Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Piotr Kotarbiński i Grzegorz Nowakowski
Nr albumu: 181154 i 181093

Implementacja serwera DNS w języku O'Caml

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki UW

Wrzesień 2004

Pracę przedkładam do oceny

Data Podpis autora pracy:

Praca jest gotowa do oceny przez recenzenta

Data Podpis kierującego pracą:

Streszczenie

W ramach niniejszej pracy zaprojektowano i zaimplementowano serwer nazw domen w języku funkcyjnym O'Caml. W części pisemnej przedstawiono usługę, protokół oraz architekturę implementacji wraz z dyskusją napotkanych problemów projektowych. Na część praktyczną składa się realizacja przedstawionego projektu oraz wykonanie testów wydajnościowych stworzonego serwera. Przedstawione zostało również porównanie z najbardziej popularną implementacją DNS jaką jest BIND.

Słowa kluczowe

serwer nazw domen, DNS, programowanie funkcyjne, O'Caml, system rozproszony, rozproszona baza danych

Klasyfikacja tematyczna

C. Computer System Organization
C.5. Computer System Implementation
C.5.5. Servers

Spis treści

Rozdział 1

Wprowadzenie

1.1. Wprowadzenie do systemu nazw domen

Adresy IP są podstawą protokołu TCP/IP. Każdy, kto korzystał z Internetu wie, że użytkownicy nie muszą pamiętać ani wpisywać tych adresów (w postaci liczbowej). Komputery, prócz adresu liczbowego (IP) mają przypisane dodatkowo nazwy symboliczne (np. `www.google.com`). Programy użytkowe pozwalają na wprowadzanie którejś z nazw symbolicznych przy identyfikowaniu określonego komputera. Przykładem może być przeglądanie stron WWW (ang. *World Wide Web*). Użytkownik wpisuje w przeglądarce nazwę serwera, który chce odwiedzić, np. `www.gegez.com`, zamiast adresu liczbowego.

Nazwy symboliczne, chociaż wygodne dla ludzi, sprawiają problemy komputerom. IP w postaci numerycznej jest bardziej zwięzły niż nazwa symboliczna, wymaga mniej obliczeń (np. porównywanie) i jest stałej długości (32 bity dla IP w wersji 4). Co więcej, adres w postaci liczbowej zajmuje mniej pamięci i wymaga mniej czasu przy transmisji przez sieć niż nazwa. Stąd, chociaż oprogramowanie użytkowe pozwala na wprowadzanie nazw symbolicznych, bazowe protokoły sieciowe wymagają używania adresów liczbowych. Program, zanim będzie mógł skorzystać przy komunikacji z nazwy, musi przetłumaczyć ją na równoważny adres IP. Najczęściej tłumaczenie to jest wykonywane automatycznie, a jego wynik nie jest użytkownikowi przedstawiany — adres IP jest zachowywany w pamięci i wykorzystywany tylko do wysyłania i odbierania datagramów (pakietów).

Oprogramowanie, które tłumaczy nazwy komputerów na równoważne adresy internetowe, stanowi interesujący przykład interakcji typu klient-serwer. Baza danych z nazwami nie jest utrzymywana na pojedynczym komputerze. Zamiast tego informacja o nazwach jest rozproszona w ramach potencjalnie dużego zbioru serwerów umieszczonego w całym Internecie. Gdy program ma przetłumaczyć nazwę, staje się klientem systemu nazw. Klient taki wysyła do serwera nazw komunikat z pytaniem, serwer zaś odnajduje odpowiedni adres i wysyła komunikat z odpowiedzią. Jeżeli nie można znaleźć odpowiedzi, to serwer nazw może stać się tymczasowo klientem innego serwera i tak do momentu, aż zostaje znaleziona odpowiedź na zadane pytanie.

1.1.1. Schemat nazw w Internecie

Powszechnie stosowany w Internecie schemat nazw określany jest mianem systemu nazw dziedzin (ang. *Domain Name System*, w skrócie DNS). Składnia nazw jest bardzo prosta: jest to ciąg znaków alfanumerycznych oddzielonych kropkami. Przykładowo na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego znajdują się komputery:

- `zodiac.mimuw.edu.pl`,
- `rainbow.mimuw.edu.pl`,
- `duch.mimuw.edu.pl`.

Mają one wspólną część dziedziny `mimuw.edu.pl`, która odpowiada temu wydziałowi. Nazwy dziedzin są zhierarchizowane, przy czym najbardziej znacząca część nazwy znajduje się po prawej stronie, natomiast lewy skrajny segment nazwy (`zodiac`, `rainbow` czy `duch`) opisuje konkretny węzeł. Pozostałe segmenty w nazwie identyfikują grupę o danej nazwie, np. `mimuw` identyfikuje Wydział MIM UW.

Organizacje zarządzające systemem nazw określają tylko sposób obierania segmentów najbardziej znaczących (w tym wypadku `.pl`, `edu.pl`) (nazywany głównym poziomem DNS lub korzeniem), natomiast nie określa dokładnej liczby segmentów w każdej nazwie ani też nie precyzuje ich znaczenia. Każda organizacja może sama zdecydować, ile segmentów będzie u siebie używała i co one reprezentują. Przykładowe dziedziny głównego poziomu to `com` (organizacje komercyjne), `edu` (instytucje edukacyjne), `gov` (instytucje rządowe), `pl` (kod kraju, w tym wypadku Polski).

Aby otrzymać dziedzinę, organizacja musi wystąpić z wnioskiem rejestracyjnym do odpowiednich władz Internetu (w Polsce jest to NASK). Każdej organizacji przyznawany jest jednoznaczny sufiks nazwy dziedziny (np. `mimuw.edu.pl` dla MIM UW).

1.2. Cel projektu

Celem naszej pracy jest stworzenie przenośnej implementacji serwera DNS o nazwie UnNamed wg specyfikacji RFC [rfc1034, rfc1035] w języku funkcyjnym O'Caml. DNS ma z założenia strukturę modularną — niektóre jego funkcjonalności są bezwzględnie wymagane, inne są opcjonalne. Naszym celem nie jest stworzenie pełnej implementacji a jedynie części podstawowej wraz z rozszerzeniami zwiększającymi wydajność systemu (inkrementalny transfer strefy [rfc1995], DNS Notify [rfc1996], dynamiczny DNS [rfc2136]). System został tak zaprojektowany, aby dodanie kolejnych rozszerzeń protokołu było jak najprostsze. Z założenia nasz serwer będzie pracować używając tylko protokołu IP w wersji 4. Kod źródłowy został tak napisany, aby można go było skompilować bez modyfikacji na następujących platformach: UNIX (POSIX), Linux, Microsoft Windows.

