Uniwersytet w Białymstoku

Wydział Matematyczno-Fizyczny

Instytut Matematyki

Ewa Chaberek

Dynamiczny PDF - moduł serwera Apache

Praca magisterska napisana
pod kierunkiem
dr. Mariusza Żynela

Białystok 2004

Spis treści

Wstęp

Rozdział 1

Wprowadzenie

1.1 PDF

PDF (Portable Document Format - format dokumentu przenośnego) jest powszechnie uznanym standardem, pozwalającym przenosić sformatowane informacje i grafikę między komputerami. Technologie Acrobat i PDF, które pojawiły się na rynku w czerwcu 1993 r., powstały jako udoskonalona wersja Adobe PostScript, uniwersalnego języka opisu strony dla drukarek, który był zarazem pierwszym oprogramowaniem firmy Adobe. Kiedy PostScript stał się standardem konwersji obrazów z ekranu komputera na dokumenty drukowane, współzałożyciel Adobe John Warnock uznał, że potrzebne jest rozwiązanie umożliwiające udostępnianie i rozprowadzadzanie dokumentów zawierających sformatowany tekst i grafikę między różnymi platformami komputerów. Pomysł Warnocka był podstawą opracowanych niedługo potem technologii Acrobat i PDF.

Od momentu powstania darmowego oprogramowania Adobe Reader, na całym świecie rozprowadzono pół miliarda jego kopii. Tym samym program stał się podstawową aplikacją do interaktywnego przeglądania i drukowania plików w formacie PDF, takich jak dokumenty, interaktywne formularze, pliki graficzne, fotografie, książki elektroniczne i wbudowane multimedia.

Acrobat Reader ma status freeware, co oznacza, że udostępniany jest użytkownikom nieodpłatnie. Może być używany jako oddzielny program lub jako plug-in integrujący się z przeglądarkami WWW, takimi jak Netscape i Microsoft Internet Explorer. Moduł plug-in pozwala oglądać pliki PDF w oknie aplikacji przeglądarki, wzbogaca ją więc o możliwość prezentowania użytkownikowi sieciowych dokumentów stworzonych w odpowiednich programach.

1.1.1 Zastosowanie PDF

Format PDF, stosowany przez instytucje państwowe i przedsiębiorstwa na całym świecie, jest uważany za niezawodny i bezpieczny środek wymiany ważnych informacji biznesowych i plików bogatych graficznie. Sądy w Stanach

Zjednoczonych wymagają składania dokumentów w formacie PDF. Staje się on także obowiązkowy w procesie rejestracji leków. W dużych firmach finansowych oraz instytucjach ochrony zdrowia format PDF usprawnia pracochłonne zadania biurowe, automatyzuje dostarczanie dokumentów i obniża koszty administracyjne.

PDF to coraz popularniejszy sposób rozpowszechniania następujących materiałów:

- katalogów,
- ulotek,
- plakatów,
- folderów,
- kalendarzy,
- papieru firmowego,
- wizytówek,
- kopert,
- innych rodzajów publikacji.

Firma Adobe doskonali format PDF, dostosowując go do wymagań użytkowników. Obecnie pozwala on m.in. na automatyczne zbieranie informacji z baz danych w celu wypełnienia i odpowiedniego skierowania formularzy elektronicznych oraz obsługę mediów dynamicznych, w tym video i dźwięku. Ponadto firma Adobe zaprezentowała niedawno nową architekturę PDF/XML, zaprojektowaną z myślą o tworzeniu bardziej inteligentnych i interaktywnych dokumentów firmowych.

1.1.2 Narzędzia do tworzenia PDF

Obok plików złożonych oraz DVI ważnym formatem używanym w systemie TEX(LATEX) jest PostScript. Oczywiście dokumenty złożone TEX-em także można zamienić na format PDF. W tym celu należy zamienić texowy plik DVI na plik w formacie PS programem dvips. Plik postscriptowy może zostać zamieniony na plik PDF albo za pomocą programu Ghostscript, albo za pomocą Acrobat Distillera. Ostatnio coraz częściej używany jest program pdfTEX, rozszerzona wersja TEX-a, który tworzy pliki PDF bezpośrednio z plików texowych. PdfTEXjest intensywnie doskonalony i obecnie jest standardowo dołączany do każdej dystrybucji systemu TEX.

1. **Plik DVI**

   Wynikiem pracy TEX-a jest zwykle kilka plików, z których najważniejszym jest plik z rozszerzeniem DVI (device independent), który może być wyświetlony na ekranie bądź wydrukowany. Do wydrukowania lub wyświetlenia na ekranie pliku DVI potrzebny jest odrębny program zwany sterownikiem, w systemach Unix takim sterownikiem ekranowym jest np. program xDVI, w Windows DVI Win lub YAP.

2. **Program dvips**

   Plik DVI można zamienić na plik w formacie PostScript. Służy do tego program dvips. Program dvips, napisany przez Toma Rokickiego, jest również oprogramowaniem Public Domain. Dvips rozumie praktycznie wszystkie komendy `\special{em:}`. Dvips rozpowszechniony jest w wersji zródłowej (napisany jest w C) i może być wykorzystywany pod UNIXem, DOSem, OS/2 i VMSem.

   Program dvips:

   - pozwala na druk w poprzek strony, inaczej mówiąc, w trybie *landscape* (`\special{landscape}`),
   - pozwala na obracanie fragmentów tekstu (pudełek) o 90, 180, 270 stopni (rotate.sty),
   - umożliwia włączanie do tekstu rysunków PostScriptowych (encapsulated PostScript), oraz ich skalowanie (epsf.sty),
   - umożliwia włączanie do tekstu zbiorów PostScriptowych (komenda `\special`) oraz ich skalowanie i obroty,
   - pozwala na włączanie do tekstu komend języka PostScript i uzyskiwanie przez to specjalnych efektów (niedostępnych normalnie w TEX-u ani w LATEX-u), np. można w ten sposób rysować koła o dowolnym promieniu, linie i wektory o dowolnym nachyleniu i dowolnej grubości (pspic.sty),
   - pozwala na wykorzystanie PostScriptowych fontów do składania tekstów (Times-Roman, AvantGarde, Bookman,...).

   Program dvips jest bardzo efektywny, pozwala na drukowanie dużych zbiorów (korzystających z bardzo wielu fontów) na drukarkach o niewielkiej pamięci. Pozwala on również na automatyczny podział tekstów zbyt wielkich dla używanej drukarki na mniejsze fragmenty. Program może współpracować z następującymi pakietami graficznymi: tpic, psfig, METAPOST.

3. **Ghostscript**

   Wynikowy plik możemy wydrukować bezpośrednio na drukarce PostScriptowej, a także wyświetlić na ekranie lub wydrukować na dowolnej

innej drukarce, wykorzystując interpreter tego języka Ghostscript. Ghostscript - rozpowszechniony jest w postaci kodu zródłowego i może być skompilowany na bardzo wielu platformach sprzętowych. Potrafi zapisywać obraz strony w postaci mapy bitowej. Rozpowszechniony wraz z zestawem fontów Public Domain.

1.2 Protokół HTTP

Protokół HTTP (Hyper Text Transfer Protocol) definiuje sposób komunikacji pomiędzy przeglądarkami i serwerami WWW. Aplikacje pracujące z protokołem HTTP wykonują trzy podstawowe operacje: poszukują zasobów, pobierają dane oraz wyświetlają dodatkowe informacje. Aby nawiązać połączenie z serwerem, przeglądarka musi znać URL (Uniform Resource Locator), który jednoznacznie określa lokalizację interesującego nas zasobu w internecie. Takim zasobem może być dokument HTML udostępniany przez serwer HTTP, jak i plik na serwerze FTP czy artykuł przechowywany na serwerze grup dyskusyjnych. Po otrzymaniu URL'a o postaci http://www.febus.pl/akademia.shtml i skontaktowaniu się z serwerem przeglądarka nawiązuje połączenie TCP. Teraz rozpoczyna się komunikacja oparta na protokole HTTP. Przeglądarka wysyła zapytania HTTP, na które serwer odpowiada. Żądanie przeglądarki i odpowiedź serwera nazywają się komunikatami HTTP. Każda linia zapytania zawiera nazwę metody, po której umieszczony jest adres żądanego dokumentu i numer wersji HTTP (używanej przez klienta).

Metody żądań w protokole HTTP:

- GET - żądanie zasobu,
- HEAD - jak GET ale zwraca tylko nagłówek,
- OPTIONS - sprawdzenie jakie metody udostępnia serwer,
- POST - przesłanie danych do serwera (używane przy formularzach).

Przykładowe żądanie może wyglądać w następujący sposób:

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

Transakcja w protokole HTTP dzieli się na cztery etapy:

1. utworzenie połączenia - tworzone jest połączenie TCP/IP (port nr 80) z serwerem HTTP,
2. wysłanie żądania (request) - zazwyczaj jest to żądanie pobrania pliku z serwera HTTP,
3. odpowiedź serwera (response) - serwer wysyła komunikat z odpowiedzią, zawierający wersję HTTP, kod stanu i opis (dane dotyczące serwera i dokumentu żądanego przez klienta,

4. zakończenie połączenia - przerwanie przez serwer połączenia TCP/IP.

Protokół HTTP jest zaliczany do protokołów stateless (bezstanowy), z racji tego, że nie zachowuje żadnych informacji o poprzednich transakcjach z klientem, po zakończeniu transakcji wszystko "przepada".

1.3 Apache - serwer HTTP

Serwer WWW jest to oprogramowanie odpowiedzialne za akceptowanie zadań klienta, odszukiwanie określonych plików, uruchamianie skryptów PHP i zwracanie ich zawartości (lub wyników działania skryptów). Większość serwerów pracujących w sieci Internet, to serwery pracujące na maszynach UNIX. Do głównych typów serwerów WWW możemy zaliczyć: Serwer NCSA, Serwer Apache, Serwer CERN i Serwery Netscape. Z wymienionych serwerów najbardziej popularnym i najczęściej stosowanym serwerem jest Apache. W pakiecie Meta-HTML dostępny jest samodzielny, w pełni funkcjonalny serwer HTTP.

