Uniwersytet w Białymstoku

Wydział Matematyczno-Fizyczny

Instytut Matematyki

Joanna Merecka

Meta-HTML
jako
dynamiczny moduł serwera Apache

Praca dyplomowa napisana
pod kierunkiem
dr hab. K. Prażmowskiego,
prof. UwB

Białystok 2003

Składam serdeczne podziękowania
mgr Mariuszowi Żynelowi
za pomoc przy pisaniu tej pracy.

Joanna Merecka

Spis treści

Wstęp

W ostatnich latach wzrosła gwałtownie popularność technologii internetowych. Internet stał się narzędziem pracy w niemal wszystkich rodzajach ludzkiej działalności. Wiodącą rolę, wśród wszystkich usług dostępnych w sieci Internet, odgrywa *World Wide Web*, w skrócie WWW. Jego zastosowanie jest bardzo wszechstronne, od prostej witryny reklamowej, przez interaktywne sklepy on-line, po zaawansowane aplikacje wyspecjalizowane dla potrzeb różnych instytucji. W niemal wszystkich zastosowaniach istotną rolę odgrywa tzw.*dynamic content*.

Jednym z narzędzi wykorzystywanych przy tworzeniu dynamicznych stron WWW jest Meta-HTML, natomiast Apache jest najpopularniejszym serwerem HTTP. Zwykle interpreter Meta-HTML pracuje jako program CGI. Daje to pewną niezależność od serwera HTTP, ale rozwiązanie to jest mało wydajne. Spodziewana jest znaczna poprawa wydajności, jeśli z interpretera Meta-HTML utworzy się moduł serwera Apache.

W chwili obecnej serwer Apache dostępny jest w dwóch różnych wersjach: 1.3 oraz nowszej, 2.0. W momencie, kiedy rozpoczęliśmy pracę nad modułem, serwer Apache 2.0 znajdował się w bardzo wczesnej fazie testów. Nie było jeszcze ustabilizowanego i udokumentowanego API. Wiadomo było natomiast, że zasadnicze zmiany pomiędzy wersją 1.3, a 2.0 dotyczyć będą koncepcji ładowalnych modułów. Nowe moduły muszą być wielowątkowe (thread-safe) z uwagi na fakt, że Apache 2.0 jest wielowątkowy. Meta-HTML nie spełnia tych wymagań, a przystosowanie będzie wymagało poważnych zmian. Pierwsza działająca wersja Apache 2.0 pojawiła się, gdy nasze prace były daleko zaawansowane nad wersją 1.3. W tej chwili wiemy jak poważne zmiany nastąpiły w realizacji ładowalnych modułów. Niestety przejście do wersji 2.0 wymagałoby zbyt dużych inwestycji. Inwestycje te są na razie nieopłacalne, gdyż wersja 2.0 będzie wolno wchodzić do produkcji, a to z uwagi na fakt, że przepisania wymagają wszystkie moduły niezbędne do pracy serwera HTTP. Wynika to z tego, że żaden z modułów z wersji 1.3 nie może współpracować z wersją 2.0. Zerwana jest tu wyraźnie kompatybilność kolejnych wersji.

Rozdział 1

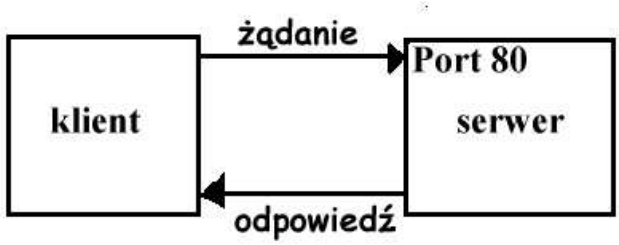
Podstawy HTTP

1.1 Protokół HTTP

HTTP (Hyper Text Transfer Protocol) jest protokołem usługi World Wide Web w warstwie aplikacji (por. [6, 7]), natomiast TCP/IP jest protokołem w warstwie transportu tej usługi. Protokół HTTP to w uproszczeniu protokół wymiany dokumentów hipertekstowych między serwerem, a przeglądarką. HTTP działa wykorzystując zasadę *żądanie–odpowiedź* (*request—reponse*), (rys. 1.1), polegającą na tym, że

- serwer nie może niczego wysłać z własnej inicjatywy,
- serwer odpowiada na żądanie klienta,
- para żądanie-odpowiedź jest wymieniana w ramach jednego połączenia TCP/IP,
- po przekazaniu przez serwer odpowiedzi transmisja zostaje zerwana.

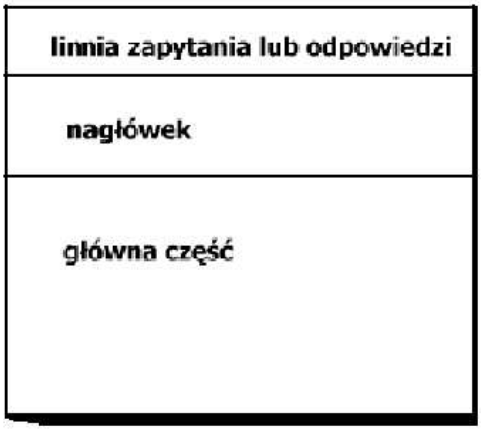
Ostatnia własność zasady żądanie–odpowiedź mówi, że każda interakcja jest niezależna od poprzedniej. Ogólnie, żądanie przeglądarki i odpowiedź serwera nazywa się komunikatami HTTP. Komunikaty te mają ściśle określoną budowę.



Rysunek 1.1: Schemat żądanie–odpowiedź HTTP.

Każdy komunikat składa się z:

1. linii żądania lub statusu,
2. listy nagłówków (headers),
3. treści komunikatu (content).



Rysunek 1.2: Budowa komunikatu HTTP.

O ile lista nagłówków i treść komunikatu mogą być puste, to linia żądania/statusu musi być niepusta. W linii żądania określona jest użyta metoda, ścieżka do żądanego dokumentu oraz wersja stosowanego przez pytającego protokołu. Ogólnie linia zapytania wygląda tak (patrz rys. 1.3):

```
<METODA> <URI> HTTP/<wersja>
```

Rysunek 1.3: Schemat linii żądania (zapytania) HTTP.

Przykład:

```
GET /index.html HTTP/1.0
```

Linia statusu natomiast składa się z numeru wersji protokołu stosowanego przez odpowiadający serwer, numeru statusu oraz jego nazwy symbolicznej np.

```
HTTP/1.1 200 OK
```

Najczęściej stosowane metody protokołu HTTP to GET i POST[1]. Od użytej metody zależy budowa komunikatu. Nagłówki mogą zawierać:

- charakterystyki przeglądarek i serwerów,
- język preferowany(zwrócony),
- dane,
- cechy dokumentu (np. typ i rozmiar),
- wartości i wymóg ustawienia *cookie*.

Najczęściej spotykane odpowiedzi serwera to:

`HTTP_OK` (kod 200) – DOCUMENT_FOLLOWS – Zapytanie powiodło się, serwer wysyła odpowiedź, która zawiera żądane dane.

`HTTP_MOVED_PERMANENTLY` (kod 301) – MOVED – Wskazany dokument został przeniesiony, a żądany URI nie jest już używany przez serwer. Nowy URI określony jest w nagłówku `Location`.

`HTTP_MOVED_TEMPORARILY` (kod 302) – REDIRECT – Wskazany dokument został tymczasowo przeniesiony w inne miejsce.

`HTTP_BAD_REQUEST` (kod 400) – BAD_REQUEST – Niepoprawne zapytanie. Serwer wykrył błąd w składni zapytania.

`HTTP_UNAUTHORIZED` (kod 401) – AUTH_REQUIRED – Brak autoryzacji. Klient nie podał poprawnie danych, które umożliwiają uwierzytelnienie.

`HTTP_NOT_FOUND` (kod 404) – NOT_FOUND – Żądany dokument nie istnieje. Serwer nie może go odnaleźć.

`HTTP_INTERNAL_SERVER_ERROR` (kod 500) – SERVER_ERROR – Wewnętrzny błąd serwera np. program CGI zawiesił się lub wystąpił w nim błąd konfiguracyjny.

[1] w ciągu całego roku 2002 do serwera math wpłynęło 368735 żądań, z czego 358277 to były żądania GET, co daje 97%.

1.2 Metoda GET

GET jest zapytaniem o informacje znajdujące się pod określonym adresem URI (Uniform Resource Identifier) na serwerze. Przeglądarka korzysta z tej metody do pobierania dokumentów z serwera HTTP. Formalnie GET jest żądaniem określonego pliku. Oto przykład kompletnego żądania metodą GET.

```
GET /index.html HTTP/1.0
Connection: Keep-Alive
User-Agent:Mozilla/2.02Gold (WinNT; I)
Host:www.uwb.edu.pl
Accept:image/gif, image/x-xbitmap, image/jpeg, */*
```

Jeśli żądany plik, w tym przypadku `index.html` istnieje na serwerze, to odpowiedź serwera może wyglądać następująco:

```
HTTP/1.0 200 OK
Date: Fri, 20 Sep 2002 09:12:56 GMT
Serwer: NCSA/1.5.2
Last-modified: Mon, 19 Sep 2002 12:00:01 GMT
Content-type: text/html
Content-lenght: 41

<html>
<body>
<h1>Test</h1>
</body>
</html>
```

W przypadku metody GET w protokole HTTP 1.0 wszystkie nagłówki są opcjonalne i niosą dodatkową informację o przeglądarce. W przypadku HTTP 1.1 wymagany jest nagłówek `Host`.

1.3 Metoda POST

Ideologiczna różnica pomiędzy metodami GET i POST jest taka, że o ile metoda GET służy do pobierania dokumentu, to przy pomocy metody POST wysyłane są dane z przeglądarki do serwera, a właściwie do programu działającego po stronie serwera, obsługującego zapytanie. W przypadku obu metod GET i POST mamy do czynienia z dwoma rodzajami danych, które trafiają z przeglądarki do serwera:

- właściwości przeglądarki, umieszczone w nagłówkach i analizowane przez serwer, oraz
- dane ignorowane przez serwer, biernie przekazywane do programów CGI i modułów obsługujących żądania.

Drugi typ danych można przekazać metodą GET w linii żądania jako fragment URI za ścieżką do konkretnego pliku np.

```
GET /publikacje/lista.mhtml?pracownik=53 HTTP/1.0
```

W ten sposób możemy jednak przesłać niewiele danych ze względu na ograniczenia rozmiaru URI. Większą ilość danych możemy przekazać metodą POST. To samo zapytanie, co wyżej, przekazane metodą POST miałoby postać:

```
POST /publikacje/lista.mhtml HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/2.02 Gold (WinNT; I)
Host: www.uwb.edu.pl
Accept: image/gif, image/x-xbitmap, image/jpeg, */*
Content-lenght: 12
Content-type: application/x-www-from-urlencoded
pracownik=53
```

Ta metoda pozwala przesyłać dane ze złożonych formularzy.
W metodzie POST obowiązkowy jest nagłówek `Content-lenght` mówiący o ilości wysłanych danych w bajtach. W powyższym przykładzie ten rozmiar to 12.

Innym ważnym argumentem przeważającym za użyciem metody POST, poza ilością danych, jest fakt, że dane przesłane przy pomocy GET widoczne są w przeglądarce razem z adresem strony, natomiast dane wysłane za pomocą POST nie są widoczne dla użytkownika, co ważniejsze mogą one być zaszyfrowane przed transmisją, jeśli używamy protokół HTTPS, tzn. szyfrowany protokół HTTP.

