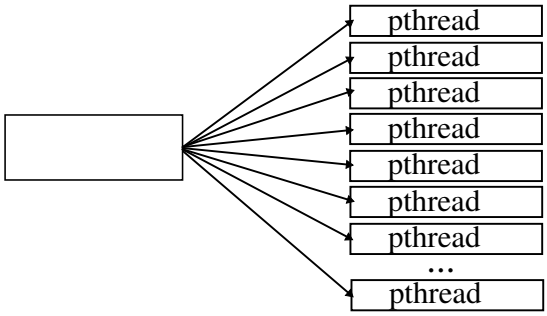
Newralgicznym puntem obsługi zapytań jest jednak bufor. Musi być dostępny dla każdego wątku. Musi gwarantować wykluczanie dostępu dla wątków ubiegających się o zgłoszenia, by jednego zgłoszenia nie obsługiwało wiele wątków. Gwarantuje też wykluczanie dostępu między wątkami obsługi, a procesem dodającym nowe zgłoszenia. Jest to więc klasyczny problem producenta i konsumenta. Rozwiązaliśmy go korzystając z semafora (pthread_mutex_t) i zmiennej warunkowej (pthread_cond_t) biblioteki pthread.
Gdy przychodzi nowe zgłoszenie, bufor jest przełączany w tryb wyłączności by je dodać. Następnie budzone są wątki oczekujące na zmiennej warunkowej, by jeden z nich mógł przejąć zadanie.
Każdy wątek w trybie wyłącznym sprawdza liczbę dostępnych zapytań w buforze i o ile nic nie znajdzie, zawiesza się w oczekiwaniu na zmiennej warunkowej.
Bufor jest dostępny w obszarze pamięci współdzielonej. W przypadku wątków jest to zresztą naturalne. Natomiast, aby procesy współdzielić pamięć, tworzone są z odpowiednimi parametrami funkcji systemowej clone.
Sam bufor działa na zasadzie kolejki. Do jej obsługi są dwie zmienne: liczba zgłoszeń w buforze oraz numer indeksu pierwszego dostępnego zgłoszenia, co w połączeniu z trzecią zmienną - rozmiarem bufora, daje cały obraz kolejki.

2.4 Obsługa zgłoszeń

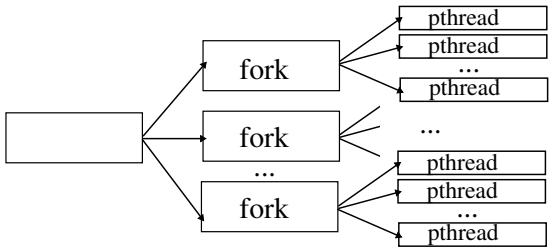
Należy jeszcze dodać kilka słów nt. grupowania wątków w procesy. Były rozważane dwie opcje (patrz także rysunki 2 i 3):

1. proces nadrzędny i wiele wątków w ramach jego
2. proces nadrzędny, procesy podrzędne, w ramach każdego procesu podrzędnego niewielka liczba wątków

Pierwsza opcja jest prostsza w implementacji. Głównych zalet drugiej opcji wypatrywano tak naprawdę w wadach wątków. To znaczy duża liczba wątków mogłaby być źle szeregowana, lub że wątki w ramach jednej grupy PID nie