Serwer Apache pochodzi od serwera NCSA (National Center for Supercomputing Applications) opracowanego w 1995 roku na zlecenie rządu Stanów Zjednoczonych. Jego dominacja na rynku serwerów WWW jest ogromna i wynosi ok. 67%[1]. Popularność serwera Apache jest pochodną popularności samego systemu Linux oraz faktu, że zarówno system, jak i serwer są całkowicie bezpłatne dla wszelkich zastosowań. Możliwości serwera pozwalają na publikację w Internecie wszystkich typów stron, począwszy od zwyczajnej strony firmowej, a skończywszy na dużych serwisach połączonych z bazami danych. Skalowalność oraz dostępność na praktycznie wszystkie liczące się platformy sprzętowe i programowe jest możliwa budowie modularnej, która pozwala na tworzenie rozwiązań wieloplatformowych, szybkich oraz stabilnych.

Serwer Apache ma minimalne wymagania sprzętowe - małe (np. testowe) serwisy WWW mogą pracować nawet na komputerach klasy 486. Minimalizuje to w znacznym stopniu inwestycje związane z uruchomieniem usługi WWW. Serwer Apache umożliwia obsługę m.in. skryptów CGI, PHP (liczniki odwiedzin, księgi gości, statystyki, formularze itp.). Stanowi podstawę do testowania nowo utworzonych stron przed publikacją ich w sieci. Serwer Apache możemy również wykorzystywać do integracji z bazami danych za pomocą Meta-HTML, PHP czy Perla. Rozwojem serwera zajmuje się zespół wybranych, doświadczonych programistów tzw. Apache Group, którzy wraz z użytkownikami rozwijają program jako Apache Project, a główna baza znajduje się pod adresem http://www.apache.org/, gdzie znajdziemy informacje na temat samego serwera.

Do głównych zadań serwera WWW należy:

- pracować szybko bez powodowania obciążeń komputera, na którym jest uruchomiony,

[1]Dane z maja 2004 roku uzyskane na podstawie [10].

- tryb wielozadaniowy: możliwość obsługi jednocześnie większej ilości zadań,
- kontrola nad użytkownikami,
- uzgodnienie formy i języka komunikacji co sprawia zdolność serwera do porozumienia się z klientem w danym języku,
- udostępnianie danych w różnych formatach,
- praca w charakterze serwera pośredniczącego (proxy server),
- bezpieczeństwo danych.

1.4 Moduły serwera Apache

Serwer Apache składa się z modułów oprogramowania. Moduły są samodzielnymi blokami kodu źródłowego, obsługującymi określone funkcje programu Apache. Poszczególne moduły mogą być w kompilowane w serwer lub załadowane w czasie działania. Możemy także tworzyć własne moduły. Aby włączyć dany moduł do kompilowanego kodu, wystarczy usunąć znak komentarza z początku odpowiedniego wiersza pliku Configuration. Spis wszystkich modułów znajduje się na serwerze FTP Apache [9].

Oprócz standardowych modułów niezbędnych do pracy, Apache dysponuje wieloma modułami dodatkowymi. Dodatkowe moduły Apache pozwalają m.in. na automatyczne poprawianie błędów w nazwach URL, modyfikację standardowych nagłówków HTTP, dają rozszerzone możliwości raportowania pracy serwera, uruchomienia funkcji proxy. Możemy je dynamicznie dołączać do programu dzięki wykorzystaniu techniki DSO (Dynamic Shared Object), co przyczynia się do optymalnego wykorzystania pamięci komputera.

Przy instalacji serwera powinniśmy wiedzieć, które moduły będą nam potrzebne do pracy. Nie należy bezmyślnie włączać wszystkich modułów, gdyż im jest ich mniej, mniejsza jest objętość wygenerowanego kodu wynikowego, a co za tym idzie, wydajniej działa serwer. Ponadto instalacja zbędnych modułów zmniejsza potencjalne bezpieczeństwo systemu.

Rozdział 2

Ideowy schemat działania modułu

Celem mojej pracy licencjackiej jest stworzenie takiego modułu serwera Apache, który będzie realizował zadanie polegające na generowaniu dokumentów PDF. Zadaniem modułu jest utworzenie plików w formacie PDF z pliku latexowego. Sam plik latexowy, czyli plik wejściowy dla naszego modułu, tworzony jest niezależnie przy pomocy dowolnego oprogramowania. Prawdopodobnie najczęściej używany będzie Meta-HTML, PHP lub Perl jako wygodne języki skryptowe działające po stronie serwera HTTP. Generowany tekst w LaTeX-u może zawierać dane zgromadzone w bazie SQL, np. MySQL. Wspomniane języki skryptowe dostarczają wygodny interfejs dostępu do takiej bazy danych. Nie ma więc sensu tworzenie nowego interfejsu, a przez to zmuszanie potencjalnych użytkowników do uczenia się nowego języka. Zatem wygenerowanie dokumentu latexowego odbywa się poza naszym modułem. Nasz moduł ma jedynie za zadanie przetworzyć dokument latexowy w dokument PDF lub któryś z formatów pośrednich tzn. DVI albo PS.

Zgodnie z konwencją nazewnictwa modułów serwera Apache dla naszego modułu przyjęliśmy nazwę `mod_pdf`.

2.1 Przyjęte założenia dotyczące modułu

Moduł `mod_pdf`:

1. współpracuje z Meta-HTML, PHP, Perlem i innymi językami bez ograniczeń,
2. rozumie tylko żądania typu GET,
3. jest modułem typu *request-handler* i obsługuje dokumenty typu *application/pdf*, *application/postscript*, *application/x-dvi*, *application/x-tex*,
4. musi znać treść żądania,

5. musi znać ścieżkę do pliku źródłowego na podstawie którego generuje odpowiedź,

6. potrafi zinterpretować dodatkowe parametry przekazane w żądaniu,

7. dba o to, by pliki tymczasowe powstające w trakcie generowania odpowiedzi miały unikalne nazwy.

Moduły typu *request-handler* realizują żądania wysłane z przeglądarki i generują odpowiedzi. Typowym przykładem takiego modułu jest moduł PHP oraz stworzony w ramach ubiegłorocznej pracy dyplomowej moduł MetaHTML. Nazwy typów dokumentów obsługiwanych przez `mod_pdf` zostały ustalone na podstawie oficjalnego standardu znajdującego się w [12].

Dodatkowe parametry, o których wyżej mowa, to podstawowe parametry sterujące orientacją i układem stron. Moduł `mod_pdf` rozumie dwa prametry:

o — orientacja strony (skrót od ang. *orientation*). Możliwe są dwie wartości: `p` i `l` (odpowiednio *portrait* i *landscape*). Nadanie `o` wartośći `l` w żądaniu powoduje zawołanie programu `dvips` z parametrem `-t landscape`. Domyślna wartość `o` to `p`.

l — układ strony (skrót od ang. *layout*). Ten parametr może przyjąć jedną z trzech wartości: `1`, `2` i `b`. Oznaczają one odpowiednio: jedna strona na arkusz, dwie strony na arkusz i *booklet* czyli takie przetasowanie stron, aby umożliwić złożenie arkusza A4 w połowie i uzyskanie czegoś w rodzaju książki. Zauważmy, że opcja `b` pociąga za sobą `2`. Domyślna wartość `l` to `1`.

Powyższe parametry przekazuje się w żądaniu w następujący sposób:

```
GET /dokument.pdf?o=l&l=2 HTTP/1.0
```

przy czym kolejność parametrów i wielkość liter nie ma znaczenia. Opuszczenie, któregokolwiek z nich powoduje ustawienie wartości domyślnej.

Przypomnijmy, że w żądaniu HTTP znaki: `?` i `&` mają specjalne znaczenie. Znak `?` oddziela nazwę żądanego dokumentu od parametrów, natomiast znak `&` oddziela parametry od siebie. Jak się łatwo domyśleć napis znajdujący się z lewej strony znaku `=` jest nazwą parametru, a napis z jego prawej strony to wartość parametru. Zarówno nazwa, jak i wartość, może być dłuższym napisem niż jeden znak. Używamy tutaj jednoliterowych oznaczeń dla uproszczenia implementacji modułu i zaoszczędzenia na bajtach przesyłanych w sieci Internet.

Mówiąc o założeniach co do działania modułu przypomnijmy, zapytania dotyczące jednego dokumentu mogą napływać równocześnie. Dlatego musimy zadbać o to, by ewentualne pliki tymczasowe związane z jednym zapytaniem

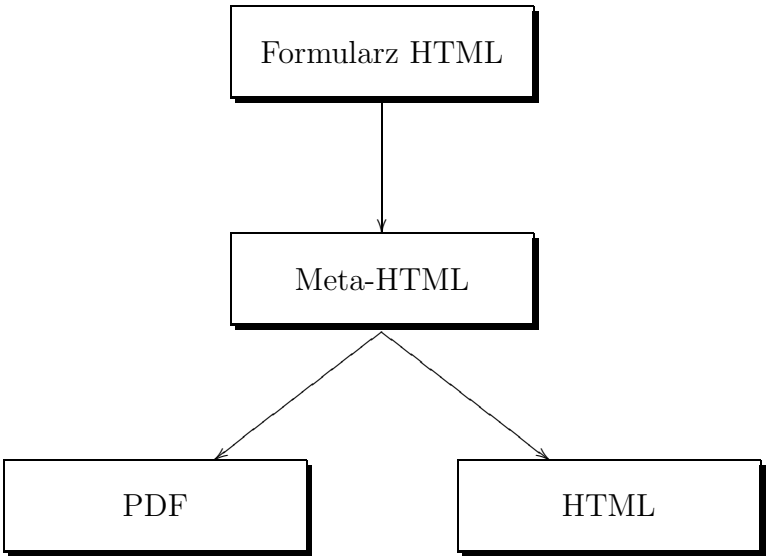
nie kolidowały z innym zapytaniem. Z uwagi na dynamiczny charakter odpowiedzi naszego modułu nie jest możliwe, aby wykorzystać wielokrotnie raz utworzoną odpowiedź w postaci pliku DVI, PS lub PDF.

Warto być może również w tym miejscu przypomnieć, że pliki źródłowe tworzone przez programy zewnętrzne, czyli Meta-HTML, PHP, itp., także powinny mieć unikalne nazwy, nawet jeśli związane są z jednym formularzem i jednym zapytaniem do bazy danych. Dość dobrym rozwiązaniem, ale niedoskonałym, jest umieszczanie daty i godziny w nazwie pliku.