Tak więc np. podczas logowania użytkownika dane z formularza zawierającego hasła powinny być wysłane metodą POST, najlepiej w protokole HTTPS.

Rozdział 2

Modularna budowa i działanie serwera Apache 1.3

2.1 Biblioteki dynamiczne

Jednym ze sposobów oszczędzania zasobów komputera, zarówno jeśli chodzi o pamięć masową jak i operacyjną, a także podnoszenia wydajności systemu jest stosowanie bibliotek dynamicznych. W przypadku systemu Unix mówimy o DSO, czyli Dynamic Shared Object, w Windows o DLL czyli Dynamically Linked Libray. Mimo różnic w terminologii i wewnętrznej implementacji zasada pozostaje taka sama w obu przypadkach, a mianowicie część kodu wspólnego dla różnych programów umieszczona jest poza programem, w bibliotece. Ta prosta koncepcja ma daleko idące konsekwencje dla systemu. Po pierwsze oszczędzamy w ten sposób miejsce na dysku, po drugie zmniejszamy zużycie pamięci operacyjnej, gdyż, co najwyżej jedna kopia biblioteki dynamicznej może znajdować się w tej pamięci, a jednocześnie wiele programów może korzystać z jej kodu.

Główną wadą bibliotek statycznych jest to, że jeśli chcemy ulepszyć taką bibliotekę to trzeba zrekompilować aplikację, aby ją wykorzystywać. W przypadku biblioteki dynamicznej wystarczy, zrekompilować samą bibliotekę. Statyczne biblioteki są linkowane w program wykonywalny przez linkera w procesie kompilacji programu. Rozmiar ostatecznego kodu wzrasta przez ilość kodu, jaką linker musi umieścić z bibliotek. Dynamiczne biblioteki nie są linkowane w programie wykonywalnym w czasie kompilacji, lecz podczas uruchamiania programu poprzez system operacyjny.

Dynamiczne biblioteki są znacznie lepsze niż biblioteki statyczne i powinny być użyte gdziekolwiek jest to możliwe, z następujących powodów:

- Programy współdzielą dynamiczne biblioteki, ale nie współdzielą statycznych. Ponieważ wiele z powszechnych bibliotek jest załadowanych do systemu, oraz wiele różnych programów używa tych bibliotek, więc dynamiczne biblioteki znacznie oszczędzają zużycie pamięci.

- Biblioteka dynamiczna nie jest linkowana w program wykonywalny i przez to program wykonywalny jest znacznie mniejszy. Oszczędza się w ten sposób miejsce na dysku, a program wykonywalny jest szybciej ładowany do pamięci podczas uruchamiania.

- Biblioteki dynamiczne mogą być ładowane na żądanie programu w trakcie jego działania. Oznacza to, że z poziomu uruchomionej aplikacji możemy ładować i zwalniać dodatkowy kod, co pozwala dynamicznie modyfikować funkcjonalność aplikacji, bez potrzeby jej rekompilacji.

Niestety biblioteki dynamiczne nie są doskonałe.

- Potrzebny jest dodatkowy czas procesora by załadować i zlinkować biblioteki podczas pracy systemu. Także nierelokowalny kod może być wykonywany wolniej niż kod relokowalny w statycznych bibliotekach.

- Jeśli program wykonywalny i biblioteki są w różnych systemach plików., a partycja, na której są niezbędne biblioteki jest niedostępna to aplikacja jest również niedostępna. Z tego właśnie powodu powłoka (shell) root'a jest linkowana statycznie.

Technika dynamicznie ładowalnych bibliotek przez uruchomiony program, została wykorzystana w serwerze Apache. Mówi się dlatego, że ma on budowę modularną. Moduły serwera Apache to dynamiczne biblioteki. Dobrze zaprojektowane API serwera Apache powoduje, że stosunkowo łatwo i szybko można wzbogacić go o dodatkowe funkcje i możliwości. Do tych najbardziej popularnych i dość dobrze znanych należą `mod_perl` zwiększający wydajność programów CGI napisanych w Perlu, oraz PHP —bardzo popularny interpreter skryptów włączanych do HTML, wykonywanych po stronie serwera.

2.2 Porównanie — CGI a moduł

Common Gateway Interface, w skrócie CGI, jak sama nazwa mówi to zestandaryzowany sposób komunikacji serwera HTTP z niezależnymi programami realizującymi żądania HTTP i generującymi odpowiedzi. Program CGI to dowolny, wykonywalny program, od skryptu shell-owego, przez skrypty w Perlu, do skompilowanych programów pisanych w dowolnym języku kompilowanym. Aby program mógł być programem CGI musi spełniać dwa warunki:

- musi działać na tej samej platformie, co serwer HTTP, oraz

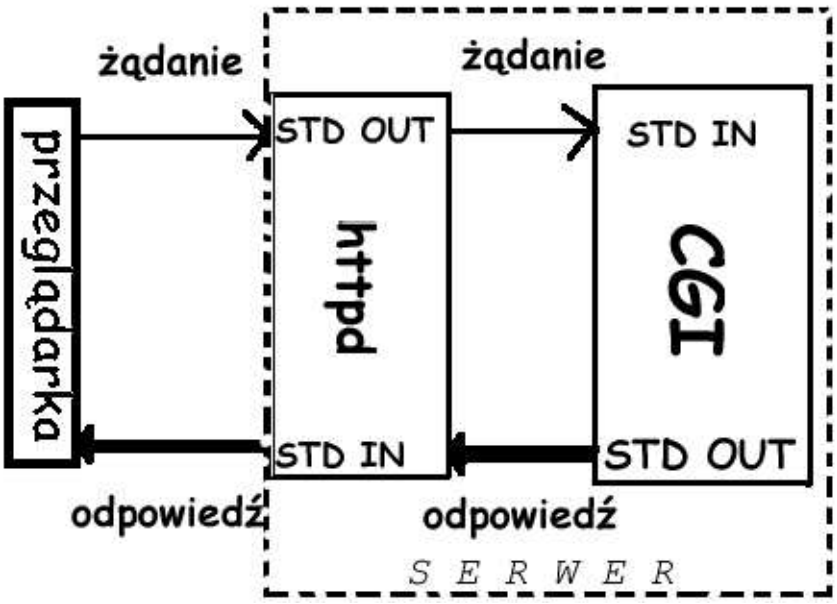
- musi umieć komunikować się z serwerem HTTP w odpowiedni sposób.

Pierwszy warunek ogranicza wybór kompilatora, czy interpretera.
Komunikacja pomiędzy programem CGI a serwerem HTTP odbywa się za pośrednictwem standardowego wejścia i wyjścia. Serwer przekazuje kompletne

żądanie, jakie otrzymał od przeglądarki, natomiast program CGI na podstawie tego żądania generuje odpowiedź, którą z kolei serwer przekaże do przeglądarki. Odpowiedź to dokument lub inny zbiór danych obudowany nagłówkami HTTP. Wśród nagłówków zwracany jest typ przekazywanego dokumentu lub zbioru danych oraz jego rozmiar w bajtach. Najczęściej w odpowiedzi od programu CGI przeglądarka dostaje dokument HTML, ale może to być bitmapa, skompresowany plik lub plik multimedialny.

Może się wydawać, że przy takim scenariuszu serwer HTTP jest niemal bezczynny. Otóż jest to złudzenie, gdyż wykonuje on niemal dokładnie tę samą pracę, co w przypadku statycznego dokumentu HTML. Różnica polega na tym, że dokument statyczny to plik znajdujący się w systemie plików, a odpowiedź z programu CGI to plik specjalny. W pierwszym przypadku serwer HTTP sam musi zadbać o dołączenie nagłówków HTTP.

Program CGI uruchamiany jest przez serwer HTTP, gdy wymaga tego realizacja żądania. Po uruchomieniu, na standardowym wejściu CGI odbiera żądanie, tworzy odpowiedź i przekazuje ją serwerowi na standardowym wyjściu (rys.2.1). Po czym program kończy działanie i zwalnia zasoby.



Rysunek 2.1: Schemat współdziałania programu CGI z serwerem HTTP.

Jak widać ze schematu działania CGI programy CGI działają niezależnie od serwera HTTP. W zasadzie mogą być uruchamiane bez niego np. w celu testów. Jest to poważna zaleta, gdyż możliwe jest użytkowanie jednego programu we współpracy z serwerem HTTP i osobno. Interakcja pomiędzy programem CGI, a serwerem HTTP jest mocno ograniczona. Użycie standardowego wejścia i wyjścia, czyli poprzez pliki mocno spowalnia wymianę danych, natomiast

uruchomienie nowego procesu i jego zakończenie jest w skali czasu działania tego procesu długotrwałe i poważnie obciąża system operacyjny.

Moduły serwera Apache mogą realizować te same zadania, co programy CGI. Nie ma tutaj ograniczeń funkcjonalnych, jedynie implementacyjne. Moduł jest biblioteką dynamiczną na platformie, na której działa serwer Apache. Biblioteka zawiera kod w języku maszyny zatem jest to plik binarny. Zwykle powstaje w procesie kompilacji na podstawie kodu w języku wyższego poziomu (C, Pascal, C++, Java). Skrypty z natury rzeczy to pliki tekstowe, interpretowane w trakcie uruchamiania (można powiedzieć kompilacja w locie na zawołanie). Stąd rozróżnienie na kompilatory i interpretery. Przykładem kompilatora jest gcc, natomiast przykładami interpreterów są Shell, Perl, Meta-HTML, PHP. Na ogół ze skryptu nie da sie wygenerować biblioteki, w sensie systemu Unix np. Solaris. Wyjątek stanowi Java. Program napisany w Java może być interpretowany przez tzw. maszynę wirtualną lub może być skompilowany. Tak więc żaden z języków interpretowanych nie nadaje się, aby napisać w nim moduł serwera Apache. Nie ma natomiast ograniczeń w przypadku kompilatorów, gdyż niezbędny do utworzenia biblioteki linker jest na ogół dołączany do systemu operacyjnego[1]. W przypadku Apache 2.0 moduł musi być wielobieżny (thread-safe). Wymusza to na autorze modułu odpowiednią dyscyplinę podczas implementacji. W Apache 1.3 wystarcza, aby moduł zwalniał zasoby po przetworzeniu żądania. Zaleca się oczywiście, aby biblioteki były wielobieżne.

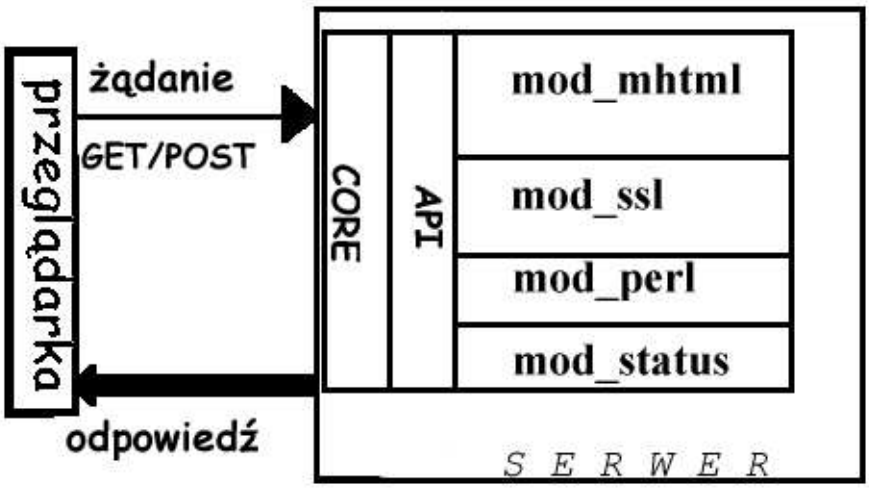
Moduły, które realizują żądania i generują odpowiedź nazywa się modułami obsługi żądań (`request handlers`). Moduł `mod_mhtml` jest typowym przykładem tego rodzaju modułów. Innym przykładem jest moduł PHP.