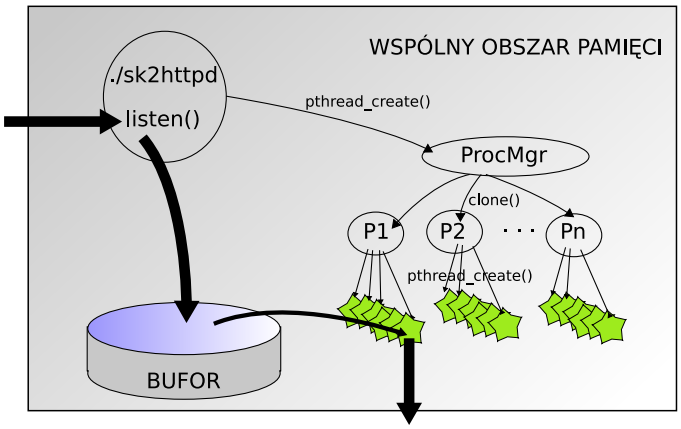


Rysunek 2: Model 1 proces-wątek



Rysunek 3: Model 2 proces-proces-wątek

otrzymają tyle zasobów, co w ramach kilku grup o różnych PID. Ostatecznie zaimplementowano drugą wersję (procesprocesy-wątki). Manager procesów tworzy, lub usuwa procesy obsługi, które uruchamiają określoną liczbę wątków. Model ten w prosty sposób można uprościć do modelu pierwszego (proces-wątki), poprzez redukcję liczby procesów podrzędnych do 1.
Struktura serwera z punktu widzenia systemu operacyjnego została przedstawiona na rysunku 4. Zaznaczono tam także przepływ przykładowego zapytania. Dla uproszczenia, nie widać natomiast dodatkowych procesów związanych z wykonywaniem aplikacji CGI. Wkrótce zostanie to dokładniej omówione.



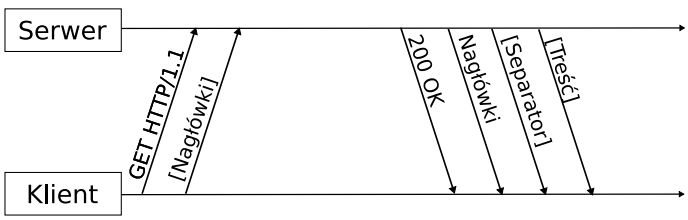
Rysunek 4: Struktura serwera z punktu widzenia systemu

2.5 Algorytm dopasowania liczby procesów

Jak wspomniano, serwer dostosowuje liczbę działających procesów z wątkami obsługi, do rzeczywistego zapotrzebowania. Administrator ma możliwość wskazania przedziału, w jakim liczba procesów powinna się znajdować. Początkowo manager procesów uruchamia minimalną obowiązkową liczbę procesów i ze stałym odstępem czasu monitoruje parametry kolejki. Dąży do tego, by kolejka była pusta, tzn. by zgłoszenia czekały minimalną ilość czasu. Jeśli w kolejce przez dłuższy okres czasu utrzymują się nie obsłużone zgłoszenia, uruchamiany jest dodatkowy proces z nowym pakietem wątków obsługi. Z drugiej strony, gdy w kolejce jest pusto, najpóźniej utworzony proces jest zatrzymywany.
Zanim wybraliśmy taką implementację, rozważana była też opcja, bardziej dokładnego dostrajania - przez dodawanie i usuwanie pojedynczych wątków obsługi. Jednak wówczas nie było jasne, kiedy należałoby dodawać pojedyńczy wątek, a kiedy cały proces - to znaczy jak duży musi być wzrost obciążenia kolejki. Dzięki temu, manager procesów może w większych odstępach czasu sprawdzać stan kolejki.

3 Protokół HTTP

Na etapie implementacji obsługi zgłoszeń zdecydowaliśmy się na wsparcie najnowszej wersji protokołu: HTTP/1.1. Protokół ten jest bardzo rozbudowany, definiuje m.in. obsługę klientów typu Proxy, metody autoryzacji, wspiera cache'owanie zasobów. Są to istotne elementy działania serwera WWW, jednak dla naszego klienta został wybrany tylko pewien zakres funkcjonalności. Mimo tego, akceptowane są zgłoszenia zarówno w protokole HTTP/1.0 jak i HTTP/1.1. Schemat połączenia przedstawiono na rysunku 5.