Dodatkowym celem pracy jest sprawdzenie przydatności języka O'Caml do pisania aplikacji sieciowych. W testach zamierzamy między innymi zmierzyć wpływ liczby klientów (jednoczesnych żądań) na czas odpowiedzi.

1.3. Motywacje

W sytuacji obecnej istnieje wiele zarówno komercyjnych jak i takich z ogólnie dostępnym kodem źródłowym implementacji DNS. Najbardziej znaną i rozpowszechnioną jest BIND (ang. *Berkeley Internet Name Domain*). Są one napisane w językach dość niskopoziomowych (głównie C). Efekt jest taki, że są one często dziurawe i niestabilne, czego najlepszym przykładem jest liczba złośliwych programów powodujących błędną pracę serwera (ang. *exploits*). W Internecie można znaleźć setki takich programów dla każdej znaczącej implementacji. Zastosowanie języków niskiego poziomu powoduje również, że kod tych implementacji jest mało przejrzysty i trudny do ogarnięcia, przez co często wymyka się spod kontroli i ciężko jest go uporządkować.

Chodziły również słuchy, że kod najpopularniejszej implementacji DNS jaką jest BIND, może przestać być publicznie dostępny. Miałoby to zapobiec tworzeniu programów wykorzystujących dziury w kodzie.

Paradygmaty programowania funkcyjnego oraz natura języka O'Caml wymuszają dużą modularność pisanych programów oraz rozłączność modułów, dzięki czemu kod jest bardziej przejrzysty i lepiej uporządkowany. Wysoka modularnośc ułatwia też rozwijanie kodu (dodawanie nowych modułów), a w połączeniu z deklaratywną naturą języków funkcyjnych, weryfikowalność i pielęgnację istniejącego kodu. Dodatkowo kod w językach deklaratywnych jest zazwyczaj dużo krótszy niż analogiczny kod napisany w językach imperatywnych.

Inną ważną własnością języków funkcyjnych jest bardzo silny system typów. Oznacza to, że typy większości wyrażeń są wyliczane podczas kompilacji. Pozwala to wykryć większość błędów właśnie na etapie kompilacji, a nie w czasie użytkowania czy odpluskwiania.

Języki wysokiego poziomu są zazwyczaj jednak mało wydajne, przez co nie zawsze nadają się do zastosowania przy pisaniu serwerów, które powinny reagować na zdarzenia w możliwie krótkim czasie, przy stosunkowo dużym obciążeniu. O'Caml natomiast wyróżnia się w tej klasie języków dużą wydajnością, porównywalną z C++. W przeciwieństwie do większości innych języków nie imperatywnych posiada on od dawna kompilator (a nie tylko interpreter), zarówno do kodu natywnego, jak i specyficznego dla siebie kodu bajtowego. Obecnie istnieją maszyny wirtualne oraz kompilatory na najbardziej popularne platformy, jakimi są Microsoft Windows, UNIX (między innymi Linux) oraz MacOS. Zarówno kod źródłowy, jak i kod bajtowy może być w znaczącej większości przypadków przenoszony pomiędzy tymi platformami bez żadnych modyfikacji. Kod napisany w języku O'Caml wykonuje się średnio trzykrotnie wolniej niż analogiczny w C, a trzykrotnie szybciej niż w napisany w języku Java — innym popularnym języku kompilowanym do kodu bajtowego.

Bardzo ważną rzeczą z punktu widzenia stabilności i czystości kodu jest prawidłowe zarządzanie pamięcią. W językach C/C++ obowiązek alokacji oraz zwalniania potrzebnej pamięci spada całkowicie na programistę. Nieprawidłowe przydzielanie/zwalnianie pamięci jest bardzo trudne do wykrycia i jest najczęściej wykorzystywanym błędem w programach mającym na celu spowodowanie błędnej pracy serwerów. W O'Camlu przydzielaniem pamięci zajmuje się niejawnie kompilator i ewentualnie maszyna wirtualna (w przypadku kodu bajtowego). Dzięki temu prawie niemożliwe jest przepełnienie bufora (ang. *buffer-overflow*), czyli pisanie poza przydzielony obszar. Innym często spotykanym błędem jest nie zwalnianie pamięci (ang. *memory leak*), gdy już nie jest potrzebna. W takim przypadku zużycie pamięci zaczyna rosnąć, co może doprowadzić do niestabilności rozwiązania. O'Caml, w odróżnieniu od wielu innych popularnych języków posiada odśmiecacz (ang. *garbage collector*). Jest to dodatkowy moduł zajmujący się śledzeniem wykorzystania pamięci i zwalniania jej w przypadku wykrycia, że pewien obszar nie jest już wykorzystywany (brak do niego referencji). Odśmiecacz w O'Camlu jest hybrydą odśmiecaczy typu pokoleniowego i inkrementalnego. W przeciwieństwie do odśmiecacza w języku Java nie alokuje on dużych obszarów pamięci podczas startu ani nie ma zaszytych limitów, które trzeba ręcznie konfigurować. W wielu przypadkach dobry odśmiecacz może być szybszy niż ręczne zarządzanie pamięcią – odśmiecacz wykonuje się rzadziej niż ręczna dealokacja pamięci, co powoduje mniejszy narzut na analizę złożonych struktur danych alokatora oraz możliwość optymalizacji struktury kopca.

Więcej informacji na temat języka O'Caml można znaleźć w [OCaml].

1.4. Praca zespołowa

Praca została wykonana w dwuosobowym zespole. Obaj współwykonawcy po równo uczestniczyli w czterech częściach projektu:

- zaprojektowanie systemu,
- implementacja systemu,
- testy,
- opisanie całości w niniejszym dokumencie.

W procesie projektowania trudno wyodrębnić udział w poszczególnych pracach. Schemat modułów, ich zależności, model wątków i serwera sieciowego oraz założenia funkcjonalne systemu są efektem wspólnej dyskusji. Autorem interfejsów poszczególnych modułów jest w większości przypadków Piotr Kotarbiński.

Poszczególnym modułom można przypisać już jednoznacznie autora, choć późniejsze poprawki były wprowadzane wspólnie i niezależnie od pierwotnego twórcy.

Moduł	Autor
Start	Piotr Kotarbiński
ZoneParser	Piotr Kotarbiński
Timer	Piotr Kotarbiński
Network	Piotr Kotarbiński
ModuleDispatcher	Piotr Kotarbiński
StdQuery	Piotr Kotarbiński
Axfr	Piotr Kotarbiński
Ixfr	Piotr Kotarbiński
Xfrer	Piotr Kotarbiński
Notify	Piotr Kotarbiński
Update	Grzegorz Nowakowski
Rrdb	Grzegorz Nowakowski
UpdateHistory	Grzegorz Nowakowski
PrioQueue	Grzegorz Nowakowski
Msg	Grzegorz Nowakowski

Testy wydajnościowe zostały przeprowadzone przez Piotra Kotarbińskiego, a ich wyniki zebrane i zilustrowane wykresami przez Grzegorza Nowakowskiego.