Fakt, że moduł jest integralną częścią serwera pociąga za sobą dwie istotne konsekwencje:

- w trakcie obsługi żądania nie jest konieczne tworzenie nowego procesu w systemie, a niezbędne zasoby są przydzielane tylko raz podczas startu serwera Apache,
- komunikacja moduł–serwer odbywa się poprzez funkcje API [3] serwera Apache, czyli bezpośrednio poprzez pamięć procesu (elementy API serwera Apache zostały zebrane na stronie 20).

Obie z wymienionych właściwości modułu skracają czas potrzebny na obsłużenie żądania. Schemat obsługi żądania przez moduł przedstawiono na rys. 2.2.

[1]Dotyczy to Solaris, Linux i innych systemów Unix. W przypadku Windows linker nie jest zawarty w systemie.



Rysunek 2.2: Schemat współpracy modułu z serwerem Apache.

Podsumowując, zarówno CGI jak i moduły mają swoje unikalne zalety. Jeśli jednak za decydujący czynnik wybrać wydajność serwera HTTP, to lepszym okazuje się moduł.

Rozdział 3

Kompilacja i uruchamianie modułu

3.1 Kompilacja

Kompilacja jest dość złożonym procesem, szczególnie gdy kompilujemy bibliotekę dynamiczną. Jednym z programów wspomagających programistę jest narzędzie `make`. Program ten sprawdza zależności pomiędzy elementami kodu i woła kompilator, linker i inne niezbędne programy do kompilacji na podstawie tzw. `Makefile`.

Do kompilacji `mod_mhtml` używany jest następujący plik `Makefile`:

```
# Simplistic Makefile for mod_mhtml (Apache versions 1.3x)
#
APXS            = apxs
APXS_STDFLAGS = -a -c
APXS_INCFLAGS = -I ../.. -I ../../libutils
APXS_MHTML      = -L ../../libmhtml -l mhtml
APXS_SERVE      = -L ../../libserver -l server
APXS_UTILS      = -L ../../libutils -l utils
#APXS_CRYPT      = -L ../../../libdes-4.04b -l des
APXS_GDBM       = -L/usr/local/lib -Wl,-R/usr/local/lib  -l gdbm
APXS_LIBFLAGS = $(APXS_MHTML) $(APXS_SERVE) $(APXS_UTILS) \
                  $(APXS_GDBM) $(APXS_CRYPT)
APXS_EXTRAFLAGS = $(APXS_INCFLAGS) $(APXS_LIBFLAGS)
APXS_OPTIMIZATION = -Wc,-O4 -Wc,-march=i686 -Wc,-mcpu=i686\
        -Wc,-fomit-frame-pointer
APXS_COMPILE_FLAGS = $(APXS_STDFLAGS) $(APXS_EXTRAFLAGS) \
        $(APXS_OPTIMIZATION)
APXS_INSTALL_FLAGS = -i

all: mod_mhtml.so

mod_mhtml.so: mod_mhtml.c mod_mhtml.h ../../libmhtml/libmhtml.a \
        ../../libutils/libutils.a  Makefile
        $(APXS) $(APXS_COMPILE_FLAGS) mod_mhtml.c

install: all
        $(APXS) $(APXS_INSTALL_FLAGS) mod_mhtml.so

realclean clean distclean: FORCE
        rm -f *.so *.o *~
        FORCE:
```

Należy zwrócić uwagę na to, że nigdzie w tym pliku nie jest użyty explicite kompilator, ani linker. Kompilacja wykonywana jest przez program APXS dołączony do serwera Apache. Jego zadaniem jest wywołanie kompilatora (np. gcc) i linkera z odpowiednimi parametrami tak, aby utworzyć standardowy moduł Apache. Ten sposób kompilacji jest zalecany przez autorów Apache [2].

3.2 Uruchamianie i debugging

Powstająca w wyniku kompilacji kodu `mod_mhtml.c` biblioteka dynamiczna `mod_mhtml.so` jest to kompletny silnik Meta-HTML, wyposażony w interfejs pozwalający mu współpracować z serwerem Apache. Ten interfejs napisany jest w `mod_mhtml.c`. Silnik Meta-HTML "montowany" jest do modułu w trakcie kompilacji, formalnie do `mod_mhtml.so` linkowane są statyczne biblioteki Meta-HTML: `libmhtml.a` i `libutils.a`, zawierające wszystkie funkcje jądra Meta-HTML. Warto tu wspomnieć, że Meta-HTML ma możliwość rozbudowy o swoje własne biblioteki dynamiczne, tzw. *moduły Meta-HTML*. Funkcje w nich zawarte (np. matematyczne `sin()`, `cos()`) nie są linkowane w `mod_mhtml.so` w trakcie kompilacji, lecz w trakcie uruchamiania serwera Apache i inicjalizacji modułu `mod_mhtml`.

Przy pisaniu modułu wzorowaliśmy się na przykładowych modułach dostarczanych z oprogramowaniem Apache i zawiera odwołania do API Apache. Opisy wołanych funkcji API serwera Apache zostały zebrane w dodatku A. Kod modułu zawiera również fragmenty kodu interpretera Meta-HTML. Autorem Meta-HTML jest Brian Fox, z którym korespondowaliśmy w trakcie prac nad modułem. Opisy funkcji Meta-HTML wykorzystanych bezpośrednio w `mod_mhtml.c` umieszczone są w dodatku B.

W odróżnieniu od zwykłego programu, czy skryptu, uruchamianie biblioteki jest dużo trudniejsze i wymaga większego nakładu pracy. Aby w ogóle zacząć testy moduły musieliśmy odpowiednio skonfigurować serwer Apache, tak aby używał modułu `mod_mhtml`. Ponadto napisaliśmy testowe skrypty w języku Meta-HTML, testujące wszystkie ważne funkcje interpretera.

Konflikt w bibliotece `regexp`

Program właściwy serwera Apache (`httpd`) używa funkcji `regexec` i pochodnych. W module `mod_mhtml` również używana jest funkcja o tym samym symbolu (nazwie), ale z inną realizacją niż w `httpd`. Powstaje, zatem konflikt. Problem ten objawił się w późniejszej fazie pracy nad modułem `mod_mhtml`, kiedy zaczęliśmy uruchamiać program w Meta-HTML używający regularnych wyrażeń. Jak widać był to problem w czasie pracy modułu, nie przy jego kompilacji. Aby rozwiązać ten problem należało zmienić nazwę `regexec` na inną np. `mhtml_regexec` tak, aby uniknąć konfliktu. Najprościej można to zrobić

korzystając z dobrodziejstwa preprocesora języka C. W tym celu został utworzony plik `regex-extra.h` o następującej treści:

```
#undef regexec
#undef regerror
#undef regfree
#undef regcomp
#define regexec mhtml_regexec
#define regerror mhtml_regerror
#define regfree mhtml_regfree
#define regcomp mhtml_regcomp
```

Natomiast w `regex.h` dołączono dyrektywę:

```
#include "regex-extra.h"
```

Powoduje to podstawienie za nazwę `regexec` nazwy `mhtml_regexec` przez preprocesor w obrębie kodu modułu `mod_mhtml`.

Obsługa metody POST

Sporo trudności, podczas uruchamiania modułu, mieliśmy z obsługą metod typu POST. Kłopot polega na tym, że aby poprawnie takie żądanie obsłużyć, należy pobrać od klienta (przeglądarki) dodatkowe dane. W przypadku modułu, musimy to zrobić za pośrednictwem serwera Apache. Z powodu nader skąpej dokumentacji sposobu obsługi żądań POST przez API Apache, zmuszeni byliśmy poszukiwać wzorców w działających modułach. Doskonałym przykładem okazał się moduł PHP, z którego został zaczerpnięty pomysł na obsługę POST.

Pobranie danych od klienta rozpoczyna się wywołaniem funkcji z API Apache `ap_setup_client_block()`, kóra inicjalizuje odpowiednie struktury danych odpowiedzialne za przekazanie danych. Sam odczyt danych POST wykonywany jest w funkcji `mhtml_read_content_from_ap()`. Najpierw od klienta pobierana jest długość bloku danych do przeczytania i jednocześnie klient jest zapraszany do wysłania tych danych. Odpowiedzialna jest za to funkcja `ap_should_client_block()`. Następnie funkcja `apache_read_post()` pobiera dane od klienta wołając w pętli `ap_get_client_block()` do momentu osiągnięcia końca bloku danych, lub pojawienia się błędu transmisji.

Sesje

Najpoważniejszą usterką, z którą borykaliśmy się najdłużej, było nieprawidłowe działanie *sesji*.

Sesją określa się zestaw danych trzymanych w postaci par *klucz*, *wartość* po stronie serwera. Każda sesja ma swój unikalny identyfikator SID (Session

Identifier). Przy pomocy mechanizmu *Cookie*, SID jednoznacznie identyfikuje przeglądarkę, która sesję zainicjowała. Pozwala to na komunikację pomiędzy kolejnymi transmisjami żądanie–odpowiedź.

Sztandarowym przykładem zastosowania mechanizmu Cookie i sesji jest bank internetowy. Po zalogowaniu się użytkownika "serwer" pamięta nazwę użytkownika, aż do momentu wylogowania się, mimo że pomiędzy zalogowaniem a wylogowaniem otwieranych jest wiele różnych stron.

Sesje nie są niezbędne do prawidłowego działania samego interpretera. Wpływają jednak na to jakie zadania można przy jego pomocy rozwiązać. W naszym wypadku "bank" napisany w Meta-HTML i obsługiwany przez moduł, nie działałby prawidłowo. Działałby natomiast prawidłowo z Meta-HTML jako CGI.

Problem z sesjami tkwił w tym, że nie były resetowane wszystkie struktury globalne w module. Dokładnie chodzi tu o listę zmiennych środowiskowych `new_env`, na której w szczególności znajduje się SID. Ponieważ nie była ona zerowana po obsłużeniu żądania, różne przeglądarki dostawały jeden SID i w efekcie klienci "banku" mogli zaglądać do cudzych kont.

Rozdział 4

Działanie, instalacja i użytkowanie modułu

4.1 Proces `httpd`

W serwerze Apache na platformie UNIX wykorzystano mechanizm *preforking*. Funkcja `fork()` jest to standardowa funkcja systemu operacyjnego UNIX do tworzenia nowych procesów [5].

Demon serwera Apache (`httpd`) po uruchomieniu woła funkcję `fork()`, do utworzenia kilku swoich procesów potomnych. Procesy potomne są to dokładne kopie procesu macierzystego tzn. kopiowany jest:

- segment kodu, oraz
- segment danych (zmienne i stałe globalne).

Nie jest kopiowany segment stosu (zmienne lokalne, adresy skoków do funkcji).