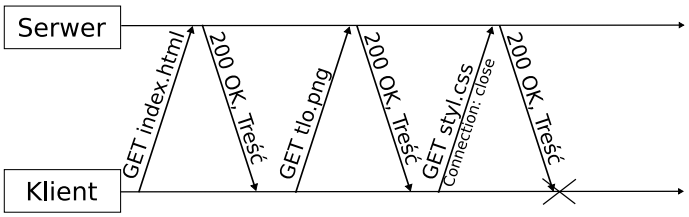


Rysunek 5: Komunikacja w HTTP

Klient w zgłoszeniu wskazuje nazwę żądanego zasobu (*URI - Unified Resource Identifier*), oraz dodatkowe opcje (nieobowiązkowe). Rysunek przedstawia przykład zapytania typu GET. Jednak w niektórych innych typach zapytań, np. w POST klient przesyła także treść, poprzedzoną separatorem. Rolę separatora w HTTP stanowią dwa kolejne znaki nowej linii `\r\n\r\n`.
Serwer po określeniu lokalizacji pliku na dysku, na podstawie adresu URI, oraz sprawdzeniu możliwości dostępu, generuje odpowiedź. Odpowiedź zawiera kod wynikowy, mówiący czy operacja się powiodła, czy nie, informacje na temat zasobu i połączenia (w nagłówkach), oraz treść zasobu.

3.1 Persistent connections

Już w czasach standardu HTTP/1.0 dostrzeżono, że tworzenie nowego połączenia dla każdego kolejnego zasobu jest bardzo drogie (podobnie jak tworzenie osobnego wątku obsługi). Stąd niektóre serwery zaimplementowały strategię *Keep-Alive*, dalej w HTTP/1.1 przemianowaną na *Persistent connections*.
Połączenia typu persistent pozwalają klientowi pobrać wiele zasobów w trakcie jednej komunikacji. Serwer przed zamknięciem połączenia czeka chwilę. Klient może sterować połączeniem dzięki nagłówkom `Connection` oraz `Keep-Alive`. Scenariusz takiego połączenia przedstawia rysunek 6. W standardzie HTTP/1.1, połączenia persistent są domyślne, natomiast w 1.0, trzeba je było wymusić nagłówkiem `Connection: keep-alive`. Aby zakończyć połączenie, przeglądarka przy ostatnim zapytaniu powinna wysłać nagłówek `Connection: close`. W innym przypadku, serwer po przekroczeniu pewnego czasu sam się rozłączy. Ponadto klient może zaproponować czas oczekiwania umieszczając w nagłówku `Keep-Alive: czas`, czas jest liczony w sekundach.



Rysunek 6: Persistent connection

W związku z implementacją połączeń typu persistent pojawił się inny problem. Dotychczas klient rozpoznawał koniec zasobu po tym, że serwer zamknął połączenie. Standard HTTP/1.1 proponuje kilka sposobów realizacji takich połączeń, np. chunked. Jednak w sk2httpd problem ten rozwiązano dodając w odpowiedzi serwera nagłówek `Content-length`. Jego implementacja wymaga, aby serwer znał na przód rozmiar pliku, co np. w przypadku zasobów generowanych dynamicznie (CGI) wymaga dodatkowego buforowania.

3.2 Udostępniane zasoby

Serwer sk2httpd oferuje trzy podstawowe typy zasobów: pliki, aplikacje CGI oraz listingi katalogów. Ponadto zamiast zasobu serwer może czasem zwrócić błąd, np. gdy plik jest niedostępny. Serwer nie daje dostępu do wszystkich plików na dysku. Rozstrzyganie o tym, który plik wyświetlić w odpowiedzi na dane zapytanie, odbywa się poprzez dopasowywanie adresu URI. W pliku konfiguracyjnym można zdefiniować dowolną liczbę bazowych ścieżek URI oraz powiązać z nimi katalogi na dysku serwera. Wówczas po otrzymaniu zgłoszenia serwer dopasowując zapytanie do kolejnych ścieżek wybierze pierwszą odpowiadającą i przekształci adres URI na bezwzględny adres pliku. Oto przykładowa konfiguracja:

Dopasowanie URI	Ścieżka na dysku
/ jacek/	/home/jacek/public_html/
/	/var/www/htdocs/

Tabela 1: Konfiguracja dopasowań URI, odpowiadająca popularnym ustawieniom serwerów www.