2.2 Typowe zastosowanie modułu

Możliwe jest zastosowanie modułu `mod_pdf` do tworzenia dokumentów w formacie DVI, PS lub PDF na podstawie statycznych plików źródłowych. Jeśli jednak taki plik latexowy rzadko się zmienia nie warto obciążać serwer, a raczej należy go ręcznie przetworzyć do formatu w jakim będzie pobierany przez Internet. Typowym zastosowaniem modułu jest sytuacja, gdy plik źródłowy ma charakter dynamiczny i zmienia się przy każdym żądaniu.

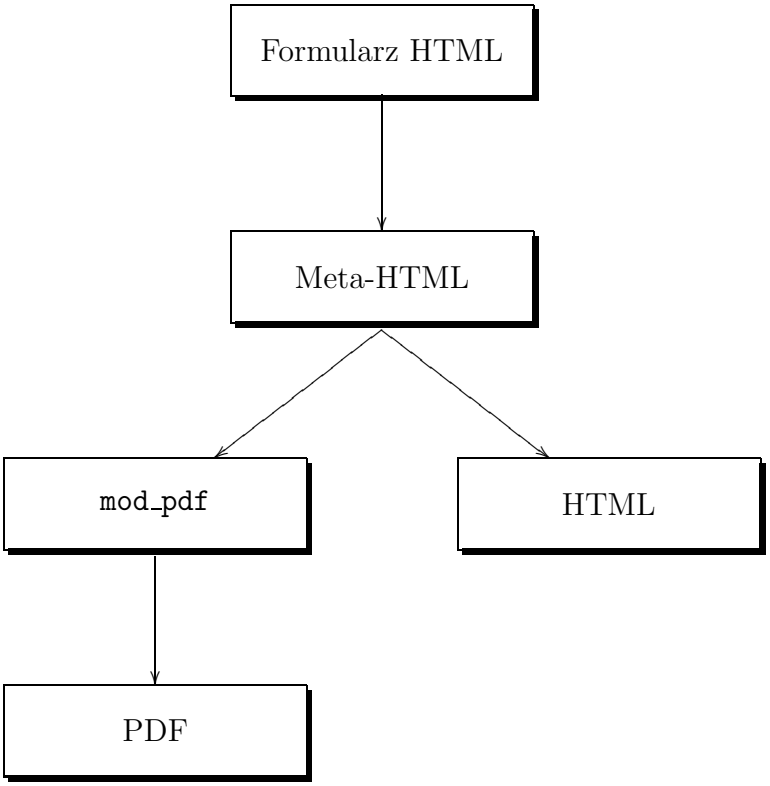
Funkcja jaką pełni nasz moduł jest dobrze widoczna na poniższym przykładzie, gdzie najpierw dokument PDF jest w całości generowany przy pomocy Meta-HTML, a następnie wykorzystywany jest moduł `mod_pdf`.



Rysunek 2.1: Przykładowy scenariusz generowania dokumentu PDF przy pomocy Meta-HTML.

W formularzu HTML wybieramy zakres danych z bazy, które chcemy mieć w dokumencie i określamy tam również parametry, co do sposobu tworzenia takiego dokumentu, np. orientację tekstu, umieszczenie jednej lub dwóch stron

na jednym arkuszu, format dokumentu (LaTeX, DVI, PS lub PDF). Zawartość formularza otrzymuje Meta-HTML. W następnym kroku Meta-HTML generuje tekst źródłowy, odwołując się przy tym do bazy danych i dalej woła LaTeX, dvips oraz ps2pdf. W ten sposób powstaje dokument w formacie PDF. Jednocześnie Meta-HTML tworzy stronę HTML, za pośrednictwem której użytkownik może pobrać dokument PDF. Zazwyczaj umieszcza się na tej stronie odnośnik do utworzonego dokumentu PDF. Nie jest możliwe pominięcie generowania strony HTML, ponieważ Meta-HTML jako odpowiedź dla przeglądarki może odesłać jedynie tekst HTML. Może to być tekst pusty, ale nie będzie wtedy możliwe odwołanie się do dokumentu PDF.



Rysunek 2.2: Przykład zastosowania modułu `mod_pdf`.

W podanym przykładzie na rysunku 2.2 moduł `mod_pdf` przejmie część zadań Meta-HTML związanych z przetwarzaniem tekstu latexowego do formatu PDF. W stosunku do sytuacji z rysunku 2.1 zmienia się wyłącznie sposób tworzenia dokumentu w formacie PDF. Wołanie programów LaTeX, `dvips`, `ps2pdf` przejmuje od Meta-HTML moduł `mod_pdf`, który nie musi nic wiedzieć o Meta-HTML, czy innym języku. Jest mu potrzebny jedynie tekst latexowy. Między danym modułem a Meta-HTML nie ma bezpośredniej komunikacji. `mod_pdf` nie komunikuje się również z formularzem HTML. Celem jest nie tylko odciążenie Meta-HTML, PHP czy Perla. Chcemy z jednej strony zestandaryzować proces, z drugiej dać możliwość łatwiejszego i skuteczniejszego zarządzania

tym procesem (dubugging, logi).

2.3 Przykładowa realizacja żądania

Aby lepiej zrozumieć sposób działania modułu `mod_pdf` prześledzimy na przykładzie kolejne akcje od żądania do odpowiedzi.

1. **Żądanie HTTP**

   ```
   GET /dokument.pdf HTTP/1.0
   ```

   Klient łączy się z serwerem. Następnie wysyła żądanie dokumentu, zawierające polecenie HTTP z nazwą metody GET, po której umieszczony jest adres dokumentu i numer wersji HTTP. Metoda GET jest typową metodą używaną przez przeglądarki do pobierania dokumentów z serwera HTTP.

2. **Uruchomienie modułu**

   Kiedy serwer Apache odbierze żądanie pobrania pliku `dokument.pdf` od przeglądarki, uruchamia moduł `mod_pdf` do obsługi tego żądania.

3. **Odszukiwanie dokumentu określonego w żądaniu**

   Zadaniem modułu jest zlokalizowanie odpowiedniego dokumentu latexowego, na podstawie którego ma być wygenerowany `dokument.pdf`. W konfiguracji serwera Apache określony jest katalog, w którym serwer będzie szukał pliku `dokument.pdf`, dla ustalenia uwagi niech to będzie `/export/home1/httpd/htdocs`. Moduł sprawdza, czy w tym katalogu jest plik `dokument.pdf`. Jeśli tak, to zwraca go klientowi i kończy działanie. W przeciwnym razie przyjmuje się, że plikiem źródłowym jest `/export/home1/httpd/htdocs/dokument.tex`.

4. **Przetwarzanie pliku źródłowego**

   Zakładamy w tym miejscu, że nie było gotowego pliku `dokument.pdf`. Musimy zatem go wygenerować. W tym celu z poziomu modułu wołany jest LaTeX, `dvips`, `ps2pdf`. Jeśli nie pojawiły się żadne błędy w czasie przetwarzania, to wynikowy dokument wysyłany jest do klienta i moduł kończy działanie.

W podanym powyżej przykładzie celowo unikamy różnych szczegółów technicznych tak, aby dać ogólny zarys sposobu działania modułu. Dla uproszczenia pominęliśmy zupełnie kwestię dodatkowych parametrów sterujących orientacją i układem strony oraz związane z nimi dodatkowe programy `psbook` i `psnup`.

Rozdział 3

Implementacja i uruchamianie

3.1 Struktura `module`

Implementację modułu serwera Apache zaczyna się od wypełnienia struktury `module`, która stanowi coś w rodzaju metryczki modułu. Za pośrednictwem tej struktury jądro serwera wywołuje funkcje z modułu. Przetwarzanie żądania HTTP w serwerze Apache podzielone jest na dziesięć etapów i każdemu z nich odpowiada pole w omawianej strukturze. W naszym wypadku wypełniamy tylko pola odpowiedzialne za: inicjalizację modułu, spis opcji oraz listę programów obsługi. Wypełniona struktura przedstawiona jest poniżej.

```
module MODULE_VAR_EXPORT pdf_module =
  {
    STANDARD_MODULE_STUFF,
    pdf_init,                          /* module initializer */
    NULL,                              /* per-directory config creator */
    NULL,                              /* dir config merger */
    NULL,                              /* server config creator */
    NULL,                              /* server config merger */
    pdf_cmds,                          /* command table */
    pdf_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
};
```

Najistotniejsze dla nas jest pole listy programów obsługi (oznaczone liczbą

9). W polu tym znajduje się wskaźnik do listy przedstawionej poniżej.

```
static const handler_rec pdf_handlers[] =
  {
    { "application/pdf", pdf_handle_req_pdf },
    { "application/postscript", pdf_handle_req_ps },
    { "application/x-dvi", pdf_handle_req_dvi },
    { "application/x-tex", pdf_handle_req_tex },
    { NULL }
  };
```

Dla każdego z obsługiwanych przez nasz moduł typów dokumentów przypisujemy oddzielną funkcję. Odpowiednio skonfigurowany serwer Apache rozpozna typ żądanego przez klienta dokumentu na podstawie rozszerzenia pliku i wywoła związaną z tym typem funkcję. Poniżej zamieszczamy wymagany fragment konfiguracji serwera Apache.

```
AddType application/pdf .pdf
AddType application/postscript .ps
AddType application/x-dvi .dvi
AddType application/x-tex .tex
```

Wartość `pdf_init` w srtukturze `module` to wskaźnik do funkcji inicjalizującej nasz moduł. Dopisuje ona numer wersji modułu `mod_pdf` do nazwy serwera Apache przekazywanej klientom w nagłówkach odpowiedzi. Natomiast wartość `pdf_cmds` to wskaźnik do listy dyrektyw konfiguracyjnych naszego modułu. Zaimplementawaliśmy następujące dyrektywy:

`PDFWorkingDirectory` — ścieżka do katalogu roboczego, domyślnie `/tmp`,

`PDFExecPath` — lista ścieżek rozdzielonych znakiem `:`, w których znajdują się wykonywalne programy, domyślnie `/usr/bin`,

`PDFlatex` — nazwa lub pełna ścieżka do programu LaTeX, domyślnie `latex`,

`PDFdvips` — nazwa lub pełna ścieżka do programu DVIPS, domyślnie `dvips`,

`PDFpsbook` — nazwa lub pełna ścieżka do programu PSBOOK, domyślnie `psbook`,

`PDFpsnup` — nazwa lub pełna ścieżka do programu PSNUP, domyślnie `psnup`,

`PDFgs` — nazwa lub pełna ścieżka do programu GS, domyślnie `gs`,

`PDFDebugLevel` — liczba określająca poziom debuggowania, domyślnie 0, czyli brak debuggowania.