Efekt działania `fork()` widać na liście procesów `httpd`:

```
  UID   PID  PPID  C    STIME TTY      TIME CMD
 root 18382     1  0   kwi 06 ?        0:01 /usr/local/sbin/httpd
nobody 18407 18382  0   kwi 06 ?        0:26 /usr/local/sbin/httpd
nobody 18395 18382  0   kwi 06 ?        0:25 /usr/local/sbin/httpd
nobody 18408 18382  0   kwi 06 ?        0:25 /usr/local/sbin/httpd
nobody 18700 18382  0   kwi 06 ?        0:25 /usr/local/sbin/httpd
nobody 18471 18382  0   kwi 06 ?        0:25 /usr/local/sbin/httpd
nobody 18392 18382  0   kwi 06 ?        0:29 /usr/local/sbin/httpd
nobody 18394 18382  0   kwi 06 ?        0:25 /usr/local/sbin/httpd
nobody 18393 18382  0   kwi 06 ?        0:26 /usr/local/sbin/httpd
nobody 18474 18382  0   kwi 06 ?        0:23 /usr/local/sbin/httpd
nobody 18396 18382  0   kwi 06 ?        0:26 /usr/local/sbin/httpd
```

`UID` (User Identifier) oznacza tutaj nazwę użytkownika. Z przywilejami tego użytkownika działa dany proces. `PID` (Process Identifier) to indyfikator procesu, natomiast `PPID` (Parent Process Identifier) to indyfikator procesu rodzicielskiego, tzn. tego, który dany proces uruchomił. Proces `httpd` o numerze `PID 18382` jest w tym wypadku procesem macierzystym działający z uprawnienia `root'a`. Jego rodzicem jest proces `scheduler'a` o numerze `PID 1`. Pozostałe procesy `httpd` mają `PPID` równe `18382`, a więc są procesami potomnymi, czas powstania ich jest późniejszy niż czas procesu macierzystego i jeśli proces `18382` zostanie zamknięty wszystkie procesy potomne też zostaną zamknięte.

Głównym zdaniem procesu macierzystego jest odbieranie żądań przychodzących z Internetu i przydzielanie procesów potomnych do obsługi tych żądań. Proces macierzysty nie obsługuje żadnych żądań. Proces potomny po obsłużeniu żądania nie jest zwalniany (likwidowany) i czeka na nowe żądania.

4.2 Realizacja żądania przez moduł `mod_mhtml`

W przypadku, gdy żądanie dotyczy dokumentu mhtml i używany jest moduł Meta-HTML, ta obsługa żądania wygląda następująco:

1. Wywołanie funkcji `mhtml_handle_req()` odpowiedzialnej za obsługę żądania.

2. Ustalenie metod obsługiwanych przez `mod_mhtml`. Są to GET i POST. Pozostałe nie są obsługiwane przez ten moduł i zgłaszany jest dla nich błąd – metoda niedozwolona.

3. Sprawdzenie czy nie ma danych do przeczytania w przypadku metody POST.

4. Inicjalizacja struktury `req` przy pomocy funkcji `http_parse_request()`, która ustala metodę, URI, protokół i wersję protokołu.

5. Ustalenie danych o kliencie, tzn. adresu IP i nazwy hosta.

6. Ustawienie zmiennych środowiskowych CGI (por. `ap_add_cgi_vars()` i `ap_add_common_vars()` na stronie 20).

7. Przekopiowanie wszystkich nagłówków z żądania HTTP do struktury `req->headers` Meta-HTML.

8. Wywołanie `initialize_engine()`. Podłączenie modułów Meta-HTML, wczytanie makr napisanych w Meta-HTML i odpowiednie wypełnienie listy `ALLPackages`, w której spisane są wszystkie funkcje i zmienne Meta-HTML.

9. Ustawienie zmiennych Meta-HTML niezbędnych ze względu na kompatybilność z poprzednimi wersjami języka Meta-HTML.
10. Wywołanie funkcji `mhtml_set_remote_user()`, która pobiera z żądania nazwę użytkownika oraz hasło o ile występuje nagłówek `Authorization`.
11. Przygotowanie Meta-HTML do przetworzenia żądanego dokumentu, inicjalizacja struktury `result` za pomocą funkcji `mhtml_make_result()`.
12. Ustalenie lokalizacji żądanego dokumentu na podstawie URI z żądania, wywołanie `mhttpd_resolve_location()`.
13. Wczytanie danych przesłanych metodą POST, wykonywane w funkcji `mhtml_read_content_from_ap()`.
14. Właściwe przetworzenie żądanego dokumentu i wygenerowanie strony HTML, wywołanie funkcji `mhttpd_metahtml_engine()`.
15. Obudowanie odpowiedzi odpowiednimi nagłówkami HTTP.
16. Zwolnienie pamięci zajmowanej przez strukturę `req`.
17. Wysłanie odpowiedzi do klienta.
18. Ustawienie kodu odpowiedzi i przekazanie go do jądra serwera Apache (por. strona 31).
19. Zwolnienie pamięci zajmowanej przez strukturę `result` oraz wszystkie makra i zmienne Meta-HTML poza prymitywami Meta-HTML
20. Wyczyszczenie bufora informacji o błędach Meta-HTML tak, aby przy kolejnych żądaniach bufor był pusty.
21. Zakończenie obsługi żądania, zwrócenie odpowiedniego kodu wyniku do jądra serwera Apache. Może to być `OK`, gdy żądanie zostało obsłużone pomyślnie, `REDIRECT`, gdy wystąpiła redyrekcja (np. przez wywołanie makra `<redirect>`), lub kod błędu w pozostałych przypadkach.

4.3 Użytkowanie modułu `mod_mhtml`

Kod źrodłowy Meta-HTML, wraz z napisanym przez nas modułem `mod_mhtml` można pobrać ze strony [8]. Kompilacja Meta-HTML przebiega zgodnie ze standardem GNU. Po skompilowaniu, moduł `mod_html`, podobnie jak inne moduły serwera Apache, umieszcza się w katalogu `/opt/libexec`. Do działania tego modułu wymagane są jego własne biblioteki, które zwykle umieszcza sie w katalogu `/opt/metahtml/lib`.

Aby moduł `mod_mhtml` został załadowany przez serwer Apache podczas startu należy do konfiguracji serwera (`httpd.conf`) dopisać następujące linie:

```
LoadModule mhtml_module      libexec/mod_mhtml.so
AddModule mod_mhtml.c
```

Standardowym rozszerzeniem dokumentów napisanych w Meta-HTML jest `.mhtml`. W konfiguracji Apache należy przypisać temu rozszerzeniu (lub innemu, którego chcemy używać z modułem) typ `metahtml/interpreted`. Dokumenty tego typu będą obsługiwane przez moduł `mod_mhtml`.

Do konfiguracji Apache dodajemy linię:

```
AddType metahtml/interpreted .mhtml
```

Warto nadać specjalne znacznie dla dokumentu `index.mhtml`, który będzie plikiem startowym katalogu:

```
DirectoryIndex index.html index.mhtml welcome.mhtml index.php
```

W tym wypadku najwyższy priorytet ma plik `index.html`, a najniższy `index.php`.

Moduł `mod_mhtml` jest konfigurowalny z poziomu konfiguracji serwera Apache, tak jak inne moduły. Do dyspozycji mamy następujące dyrektywy:

`Meta-HTMLModuleDirectories` — lista ścieżek, w których znajdują się biblioteki modułu `mod_mhtml`,

`Meta-HTMLPublicHTML` — nazwa katalogu, w których zwyczajowo użytkownicy zakładają swoje strony prywatne (por. z dyrektywą `UserDir` serwera Apache),

`Meta-HTMLDirectoryIndex` — lista nazw plików, które są plikami startowymi katalogów (por. z `DirectoryIndex` powyżej),

`Meta-HTMLRequireDirectories` — lista katalogów, w których znajdują się biblioteki napisane w języku Meta-HTML,

`Meta-HTMLExecPath` — ścieżka `PATH`, której używa moduł; istotna, gdy z poziomu modułu uruchamiamy inne programy,

`Meta-HTMLFileExtensions` — lista rozszerzeń plików, które będą interpretowane przez moduł.

Wszystkie dyrektywy konfiguracji modułu są opcjonalne, a ich domyślne wartości są następujące:

```
<IfModule mod_mhtml.c>
  Meta-HTMLModuleDirectories /usr/lib/metahtml /usr/local/metahtml/lib
  Meta-HTMLPublicHTML public_html
  Meta-HTMLDirectoryIndex welcome.mhtml index.mhtml
  Meta-HTMLRequireDirectories lib tagsets macros include
  Meta-HTMLExecPath /usr/bin:/usr/local/bin:/usr/local/metahtml/bin
  Meta-HTMLFileExtensions .mhtml
</IfModule>
```

Dodatek A

API serwera Apache

Poniżej przedstawione są funkcje API serwera Apache 1.3 użyte w module `mod_mhtml`.

`void ap_add_cgi_vars(request_rec *r)`
: Zapamiętuje zmienne związane ze skryptami CGI w środowisku podprocesu (potomka) przez ustawienie ich w tabeli o nazwie `subprocess_env` w rekordzie `request_rec`. Zapamiętywane zmienne to:

  - GATEWAY_INTERFACE
  - PATH_INFO – dodawana jest wtedy i tylko wtedy, jeśli w URI żądania jest path-info
  - PATH_TRANSLATED
  - QUERY_STRING
  - REQUEST_METHOD
  - REQUEST_URI
  - SCRIPT_NAME
  - SERVER_PROTOCOL

`void ap_add_common_vars(request_rec *r)`
: Dodaje standardowe zmienne do środowiska podprocesu (potomka) poprzez wpisanie ich do tabeli o nazwie `subprocess_env`. Te zmienne to:

  - CONTENT_TYPE
  - CONTENT_LENGTH
  - PATH
  - vSERVER_NAME
  - SERVER_PORT
  - REMOTE_HOST

- REMOTE_ADDR
- DOCUMENT_ROOT
- SERVER_ADMIN
- SCRIPT_FILENAME
- REMOTE_PORT
- REMOTE_USER
- REMOTE_IDENT

Jeśli żądanie jest rezultatem przekierowania, to następujące zmienne będą także ustawione:

- REDIRECT_QUERY_STRING
- REDIRECT_URL

`void ap_add_version_component(const char *component)`
Przekazany argument dodawany jest do łańcucha używanego jako wartość pola `Server` w nagłówku odpowiedzi. Funkcja może być wołana wyłącznie w fazie inicjalizacji modułu, a podana wartość musi być albo poprawnym łańcuchem wersji komponentu (tzn. "component-name/n.n"), albo komentarzem ujętym w nawiasy (tzn. "(comment)"). To czy dodatkowa identyfikacja serwera będzie używana w nagłówkach odpowiedzi serwera zależy od ustawienia dyrektywy `ServerTokens` w konfiguracji serwera.

`const char *ap_document_root(request_rec *r)`
Funkcja zwraca ścieżkę podaną w dyrektywie `DocumentRoot` w konfiguracji serwera lub wirtualnego hosta, do którego dane żądanie jest skierowane.

`char *ap_get_server_name(const request_rec *r)`
Zwraca nazwę hosta, na którym działa serwer Apache. Są dwie możliwości na to, co oznacza nazwa hosta. Jest to nazwa kanoniczna (canonical) zdefiniowana przez `SerwerName` i `Port`, albo też, wartość podana w nagłówku `Header` lub w URI zapytania.

`unsigned ap_get_server_port(const request_rec *r)`
Zwraca numery portu TCP, na którym nasłuchuje serwer Apache. Przekazany port w nagłówku przez klienta nie jest wiarygodny. Używany jest port właściwego gniazda (socket).

`void ap_kill_timeout(request_rec *r)`
Zeruje zmienną `timeout` w rekordzie `request_rec`.

`int ap_rflush(request_rec *r)`
Przesłanie zawartości bufora odpowiedzi do klienta.

`int ap_rwrite(const void *buf, int nbyte, request_rec *r)`
Podany łańcuch dopisywany jest do bufora, w którym przechowywana jest odpowiedź serwera.