Część pisemną pracy również można wyraźnie podzielić między autorów poszczególnych rozdziałów. Piotr Kotarbiński jest autorem rozdziałów: 2, 4.1, 4.2 oraz 4.3.1. Grzegorz Nowakowski jest autorem rozdziałów: 1, 3, 4 (poza 4.1 i 4.2.1) i 5. Całość została złożona w LaTeXu przez Grzegorza Nowakowskiego.

Rozdział 2

Ogólne założenia systemu DNS

W niniejszym rozdziale opiszemy pobieżnie specyfikację systemu DNS w stopniu niezbędnym do zrozumienia dalszej części pracy. Będziemy w miarę możliwości unikać szczegółów technicznych w celu osiągnięcia jak największej przejrzystości. Zainteresowanego czytelnika odsyłamy do dokumentów [rfc1034, rfc1035, rfc2181].

2.1. Struktura przestrzeni nazw

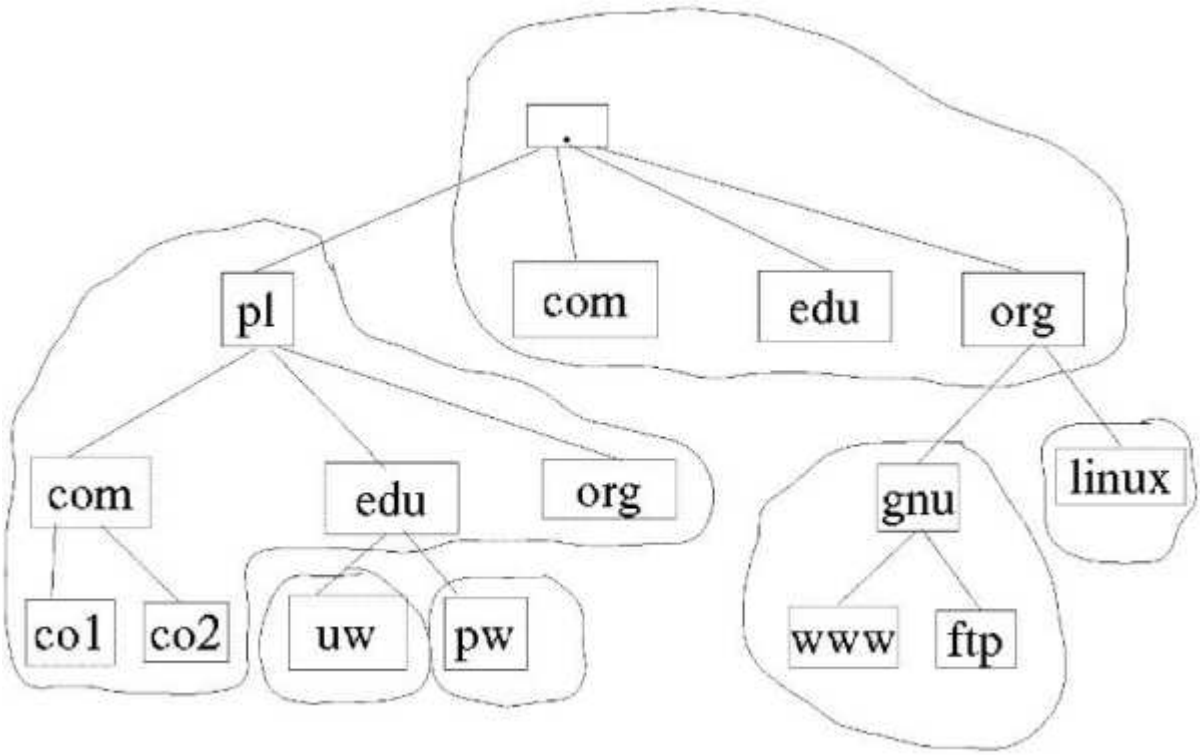
Przestrzeń nazw domen ma strukturę drzewa, w którym każdy węzeł ma etykietę (ang. *label*), identyfikującą go jednoznacznie wśród dzieci węzła macierzystego. Nazwa węzła składa się natomiast z ciągu oddzielonych kropkami etykiet węzłów znajdujących się na ścieżce od danego węzła do korzenia.

Przestrzeń nazw jest podzielona na strefy, które są wewnętrznie spójnymi wycinkami drzewa nazw (takimi, w których węzły znajdujące się na najkrótszej ścieżce, między dwoma dowolnymi węzłami należącymi do tej strefy, również należą do tej strefy). Przykładowy podział drzewa na strefy ukazany jest na diagramie 2.1. Nazwa strefy jest tożsama z nazwą węzła w jej korzeniu.

2.2. Struktura systemu DNS

System DNS można traktować jak rozproszoną bazę danych. Poszczególne serwery nie przechowują informacji o całej przestrzeni nazw, a jedynie o grupie stref. Każda strefa obsługiwana jest przez przynajmniej dwa serwery, z których jeden jest główny (ang. *master*), a pozostałe pomocnicze (ang. *slave*). Takie rozwiązanie ma na celu utrzymanie integralności całego systemu w przypadku awarii (w momencie niedostępności serwera głównego zapytanie można wysłać do jednego z pomocniczych). Wszystkie dane jakie serwer posiada o obsługiwanych przez siebie strefach nazywane są autorytatywnymi, a pozostałe dane dotyczące przestrzeni nazw nazywane są nieautorytatywnymi.

Każda strefa ma dowiązania do stref znajdujących się bezpośrednio pod nią. Umożliwia to odesłanie klienta pytającego się o węzeł znajdujący się w tym samym poddrzewie, ale poniżej granicy obsługiwanej strefy do odpowiedniego serwera. Każdy serwer zna również adresy serwerów z korzenia drzewa nazw — jeśli serwer otrzyma zapytanie o węzeł znajdujący się poza poddrzewami zaczynającymi się od obsługiwanych stref i nie posiada odpowiedzi w pamięci podręcznej, to przekierowuje klienta właśnie do serwerów z korzenia.



Rysunek 2.1: Diagram przykładowego podziału drzewa na strefy

2.3. Reprezentacja danych o przestrzeni nazw

Dane związane z poszczególnymi węzłami przestrzeni przechowywane są w rekordach zasobów (ang. *resource record*, w skrócie RR). Każdy taki rekord składa się z:

- nazwy właściciela (czyli węzła, którego dotyczy),
- typu,
- klasy (obecnie istnieje tylko jedna klasa IN symbolizująca Internet),
- czasu żywotności w pamięci podręcznej (ang. *time-to-live*, w skrócie TTL),
- samych danych (Rdata).