3.2 Funkcje modułu `mod_pdf`

W skład naszego modułu wchodzą następujące pliki:

`mod_pdf.c` — podstawowy plik źródłowy modułu, tutaj zawarte są wszystkie funkcje bezpośrednio odwołujące się do API serwera Apache,

`utils.c` — zestaw użytecznych funkcji, które moga być wykorzystane w dowolnym programie, niezależnie od środowiska serwera Apache,

`utils.h` — plik nagłówkowy, deklarujący stałe i prototypy funkcji z `utils.c`,

`Makefile` — zestaw reguł dla programu `make` automatyzującego proces kompilacji kodu źródłowego naszego modułu.

Powyższe pliki zamieszczono w dodatku C na stronie 31.

Elementy API serwera Apache użyte w module `mod_pdf` opisane zostały w dodatku A na stronie 26. Funkcje z tego API prefiksowane są przez `ap_`, co ułatwia ich rozpoznanie w kodzie.

Działanie modułu sprowadza się do wołania dwóch funkcji w zależności od typu żądanego dokumentu. Jeśli żądanym dokumentem jest plik z rozszerzeniem `tex`, to wołana jest funkcja `pdf_handle_req_tex()`. Nie robi ona nic innego poza tym, że odsyła żądany plik do klienta, ewentualnie odnotowuje w logu błąd jeśli takiego pliku nie ma. W przypadku natomiast, gdy klient żąda plik o rozszerzeniu `dvi`, `ps` lub `pdf`, to realizacja żądania sprowadza się do funkcji `generate_requestet_file()`. Ponieważ funkcja ta jest najważniejszą częścią modułu przedstawimy ją w postaci komentarzy do kodu źródłowego (opuszczamy komentarze napisane w języku angielskim). Prototyp funkcji wygląda następująco:

```
static int
generate_requested_file(request_rec *r,
                        int reqformat,
                        const char *reqext)
```

Funkcja pobiera:

- wskaźnik do struktury `request_rec` zawierającej wszelkie dane na temat żądania i klienta,
- liczbę określającą format żądanego dokumentu, 0 dla `tex`, 1 dla `dvi`, 2 dla `ps` lub 3 dla `pdf`,
- rozszerzenie żądanego pliku.

Jako wynik funkcja zwraca liczbę całkowitą odpowiadającą statusowi HTTP. Wartość ta zależy od tego, czy pojawiły się błędy i jakie to były błędy. Wartość

0 oznacza pomyślne wykonanie funkcji i odpowiada jej status `HTTP_OK` (w skrócie `OK`). Lista wszystkich możliwych wartości podana jest w dodatku B.

Zaczynamy od deklaracji niezbędnych zmiennych:

```
char *srcfile = (char *)NULL;
char *trgfile = (char *)NULL;
char *tmpdir = (char *)NULL;
char *command = (char *)NULL;
char *output = (char *)NULL;
char *errors = (char *)NULL;
char orientation = PORTRAIT;
char layout = ONEPAGE;
FILE *fd;
int result = SERVER_ERROR;
```

Mają one następujące znaczenie:

`srcfile` — pełna ścieżka do pliku źródłowego `tex`,

`trgfile` — pełna ścieżka do generowanego pliku,

`tmpdir` — pełna ścieżka do katalogu roboczego,

`command` — skrypt shellowy generujący żądany plik,

`output` — tekst wypisywany przez powyższy skrypt na STDOUT,

`errors` — tekst wypisywany przez powyższy skrypt na STDERR

`orientation` — orientacja strony wynikowego dokumentu,

`layout` — układ strony wynikowego dokumentu,

`fd` — deskryptor pliku,

`result` — wynik działania funkcji.

Stałe `PORTRAIT` i `ONEPAGE` zostały zdefiniowane dla poprawienia czytelności kodu. Mają one wartość odpowiednio `'p'` i `'1'`. Poza nimi są jeszcze stałe `LANDSCAPE`, `TWOPAGE` i `BOOKLET` o wartościach `'l'`, `'2'` i `'b'`. Odpowiadają one parametrom sterującym tworzenie PDF.

Przed wykonaniem zadania sprawdzamy, czy użyta w żądaniu metoda jest metodą GET. Jeśli nie, to kończymy wykonanie funkcji, gdyż nasz moduł obsługuje tylko metodę GET.

```
if ((result = check_method(r)) != OK) {
    return result;
}
```

Sprawdzamy czy żądany plik istnieje. Jeśli tak, to wysyłamy go do klienta i opuszczamy funkcję.

```
if (r->finfo.st_mode != 0)
    return send_file_to_client(r, r->filename);
```

W przypadku gdy poziom debuggowania jest niezerowy w logu zapisujemy pełną ścieżkę do żądanego przez klienta pliku.

```
if (0 < pdf_debug_level)
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
            "Requested file: %s", r->filename);
```

Wyliczamy ścieżkę do pliku źródłowego:

```
if ((srcfile = replace_file_ext(r->filename, reqext, ".tex"))
                                                    == NULL) {
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
            MODNAME ": cannot convert filename:
                    %s from %s to .tex", r->filename, reqext);
    return SERVER_ERROR;
}
```

Funkcja `replace_file_ext()` w żądanym pliku `r->filename` w miejsce rozszerzenia podanego w `reqext` wstawia `.tex`. Uzyskujemy w ten sposób ścieżkę do pliku źródłowego i zapamiętujemy ją w `srcfile`. Jeśli roszerzenie do zamiany nie zostanie znalezione lub zabraknie pamięci aby zaalokować nowy napis `replace_file_ext()` zwróci NULL.

Sprawdzamy, czy plik źródłowy istnieje. W tym celu próbujemy otworzyć go do czytania. Jeśli się, zamykamy go i przechodzimy dalej. W przeciwnym razie zgłaszamy błąd w logu i przerywamy.

```
if ((fd = ap_pfopen(r->pool, srcfile, "r")) == NULL) {
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
            MODNAME ": file not found: %s", srcfile);
    free(srcfile);
    return NOT_FOUND;
}
ap_pfclose(r->pool, fd);
```

Tworzymy nazwę tymczasowego podkatalogu w katalogu roboczym określonym w zmiennej globalnej `pdf_working_directory`.

```
tmpdir = tempnam(pdf_working_directory, "dpdf");
```

Wołamy w tym celu funkcję systemową `tempnam()`, podając jako katalog, gdzie należy utworzyć plik o unikalnej nazwie, wartość `pdf_working_directory` i jako stały prefiks napis `dpdf` (skrót od „Dynamiczny PDF"). Jeśli ciąg wskazywany przez `pdf_working_directory` ma wartość `/tmp` to uzyskamy np. `/tmp/dpdfAAA.LaWVd`.

Przy odpowiednim poziomie debuggowania w logu zapisujemy wartość `tmpdir`.

```
if (0 < pdf_debug_level)
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
            "Temporary file: %s", tmpdir);
```

Tworzymy nazwę pliku wynikowego w ten sposób, że do `tmpdir` dopisujemy rozszerzenie podane w `reqext` odpowiadające formatowi żądanego dokumentu. Najpierw rezerwujemy odpowiednią ilość pamięci. Jeśli się nie uda, zwalniamy zajętą do tej pory pamięć i wychodzimy.

```
if ((trgfile = (char *)malloc(strlen(tmpdir) +
                                    strlen(reqext) + 2)) == NULL) {
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
          MODNAME ": insufficient memory to allocate: trgfile");
    free(srcfile);
    free(tmpdir);
    return SERVER_ERROR;
}
```

Następnie korzystamy z systemowej funkcji `sprintf()` aby połączyć dwa napisy: `tmpdir` i `reqext`. Wynik umieszczany jest w zmiennej `trgfile`.

```
sprintf(trgfile, "%s%s", tmpdir, reqext);
```

Jeśli potrzeba, to zapisujemy nazwę tego pliku w logu do ewnetualnej diagnostyki.

```
if (0 < pdf_debug_level)
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
            "Target file: %s", trgfile);
```

Teraz utworzymy skrypt shellowy do wygenerowania żądanego pliku. Jak zwykle należy zacząć od zaalokowania pamięci. Rezerwujemy 1KB-owy blok pamięci i jego adres zapamiętujemy w zmiennej `command`. Gdy nie ma tyle wolnej pamięci operacyjnej zwalniamy zajętą do tej pory pamięć i przerywamy.

```
if ((command = (char *)malloc(BIGSTRING)) == NULL) {
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
          MODNAME ": insufficient memory to allocate: command");
    free(srcfile);
    free(trgfile);
    free(tmpdir);
    return SERVER_ERROR;
}
```

Na początku nasz skrypt ma utowrzyć tymszasowy katalog roboczy, wejść do niego i skopiować tam plik źródłowy nadając mu nazwę `src.tex`.

```
snprintf(command, BIGSTRING-1, "mkdir %s; cd %s; cp %s src.tex",
        tmpdir, tmpdir, srcfile);
```

Wykorzystujemy tutaj funkcję systemową `snprintf()`. Jest to podobna funkcja do `sprintf()` z tym, że kontroluje ile bajtów można maksymalnie zapisać do bufora wynikowego, tutaj jest nim obszar pamięci wskazywany przez zmienną `command`.

Kolejne wołania `snprintf()` dopisują do tego co już jest w `command` kolejne fragmenty skryptu.

```
if (DVI <= reqformat) {
    snprintf(command, BIGSTRING - 1, "%s; %s
            --interaction batchmode src", command, pdf_latex);
    snprintf(command, BIGSTRING - 1, "%s; %s
            --interaction batchmode src", command, pdf_latex);
    snprintf(command, BIGSTRING - 1, "%s; %s
            --interaction batchmode src", command, pdf_latex);
}
```

Jeśli formatem pośrednim do docelowego jest DVI, to wołamy LaTeX, trzykrotnie, aby upewnić się, że wszystkie spisy i referencje są poprawne. Opcja `--interaction batchmode` mówi programowi TeX, aby nie wypisywać nic na standardowe wyjście (w przypadku konsoli – na ekran), inaczej mówiąc jest to tzw. tryb *quiet*.