`ap_setup_client_block()`
Przygotowuje moduł do odbioru danych od klienta, zwykle gdy klient wysłał żądanie PUT lub POST. Funkcja powinna być wołana przed `ap_should_client_block()`. Zwraca OK jeśli jest zezwolenie na odbiór danych od klienta lub kod statusu w przeciwnym razie. OK jest zwracane nawet, gdy żądanie nie zawiera żadnych danych (Content-Length jest 0).

`ap_should_client_block()`
Sprawdza czy klient będzie przesyłał dane i ewentualnie zaprasza go do kontynuowania. Wysyła odpowiedź `Continue` o kodzie 100 gdy używany jest protokół HTTP/1.1 lub wyższy. Ta funkcja powinna być używana po `ap_setup_client_block()` i przed `ap_get_client_block()`. Zwraca 1, jeśli klient powinien przesłać dane, 0 jeśli nie.

`void ap_send_http_header(request_rec *r)`
Przesyła nagłówki HTTP z `r->headers_out` i `r->err_headers_out` do klienta. Dodatkowe pole nagłówka HTTP może być dodane przez serwer, w zależności od wersji protokołu HTTP. Funkcja ta musi być zawołana przez moduł obsługi żądań zanim wyśle on jakiekolwiek dane. Po zawołaniu tej funkcji, dalsze zmiany w tablicach `r->headers_out` i `r->err_headers_out` są ignorowane.

`int ap_should_client_block(request_rec *r)`
Zwraca wartość klucza ze wskazanej tabeli.

`void ap_table_add(table *t, const char *k, const char *val)`
Dodaje nową wartość `val` do tabeli `t`, związaną z kluczem `key`. Zarówno klucz jak i wartość są kopiowane do tablicy poprzez użycie `ap_pstrdup()` i używane są wskaźniki do kopii. Jeśli obie wartości są łańcuchami stałymi, powinno się używać `ap_table_addn()`. Zawsze tworzony jest nowy wpis w tablicy, nawet, jeśli istnieje już jeden albo więcej wpisów z tym samym kluczem. Aby uaktualnić istniejący wpis, albo stworzyć go, gdy taki nie istnieje, należy użyć `ap_table_set()`. Należy uważać, gdyż `ap_table_set()` skasuje wszystkie inne wpisy z tym samym kluczem. Jeśli tablica ma wiele wpisów dla podanego klucza, tylko pierwszy będzie zwrócony przez bezpośrednie funkcje wyszukiwania. Jedynym sposobem odczytania powtórzonych wpisów jest przebiegnięcie całej tablicy przy pomocy funkcji `ap_table_do()`.

`void ap_table_merge(table *t, const char *k, const char *more_val)`
Jeśli nazwa istnieje jako wartość klucza w tablicy `t`, wówczas `more_val` jest dodawane do tablicy. W przeciwnym razie tworzony jest nowy wpis.

Jeśli istnieje więcej niż jeden wpis, z kluczem `k`, tylko pierwszy będzie zmodyfikowany.

`void ap_table_set(table *t, const char *k, const char *val)`
Wstawia do tablicy `t` wpis z kluczem `k` o wartości `val`. Jeśli taki wpis nie istnieje, to zostaje utworzony. Jeśli natomiast wpis istnieje jego wartość jest zastępowana. Jeśli jest kilka wpisów o kluczu `k`, pierwszy będzie zmodyfikowany, a pozostałe pasujące wykasowywane z tablicy. Wartość zawsze jest duplikowana z podanego argumentu do komórki tablicy, przy pomocy `ap_pstrdup()` i wskaźnik do kopii jest używany. Jeśli nowy wpis musi być stworzony, to również łańcuch klucza jest kopiowany. Jeśli zarówno klucz i wartość są łańcuchami stałymi, to lepiej użyć `ap_table_setn()`.

Do ważnych struktur danych API serwera Apache 1.3 należy struktura `request_rec`, do której często odwołujemy się podczas implementacji modułu `mod_mhtml`:

```
typedef struct request_rec request_rec;
struct request_rec {
    pool *pool;
    conn_rec *connection;
    server_rec *server;
    request_rec *next;
    request_rec *prev;
    request_rec *main;
    char *the_request;
    int assbackwards;
    enum proxyreqtype proxyreq;
    int header_only;
    char *protocol;
    int proto_num;
    char *hostname;
    time_t request_time;
    const char *status_line;
    int status;
    const char *method;
    int method_number;
    int allowed;
    int sent_bodyct;
    long bytes_sent;
    time_t mtime;
    int chunked;
    int byterange;
    char *boundary;
    const char *range;
    long clength;
    long remaining;
    long read_length;
    int read_body;
    int read_chunked;
    unsigned expecting_100;
    table *headers_in;
    table *headers_out;
    table *err_headers_out;
    table *subprocess_env;
    table *notes;
    const char *content_type;
    const char *handler;
```

```
    const char *content_encoding;
    const char *content_language;
    array_header *content_languages;
    char *vlist_validator;
    int no_cache;
    int no_local_copy;
    char *unparsed_uri;
    char *uri;
    char *filename;
    char *path_info;
    char *args;
    struct stat finfo;
    uri_components parsed_uri;
    void *per_dir_config;
    void *request_config;
    const struct htaccess_result *htaccess;
    char *case_preserved_filename;
};
```

Interfejsem pomiędzy modułem serwera Apache 1.3, a jego jądrem jest struktura `module`, która składa się prawie wyłącznie ze wskaźników do funkcji obsługujących poszczególne fazy przetwarzania żądania. Wskaźnik NULL oznacza, że dana faza nie jest obsługiwana przez moduł. Liczby w komentarzach oznaczają kolejność faz.

```
module MODULE_VAR_EXPORT mhtml_module =
  {
    STANDARD_MODULE_STUFF,
    mhtml_init,           /* module initializer */
    NULL,                 /* per-directory config creator */
    NULL,                 /* dir config merger */
    NULL,                 /* server config creator */
    NULL,                 /* server config merger */
    mhtml_cmds,           /* command table */
    mhtml_handlers,       /* [9] list of handlers */
    NULL,                 /* [2] filename-to-URI translation */
    NULL,                 /* [5] check/validate user_id */
    NULL,                 /* [6] check user_id is valid *here* */
    NULL,                 /* [4] check access by host address */
    NULL,                 /* [7] MIME type checker/setter */
    NULL,                 /* [8] fixups */
    NULL,                 /* [10] logger */
#if MODULE_MAGIC_NUMBER >= 19970103
    NULL,                 /* [3] header parser */
#endif
#if MODULE_MAGIC_NUMBER >= 19970719
    NULL,                 /* process initializer */
#endif
#if MODULE_MAGIC_NUMBER >= 19970728
    NULL,                 /* process exit/cleanup */
#endif
#if MODULE_MAGIC_NUMBER >= 19970902
    NULL                  /* [1] post read_request handling */
#endif
}
```

W przypadku modułu `mod_mhtml` zadeklarowane są wyłącznie trzy wskaźniki w strukturze `module`. Są to wskaźniki do funkcji `mhtml_init()` inicjalizującej moduł, do struktury `mhtml_cmds` opisującej dyrektywy konfiguracji modułu oraz wskaźnik do listy `mhtml_handlers` funkcji obsługujących żądania.

```
command_rec mhtml_cmds[] =
```

```
  {
    { "Meta-HTMLModuleDirectories", set_mhtml_module_directories,
        NULL, RSRC_CONF, RAW_ARGS, "Set MHTML module directories" },
    { "Meta-HTMLPublicHTML", set_mhtml_public_html,
        NULL, RSRC_CONF, RAW_ARGS, "Set MHTML user public directory" },
    { "Meta-HTMLDirectoryIndex", set_mhtml_directory_index, NULL,
        RSRC_CONF, RAW_ARGS, "Set MHTML directory default filenames" },
    { "Meta-HTMLRequireDirectories", set_mhtml_require_directories,
        NULL, RSRC_CONF, RAW_ARGS, "Set MHTML standard directories to
          look for libraries" },
    { "Meta-HTMLExecPath", set_mhtml_exec_path, NULL, RSRC_CONF,
          RAW_ARGS, "Set default list of paths to look for
        executables" },
    { "Meta-HTMLFileExtensions", set_mhtml_file_extensions,
        NULL, RSRC_CONF, RAW_ARGS, "Set default extensions for
                Meta-HTML documents" },
    { NULL }
  };

static const handler_rec mhtml_handlers[] =
  {
    { "metahtml/interpreted", mhtml_handle_req },
    { NULL }
  };
```

Dodatek B

API Meta-HTML

Zmienne globalne

`AllPackages`
: Wszystkie funkcje i zmienne Meta-HTML przechowywane są w postaci pakietów (package). `AllPackages` to wskaźnik na listę wszystkich pakietów.

`int AP_index`
: Ilość pakietów na liście `AllPackages`.

`int AP_slots`
: Ilość bloków pamięci zarezerwowanej na pakiety znajdujące się na liście `AllPackages`. Pamięć potrzebna na przechowywanie pakietów alokowana jest w blokach, każdy taki blok mieści 10 pakietów.

`sv_DocumentRoot`
: Pełna ścieżka do dokumentów HTML/MHTML na serwerze HTTP (por. `ap_document_root()` w A str. 20).

`gbl_passed_sid`
: Wartość SID (Session Identifier) jaką przekazuje w postaci *cookie* przeglądarka lub ustawiana w makrze Meta-HTML `session::initialize`.

`new_env`
: Wskaźnik do listy zawierającej zmienne środowiskowe i ich wartości, związane z żądaniem HTTP, takie jak:
SERVER_NAME, SERVER_PORT, REMOTE_HOST, REMOTE_USER, HTTP_USER_AGENT, HTTP_COOKIE, SID.

`new_env_index`
: Ilość elementów na liście wskazywanej przez `new_env`.

`new_env_slots`
: Ilość bloków pamięci przydzielonych liście wskazywanej przez `new_env`.

Dla zminimalizowania granulacji pamięci rezerwowane są bloki po 10 elementów listy każdy.

Funkcje

`static char *mhtml_evaluate_string(char *body)`
: Zinterpretowanie przez Meta-HTML napisu (tekstu) przekazanego w `body` i zwrócenie wyniku.

`HTTP_RESULT *mhtml_make_result(void)`
: Zaalokowanie (przydzielenie) pamięci dla struktury `HTTP_RESULT` i wyzerowanie tej struktury.

`void mhttpd_free_request(HTTP_REQUEST *request)`
: Zwalnienie pamięci zajmowanej przez strukturę `request`.

`char *mhttpd_get_mime_header(MIME_HEADER **headers, char *which)`
: Lista `headers` zawiera elementy typu `MIME_HEADERS`, które są strukturami złożonymi z dwóch pól: `tag` oraz `value`. Funkcja ta przeszukuje listę `headers` i jeśli znajdzie element, którego pole `tag` jest równe `which` to zwraca `value`, w przeciwnym razie zwraca NULL.

`void mhttpd_metahtml_engine(HTTP_RESULT *result)`
: Przetworzenie przez Meta-HTML żądania o parametrach przekazanych w strukturze `result`. Powstała w efekcie działania interpretera strona HTML zwracana jest w `result->page->buffer` oraz ustawienie kodu odpowiedzi w `result->result_code`.

`Package *mhttpd_mime_headers_to_package(MIME_HEADER **headers, char *packname)`
: Wypełnia pakiet o nazwie `packname` wartościami przekazanymi w liście `headers`.