Typ rekordu zasobu mówi, jakiego rodzaju dane przechowywane są w sekcji Rdata. Do najważniejszych typów należą:

- A – adres IP właściciela rekordu,
- CNAME – kanoniczna nazwa maszyny. Do danej maszyny może być przypisane wiele nazw, z czego tylko jedna jest kanoniczna. W prawidłowo skonfigurowanej strefie rekordy typu A powinny istnieć tylko dla nazwy kanonicznej, dla pozostałych nazw (ang. *aliasów*) powinny istnieć rekordy typu CNAME wskazujące na nazwę kanoniczną. Załóżmy, że mamy maszynę o dwóch adresach IP (10.0.0.1 oraz 10.0.0.2) oraz dwóch nazwach (*Ptasie* oraz *Mleczko*). W takiej sytuacji jedna z nich powinna być kanoniczna (np. *Ptasie*), a druga aliasem. Poprawny wpis w przestrzeni nazw dla tej sytuacji przedstawiony jest w tabelce poniżej:

Właściciel	Typ	Dane
Ptasie	A	10.0.0.1
Ptasie	A	10.0.0.2
Mleczko	CNAME	Ptasie

- NS – Name Server. Ten typ rekordu występuje w wierzchołku strefy oraz w liściach jako dowiązania do podstref. Rekord ten zawiera nazwę autorytatywnego serwera nazw dla danej strefy.
- SOA – Start Of Authority. Znajduje się wyłącznie w korzeniu strefy, wskazuje, który serwer nazw jest główny dla danej strefy i opisuje jej różne parametry konfiguracyjne.

Rekordy RR przechowywane są w tak zwanych plikach strefowych, których struktura wyspecyfikowana jest w [rfc1035]. Pliki te są jedynym trwałym nośnikiem danych o strefie. Są one ładowane do pamięci w momencie startu serwera. Każda zmiana w strefie powinna zostać odwzorowana w pliku strefowym.

2.4. Protokół

W systemie DNS zapytania do serwerów oraz odpowiedzi od nich są przekazywane w dokładnie tym samym formacie. Do transportu wiadomości stosowany jest TCP oraz UDP. Ze względu na duży narzut wynikający z ustanawiania połączeń w TCP stosuje się go praktycznie tylko wtedy, gdy wiadomość przekracza ustaloną maksymalną wielkość pakietu dla UDP (512 bajtów). Klient powinien zadać pytanie protokołem UDP (przy założeniu, że mieści się ono w limicie 512 bajtów). Jeżeli odpowiedź również mieści się w tym limicie, to jest ona przesyłana protokołem UDP, w przeciwnym przypadku serwer wysyła obciętą odpowiedź wraz z informacją o tym, że jest ona niekompletna. Klient w takim przypadku powinien ponowić zapytanie używając TCP.

Zapytanie składa się z trzech części:

- nazwy węzła, o który pytamy,
- klasy strefy,
- typu interesującego nas rekordu, np. A czy MX (poczta elektroniczna).

Do synchronizacji danych między obsługującymi tę samą strefę serwerami używa się mechanizmu zwanego transferem stref. Serwery pomocnicze odpytują periodycznie serwer główny, czy strefa nie uległa zmianie. Jeżeli tak, to wysyłają żądanie transferu strefy za pomocą specjalnego typu zapytań (AXFR). W odpowiedzi otrzymują wszystkie rekordy z danej strefy. Możliwe jest również odpytywanie w celu pobrania strefy innych serwerów pomocniczych. Stosuje się to w celu odciążenia serwera głównego.

2.5. Rozszerzenia protokołu

2.5.1. Inkrementalny transfer strefy

Wielkość strefy dużej organizacji może sięgać kilkunastu tysięcy węzłów. Każdy węzeł ma przeciętnie kilka rekordów. Rekordy mogą zmieniać się nawet parę razy dziennie. W celu ograniczenia ruchu sieciowego i zbędnego obciążenia serwerów (generowanie odpowiedzi), zostało stworzone rozszerzenie zwane inkrementalnym transferem strefy. Polega ono na tym,

że zamiast przesyłać zawartość całej strefy przy każdej, najmniejszej zmianie, przesyłane są tylko różnice między danymi wersjami strefy.

Serwer, aby móc odpowiadać na zapytanie inkrementalnego transferu strefy (typ zapytania IXFR), musi przechowywać zmiany, które nastąpiły w kolejnych wersjach strefy. Różne serwery mogą z różną częstotliwością odświeżać strefę, a zatem mogą żądać zmian pomiędzy innymi wersjami, nie zawsze pomiędzy najnowszą a poprzednią.

Rozszerzenie to jest opcjonalne. Jeżeli serwer nie wspiera go lub nie jest przygotowany do jego używania, to używa się zwykłego transferu strefy. Jego specyfikacja znajduje się w [rfc1995].

2.5.2. DNS Notify

Każdy serwer pomocniczy powinien co pewien czas, określony w konfiguracji strefy, odpytywać serwer główny czy dana strefa zmieniła się od poprzedniego transferu. Odbywa się to przez ściągnięcie i porównanie numerów seryjnych strefy (SOA). Może to powodować niepotrzebny ruch oraz duże opóźnienia w propagacji nowych danych, które mogą być bardzo istotne (dane mogą zmieniać się nieregularnie, więc skrócenie czasu odświeżania strefy może spowodować zwiększony ruch). W celu rozwiązania tych problemów zostało stworzone nowe rozszerzenie protokołu, jakim jest DNS Notify. Polega ono na tym, że serwer w momencie zmiany strefy (a co za tym idzie jej numeru seryjnego SOA) wysyła do serwerów pomocniczych informacje o zmianie strefy. W wyniku tego serwery powinny odświeżyć sobie daną strefę za pomocą żądania transferu strefy. Rozszerzenie to jest zdefiniowane w [rfc1996].

2.5.3. Dynamiczny DNS

Dane niektórych węzłów zmieniają się bardzo często, np. adres IP maszyny w połączeniach komutowanych. Aby móc poinformować serwer o takiej zmianie powstało rozszerzenie "Dynamiczny DNS". Klient za pomocą odpowiedniego komunikatu (komunikat DNS o typie UPDATE) informuje serwer o zmianach w rekordach (RR). Rozszerzenie to może być również użyte przez administratora systemu do zdalnego uaktualnienia strefy bez konieczności restartu serwera i modyfikacji pliku strefowego. Komunikat taki składa się z czterech zestawów rekordów:

- tych, które muszą istnieć w aktualnej wersji strefy, by można było dokonać modyfikacji,
- tych, które nie mogą istnieć,
- tych, które należy dodać do strefy,
- tych, które należy usunąć.