Poniżej natomiast przykład kilku dopasowań, jakie zastosuje serwer, by znaleźć właściwy zasób:
Taki sposób organizacji zasobów jest dużo bardziej elastyczny, niż wskazanie jednego punktu korzenia (*Document root*), ma za to zbliżone możliwości do koncepcji katalogów z Apache.
Widać także, że rozszerzenie mechanizmów dopasowania, tak by rozpatrywały URI, oraz adres domeny, w przyszłej wersji projektu pozwoliłoby obsługiwać wirtualne domeny.

URL zapytania	Znaleziony zasób
http://localhost/pliki.html	/var/www/htdocs/pliki.html
http://localhost/documents/sk2httpd	/var/www/htdocs/documents/sk2httpd
http://localhost/ jacek/podkatalog/pliki.html	/home/jacek/public_html/podkatalog/pliki.html

Tabela 2: Dopasowania zastosowane przez serwer sk2httpd.

3.3 Typy MIME

Serwer ma także za zadanie poinformować klienta jakiego typu jest zasób, tzn. czy jest to grafika, tekst, strona html, czy plik binarny. Identyfikacja typu zasobu odbywa się podobnie, poprzez znajdowanie dopasowań. Są one jednak bardziej rozbudowane, gdyż typ MIME pliku można określić nie tylko na podstawie jego lokalizacji, ale także rozszerzenia. Przykładowa konfiguracja:

Dopasowanie URI	rozszerzenie	Typ pliku
/documents/	-	application/pdf
/cgi-bin/	.cgi	application/cgi
-	.html	text/html
-	-	text/plain

Tabela 3: Konfiguracja dopasowań typów plików

Dopasowanie URI ani dopasowanie rozszerzenia nie jest obowiązkowe. Należy pamiętać, że tu także URI zapytania jest porównywane kolejno od pierwszej reguły konfiguracyjnej. Tak więc uzasadnione jest podawanie najpierw szczegółowych deklaracji, następnie coraz bardziej ogólnych, a na końcu domyślnej, która dopasuje się zawsze i sprawi, że niezidentyfikowany plik będzie typu text/plain.
Na pierwszy rzut oka możliwość dopasowywania typu pliku na podstawie katalogu wydaje się nie mieć zastosowania. Jednak chcieliśmy przy tej okazji upiec dwie pieczenie na jednym ogniu. A mianowicie zapewnić bezpieczeństwo wykonywania plików CGI.

3.4 Pliki CGI

Serwer pozwala na uruchamianie plików wykonywalnych dzięki mechanizmowi CGI. Jednak pliki te muszą spełniać pewne warunki: Po pierwsze, muszą być dostępne do odczytu, muszą mieć prawo do wykonania dla wszystkich (`------x`), a po trzecie muszą być typu `application/cgi`. To serwer decyduje o przyznaniu plikowi typu application/cgi. A administrator może skonfigurować serwer tak, by tylko pliki w określonych katalogach były traktowane jako CGI. Prawdziwy typ dokumentu zostanie natomiast wygenerowany dopiero przez samą aplikację.
Wygenerowanie odpowiedzi dla pliku wykonywalnego jest trochę bardziej złożone, niż dla zwykłego pliku. Po upewnieniu się, że spełnia on w/w wymagania, wątek obsługi tworzy nowy proces (`fork()`), w zmiennych środowiskowych zapisuje parametry aplikacji (`QUERY_STRING`), oraz inne dane dot. zapytania, przekierowuje standardowe wyjście aplikacji do bufora i wykonuje program (`execv()`). W dalszych krokach, do klienta odsyłany jest rozmiar odpowiedzi, oraz sama treść odpowiedzi wraz z nagłówkami wygenerowanymi przez serwer.