`MIME_HEADER **mime_headers_from_string(char *string, int *last_char)`
: Skanowanie tekstu `string` w poszukiwaniu nagłówków HTTP. Znalezione nagłówki kolekcjonowane są w liście, do której wskaźnik zwracany jest jako wynik tej funkcji. `last_char` ustawiany jest na pierwszy znak za końcem nagłówków w tekście `string`.

`char *mr_engine(HTTP_RESULT *result, char *existing_location)`

`char *mr_expires(HTTP_RESULT *result, char *existing_location)`

`char *mr_last_modified(HTTP_RESULT *result, char *existing_location)`

`char *mr_location(HTTP_RESULT *result, char *existing_location)`

`char *mr_set_cookie(HTTP_RESULT *result, char *existing_location)`

`char *mr_status(HTTP_RESULT *result, char *existing_location)`
Powyższe funkcje zwracają wartości przekazywane w nagłówkach HTTP odpowiedzi, na podstawie struktury `result`.

`void package_pdl_remove(Package *p)`
Zwalnia pamięć zajmowaną przez `p`.

Dodatek C

Nagłówki HTTP

Najczęściej występujące nagłówki HTTP:

`Accept` – Określa typ danych, które preferuje klient.

`Accept-Encoding` – Określa typ kodowania, który preferuje klient, taki jak `compress` lub `gzip`. Jeśli nie jest wymieniony żaden schemat kodowania, oznacza to, że żaden typ kodowania nie jest akceptowany przez klienta.

`Accept-Language` – Określa język preferowany przez klienta. Nazwy języków zapisywane są za pomocą ich dwuliterowych skrótów (np. `en` dla języka angielskiego).

`Authorization` – Zapewnia uwierzytelnienie klienta przy dostępie do danych. Kiedy określony w żądaniu dokument wymaga autoryzacji, serwer zwraca nagłówek `WWW-Authenticate`. Następnie klient odpowiada podając odpowiednie informacje.

`Connection` – Określa opcje ważne dla bieżącego połączenia. Połączenie z opcją `close` oznacza, że albo klient, albo serwer chce zakończyć połączenie.

`Content-Length` – Nagłówek ten określa długość danych (w bajtach) znajdujących się w głównej części.

`Content-Type` – Opisuje typ i podtyp danych znajdujących się w głównej części.

`Cookie` – Nagłówek ten zawiera informacje w postaci pary nazwa/wartość dla danego URL.

`Host` – Określa nazwę hosta i numer portu w adresie URI

`If-Modified-Since` – Informuje, że dane określone w URI mają być przesłane tylko wtedy, gdy zostały zmodyfikowane od czasu podanego w postaci

wartości tego nagłówka. Jeśli dokument nie został zmodyfikowany to serwer zwraca kod 304.

`Referer` – Podaje URL dokumentu odpowiadającego zadanemu URI.

`User-Agent` – Podaje informacje identyfikujące program klienta. Treść tego nagłówka zapisywany jest za pomocą ciągu znaków.

Dodatek D

Kody odpowiedzi serwera HTTP

Najczęściej spotykane odpowiedzi HTTP:

`HTTP_OK` (kod 200) – DOCUMENT_FOLLOWS – Zapytanie powiodło się, serwer wysyła odpowiedź, która zawiera żądane dane.

`HTTP_CREATED` (kod 201) – Utworzony został nowy URI. Przesyłany jest też nagłówek `Location`, który określa miejsce umieszczenia nowych danych.

`HTTP_ACCEPTED` (kod 202) – Żądanie jest zaakceptowane, ale serwer nie od razu je przetworzył.

`HTTP_NON_AUTHORITATIVE` (kod 203) – Informacja jest nieautoryzowana. W zasadniczej części odpowiedź pochodzi z innego komputera, a nie z serwera, do którego kierowane było żądanie.

`HTTP_NO_CONTENT` (kod 204) – Żądanie powiodło się, ale w odpowiedzi nie ma części zasadniczej. Przeglądarka nie powinna uaktualniać wyświetlanego dokumentu.

`HTTP_PARTIAL_CONTENT` (kod 206) – PARTIAL_CONTENT – Serwer przesyła tylko część danych, które żądał klient. W odpowiedzi musi być określony zakres danych wraz z nagłówkiem `Content-Range`.

`HTTP_MULTIPLE_CHOICES` (kod 300) – MULTIPLE_CHOICES – URI w zapytaniu odnosi się do więcej niż jednego zasobu. W odpowiedzi serwer wysyła listę z bardziej szczegółowymi informacjami, jak poprawnie należy określić żądany zasób.

`HTTP_MOVED_PERMANENTLY` (kod 301) – MOVED – Wskazany dokument został przeniesiony, a żądany URI nie jest już używany przez serwer. Nowy URI określony jest w nagłówku `Location`.

`HTTP_MOVED_TEMPORARILY` (kod 302) – REDIRECT – Wskazany dokument został tymczasowo przeniesiony w inne miejsce.

`HTTP_NOT_MODIFIED` (kod 304) – USE_LOCAL_COPY – Jest to kod odpowiedzi na żądanie zawierające nagłówek `If-Modified-Since`, gdy określony URI nie został zmodyfikowany od określonej daty. Odpowiedź nie zawiera głównej części.

`HTTP_BAD_REQUEST` (kod 400) – BAD_REQUEST – Niepoprawne zapytanie. Serwer wykrył błąd w składni zapytania.

`HTTP_UNAUTHORIZED` (kod 401) – AUTH_REQUIRED – Brak poprawnej autoryzacji. Klient nie podał poprawnie danych, które umożliwiają uwierzytelnienie.

`HTTP_PAYMENT_REQUIRED` (kod 402) – Wymagana jest płatność. Ten kod nie jest jeszcze zaimplementowany w protokole HTTP.

`HTTP_FORBIDDEN` (kod 403) – FORBIDDEN – Żądanie zostało odrzucone, ponieważ serwer nie przyjmuje zapytań od tego klienta lub nie może określić, kto wysłał żądanie. Zasób jest zakazany.

`HTTP_NOT_FOUND` (kod 404) – NOT_FOUND – Żądany dokument nie istnieje. Serwer nie może go odnaleźć.

`HTTP_METHOD_NOT_ALLOWED` (kod 405) – METHOD_NOT_ALLOWED – Metoda jest niedozwolona. Kod ten jest przekazywany wraz z nagłówkiem `Allow`.

`HTTP_NOT_ACCEPTABLE` (kod 406) – Żądanie nie jest akceptowane. URI istniej, ale nie w formacie preferowanym przez klienta. Razem z tym kodem serwer przesyła nagłówki: `Content-Language, Content-Encoding` i `Content-Type`.

`HTTP_PROXY_AUTHENTICATION_REQUIRED` (kod 407) – Konieczne jest uwierzytelnienie serwera proxy, który musi autoryzować żądanie przed przekazaniem go. Kod używany jest wraz z nagłówkiem `Proxy-Authenticate`.

`HTTP_REQUEST_TIME_OUT` (kod 408) –Klient nie przygotował pełnego zapytania w określonym przez serwer czasie. Serwer zrywa połączenie sieciowe.

`HTTP_GONE` (kod 410) – URI umieszczony w żądaniu już nie istnieje i został nieodwracalnie usunięty z serwera.

`HTTP_PRECONDITION_FAILED` (kod 412) – PRECONDITION_FAILED – Warunek określony przez nagłówek umieszczony w żądaniu nie został spełniony.

`HTTP_REQUEST_ENTITY_TOO_LARGE` (kod 413) –Serwer nie może przetworzyć żądania, ponieważ wielkość jego głównej części jest za duża.

`HTTP_REQUEST_URI_TOO_LARGE` (kod 414) – Serwer nie może przetworzyć żądania, ponieważ określony w nim URI jest za długi.

`HTTP_UNSUPPORTED_MEDIA_TYPE` (kod 415) – Serwer nie może przetworzyć żądania, ponieważ dane w nim przesłane zapisane są w formacie, który serwer nie obsługuje.

`HTTP_INTERNAL_SERVER_ERROR` (kod 500) – SERVER_ERROR – Wewnętrzny błąd serwera np. program CGI zawiesił się lub wystąpił w nim błąd konfiguracyjny.

`HTTP_NOT_IMPLEMENTED` (kod 501) – NOT_IMPLEMENTED – Żądanie klienta dotyczy działań, które nie mogą być wykonane przez serwer.

`HTTP_BAD_GATEWAY` (kod 502) – BAD_GATEWAY – Brama jest niepoprawna. Serwer (lub `proxy`)otrzymał niepoprawną odpowiedź od innego serwera (lub `proxy`).

`HTTP_SERVICE_UNAVAILABLE` (kod 503) – Usługa jest niedostępna, chociaż w przyszłości powinno to się zmienić. Jeśli serwer wie, kiedy to nastąpi to do odpowiedzi dołączony jest nagłówek `Retry-After`.

`HTTP_GATEWAY_TIME_OUT` (kod 504) – Ten kod podobny jest do kodu 408 (Wymagania czasowe), lecz wynika ze stanu bramy lub `proxy`.

`HTTP_VERSION_NOT_SUPPORTED` (kod 505) – Wersja HTTP jest nie obsługiwana przez serwer.