Dynamiczny DNS jest opisany dokładnie w [rfc2136].

Rozdział 3

Istniejące implementacje serwera nazw

3.1. Berkeley Internet Name Domain

Najpopularniejszą i jedną z najstarszych implementacji serwera nazw jest BIND (ang. *Berkeley Internet Name Domain*). Jest napisany w całości w języku C. Pierwsza wersja, będąca pracą dyplomową na Uniwersytecie Kalifornijskim w Berkeley, ukazała się jeszcze w latach osiemdziesiątych. Kolejna, oznaczona jako wersja 8, wyszła w drugiej połowie lat dziewięćdziesiątych. Po serii błędów, jakie zostały wykryte, autorzy implementacji BIND napisali praktycznie całe oprogramowanie od nowa (jedynie biblioteka kliencka DNS pozostała stara). W przeciągu kilkunastu lat odkąd pierwsza wersja ujrzała światło dzienne, została zgłoszona niezliczona liczba błędów przepełnienia bufora oraz podatności na atak typu DoS. Nawet w najnowszych wersjach wciąż pojawiają się informacje o nowo wykrytych błędach. Z racji tego, że przez tak długi czas autorom nie udało się uporać z błędami, wiele osób odradza korzystania z tego serwera.

BIND w wersji 9, będąc najbardziej rozbudowaną implementacją DNS w historii (implementuje większość używanych rozszerzeń protokołu wyspecyfikowanych w RFC, typu DNSSEC, DNS Notify, inkrementalny transfer strefy, DNS w IPv6, nazwy domen w unikodzie, profilowanie odpowiedzi w zależności od klienta) niestety jest kiepsko udokumentowany (od strony technicznej). Poza komentarzami w samym kodzie, nie udało nam się dotrzeć do dokumentacji technicznej.

Do wersji 9.2 BIND był rozwijany osobno dla różnych platform (MS Windows, Linux). Od wersji 9.2 źródła, które można ściągnąć, są wspólne, choć proces kompilacji i uruchamiania jest różny [BIND].

3.2. djbdns (tinydns)

Serwer djbdns napisany został przez amerykańskiego profesora matematyki Daniela J. Bernsteina. Na serwer ten składa się kolekcja programów. Osobnymi programami są: moduł zapytań autorytatywnych, zapytań rekurencyjnych, moduł transferu stref, odwrotny dns (zwany dnswall); nawet moduł odpowiedzialny za komunikację za pomocą TCP jest osobnym programem.

Autor napisał pakiet djbdns mając na uwadze dwie rzeczy, szybkość i bezpieczeństwo. Niestety spowodowało to, że w kilku miejscach serwer ten jest niezgodny ze specyfikacją pochodzącą z RFC [rfc1034, rfc1035]. Autor nie zastosował klasycznych plików stref, tylko

swój format, który jest kompilowany do postaci binarnej. Dzięki temu osiągnął między innymi to, że serwer nie trzyma drzewa stref ani samych stref w pamięci, a jedynie czyta potrzebne dane z dysku (dane są buforowane przez system operacyjny). Stosując taki model, serwer ma minimalne wymagania co do wielkości dostępnej pamięci i może działać w środowisku, gdzie pamięć dostępna dla pakietu będzie mniejsza niż sam BIND (sam program załadowany do pamięci, nie licząc danych pochodzących ze stref). Jednak takie rozwiązanie, pomimo dużej wydajności, posiada również wady, główną jest brak jakiejkolwiek zgodności tego formatu (zarówno wejściowego tekstowego, jak i ostatecznego binarnego) z przyjętymi standardami. Jedyną możliwością przekazania strefy jest jej transfer sieciowy. Inną wadą jest trudność w zaimplementowaniu dodatków do protokołu takich jak inkrementalny transfer stref.

Profesor Bernstein projektując serwer maksymalnie zmodularyzował go (pomimo implementacji w C). Dzięki temu uzyskał bardzo duży poziom bezpieczeństwa. Od pierwszej, publicznej wersji nigdy nie został oficjalnie ogłoszony jakikolwiek błąd krytyczny (przepełnienie bufora, wprowadzanie do pamięci podręcznej fałszywych informacji itp.). Co więcej, autor jest na tyle pewny swojego dzieła, że ogłosił nagrodę pieniężną ($500) dla pierwszej osoby, która taki błąd odkryje i opisze.

W większości testów ogólnodostępnych serwerów nazw djbdns ma od lat pierwsze miejsce. Niestety osoba profesora Bernsteina zniechęca sobą wiele osób, a przez to do świetnego oprogramowania, jakie on pisze. Jego polityka nie zmieniania (poprawiania) działających programów powoduje, że praktycznie żadne opcjonalne dodatki nie są dostępne oficjalnie, a jedynie w formie łat i nakładek pochodzących od innych osób. Serwer dzięki temu jest bardzo bezpieczny, jednak wspiera wyłącznie podstawową funkcjonalność.

Pakiet djbdns działa wyłącznie (informacje z oficjalnej strony projektu) na platformie UNIX (oprócz SCO UnixWare). Kompiluje się i działa poprawnie również pod systemem Linux [djbdns].

Rozdział 4

Projekt i implementacja serwera UnNamed

W tym rozdziale przedstawiamy projekt i implementację serwera UnNamed. Opisane są tu ważne decyzje projektowe, sposób realizacji głównych modułów, napotkane w czasie projektowania i implementacji problemy i sposób ich rozwiązania wraz z dyskusją ewentualnych alternatywnych metod rozwiązania.