Dalej, jeśli jest taka potrzeba generujemy Postscript. Zaczynamy od sprawdzenia dodatkowych parametrów z żadania od klienta. Funkcja `parse_request()` jest odpowiedzialna za analizę treści żadania `r->the_request` i odpowiednie ustawienie wartości zmiennych `orientation` i `layout`. Funkcja ta została zamieszczona w pliku `utils.c`.

```
if (PS <= reqformat) {
  if ((result = parse_request(r->the_request,
                              &orientation, &layout)) != OK)
      return result;

  if (0 < pdf_debug_level)
      ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
          "Request: %s, orientation: %c, layout: %c",
          r->the_request,
          orientation == '\0' ? PORTRAIT : orientation,
          layout == '\0' ? ONEPAGE : layout);

  if (orientation == LANDSCAPE)
      snprintf(command, BIGSTRING - 1, "%s; %s -q -o src.ps
          -t landscape src", command, pdf_dvips);
  else
      snprintf(command, BIGSTRING - 1, "%s; %s -q -o src.ps
          src", command, pdf_dvips);

  if (layout == BOOKLET)
      snprintf(command, BIGSTRING - 1, "%s; %s -q src.ps
          src-b.ps; mv src-b.ps src.ps", command, pdf_psbook);
```

```
    if (layout == TWOPAGE || layout == BOOKLET)
        snprintf(command, BIGSTRING - 1, "%s; %s -q -2 src.ps
                src-2.ps; mv src-2.ps src.ps", command, pdf_psnup);
}
```

Jeśli debuggowanie jest włączone, w logu zapisywana jest treść żądania oraz wartości zmiennych `orientation` i `layout`. Jeśli nie podano podano orientacji i układu, wypisywane są wartości domyślne.

Jeśli orientacja strony ma być pozioma program `dvips` wołany jest z opcją `-t landscape`. Parametr `-q` wymusza tryb *quiet*, natomiast `-o` wskazuje plik wynikowy. Używamy opcję `-o`, gdyż standardowo `dvips` wysyła wynik na drukarkę, nie do pliku.

Gdy żadanao układu typu *booklet*, wołany jest program `psbook`. Program `psnup` z opcją `-2` wołany jest gdy mają być umieszczone dwie strony na arkuszu, albo gdy układ jest typu *booklet*. W każdym wypadku używamy opcji `-q` by wymusić tryb *quiet*.

Ostatecznie zamiast programu `ps2pdf` postanowiliśmy bezpośrednio korzystać z programu `gs`. Daje to lepszą kontrolę nad tym co się dzieje.

```
if (PDF <= reqformat) {
    snprintf(command, BIGSTRING - 1, "%s; %s -q
        -dSAFER -dNOPAUSE -dBATCH -sDEVICE=pdfwrite
        -dCompatibilityLevel=1.4
        -dEmbedAllFonts=true -sPAPERSIZE=a4
        -sOutputFile=src.pdf src.ps", command, pdf_gs);
}
```

Znaczenie poszczególnych opcji jest następujące:

`-q` — tryb *quiet*,

`-dSAFER` — wszystkie pliki poza wynikowym otwierane są tylko do odczytu,

`-dNOPAUSE` — powoduje, ze kolejne strony są przetwarzane bez przerw,

`-dBATCH` — wyjście po skończeniu przetwarzania, w przeciwnym razie `gs` czeka na kolejne polecenia,

`-sDEVICE` — określenie urządzenia wynikowego, tutaj jako formatu PDF,

`-dCompatibilityLevel` — wersja formatu PDF,

`-dEmbedAllFonts` — włączanie definicji fontów do dokumentu PDF,

`-sPAPERSIZE` — rozmiar papieru,

`-sOutputFile` — nazwa pliku wyjściowego.

Na zakończenie, przenosimy plik wynikowy z tymczasowego katalogu roboczego o poziom wyżej i nadajemu my nazwę określoną w `trgfile`. Usuwamy katalog roboczy.

```
snprintf(command, BIGSTRING - 1, "%s; mv src%s %s; cd %s;
   rm -rf %s", command, reqext, trgfile, pdf_working_directory,
                                                        tmpdir);
```

Wygenerowany skrypt zapisujemy w logu, jeśli tak zdecydowano w konfiguracji modułu.

```
if (0 < pdf_debug_level)
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
            "Command to execute: %s", command);
```

Teraz wywołujemy nasz skrypt przy pomocy funkcji `shell_execute()`.

```
if (! shell_execute(command, pdf_exec_path,
                                        &output, &errors, 1)) {
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
        MODNAME ": command %s failed: output %s, errors: %s ",
            command,
            output ? output : "(NULL)",
            errors ? errors : "(NULL)");
    free(srcfile);
    free(trgfile);
    free(tmpdir);
    free(command);
    return SERVER_ERROR;
}
```

Jeśli w trakcie wykonania skryptu powstanie jakiś problem lub błąd, zwalniamy pamięć i kończymy zgłaszając błąd.

Ważne jest to, co zawiera `errors`. Spisane są tam ewentualne błędy wykonania skryptu, czyli błędy zgłaszane przez kolejno wołane programy.

Funkcja `shell_execute()` wraz z jej funkcjami pomocniczymi zostały zapożyczone z kodu programu Meta-HTML. Ponieważ udostępniamy nasz program wraz ze źródłami i przyznajemy się do zapożyczenia nie naruszamy zasad licencji Meta-HTML.

`shell_execute()` ma bardzo złożony przebieg dla początkujących programistów jkimi jesteśmy. Dla zrozumienia działania modułu wystarczy powiedzieć, że uruchamiany jest skrypt przekazany w zmiennej `command`, obowiązywać będą ścieżki do programów z `pdf_exec_path`, to co zostaje skierowane na wyjście STDOUT umieszczone zostanie pod `output`, a zawartość STDERR pod `errors`. Ostatni parametr, jeśli jest różny od 0 powoduje, że wykonanie funkcji, którą opisujemy, nastąpi dopiero po zakończeniu `shell_execute()`. Dokładniej ten problem opisujemy w 3.4 na stronie 23.

Teraz, gdy skrypt już został wykonany, o ile włączony jest debugging, zapisujemy w logu to co pojawiłoby się na ekranie konsoli, gdyby skrypt uruchamiać ręcznie.

```
if (0 < pdf_debug_level) {
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
            "Output: %s", output ? output : "(NULL)");
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
            "Errors: %s", errors ? errors : "(NULL)");
}
```

W tym miejscu mamy już plik wynikowy, do którego pełna ścieżka podana jest w `trgfile`. Wysyłamy go do klienta.

```
result = send_file_to_client(r, trgfile);
```

Gdyby w trakcie wysyłania danych pojawił się błąd, zostanie on zapamiętany w zmiennej `result`.

Po wysłaniu, usuwamy plik wynikowy i zwalniamy zaalokowana pamięć.

```
remove(trgfile);

free(srcfile);
free(trgfile);
free(tmpdir);
free(command);

return result;
```

Funkcja zwraca jako wynik wartość `result`, która zeleży od tego jak przebiegło przesyłanie pliku do klienta.

3.3 Kompilacja

Praktycznie cały proces od założenia plików źródłowych modułu, przez kompilację do uruchamiania odybwał się w laboratorium komputerowym Instytutu Matematyki pod systemem Sun Solaris 9. Używaliśmy kompilatora C firmy Sun z pakietu Sun ONE Studio 8, Compiler Collection. Jest to komercyjny kompilator, ale dostępny dla studentów i pracowników naukowych bezpłatnie.

Początkowo cały kod modułu znajdował się w jednym pliku `mod_pdf.c`. W miarę rozbudowy wydzieliliśmy uniwersalne funkcje i umieściliśmy je w nowym pliku `utils.c`. Aby możliwa była kompilacja należało założyć plik `utils.h` i zamieścić w nim protoypy funkcji, z których chcemy korzystać poza `utils.c`. Taki plik nagłówkowy (rozszerzenie `.h` pochodzi od ang/ *header*) włącza się makropoleceniem `#include` preprocesora C tam, gdzie korzystamy z funkcji, stałych i zmiennych globalny z `utils.c`.

W celu usprawnienia kompilacji napisaliśmy kilka regół dla programu `make` i umieściliśmy je w `Makefile`. Pozbyliśmy się w ten sposób problemu z pamiętaniem, które pliki się zmieniły, co trzeba skompilować i w jakiej kolejności. Jesli nie ma błędów, moduł kompiluje się wołając

```
make
```

w katalogu gdzie znajduje się kod źródłowy.

Plik `Makefile` załączony jest w C.4 na stronie 45. Tajemniczy program `apxs` pochodzi z pakietu serwera Apache i wspomaga kompilacje oraz instalację modułów. Tak na prawdę jest to skrypt Perlowy, który woła kompilator C z odpowiednimi dla Apache'a parametrami.

3.4 Konfiguracja i uruchamianie

Aby uruchomić nasz moduł, należało po kompilacji umieścić plik biblioteczny `mod_pdf.so` w odpowiednim miejsu systemu, tak aby mógł być znaleziony przez program serwera Apache. Instalacje ułatwia wspomniany program `apxs`. Wystarczy wywołać

```
make install
```

Poza określeniem dodatkowych typów dokumentów w konfiguracji Apache, o czym była mowa na stronie 14, moduł `mod_pdf` może być konfigurowany. Dostępne opcje opisane były na stronie 14.

W trakcie uruchamiania do konfiguracji Apache dodaliśmy takie oto dyrektywy:

```
<IfModule mod_pdf.c>
  PDFWorkingDirectory /tmp
  PDFExecPath /usr/bin:/opt/bin:/opt/cfw/bin:/usr/sfw/bin:/opt/sfw/bin
  PDFDebugLevel 255
</IfModule>
```

Bardzo ważna jest dyrektywa `PDFExecPath`, gdyż od niej zależy, czy wszystkie programy, nawet jeśli są poprawnie zainstalowane, zostaną znalezione podczas przetwarzania w module.