3.5 Listingi katalogów

Dla zapytań nie wskazujących konkretnych plików, a całe katalogi serwer sam generuje plik html zawierający odnośniki do zasobów w katalogu. Jedynym wyjątkiem, jaki należało obsłużyć dla zgodności z protokołem HTTP jest obsługa niepełnych zapytań, nie zawierających na końcu URI slasha, np. `GET http://localhost/pliki/katalog`. Z punktu widzenia serwera jest to odwołanie do zasobu `katalog`, w `/pliki/`. Jeżeli jednak serwer od razu zwróci indeks dla danego katalogu, po stronie przeglądarki wystąpi błąd, gdyż wszelkie odnośniki relatywne (np. `styl.css`) w indeksie będą poszukiwane w katalogu `/pliki/` (`/pliki/styl.css`), podczas gdy powinny być wyszukiwane w katalogu `/pliki/katalog/` (`/pliki/katalog/styl.css`). Protokół HTTP proponuje w tym miejscu, by serwer zwracał przeglądarce kod odpowiedzi `301 Moved Permanently`, oraz w nagłówku `Location` zamieszczał poprawny adres, tj. `Location: /pliki/katalog/`. Wówczas przeglądarka ponowi zapytanie z poprawnym adresem. Zachowanie to jest prawie nie widoczne dla użytkowników popularnych przeglądarek, gdyż po otrzymaniu takiej odpowiedzi, korygują one adres w polu edycji adresu dodając brakujący slash. Jednak całe rozwiązanie wymaga dodatkowej komunikacji z serwerem.

Inny mechanizm implementowany przez nas w ramach listingu katalogów, to możliwość domyślnego otwierania wybranego pliku, zamiast listowania zawartości katalogu. Konfigurowany jest poprzez odpowiednie opcje serwera i można określić całą hierarchię plików, np. `index.html`, `index.htm`, `default.htm`.

4 Konfiguracja

Plik konfiguracyjny serwera `sk2httpd.xml` umieszczony w katalogu serwera jest plikiem XML. Zawiera dwa główne typy ustawień: parametry pracy serwera, specyficzne dla całego programu, oraz opcje zasobów, specyficzne dla zasobów które serwer udostępnia.

4.1 Parametry pracy serwera

4.1.1 Numer portu

```
<sk2httpd port="80">
```

Parametr określa numer portu, na którym nasłuchuje serwer. Standardowo dla protokołu HTTP jest to port 80. Opcja ta, inaczej niż pozostałe, definiowana jest jako argument znacznika root dokumentu XML, sk2httpd.

4.1.2 Interfejs sieciowy

```
<option name="interface" content="127.0.0.1"/>
```

Parametr określa, na jakim interfejsie sieciowym ma nasłuchiwać serwer. Jeśli opcja nie jest podana, Serwer działa na wszystkich interfejsach, w przeciwnym wypadku nasłuchuje tylko na tym, który ma adres IP zgodny z tym podanym w argumencie content.

4.1.3 Plik logu

```
<option name="logfile" content="log.txt"/>
```

Parametr określa, w jakim pliku ma być przechowywany dziennik zdarzeń serwera. W dzienniku zdarzeń gromadzona jest historia odwołań do zasobów serwera. Rejestrowane są także błędy, które nie powodowały zakończenia aplikacji. Jeśli parametr nie jest podany, rejestr zdarzeń przekierowywany jest na standardowe wyjście.

4.1.4 Rozmiar kolejki

```
<option name="queuesize" content="20000"/>
```

Rozmiar kolejki zgłoszeń. Jest to liczba zgłoszeń, jakie mogą jednocześnie oczekiwać na odpowiedź. Przy dobieraniu tej wartości należy skorelować ją z liczbą wątków (opcja `threads`) i procesów (opcje `minprocs` i `maxprocs`). Zgłoszenia nie mieszczące się w kolejce będą odrzucane. Dobierając tą wartość należy także zwrócić uwagę na stan ograniczeń systemu (`ulimit`), m.in. ograniczeń otwartych deskryptorów, ponieważ każde otwarte połączenie oczekuje w kolejce.