Dodatek E

Kod źródłowy mod_mhtml.c

```
/* mod_mhtml.c: -*- C -*-  Meta-HTML Apache (1.3) module.

   This file is part of <Meta-HTML>(tm), a system for the rapid
   deployment of Internet and Intranet applications via the use
   of the Meta-HTML language.

   Meta-HTML is copyright (c) 1995, 2003, Brian J. Fox (bfox@ai.mit.edu).
   This module is copyright (c) 2003 Joanna Merecka & Mariusz Zynel
   (mariusz@math.uwb.edu.pl), except portions written by Brian J. Fox,
   which are copyright (c) 1998, 2003.

   Meta-HTML is free software; you can redistribute it and/or modify
   it under the terms of the General Public License as published
   by the Free Software Foundation; either version 1, or (at your
   option) any later version.

   This program is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
   FSF GPL for more details. */

#define HAVE_CONFIG_H
/* # include "limits.h" */
#define NO_REGEX_H 1
#include "../../libmhtml/language.h"
#undef NO_REGEX_H
#include "../../libserver/http.h"
#include "../../libserver/globals.h"
#include "../../libserver/logging.h"
#include "mod_mhtml.h"

#include "httpd.h"
#include "http_config.h"
#include "http_core.h"
#include "http_log.h"
#include "http_main.h"
#include "http_protocol.h"
#include "util_script.h"

module MODULE_VAR_EXPORT mhtml_module;

#if !defined (MHTML_VERSION_STRING)
#  define MHTML_VERSION_STRING "6.11"
#endif

static char *mhtml_version_string = MHTML_VERSION_STRING;
```

```
static char *mhtml_module_directories = NULL;
static char *mhtml_public_html = NULL;
static char *mhtml_directory_index = NULL;
static char *mhtml_require_directories = NULL;
static char *mhtml_exec_path = NULL;
static char *mhtml_file_extensions = NULL;

static MIME_RESOLVER mime_resolvers[] = {
  { "Location",          mr_location, 1 },
  { "Meta-HTML-Engine",     mr_engine, 1 },
  { "Set-cookie",     mr_set_cookie, -1 },
  { "Expires",          mr_expires, 0 },
  { "Last-modified",     mr_last_modified, 0 },
  { "Status",          mr_status, 1 },
  { (char *)NULL,     (GFunc *)NULL, 0 }
};


char *
str_to_array (char *s)
{
  char *t = NULL;
  int i = 0;
  int j = 0;

  if (t = (char *)malloc(strlen(s) + 1))
    {
      while (s[i] != '\0')
        {
          if (s[i] == ' ' && s[i+1] != ' ')
            {
              t[j++] = '\n';
            }
          else if (s[i] != ' ')
            {
              t[j++] = s[i];
            }
          i++;
        }
    }
  t[j] = '\0';
  return t;
}


static void
mhtml_init (server_rec *s, pool *p)
{
  ap_add_version_component ("MetaHTML/" MHTML_VERSION_STRING);
}

static void
mhtml_destroy_all_packages (request_rec *r)
{
  Package *package;
  int i, j;

  if (AllPackages != (Package **)NULL)
    {
      for (j = AP_index; j > -1; j--)
    {
      if ((package = AllPackages[j]) != (Package *)NULL)
        {
          if (package == mhtml_function_package)
        continue;
```

```
          package_pdl_remove (package);

          /* Now, simply free the package contents. */
          if (package->name)
        free (package->name);

      if (package->table != (SymbolTable *)NULL &&
      package->table->symbol_list != (SymbolList **)NULL)
    {
      for (i = 0; i < package->table->rows; i++)
        {
         if (package->table->symbol_list[i] != (SymbolList *)NULL)
        {
          SymbolList *list = package->table->symbol_list[i];
          while (list)
            {
              Symbol *sym = list->symbol;
              SymbolList *thissym = list;
              list = list->next;
              symbol_free (sym);
              free (thissym);
            }
        }
        }

      free (package->table->symbol_list);
      free (package->table);
    }
      free (package);
    }
}

      free (AllPackages);
      AP_index = 0;
      AP_slots = 0;

      if (mhtml_function_package != (Package *)NULL)
    {
      AllPackages = (Package **)xrealloc
        (AllPackages, (AP_slots += 10) * sizeof (Package *));
      AllPackages[AP_index++] = mhtml_function_package;
      AllPackages[AP_index] = (Package *)NULL;
    }
      else
    {
      AllPackages = (Package **)NULL;
    }
    }

  CurrentPackage = (Package *)NULL;
  PageVars = (Package *)NULL;
  mhtml_user_keywords = (Package *)NULL;
  mhtml_flag_newly_interned_symbols = 0;
}

/* Modified initialize_engine () from metahtml/engine/engine.c */

static void
initialize_engine (request_rec *r)
{
  char *temp;
  int i;
  char buf[10];
  char *include_prefix = (char *)NULL;

  /* Reset globals */
```

```
sv_DocumentRoot = "";
gbl_passed_sid = (char *)NULL;
mhttpd_debugging = 0;
mhttpd_log_performance = 0;
mhttpd_log_referer = 0;
mhttpd_log_agent = 0;
mhttpd_per_request_function = (char *)NULL;
mhttpd_ssl_server = 0;
mhtml_cookie_compatible = 0;
new_env = (char **)NULL;
new_env_slots = 0;
new_env_index = 0;

pagefunc_set_variable ("mhtml::program-name", "mod_mhtml");
pagefunc_set_variable ("mhtml::version", mhtml_version_string);
pagefunc_set_variable ("mhttpd::copyright-string",
   metahtml_copyright_string);
pagefunc_set_variable ("mhtml::system-type", "linux-i386");

if (mhtml_module_directories)
  {
    pagefunc_set_variable ("mhtml::module-directories[]",
    str_to_array(mhtml_module_directories));
  }
else
  {
    pagefunc_set_variable ("mhtml::module-directories[]",
    "/usr/lib/metahtml\n/usr/local/metahtml/lib");
  }

if (mhtml_public_html)
  {
    pagefunc_set_variable ("mhtml::~directory", mhtml_public_html);
  }
else
  {
    pagefunc_set_variable ("mhtml::~directory", "public_html");
  }

bootstrap_metahtml (0);

/* Quickly make a package containing the minimum mime-types. */
pagefunc_set_variable ("mime-type::.mhtml", "metahtml/interpreted");
pagefunc_set_variable ("mime-type::.jpeg", "image/jpeg");
pagefunc_set_variable ("mime-type::.jpg", "image/jpeg");
pagefunc_set_variable ("mime-type::.gif", "image/gif");
pagefunc_set_variable ("mime-type::.txt", "text/plain");
pagefunc_set_variable ("mime-type::.mov", "video/quicktime");
pagefunc_set_variable ("mime-type::.default", "text/plain");
pagefunc_set_variable ("mime-type::.html", "text/html");
pagefunc_set_variable ("mime-type::.htm", "text/html");

/* The minimum startup documents. */
if (mhtml_directory_index)
  {
     pagefunc_set_variable ("mhtml::default-filenames[]",
     str_to_array(mhtml_directory_index));
  }
else
  {
     pagefunc_set_variable ("mhtml::default-filenames[]",
     "welcome.mhtml\nindex.mhtml");
  }

 /* The default extensions for running files through the engine. */
 if (mhtml_file_extensions)
```

```
    {
      pagefunc_set_variable ("mhtml::metahtml-extensions[]",
      str_to_array(mhtml_file_extensions));
    }
  else
    {
      pagefunc_set_variable ("mhtml::metahtml-extensions[]",
      ".mhtml");
    }

include_prefix = (char *)ap_document_root (r);
pagefunc_set_variable ("mhtml::include-prefix", include_prefix);
pagefunc_set_variable ("mhtml::document-root", include_prefix);

/* Even if there is something to setup in a configuration file,
this should rather be done once in initialization of the
module - here it's a pure waste of time here. So, the code to
read it is dropped. */

/* Define a default function for handling a missing page.
This will be used unless we find a configuration file. */
{
  char *x = mhtml_evaluate_string
    ("<defun mhttpd::default-document> <html>
    <body bgcolor=white> <dump-package mhttpd mhtml env>
    </body> </html> </defun>");
  xfree (x);
}

/* If the user didn't set mhtml::require-directories[],
give a reasonable value here. */
temp = pagefunc_get_variable ("mhtml::require-directories");
if (temp == (char *)NULL)
  {
    if (mhtml_require_directories)
      {
    pagefunc_set_variable ("mhtml::require-directories[]",
    str_to_array(mhtml_require_directories));
  }
    else
      {
        pagefunc_set_variable ("mhtml::require-directories[]",
        "lib\ntagsets\nmacros\ninclude");
      }
  }

/* Now set our local variables. */
pagefunc_set_variable ("mhtml::server-name",
(char *)ap_get_server_name (r));

i = ap_get_server_port (r);
snprintf (buf, 10, "%d", i);
pagefunc_set_variable ("mhtml::server-port", buf);

sv_DocumentRoot = strdup (ap_document_root (r));
chdir (sv_DocumentRoot);

mhttpd_per_request_function =
  pagefunc_get_variable ("mhttpd::per-request-function");

set_session_database_location
  (pagefunc_get_variable ("mhttpd::session-database-file"));

/* Create a reasonable default PATH variable if the user
didn't do so. */
```

```
  if (pagefunc_get_variable ("mhtml::exec-path") == (char *)NULL)
    {
      if (mhtml_exec_path)
        {
      pagefunc_set_variable ("mhtml::exec-path", mhtml_exec_path);
    }
      else
        {
      pagefunc_set_variable ("mhtml::exec-path",
      "/usr/bin:/usr/local/bin:/usr/local/metahtml/bin");
        }
     }
}

/* sapi_apache_read_post() from php/sapi/apache/mod-php4.c.
   Call ap_get-client () as many times as required to read
   count_bytes characters. */

static int
apache_read_post (request_rec *r, char *buffer, int count_bytes)
{
  uint total_read_bytes = 0, read_bytes;
  void (*handler) (int);

  handler = ap_signal (SIGPIPE, SIG_IGN);

  while (total_read_bytes < count_bytes)
    {
      /* Start timeout timer. */
      ap_hard_timeout ("Read_POST_data", r);
      read_bytes = ap_get_client_block (r, buffer + total_read_bytes,
                   count_bytes - total_read_bytes);
      ap_reset_timeout (r);

      if (read_bytes <= 0)
    {
      break;
    }

      total_read_bytes += read_bytes;
    }

  ap_signal (SIGPIPE, handler);

  return (total_read_bytes);
}

/* Modified mhtml_read_content_from_fd() from metahtml/libserver/http.c
   Read POST data from the client. Content-Length is taken from headers
   sent by the client. */

static void
mhtml_read_content_from_ap (HTTP_RESULT *result, request_rec *r)
{
  char *content_length =
    (char *)ap_table_get (r->headers_in, "Content-Length");

  if (content_length && ap_should_client_block (r))
    {
      result->spec->content_length = atoi (content_length);

     /* I just hate the whole world.  Why O Why is there an extra
     CR/LF at the end of this fucking post that doesn't show up
     in the Content-Length? */
      result->spec->content = (char *)xmalloc
    (3 + result->spec->content_length);
```

```
      apache_read_post (r, result->spec->content,
            result->spec->content_length);

      result->spec->content[result->spec->content_length] = '\0';
    }
}

/* Decode the username from Authorization header sent by the client. */

static void
mhtml_set_remote_user (HTTP_REQUEST *req, request_rec *r)
{
  int i;
  char *decoded, *username, *password;
  char *auth = (char *)ap_table_get (r->headers_in, "Authorization");

  if (auth != (char *)NULL)
    {
      /* Skip past "Basic", and any following whitespace. */
      for (i = 5; whitespace (auth[i]); i++);

      /* Get the decoded string. */
      decoded = mhtml_base64decode (auth + i, (int *)NULL);

      /* Check the authorization string here. */
      username = decoded;
      password = strchr (username, ':');

      if (password != (char *)NULL)
    {
      *password = '\0';
      password++;
    }

      pagefunc_set_variable_readonly ("env::remote_user", username);
      pagefunc_set_variable_readonly ("mhtml::remote-user", username);
    }
}

/* Strips off headers from resulting page if any and puts them into
Apache's headers_out table to send back to client. */

static void
mhtml_handle_headers (HTTP_RESULT *result, request_rec *r)
{
  register int i;
  MIME_HEADER **present = (MIME_HEADER **)NULL;
  int end_of_headers;

/* If this page already has an HTTP result line, snarf the result
   code. */

  if (result->page && result->page->buffer &&
      strncasecmp (result->page->buffer, "HTTP", 4) == 0)
    {
      char *buffer = result->page->buffer;
      for (i = 0; !whitespace (buffer[i]); i++);
      result->result_code = atoi (buffer + i);
      for (; buffer[i] != '\0' && buffer[i] != '\n'; i++);
      if (buffer[i] == '\n') i++;
      bprintf_delete_range (result->page, 0, i);
    }

  if (!result->page)
    result->page = page_create_page ();
```

```
present = mime_headers_from_string (result->page->buffer, &end_of_headers);

  if (end_of_headers)
    bprintf_delete_range (result->page, 0, end_of_headers);

  /* Next, make sure that the required headers are there. */
  for (i = 0; mime_resolvers[i].mime_name != (char *)NULL; i++)
    {
      int val_initially_present = 1;
      char *orig_value =
    mhttpd_get_mime_header (present, mime_resolvers[i].mime_name);
      char *value = orig_value;

      if (value == (char *)NULL)
    {
      val_initially_present = 0;
      value = (*mime_resolvers[i].value_generator) (result, (char *)NULL);
    }
      else if (mime_resolvers[i].call_anyway)
    {
      value = (*mime_resolvers[i].value_generator) (result, value);
    }

      if (value)
    ap_table_set(r->headers_out, mime_resolvers[i].mime_name, value);

      if (val_initially_present && (mime_resolvers[i].call_anyway > -1))
    *orig_value = '\0';
    }

  /* Now, insert all of the headers that were present, but that were not
     required. */
  for (i = 0; present && present[i]; i++)
   if ((present[i]->value != (char *)NULL) && (present[i]->value[0] != '\0'))
      ap_table_set (r->headers_out, present[i]->tag, present[i]->value);

  for (i = 0; present && present[i]; i++)
    {
      if (present[i]->tag) free(present[i]->tag);
      if (present[i]->value) free(present[i]->value);
      if (present[i]) free(present[i]);
    }

  if (present) free (present);
}

static const char *
set_mhtml_module_directories (cmd_parms *cmd, void* empty, char* text)
{
  mhtml_module_directories = (char*)ap_pstrdup(cmd->pool, text);

  return ((const char *)NULL);
}

static const char *
set_mhtml_public_html (cmd_parms *cmd, void* empty, char* text)
{
  mhtml_public_html = (char*)ap_pstrdup(cmd->pool, text);

  return ((const char *)NULL);
}

static const char *
set_mhtml_directory_index (cmd_parms *cmd, void* empty, char* text)
{
```

```
  mhtml_directory_index = (char*)ap_pstrdup(cmd->pool, text);

  return ((const char *)NULL);
}

static const char *
set_mhtml_require_directories (cmd_parms *cmd, void* empty, char* text)
{
  mhtml_require_directories = (char*)ap_pstrdup(cmd->pool, text);

  return ((const char *)NULL);
}

static const char *
set_mhtml_exec_path (cmd_parms *cmd, void* empty, char* text)
{
  mhtml_exec_path = (char*)ap_pstrdup(cmd->pool, text);

  return ((const char *)NULL);
}

static const char *
set_mhtml_file_extensions (cmd_parms *cmd, void* empty, char* text)
{
  mhtml_file_extensions = (char*)ap_pstrdup(cmd->pool, text);

  return ((const char *)NULL);
}


static int
mhtml_handle_req (request_rec *r)
{
  HTTP_REQUEST *req = (HTTP_REQUEST *)NULL;
  HTTP_RESULT *result = (HTTP_RESULT *)NULL;
  MIME_HEADER *header;
  array_header *hdrs_arr = ap_table_elts (r->headers_in);
  table_entry *hdrs = (table_entry *) hdrs_arr->elts;
  int hdrs_index = 0;
  int hdrs_slots = 0;
  int retval, i;
  char buf[256];

  if (r->method_number == M_OPTIONS)
    {
      /* We only support GET and POST methods. */
      r->allowed |= (1 << M_GET);
      r->allowed |= (1 << M_POST);
      return (DECLINED);
    }

  if (r->method_number != M_GET && r->method_number != M_POST)
    {
      return (METHOD_NOT_ALLOWED);
    }

  /* Before we initialize the engine, check to see if there is POST
  data to read. If it is chunked transfer-coding reject it, so the
  policy is REQUEST_CHUNKED_ERROR. */
  if ((retval = ap_setup_client_block (r, REQUEST_CHUNKED_ERROR)))
    return (retval);

  /* We need to initialize req data structure and set method, protocol,
  protocol_version and location adequately. Content-Length header
  will be necessary later to read POST data. */
```

```
req = http_parse_request (r->the_request);

if (req != (HTTP_REQUEST *)NULL)
  {
    char *t = (char *)r->connection->remote_host;

    if (t != (char *)NULL) t = strdup (t);
    req->requester = t;

    t = (char *)r->connection->remote_ip;
    if (t != (char *)NULL) t = strdup (t);
    req->requester_addr = t;

    /* Additional CGI environment variables can be put
   into r->subprocess_env table by calling:
   ap_add_common_vars(r);
   ap_add_cgi_vars(r);
    */

    /* Copy all the headers from Apache to MHTML. */
    for (i = 0; i < hdrs_arr->nelts; ++i)
     {
       if (!hdrs[i].key)
        {
          continue;
        }

    header = (MIME_HEADER *)xmalloc (sizeof (MIME_HEADER));

    header->tag = strdup (hdrs[i].key);
    header->value = strdup (hdrs[i].val);

       if (hdrs_index + 2 > hdrs_slots)
    req->headers = (MIME_HEADER **)xrealloc
       (req->headers, (hdrs_slots += 10) * sizeof(MIME_HEADER *));

    req->headers[hdrs_index++] = header;
    req->headers[hdrs_index] = (MIME_HEADER *)NULL;

      }

/* Now it's time to initialize MetaHTML engine. */
initialize_engine (r);

/* Set required variables, mostly for compatibility reasons */
mhttpd_mime_headers_to_package (req->headers,
                                "mhttpd-received-headers");
pagefunc_set_variable ("mhttpd::method", req->method);
pagefunc_set_variable ("mhttpd::protocol", req->protocol);
pagefunc_set_variable ("mhttpd::protocol-version",
                       req->protocol_version);
pagefunc_set_variable ("mhttpd::location", req->location);
pagefunc_set_variable ("mhttpd::requester", req->requester);
pagefunc_set_variable ("mhttpd::requester-addr", req->requester_addr);

/* Where the two are set in MetaHTML engine? Better set it here. */

pagefunc_set_variable_readonly
 ("env::server_protocol", req->protocol);
pagefunc_set_variable_readonly
 ("env::protocol_version", req->protocol_version);

 /* Set mhtml::remote-user variable */

 mhtml_set_remote_user (req, r);
```

```
  /* Prepare Meta-HTML to request processing */

  result = mhtml_make_result ();

  if (result != (HTTP_RESULT *)NULL)
  {
    result->request = req;
    result->spec = mhttpd_resolve_location (req);

    /* Read in POST data */

    mhtml_read_content_from_ap (result, r);

    /* Process the request and return resulting page */

    mhttpd_metahtml_engine (result);

    if (pagefunc_get_variable
        ("mhtml::server-pushed") == (char *)NULL)
      mhtml_handle_headers (result, r);

    mhttpd_free_request (req);

    /* If everything went well send the result back to the client. */
    if (result->result_code == res_SUCCESS &&
          result && result->page && result->page->buffer)
        {
          r->content_type = "text/html";
          ap_set_content_length (r, result->page->bindex);

    /* First send headers to the client */

    ap_send_http_header (r);

    /* Now send the resulting page */
    if (!r->header_only)
        {
      ap_rwrite (result->page->buffer, result->page->bindex, r);
      ap_rflush (r);
         }
      }
   }
}

/* Report the result code to Apache core. */
r->status = result->result_code;

/* Destroy result structure, */
if (result)
  {
    if (result->page)
  page_free_page (result->page);

    free (result);
  }

/* Cleanup variables and user macros. */
mhtml_destroy_all_packages (r);

/* Cleanup debugging and error messages */
page_debug_clear ();
page_syserr_clear ();

ap_kill_timeout (r);

if (r->status == res_MOVED_TEMPORARILY)
```

```
    return (REDIRECT);
  else
  if (r->status != res_SUCCESS)
    return (r->status);
  else
    return (OK);
}
command_rec mhtml_cmds[] =
  {
    { "Meta-HTMLModuleDirectories", set_mhtml_module_directories,
        NULL, RSRC_CONF, RAW_ARGS, "Set MHTML module directories" },
    { "Meta-HTMLPublicHTML", set_mhtml_public_html,
        NULL, RSRC_CONF, RAW_ARGS, "Set MHTML user public directory" },
    { "Meta-HTMLDirectoryIndex", set_mhtml_directory_index, NULL,
        RSRC_CONF, RAW_ARGS, "Set MHTML directory default filenames" },
    { "Meta-HTMLRequireDirectories", set_mhtml_require_directories,
        NULL, RSRC_CONF, RAW_ARGS, "Set MHTML standard directories to
          look for libraries" },
    { "Meta-HTMLExecPath", set_mhtml_exec_path, NULL, RSRC_CONF,
        RAW_ARGS, "Set default list of paths to look for
          executables" },
    { "Meta-HTMLFileExtensions", set_mhtml_file_extensions,
        NULL, RSRC_CONF, RAW_ARGS, "Set default extensions for
          Meta-HTML documents" },
    { NULL }
  };


static const handler_rec mhtml_handlers[] =
  {
    { "metahtml/interpreted", mhtml_handle_req },
    { NULL }
  };

module MODULE_VAR_EXPORT mhtml_module =
  {
    STANDARD_MODULE_STUFF,
    mhtml_init,         /* module initializer */
    NULL,               /* per-directory config creator */
    NULL,               /* dir config merger */
    NULL,               /* server config creator */
    NULL,               /* server config merger */
    mhtml_cmds,         /* command table */
    mhtml_handlers,     /* [9] list of handlers */
    NULL,               /* [2] filename-to-URI translation */
    NULL,               /* [5] check/validate user_id */
    NULL,               /* [6] check user_id is valid *here* */
    NULL,               /* [4] check access by host address */
    NULL,               /* [7] MIME type checker/setter */
    NULL,               /* [8] fixups */
    NULL,               /* [10] logger */
#if MODULE_MAGIC_NUMBER >= 19970103
    NULL,               /* [3] header parser */
#endif
#if MODULE_MAGIC_NUMBER >= 19970719
    NULL,               /* process initializer */
#endif
#if MODULE_MAGIC_NUMBER >= 19970728
    NULL,               /* process exit/cleanup */
#endif
#if MODULE_MAGIC_NUMBER >= 19970902
    NULL                /* [1] post read_request handling */
#endif
  };
```