Wartość `PDFDebugLevel` została ustawiona na 255 aby dostać możliwie dużo informacji o pracy modułu, nawet gdy nie ma błędów. Uzykane w ten sposób informacje w logu wyglądają jak poniżej:

```
(50):  Method: 0
(126): Requested file: /export/home/lab/public_html/cosik.pdf
(159): Temporary file: /tmp/dpdfBAAexa4Vd
(175): Target file: /tmp/dpdfBAAexa4Vd.pdf
(209): Orientation: p, layout: 2
(240): Command to execute:
  mkdir /tmp/dpdfBAAexa4Vd;
  cd /tmp/dpdfBAAexa4Vd;
```

```
  cp /export/home/lab/public_html/cosik.tex src.tex;
  latex --interaction batchmode src;
  latex --interaction batchmode src;
  latex --interaction batchmode src;
  dvips -q -o src.ps src;
  psnup -q -2 src.ps src-2.ps;
  mv src-2.ps src.ps;
  gs -q -dNOPAUSE -dBATCH -dCompatibilityLevel=1.4 -dSubsetFonts=true -dEmbedAllFonts=true
    -sDEVICE=pdfwrite -sPAPERSIZE=a4 -sOutputFile=src.pdf src.ps;
  mv src.pdf /tmp/dpdfBAAexa4Vd.pdf;
  cd /tmp;
  rm -rf /tmp/dpdfBAAexa4Vd
(257): Output: This is TeX, Version 3.14159 (Web2C 7.4.5)
This is TeX, Version 3.14159 (Web2C 7.4.5)
This is TeX, Version 3.14159 (Web2C 7.4.5)

(259): Errors:
(75): Sending file: /tmp/dpdfBAAexa4Vd.pdf
```

Liczba w nawiasie na początku oznacza numer wiersza pliku źródłowego. Dla poprawienia czytelności usunęliśmy powtarzające się w logu napisy i ręcznie złamaliśmy wiersze. Pierwszy wiersz w całości wyglądał tak:

```
[Mon Jun 26 14:47:37 2004] [debug] mod_pdf.c(50): [client 127.0.0.1] Method: 0
```

Tutaj wartość `Errors` jest pusta, więc ten przebieg był pomyślny. Jednak zanim uzyskaliśmy takie wyniki upłynęło wiele czasu. Największym problemem, było uruchomienie skryptu z poziomu modułu. Skrypt shellowy, jak nazwa wskazuje wymaga uruchomienia najpierw interpretera `/usr/bin/sh`, który pobierze nasz zestaw poleceń jako argument i wykona go.

Otóż w systemie wielozadaniowym jakim jest Unix, uruchomienie jednego programu z poziomu drugiego, nie jest łatwe, a już na pewno dla amatorów. Aby uruchomić program należy użyć systemową funkcję `exec()` lub którąś z jej odmian (u nas jest to `execve()`, która dodatkowow pozwala określić zmienne środowiskowe przed wykonaniem programu). Działa ona tak, że obraz aktualnie wykonywanego procesu, czyli przypisana mu pamięć na kod i dane, zastępowany jest obrazem uruchamianego procesu. Jest to proces nieodwracalny, zatem nie ma żadnej możliwości kontynuowania procesu, który wywołuje `exec()`. Typowy trick jaki się tutaj robi, to zawołanie wcześniej `vfork()`. Powoduje to utorzenie kopii procesu wołającego `vfork()`. Tak więc, po `vfork()` mamy w pamięci dwa procesy wykonywane równolegle od miejsca wywołania `vfork()`. Wykonują one ten sam program. Kopia, fachowo mówiąc *dziecko* (od ang. *child*) wie, że jest dzieckiem bo wartość jaką zwraca `vfork()` dla niego to 0, natomiast wartość zwrócona w procesie rodzicielskim to numer procesu dziecka czyli coś różnego od 0. Bardzo ważne w tym momencie jest to, że wykonanie procesu rodzicielskiego jest zawieszone do momentu, aż dziecko wykona `execve()`. Dzieje się tak dlatego, że oba procesu mają wspólny segment danych. Tworzenie kopii segmentu danych procesu rodzicielskiego przy większym procesie jakim jest proces `httpd` serwera Apache to kosztowna operacja (w sensie cykli zegara procesora) i nie jest potrzebna, bo za chwilę dla

potomka zostanie utworzony nowy segment danych programu `/usr/bin/sh` przy pomocy `execve()`.

Mechanizm `vfork()`, `execve()` został zaprojektowany z myślą o takim właśnie zastosowaniu jakiego potrzebowaliśmy. Dowiedzieliśmy się o nim podglądając gotowe rozwiązania. Jeszcze raz objawiła się wielka zaleta Open Source. Rozwiązanie zaczerpneliśmy z programu Meta-HTML. Stąd pochodzą funkcje `shell_execute()` i `gather_input()`. Dostosowaliśmy je do naszych potrzeb upraszczając je nieznacznie.

3.5 Rozwój modułu

Moduł `mod_pdf` jak na razie nie został użyty praktycznie na szerszą skalę. Wprawdzie został zainstalowany na serwerze `math` Instytutu Matematyki, ale oprogramowanie, które mogłoby z niego zrobić użytek nie zostało jeszcze dostosowane. W dalszym ciągu wydruki PDF tworzone są w całości za pomocą Meta-HTML.

Pisząc program modułu dokonaliśmy wielu uproszczeń, bo z jednej strony jesteśmy ograniczeni czasowo, z drugiej nie zawsze było jasne czy dodanie nowych funkcjonalności przyniesie jakikolwiek zysk, a na pewno skomplikuje kod. Praktyka zapewne wskaże słabości i podpowie ewentualne ulepszenia.

Liczne testy jakie przeprowadziliśmy w trakcie uruchamiania nie wykazały żadnych problemów. Nie ma jednak gwarancji, że program zadziała niezawodnie w każdej sytuacji, bo nie jesteśmy w stanie wszystkiego przewidzieć.

Dodatek A

API serwera Apache

A.1 Struktury danych wykorzystane w module `mod_pdf`

`request_rec`

Struktura `request_rec` jest jedną z największych i najważniejszych struktur danych API serwera Apache. Zawiera wszystkie informacje o żądaniu: gdzie, kiedy, od kogo, dlaczego, w jakim języku, jak je przetworzyć itd. Struktura `request_rec` jest zdefiniowana w pliku `src/include/httpd.h`.

```
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;
};
```

server_rec

Struktura `server_rec` zawiera m.in. nazwę serwera, adres administratora, pliki konfiguracyjne, struktury konfiguracyjne modułów, limity czasu oraz identyfikatory UID/GID serwera. Każdy rekord żądania zawiera wskaźnik do struktury `server_rec`. Dzięki temu można sprawdzić, który serwer wywołał twój moduł. Ta struktura jest zdefiniowana w pliku `src/include/httpd.h`.

```
struct server_rec {
    server_rec *next;
    const char *defn_name;
    unsigned defn_line_number;
    char *srm_confname;
    char *access_confname;
    char *server_admin;
    char *server_hostname;
    unsigned short port;
    char *error_fname;
    FILE *error_log;
    int loglevel;
    int is_virtual;
    void *module_config;
    void *lookup_defaults;
    server_addr_rec *addrs;
    int timeout;
    int keep_alive_timeout;
    int keep_alive_max;
    int keep_alive;
    int send_buffer_size;
    char *path;
    int pathlen;
    array_header *names;
    array_header *wild_names;
    uid_t server_uid;
    gid_t server_gid;
    int limit_req_line;
    int limit_req_fieldsize;
```

```
    int limit_req_fields;
};
```

A.2 Funkcje użyte w module mod_pdf

`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.

```
void ap_log_rerror(const char *file, int line, int level,
        const request_rec *r, const char *fmt, ...)
        __attribute__((format(printf,5,6)))
```
: Funkcja służy do zapisywania różnych informacji w logach serwera Apache. Parametry `file` i `line` określają odpowiednio nazwę pliku i numer wiersza, jeśli zapisywana w logu informacja dotyczy pliku konfiguracyjnego. Parametr `level` określa w jakim stopniu dana informacja jest krytyczna. Możliwe są m.in. następujące poziomy: komunikat debugowania, zwykła informacja, sytuacja godna uwagi, ostrzeżenie, błąd, błąd krytyczny, alert. W konfiguracji serwera Apache można nakazać zapisywanie do loga tylko informacji powyżej pewnego poziomu. Parametr `r` to wskaźnik do struktury opisującej żądanie. Pozostałe parametry są zgodne z systemową funkcją `printf()`.

`file *ap_pfopen(struct pool *p, const char *name, const char *fmode)`
: Otwiera plik o nazwie `name` w trybie `fmode`, podobnie jak systemowa funkcja `fopen()`. Różnica polega na tym, że otwarty plik zostanie zamknięty podczas zwalniania struktury `p`. Zwykle ma to miejsce po zakończeniu przetwarzania żądania. Zwracaną wartością jest deskryptor pliku lub wartość NULL, gdy pojawił się błąd.

`int ap_pfclose(struct pool *p, file *f)`
: Zamyka plik o deskryptorze `f`, otwarty przy pomocy `ap_pfopen()`.

`long ap_send_fd(file *f, request_rec *r)`
: Wysyła plik o deskryptorze `f` do klienta określonego w strukturze `r`. Zwraca liczbę przesłanych bajtów. Przesyłany plik musi być wcześniej otwarty do czytania.

`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.

Opracowano w oparciu o [3, 4, 5].

Dodatek B

Kody stanu HTTP

Odpowiedzi serwera HTTP występujące w module `mod_pdf` [2, 6, 7]:

`HTTP_OK` **(kod 200,** `OK`**)**
: Zapytanie powiodło się, serwer wysyła odpowiedź, która zawiera żądane dane.

`HTTP_FORBIDDEN(kod 403)`
: 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)`
: 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_SERVER_ERROR(kod 505)`
: Wewnętrzny błąd serwera, np.