4.1. O'Caml pod Windows a pod Uniksem

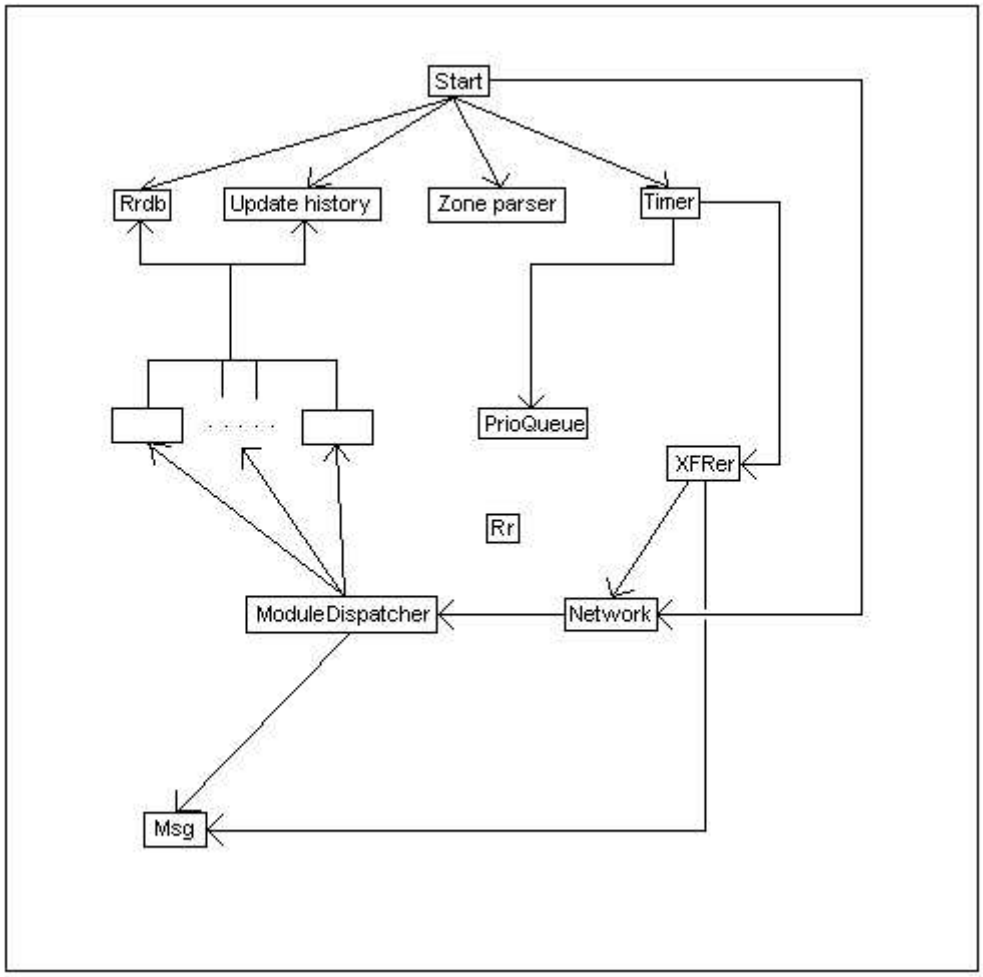
Aby w pełni zrozumieć projekt niektórych modułów należy zdawać sobie sprawę, że istnieją różnice w implementacji O'Camla na różne platformy. Większość tych różnic występuje w implementacji biblioteki Unix. W wersji na platformę Windows część funkcji ma nieco zmienioną semantykę lub ich działanie jest ograniczone, a część w ogóle nie jest zaimplementowana. Opiszemy pokrótce te różnice:

- asynchroniczne sygnały międzyprocesowe nie są w ogóle wspierane na platformie Windows – funkcja `kill` i jej pokrewne (`pause`, `alarm`) nie są zaimplementowane. Pod Windowsami proces/wątek może jedynie synchronicznie oczekiwać na nadejście sygnału;
- typ gniazd lokalnych (uniksowych) nie występuje na platformie Windows;
- znane z Uniksa systemy komunikacji międzyprocesowej/międzywątkowej, takie jak kolejki FIFO, nie działają na platformie Windows (aczkolwiek łącza nienazwane są zaimplementowane);
- działanie funkcji `select` zostało ograniczone tylko do gniazd sieciowych (czyli w szczególności nie można jej użyć do multipleksowania łączy nienazwanych);
- większość funkcji związanych z zarządzaniem uprawnieniami w systemie plików nie jest zaimplementowana bądź nie powoduje żadnych efektów – wynika to z faktu, iż system plików pod Windowsami ma dużo uboższe mechanizmy kontroli dostępu;
- podobnie sprawa ma się z funkcjami do obsługi identyfikacji użytkownika w systemie (`getuid`, `setuid` itp.);
- niskopoziomowe funkcje obsługujące tworzenie i łączenie procesów (`fork`, `wait`, itp) nie są zaimplementowane – O'Caml dostarcza wysokopoziomowe odpowiedniki.

4.2. Dekompozycja logiczna systemu

4.2.1. Podział na moduły

Na rysunku 4.1 przedstawiamy podział kodu na moduły. Został on tak zaprojektowany, by uzyskać maksymalną izolację modułów oraz by rozszerzenie funkcjonalności było możliwie łatwe i wymagało możliwie mało modyfikacji istniejącego kodu.



Rysunek 4.1: Diagram modułów

Oto krótka charakterystyka poszczególnych modułów:

- Start – jest odpowiedzialny za sparsowanie wiersza poleceń, utworzenie pozostałych wątków i obsługę sygnałów. Do jego obowiązków należy również inicjowanie pozostałych modułów.

- Rr – zdefiniowane są w nim abstrakcyjne struktury danych reprezentujące rekordy RR oraz podstawowe operacje na nich. Jest wykorzystywany przez większość pozostałych modułów – nie zostało to ukazane na rysunku, by nie utracić przejrzystości.

- Rrdb – przechowuje i udostępnia wczytane z plików strefowych rekordy RR.

- Update history – przechowuje i udostępnia historię zmian stref między wersjami. Używany podczas inkrementalnego transferu stref.
- ZoneParser – wczytuje i parsuje pliki strefowe.
- ConfParser – wczytuje i parsuje plik konfiguracyjny.
- Timer – odpowiedzialny za cykliczne, asynchroniczne inicjowanie zadań (np. transfer stref).
- PrioQueue – implementacja kolejki priorytetowej używana przez moduł Timer służąca do szeregowania zadań.
- Msg – zdefiniowane są w nim abstrakcyjne struktury danych reprezentujące komunikaty protokołu DNS oraz funkcje umożliwiające serializację i deserializację tychże komunikatów z i do postaci sieciowej (binarnej).
- Network – wielowątkowy serwer TCP/UDP. Przyjmuje i wysyła komunikaty, przekazuje je do obróbki.
- Xfrer – odpowiedzialny za transfer strefy z innego źródła.
- ModuleDispatcher – przyjmuje komunikaty od modułu Network, deserializuje je za pomocą modułu Msg i na podstawie rozpoznanego rodzaju komunikatu przekazuje go do dalszej obróbki odpowiedniemu modułowi.
- moduły odpowiedzialne za obsługę konkretnych komunikatów. Odbierają i przetwarzają komunikaty od modułu ModuleDispatcher:
  - StandardQuery – obsługuje standardowe zapytania.
  - Axfr – obsługuje przychodzące żądania AXFR transferu strefy.
  - Ixfr – obsługuje przychodzące żądania IXFR transferu strefy.
  - Notify – obsługuje żądanie DNS Notify.
  - Update – obsługuje żądanie DNS Update.

Standardowe rozszerzenie funkcjonalności o obsługę nowego rodzaju komunikatów polega na napisaniu modułu odpowiedzialnego za ich przetwarzanie (zazwyczaj znalezienie odpowiedzi), umieszczenie odwołania w module ModuleDispatcher oraz ewentualne rozszerzenie struktury komunikatów i rekordów RR.