Dodatek F

Plik nagłówkowy mod_mhtml.h

```
/* mod_mhtml.h: -*- C -*-  DESCRIPTIVE TEXT. */

/*  Copyright (c) 2003 Brian J. Fox
    Author: Brian J. Fox (bfox@ai.mit.edu) Thu Apr  3 06:57:41 2003.  */

extern char *mr_location (HTTP_RESULT *result, char *existing_location);
extern char *mr_engine (HTTP_RESULT *result, char *existing_location);
extern char *mr_set_cookie (HTTP_RESULT *result, char *existing_location);
extern char *mr_expires (HTTP_RESULT *result, char *existing_location);
extern char *mr_last_modified (HTTP_RESULT *result, char *existing_location);
extern char *mr_status (HTTP_RESULT *result, char *existing_location);
extern MIME_HEADER **mime_headers_from_string (char *string, int *last_char);
/* In ../../libmhtml/symbols.c, of course. */
extern void package_pdl_remove (Package *p);
extern int AP_index;
extern int AP_slots;

/* In ../../libserver/http.c. */
extern void mhttpd_metahtml_engine (HTTP_RESULT *result);
extern HTTP_RESULT *mhtml_make_result ();
extern char **new_env;
extern int new_env_slots;
extern int new_env_index;
```

Spis literatury

[1] Stephen Spainbour, Valerie Quercia, *Webmaster–podręcznik administratora*, Wydawnictwo RM Sp. z o.o., Warszawa 1997.

[2] Apache Modules Registry, `http://modules.apache.org/`.

[3] Apache Developer Resources, `http://httpd.apache.org/dev/`.

[4] Apache Web Server 1.3 API Dictionary, `http://httpd.apache.org/dev/apidoc`.

[5] Solaris 8 Reference Manual Collection, System Calls, `fork()`, `http://docs.sun.com/db/doc/806-0626/6j9vgh65d?q=fork`.

[6] Hypertext Transfer Protocol – HTTP/1.0, `http://www.w3.org/Protocols/rfc1945/rfc1945`.

[7] Hypertext Transfer Protocol – HTTP/1.1, `http://www.w3.org/Protocols/rfc2068/rfc2068`.

[8] The Meta-HTML Project, `http://metahtml.sourceforge.net/`.