Dodatek C

Kod źródłowy modułu

C.1 mod_pdf.c

```
/*
   Copyright (c) 2004 Ewa Chaberek & Mariusz Zynel.

   This software is FREE. You can use and/or redistribute it for any
   purpose in either, modified, or unmodified form, under the terms of the
   GNU General Public License as published by the Free Software Foundation.

   The above copyright notice and this permission notice shall be included
   in all copies or substantial portions of this software.

   THIS SOFTWARE IS PROVIDED AS IS AND COME WITH NO WARRANTY OF ANY KIND,
   EITHER EXPRESSED OR IMPLIED. IN NO EVENT WILL THE COPYRIGHT HOLDER BE
   LIABLE FOR ANY DAMAGES RESULTING FROM THE USE OF THIS SOFTWARE.
*/

#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"

#include "utils.h"

#define MODNAME              "mod_pdf"
#define PDF_VERSION_STRING   "0.1"

#define SMALLSTRING 256
#define BIGSTRING 1024

#define PORTRAIT     'p'
#define LANDSCAPE    'l'
#define ONEPAGE      '1'
#define TWOPAGE      '2'
#define BOOKLET      'b'

#define TEX 0
#define DVI 1
#define PS  2
#define PDF 3
```

```
module MODULE_VAR_EXPORT pdf_module;

/* Global variables */

char *pdf_working_directory = "/tmp";
char *pdf_exec_path = "/usr/bin";
char *pdf_latex = "latex";
char *pdf_dvips = "dvips";
char *pdf_psbook = "psbook";
char *pdf_psnup = "psnup";
char *pdf_gs = "gs";
int  pdf_debug_level = 0;


/* Check the method of the request */

static int
check_method(request_rec *r)
{
    if (0 < pdf_debug_level)
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
                "Method: %d", r->method_number);

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

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

    return OK;
}


/* Send given file back to the connected client */

static int
send_file_to_client(request_rec *r, const char *filename)
{
    FILE *fd;

    if (0 < pdf_debug_level)
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
                "Sending file: %s", filename);

    /* Open the requested file */
    fd = ap_pfopen(r->pool, filename, "r");

    /* Return error if anything goes wrong */
    if (fd == NULL) {
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
                MODNAME ": file permissions deny server access: %s",
                                                                filename);
        return FORBIDDEN;
    }

    /* Send headers back to the client */
    ap_send_http_header(r);

    /* If it's not a request of headers only send
       the file to the client */
    if (! r->header_only)
        ap_send_fd(fd, r);
```

```
    /* Close the file */
    ap_pfclose(r->pool, fd);
    return OK;
}


/* Run LaTeX, dvips, psbook, psnup and ps2pdf to generate requested file */

static int
generate_requested_file(request_rec *r, int reqformat, const char *reqext)
{
    char *srcfile = (char *)NULL;
    char *trgfile = (char *)NULL;
    char *tmpdir = (char *)NULL;
    char *command = (char *)NULL;
    char *output = (char *)NULL;
    char *errors = (char *)NULL;
    char orientation = PORTRAIT;
    char layout = ONEPAGE;
    FILE *fd;
    int result = SERVER_ERROR;

    if ((result = check_method(r)) != OK) {
        return result;
    }

    /* If the requested file exists, send it back to the client */
    if (r->finfo.st_mode != 0)
        return send_file_to_client(r, r->filename);

    if (0 < pdf_debug_level)
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
                "Requested file: %s", r->filename);

    /* Change the extension of the requested file to tex */
    if ((srcfile = replace_file_ext(r->filename, reqext, ".tex"))
                                                        == NULL) {
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
                MODNAME ": cannot convert filename: %s from %s to .tex",
                        r->filename, reqext);
        return SERVER_ERROR;
    }

    /* Check if the source file exists */
    if ((fd = ap_pfopen(r->pool, srcfile, "r")) == NULL) {
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
                MODNAME ": file not found: %s", srcfile);
        free(srcfile);
        return NOT_FOUND;
    }
    ap_pfclose(r->pool, fd);

    /* Make a unique pathname for a temporary directory */
    tmpdir = tempnam(pdf_working_directory, "dpdf");

    if (0 < pdf_debug_level)
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
                "Temporary file: %s", tmpdir);

    /* Allocate memory for target file name */
    if ((trgfile = (char *)malloc(strlen(tmpdir) + strlen(reqext) + 2))
                                                        == NULL) {
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
                MODNAME ": insufficient memory to allocate: trgfile");
        free(srcfile);
```

```
    free(tmpdir);
    return SERVER_ERROR;
}

/* Make target file name */
sprintf(trgfile, "%s%s", tmpdir, reqext);

if (0 < pdf_debug_level)
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
            "Target file: %s", trgfile);

/* Allocate memory for the command to be executed */
if ((command = (char *)malloc(BIGSTRING)) == NULL) {
    ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
            MODNAME ": insufficient memory to allocate: command");
    free(srcfile);
    free(trgfile);
    free(tmpdir);
    return SERVER_ERROR;
}

/* Prepare the command to be executed */
/* Make temporary directory and put the source file there */
snprintf(command, BIGSTRING - 1, "mkdir %s; cd %s; cp %s src.tex",
        tmpdir, tmpdir, srcfile);

/* Run latex, three times for safety */
if (DVI <= reqformat) {
    snprintf(command, BIGSTRING - 1,
        "%s; %s --interaction batchmode src", command, pdf_latex);
    snprintf(command, BIGSTRING - 1,
        "%s; %s --interaction batchmode src", command, pdf_latex);
    snprintf(command, BIGSTRING - 1,
        "%s; %s --interaction batchmode src", command, pdf_latex);
}

/* Run dvips, psbook and psnup */
if (PS <= reqformat) {
    if ((result = parse_request(r->the_request, &orientation,
                                                &layout)) != OK)
        return result;

    if (0 < pdf_debug_level)
            ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
                "Request: %s, orientation: %c, layout: %c",
                        r->the_request, orientation, layout);

    if (orientation == LANDSCAPE)
        snprintf(command, BIGSTRING - 1, "%s; %s -q -o src.ps
            -t landscape src", command, pdf_dvips);
    else
        snprintf(command, BIGSTRING - 1, "%s; %s -q -o src.ps src",
            command, pdf_dvips);

    if (layout == BOOKLET)
        snprintf(command, BIGSTRING - 1, "%s; %s -q src.ps src-b.ps;
            mv src-b.ps src.ps", command, pdf_psbook);

    if (layout == TWOPAGE || layout == BOOKLET)
        snprintf(command, BIGSTRING - 1, "%s; %s -q -2 src.ps src-2.ps;
            mv src-2.ps src.ps", command, pdf_psnup);
}

/* Run gs */
if (PDF <= reqformat) {
    snprintf(command, BIGSTRING - 1, "%s; %s -q -dSAFER -dNOPAUSE -dBATCH
```

```
            -dCompatibilityLevel=1.4 -dEmbedAllFonts=true -sDEVICE=pdfwrite
            -sPAPERSIZE=a4 -sOutputFile=src.pdf src.ps", command, pdf_gs);
    }

    /* Cleanup */
    snprintf(command, BIGSTRING - 1, "%s; mv src%s %s; cd %s; rm -rf %s" ,
            command, reqext, trgfile, pdf_working_directory, tmpdir);

    if (0 < pdf_debug_level)
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
                "Command to execute: %s", command);

    /* Execute the command */
    if (! shell_execute(command, pdf_exec_path, &output, &errors, 1)) {
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
            MODNAME ": command %s failed: output %s, errors: %s ", command,
            output ? output : "(NULL)", errors ? errors : "(NULL)");
        free(srcfile);
        free(trgfile);
        free(tmpdir);
        free(command);
        return SERVER_ERROR;
    }

    if (0 < pdf_debug_level) {
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
                "Output: %s", output ? output : "(NULL)");
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_DEBUG, r,
                "Errors: %s", errors ? errors : "(NULL)");
    }

    /* Send generated file to the client */
    result = send_file_to_client(r, trgfile);

    /* Delete the target file */
    remove(trgfile);

    free(srcfile);
    free(trgfile);
    free(tmpdir);
    free(command);

    return result;
}


static void
pdf_init(server_rec *s, pool *p)
{
    ap_add_version_component ("PDF/" PDF_VERSION_STRING);
}

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

    return ((const char *)NULL);
}

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

    return ((const char *)NULL);
```

```
}

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

    return ((const char *)NULL);
}

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

     return ((const char *)NULL);
}

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

    return ((const char *)NULL);
}

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

    return ((const char *)NULL);
}

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

    return ((const char *)NULL);
}

static const char *
set_pdf_debug_level(cmd_parms *cmd, void* empty, char* text)
{
    if (sscanf(text, "%d", &pdf_debug_level) != 1)
        return "PDFDebugLevel value must be an integer 0 - 255";

    return ((const char *)NULL);
}


static int
pdf_handle_req_pdf(request_rec *r)
{
    return generate_requested_file(r, PDF, ".pdf");
}


static int
pdf_handle_req_ps(request_rec *r)
{
    return generate_requested_file(r, PS, ".ps");
}
```

```
static int
pdf_handle_req_dvi(request_rec *r)
{
    return generate_requested_file(r, DVI, ".dvi");
}


static int
pdf_handle_req_tex(request_rec *r)
{
    int result;

    if ((result = check_method(r)) != OK) {
        return result;
    }

    /* Check if the requested file exists */
    if (r->finfo.st_mode == 0) {
        ap_log_rerror(APLOG_MARK, APLOG_NOERRNO|APLOG_ERR, r,
                MODNAME ": file not found: %s", r->filename);
        return NOT_FOUND;
    }

    return send_file_to_client(r, r->filename);
}


command_rec pdf_cmds[] =
  {
    { "PDFWorkingDirectory", set_pdf_working_directory, NULL, RSRC_CONF,
      RAW_ARGS, "Set working directory where PDF document is generated." },
    { "PDFExecPath", set_pdf_exec_path, NULL, RSRC_CONF, RAW_ARGS,
        "Set default list of paths to look for executables." },
    { "PDFlatex", set_pdf_latex, NULL, RSRC_CONF, RAW_ARGS,
        "Set path to latex executable." },
    { "PDFdvips", set_pdf_dvips, NULL, RSRC_CONF, RAW_ARGS,
        "Set path to dvips executable." },
    { "PDFpsbook", set_pdf_psbook, NULL, RSRC_CONF, RAW_ARGS,
        "Set path to psbook executable." },
    { "PDFpsnup", set_pdf_psnup, NULL, RSRC_CONF, RAW_ARGS,
        "Set path to psnup executable." },
    { "PDFgs", set_pdf_gs, NULL, RSRC_CONF, RAW_ARGS,
        "Set path to gs executable." },
    { "PDFDebugLevel", set_pdf_debug_level, NULL, RSRC_CONF, TAKE1,
        "Set debugging level." },
    { NULL }
  };

static const handler_rec pdf_handlers[] =
  {
    { "application/pdf", pdf_handle_req_pdf },
    { "application/postscript", pdf_handle_req_ps },
    { "application/x-dvi", pdf_handle_req_dvi },
    { "application/x-tex", pdf_handle_req_tex },
    { NULL }
  };

module MODULE_VAR_EXPORT pdf_module =
  {
    STANDARD_MODULE_STUFF,
    pdf_init,                       /* module initializer */
    NULL,                           /* per-directory config creator */
    NULL,                           /* dir config merger */
    NULL,                           /* server config creator */
    NULL,                           /* server config merger */
    pdf_cmds,                       /* command table */
```

```
    pdf_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
};
```