4.2.2. Podział na wątki

Wątek główny serwera po zainicjowaniu wszystkich modułów uruchamia trzy podstawowe wątki:

- wątek nasłuchujący na gnieździe TCP,
- wątek przyjmujący datagramy na gnieździe UDP,
- wątek odpowiedzialny za cykliczne inicjowanie zadań (Timer).

Ponadto uruchamiana jest konfigurowalna liczba wątków obsługujących pojedyncze komunikaty (minimum jeden).

Wielowątkowa budowa serwera umożliwia łatwą obsługę wielu równoległych żądań bez zagrożenia zagłodzeniem. Ponadto chroni serwer przed sytuacją, gdy jedno zapytanie zablokuje cały system. Uruchomienie wątku Timer umożliwia asynchroniczne wykonywanie cyklicznych zadań. Główny wątek po zainicjowaniu modułów i pozostałych wątków zasypia w oczekiwaniu na sygnał kończący działanie serwera. Jest to konieczne ze względu na implementację sygnałów na platformie Windows, gdzie sygnał nie przerywa innej funkcji systemowej (zastosowanie typowego dla Uniksa wzorca programistycznego mogłoby spowodować, że zakończenie serwera odwlekłoby się w czasie nawet o parę godzin, na przykład, gdy wszystkie wątki byłyby zawieszone podczas wykonywania funkcji systemowej typu `select` czy `sleep`).

4.3. Projekt i architektura najważniejszych modułów

4.3.1. ModulDispatcher

Moduł ten został wprowadzony do projektu w celu zwiększenia modularności i izolacji pomiędzy komponentami. Jego zadanie polega na rozpoznaniu rodzaju żądania od klienta i przekazaniu go odpowiedniemu modułowi obsługującemu ten rodzaj żądań. Dzięki temu kod jest czytelniejszy, a dodanie obsługi nowego rodzaju żądań prostsze. Wystarczy zaimplementować moduł obsługujący nowy rodzaj zapytań i umieścić informację o tym w strukturze modułu ModulDispatcher (jest to potencjalnie jedyne miejsce, gdzie trzeba modyfikować istniejący kod).

4.3.2. Timer

Moduł ten implementuje szeregowanie zadań. Warty uwagi jest w nim sposób reakcji na pomyślne lub nie zakończenie wykonywania uszeregowanego zadania. Z każdym rodzajem zadań związany jest funkcja zwrotna, która wywoływana jest w momencie zakończenia wykonywania zadania i może na przykład powodować ponowne uszeregowanie tego zadania w przypadku porażki. Dzięki temu autor piszący moduł, który być może będzie wykonywał również zadania cykliczne, musi jedynie na koniec funkcji wywołać podaną w argumencie funkcję zwrotną nie martwiąc się czy dane zadanie jest cykliczne czy nie i jak zareagować w przypadku porażki bądź sukcesu (na przykład poprzez ręczne ponowne uszeregowanie tego zadania).

4.3.3. Network

Typowe modele serwerów sieciowych

Klient korzystający z usług serwera oczekuje, że jego żądanie zostanie obsłużone w możliwie najkrótszym czasie. Dla często używanych usług (np. scentralizowane i rozproszone bazy danych, rozproszony system plików NFS), projekt serwera jest sprawą krytyczną dla wydajności całego systemu rozproszonego. Przy projektowaniu serwera należy wziąć pod uwagę typ usługi oraz oczekiwane obciążenie (liczba klientów na sekundę). Przy wyborze odpowiedniego wzorca projektowego należy wcześniej odpowiedzieć na następujące pytania: ile czasu zajmuje obsłużenie typowego żądania od klienta? Jak dużo nowych żądań może przyjść podczas obsługi typowego żądania? Jaki jest akceptowalny czas oczekiwania przez klienta na odpowiedź? Jaką złożoność budowy serwera możemy przyjąć, aby móc go zaimplementować i zweryfikować? W jakim stopniu wydajność naszego serwera wpłynie na ogólną wydajność systemu rozproszonego?

Doświadczenie pokazuje, że jedną z pierwszych decyzji projektowych jest wybór odpowiedniego modelu klient–serwer. W kolejnych podrozdziałach opiszemy typowe modele.

Jeden wątek, jeden klient Jest to najprostszy model. Serwer wykonując nieskończoną pętle słucha na porcie, akceptuje nadchodzące połączenie i obsługuje natychmiast w tym samym wątku. Następni klienci muszą czekać aż poprzednie żądanie zostanie całkowicie obsłużone.

Model ten jest bardzo prosty do zaimplementowania i wymaga mało zasobów: jednego wątku i dwóch gniazd (gniazdo nasłuchujące i symbolizujące otwarte połączenie). Użycie tego modelu sensowne jest tylko w przypadku najprostszych usług.

Jeden wątek, wielu klientów jednocześnie Model ten podobny jest do poprzedniego, gdyż również używa tylko jednego wątku z tą różnicą, że może obsługiwać wielu klientów jednocześnie, rozdzielając czas między nich. Wewnątrz pętli nieskończonej, serwer odpytuje połączenia sieciowe (w przypadku systemów Unix i Microsoft Windows stosuje się do tego funkcję `select`) — jeżeli stwierdzi, że połączenie jest gotowe do wysłania/odebrania komunikatu, to sprawdza, czy jest to nowe połączenie (i akceptuje je), czy też już istniejące połączenie (wysyła/odbiera odpowiednie dane).

Schemat ten może być użyty z powodzeniem w przypadku usług, przy których czas generowania odpowiedzi jest nieduży. W sytuacji, gdy przetwarzanie pojedynczego żądania trwa stosunkowo długo, inni klienci muszą czekać. Często sytuacja taka jest nie akceptowalna.

Jeden wątek na jednego klienta Model ten jest łatwy do implementacji i może okazać się najszybszy w przypadku, gdy maksymalna liczba jednoczesnych żądań nie przekracza kilkudziesięciu. Możliwe są dwa scenariusze: w pierwszym kod związany z obsługą komunikacji sieciowej podzielony jest między wątek główny a wątki obsługujące w ten sposób, że wątek główny tylko akceptuje połączenia, a cała dalsza obsługa wykonywana jest przez wątek obsługujący; w drugim natomiast cały kod komunikacji sieciowej wykonywany jest przez jeden wątek główny (ten sam, który nasłuchuje), a wątek obsługujący tworzony jest dopiero po otrzymaniu pełnego komunikatu.