4.1.5 Liczba wątków

```
<option name="threads" content="50"/>
```

Liczba wątków uruchamianych w ramach jednego procesu. Liczba wątków w ramach procesu jest stała, określa ile wątków w ramach jednej grupy PID będzie się ubiegać jednocześnie o zasoby systemu. Wątki są lekkie i tańsze w obsłudze od procesów, stąd wydaje się być dobrym rozwiązaniem, jeśli liczba wątków na proces jest większa niż liczba procesów.

4.1.6 Minimalna liczba procesów

```
<option name="minprocs" content="2"/>
```

Liczba minimalnie uruchomionych procesów. Niezależnie od obciążenia serwera, zawsze co najmniej tyle procesów będzie udostępniało wątki obsługi zgłoszeń. Liczbę możliwych jednocześnie obsług zgłoszeń (czyli jednoczesnych sesji) jest równa iloczynowi liczby procesów i liczby wątków. Minimalna liczba wątków obsługi to $minprocs * threads$.

4.1.7 maksymalna liczba procesów

```
<option name="maxprocs" content="20"/>
```

Liczba maksymalnie uruchomionych procesów. Jest to górna granica wyznaczająca maksymalne obciążenie serwera. Powyżej tej granicy, nawet jeśli będzie duże obciążenie, serwer http nie będzie alokował dodatkowych zasobów. Należy uważać, aby wartość ta nie była zbyt mała, gdyż może szybko doprowadzić do przepełnienia się kolejki. Z drugiej strony zwiększanie tego parametru, ponad możliwości sprzętu, także nie będzie dobrym rozwiązaniem, gdyż może prowadzić do niestabilności serwera.

4.2 Opcje zasobów

4.2.1 Indeks katalogu

```
<index-file match="nazwa"/>
```

Plik, jaki ma być domyślnie odsyłany na zapytanie o dany katalog. W pliku konfiguracyjnym można podać więcej niż jeden znacznik `index-file`, a wówczas odsyłany będzie pierwszy znaleziony plik z tej listy. Jeśli w katalogu nie będzie żadnego z plików, lub nie w pliku konfiguracyjnym nie będzie żadnego znacznika tego typu, serwer sam wygeneruje listing.

4.2.2 Dopasowywanie zasobów

```
<resource-mapping dirmatch="/sciezka/uri/" path="/sciezka/na/dysku/"/>
```

Dla zapytania, które zawiera ciąg `dirmatch`, będzie zwracany plik z dysku, o ścieżce `path` plus dalszy ciąg URI, który znajdował się po dopasowaniu (`dirmatch`). Np. Dla `/katalog/sciezka/uri/pliki/tekst.pdf` zasób to `/sciezka/na/dysku/pl`
Możliwe jest podanie wielu znaczników tego typu, zawsze będzie zwracane pierwsze poprawne dopasowanie. W pliku konfiguracyjnym musi być podany co najmniej jedno dopasowanie domyślne, zawsze pasujące, najlepiej jako ostatnie:

```
<resource-mapping dirmatch="/" path="/sciezka/na/dysku/"/>
```

4.2.3 Dopasowywanie typów mime

```
<mime-mapping dirmatch="/sciezka/uri/" extmatch=".html" type="text/html"/>
```

Dla zapytania, które zawiera ciąg `dirmatch`, oraz kończy się na `extmatch`, typ mime będzie będzie wyznaczany przez argument `type`. Argumenty `dirmatch` ani `extmatch` nie są obowiązkowe. Jeśli któregoś z nich nie ma, jest zawsze spełniony. Możliwe jest podanie wielu znaczników tego typu, zawsze będzie zwracane pierwsze poprawne dopasowanie. W pliku konfiguracyjnym musi być podany domyślny typ mime, najlepiej jako ostatni:

```
<mime-mapping type="domyslny/typ"/>
```