C.2 utils.c

```
/*
   Copyright (c) 2004 Ewa Chaberek & Mariusz Zynel, except portions written
   by Brian J. Fox (bfox@ai.mit.edu) which are copyright (c) 1998, 2003.

   This software is FREE. You can use and/or redistribute it for any
   purpose in either, modified, or unmodified form, under the terms of the
   GNU General Public License as published by the Free Software Foundation.

   The above copyright notice and this permission notice shall be included
   in all copies or substantial portions of this software.

   THIS SOFTWARE IS PROVIDED AS IS AND COME WITH NO WARRANTY OF ANY KIND,
   EITHER EXPRESSED OR IMPLIED. IN NO EVENT WILL THE COPYRIGHT HOLDER BE
   LIABLE FOR ANY DAMAGES RESULTING FROM THE USE OF THIS SOFTWARE.
*/

#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/select.h>

#include "utils.h"

#define TIMEOUT     60
#define BIGSTRING   1024


/*
  Replace extension in a given filename. Retruns a pointer to a newly
  created string that should be freed with free().
*/

char *
replace_file_ext(char *filename, const char *ext, const char *new)
{
    char *target = NULL;
    char *extpos = NULL;

    /* Allocate memory for the target filename */
    if ((target = malloc(strlen(filename) + 2)) == NULL) {
        return NULL;
    }

    /* Copy the contents of the filename */
    strcpy(target, filename);

    /* Find the last occurence of a dot in the filename */
    if ((extpos = strrchr(target, '.')) == NULL) {
        free(target);
        return NULL;
    }

    /* If it is a desired extension replace it for a new one */
    if (! strcasecmp(extpos, ext)) {
        strcpy(extpos, new);
```

```
    } else {
        free(target);
        return NULL;
    }

    return target;
}


/*
  Parse requests that come from clients. Extract orientation and layout
  parameters from the request, check if their values are valid.
*/

int
parse_request(char *request, char *orientation, char *layout)
{
    int i = 0;
    int start = 0;

    if (request == NULL) {
        return ERR;
    }

    /* Skip method */
    while (request[i] && (request[i] != ' ')) i++;

    /* Skip whitespaces and find the start of location */
    while (request[i] && (request[i] == ' ')) i++;

    start = i;

    /* Find the length of location substring */
    while (request[i] && (request[i] != '?') && (request[i] != ' ')) i++;

    if (request[i] == '?') {
        /* Find and parse query components */
        i++;
        while (request[i] && request[i] != ' ') {
            start = i;
            while (request[i] && request[i] != '&' && request[i] != ' ') i++;

            /* Parse values only if this part of query has proper
               length of 3 bytes. Ignore the others */
            if (i - start == 3) {
                if (request[start] == 'o' || request[start] == 'O') {
                    if (request[start + 1] == '=')
                        *orientation = request[start + 2];
                } else if (request[start] == 'l' || request[start] == 'L') {
                    if (request[start + 1] == '=')
                        *layout = request[start + 2];
                }
            }
            i++;
        }
    }

    /* Check if the values of orientation is valid */
    if (*orientation != 'l' && *orientation != 'p') {
        return ERR;
    }

    /* Check if the values of layout is valid */
    if (*layout != '1' && *layout != '2' && *layout != 'b') {
        return ERR;
    }
```

```
    return OK;
}


/*
  A set of functions to execute a shell script. Borrowed from Meta-HTML.
*/

typedef void(* sig_t)(int);

static int child_status = 0;

/* What to do if a child process dies. */

static void
release_child(void)
{
    wait(&child_status);
}


/*
  Collect and return the output from the specified pipe.
*/

static char *
gather_input(int fd)
{
    char *result = (char *)NULL;
    int select_result = 1;

#if defined (FD_SET)

    struct timeval timeout;
    fd_set read_fds;
    int intr = 0;

    timeout.tv_sec = TIMEOUT;
    timeout.tv_usec = 0;

    FD_ZERO(&read_fds);
    FD_SET(fd, &read_fds);

    while (intr < 2) {
        select_result =
                select(fd + 1, (fd_set *)(&read_fds), 0, 0, &timeout);
        if ((select_result == -1) && (errno == EINTR))
          intr++;
        else
          break;
    }

#else /* !FD_SET */
    select_result = 1;
#endif /* !FD_SET */

    switch (select_result) {
        case 0:
        case -1:
            break;

        default:
            {
                char *buffer = (char *)NULL;
                int bindex = 0, bsize = 0;
```

```
                int amount_read;
                int done = 0;

                while (!done) {
                    while ((bindex + 1024) > bsize)
                        buffer = (char *)realloc (buffer, (bsize += 1025));
                    buffer[bindex] = '\0';

                    amount_read = read(fd, buffer + bindex, 1023);

                    if ((amount_read < 0) && errno != EINTR) {
                        done = 1;
                    } else {
                        if (amount_read != -1) {
                            bindex += amount_read;
                            buffer[bindex] = '\0';
                        }

                        if (amount_read == 0)
                            done = 1;
                    }
                }

                result = buffer;
            }
    }

    return (result);
}

/*
  Execute a shell script using sh interpreter. Returns true if STDERR is
  empty or we don't wait for the script to finish. Parameters:
    command - the script to execute,
    path - specifies the PATH that should be used by the script,
    output - the contents of STDOUT of the script,
    errors - the contents of STDERR of the script,
    wait - wait for the script to finish, if non-zero gather
           STDOUT and STDERR.

*/

int
shell_execute(char *command, char *path, char **output, char **errors,
                                                                int wait)
{
    char *output_text = (char *)NULL;
    char *errors_text = (char *)NULL;
    char *physical_path = "/bin/sh";
    pid_t child;
    int stdout_pipe[2];
    int stderr_pipe[2];

    if (command == (char *)NULL) {
        errors_text = strdup("ERROR: shell_execute: No command given");
    } else {

        /* Make receiving pipes for stdout and stderr if the
           user wishes to receive the output. */
        if (wait) {
            pipe(stdout_pipe);
            pipe(stderr_pipe);
        }

        child = vfork();
```

```
        /* In the parent... */
        if (child != (pid_t)0) {
            /* Setup pipes for communication. */
            if (wait) {
                close(stdout_pipe[1]);
                close(stderr_pipe[1]);
             }

            /* Say what to do when a child dies. */
            signal(SIGCHLD, (sig_t)release_child);

            /* Say what to do if the pipe is broken. */
            signal(SIGPIPE, SIG_IGN);

            if (wait) {
                output_text = gather_input(stdout_pipe[0]);
                errors_text = gather_input(stderr_pipe[0]);

                close(stdout_pipe[0]);
                close(stderr_pipe[0]);
             }
        } else {  /* In the child... */
            char *argv[4];
            char *envp[2];

            argv[0] = physical_path;
            argv[1] = "-c";
            argv[2] = command;
            argv[3] = (char *)NULL;

            if (! (envp[0] = (char *)malloc(BIGSTRING))) {
                _exit(127);
            }

            snprintf(envp[0], BIGSTRING - 1, "PATH=%s", path);
            envp[1] = (char *)NULL;

            /* In the child, make STDOUT and STDERR be our pipes. */
            if (wait) {
                close(stdout_pipe[0]);
                close(stderr_pipe[0]);
                dup2(stdout_pipe[1], 1);
                dup2(stderr_pipe[1], 2);
                close(stdout_pipe[1]);
                close(stderr_pipe[1]);
            } else {
                setpgrp();
                close(0);
                close(1);
            }

            execve(physical_path, argv, envp);

            _exit(127);
        }
    }

    *output = output_text;
    *errors = errors_text;

    return (errors_text == (char *)NULL || strcmp(errors_text, "") == 0);
}
```

C.3 utils.h

```
/*
   Copyright (c) 2004 Ewa Chaberek & Mariusz Zynel.

   This software is FREE. You can use and/or redistribute it for any
   purpose in either, modified, or unmodified form, under the terms of the
   GNU General Public License as published by the Free Software Foundation.

   The above copyright notice and this permission notice shall be included
   in all copies or substantial portions of this software.

   THIS SOFTWARE IS PROVIDED AS IS AND COME WITH NO WARRANTY OF ANY KIND,
   EITHER EXPRESSED OR IMPLIED. IN NO EVENT WILL THE COPYRIGHT HOLDER BE
   LIABLE FOR ANY DAMAGES RESULTING FROM THE USE OF THIS SOFTWARE.
*/

#define OK      0
#define ERR     -1


extern char *replace_file_ext(char *filename, const char *ext,
                                                    const char *new);

extern int parse_request(char *request, char *orientation, char *layout);

extern int shell_execute(char *command, char *path, char **output,
                                            char **errors, int wait);
```

C.4 Makefile

```
# Simplistic Makefile for mod_pdf.(Apache versions 1.3x)
#
APXS                = apxs
APXS_STDFLAGS       = -a -c
APXS_OPTIMIZATION   = -Wc,-xO4 -Wc,-xtarget=pentium4 -Wc,-fast
APXS_COMPILE_FLAGS  = $(APXS_STDFLAGS) $(APXS_OPTIMIZATION)
APXS_INSTALL_FLAGS  = -i

all: mod_pdf.so

utils.o: utils.c utils.h Makefile
        $(CC) -c -o utils.o utils.c

mod_pdf.so: mod_pdf.c utils.o Makefile
        $(APXS) $(APXS_COMPILE_FLAGS) mod_pdf.c utils.o
        strip *.so

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

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

FORCE:
```

Bibliografia

[1] Lamport L., *LaTeX– a document preparation system*, Addison Wesley.

[2] Spainbour S., Quercia V., *Webmaster–podręcznik administratora*, Wydawnictwo RM Sp. z o.o., Warszawa 1997.

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

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

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

[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] Solaris 9 12/03 Reference Manual Collection,
`http://docs.sun.com/db/coll/40.13`

[9] Spis modułów serwera Apache,
`ftp://ftp.apache.org/dist/contrib/modules`

[10] Statystyki serwerów HTTP,
`http://news.netcraft.com/archives/web_server_survey.html`

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

[12] Typy dokumentów używanych w Internecie,
`http://www.iana.org/assignements/media-types/`