Pierwszy przypadek może być użyty wyłącznie w zaufanej sieci prywatnej. Łatwo sobie wyobrazić atak typu DoS (ang. *Denial of Service*), który całkowicie zablokuje taki serwer. Jednak taka konstrukcja ma również zalety. Kod jest przejrzyście podzielony, wątek główny zajmuje się akceptacją połączeń, a wątki obsługujące całą pracą związaną z żądaniem, zaczynając od znalezienia odpowiedzi, a kończąc na wysłaniu tej odpowiedzi poprzez sieć.

Drugi przypadek trochę komplikuje kod, jednak zasoby są dobrze chronione. Wątek obsługujący żądanie istnieje tylko podczas generowania odpowiedzi i natychmiast po znalezieniu jej i przekazaniu do wątku głównego zostaje zniszczony.

Wadą tego modelu (niezależnie od przyjętego scenariusza) jest mała skalowalność. Koszty związane z tworzeniem i utrzymywaniem dużej liczby wątków ograniczają mocno maksymalną liczbę jednocześnie obsługiwanych klientów.

Wątki obsługujące Model ten znacznie poprawia skalowalność rozwiązania w porównania z poprzednim. Jednak złożoność kodu również wzrasta, co powoduje, że w przypadku małej liczby jednoczesnych żądań czas odpowiedzi jest większy niż w modelu z jednym wątkiem na jednego klienta.

Tutaj również możliwe są tu dwa scenariusze: w pierwszym zadaniem głównego wątku jest odpytywanie połączeń i przekazywanie ich do wątków obsługujących, gdy tylko stanie się

możliwa operacja wejścia/wyjścia. W drugim przypadku cała komunikacja sieciowa obsługiwana jest przez wątek główny, a do wątku obsługującego przekazywany jest komunikat po jego pełnym odczytaniu.

W przypadku tego modelu problemem może być określenie prawidłowej liczby wątków obsługujących: jeżeli będzie za duża, to system stanie się niestabilny, jeżeli będzie za mała, to klienci będą musieli stosunkowo długo czekać na odpowiedź. Optymalna liczba zależy od wielu czynników, dlatego należy przeprowadzić odpowiednie testy profilujące, by ją ustalić. Na maszynach wieloprocesorowych stosuje się prostą zasadę, że wątków powinno być przynajmniej tyle ile procesorów. Liczba ta powinna być zwiększona, jeżeli którykolwiek czeka na połączeniu wejścia/wyjścia.

Przyjęte rozwiązanie

Moduł Network jest implementacją wielowątkowego serwera TCP i UDP. Jego zadaniem jest przyjmowanie i wysyłanie komunikatów protokołu DNS. W niniejszym rozdziale opiszemy historię powstawania jego projektu od wersji najstarszej aż do ostatecznej.

W wersji pierwotnej moduł był zrealizowany w postaci jednego wątku nasłuchującego, pewnej liczby wątków obsługujących oraz kolejki komunikacyjnej (model wątków obsługujących). Jego diagram przedstawiono na rysunku 4.2.

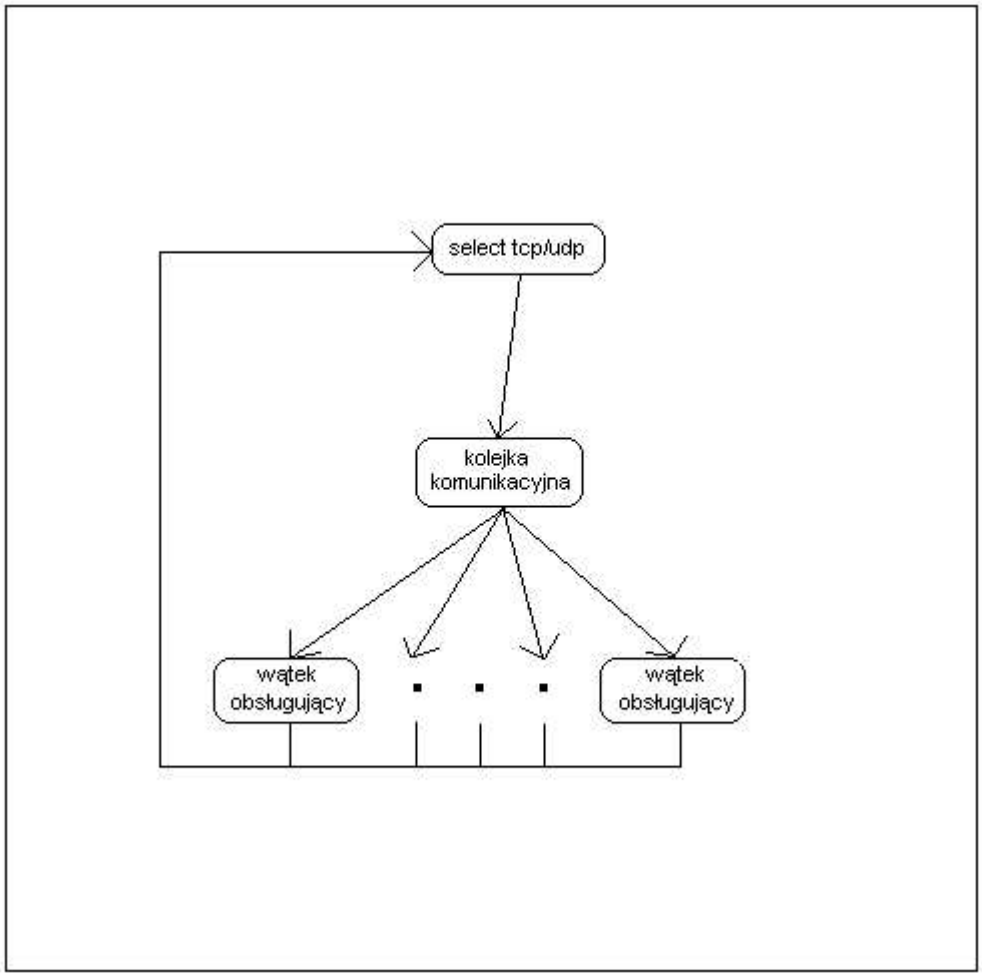
Zadaniem wątku nasłuchującego było wykonywanie w pętli systemowej funkcji `select` na gnieździe nasłuchującym (na nowe połączenia TCP), gnieździe UDP oraz wszystkich otwartych połączeniach TCP, które nie są aktualnie obsługiwane przez wątki obsługujące. W momencie uzyskania numeru deskryptora z funkcji `select`, wątek ten sprawdza czy symbolizuje on (deskryptor) otwarte połączenie TCP lub gniazdo UDP i jeżeli tak, to umieszcza je w kolejce komunikacyjnej (w tym miejscu połączenie rozumiane jest jako abstrakcyjna struktura danych). W przypadku połączenia TCP dodatkowo zaznaczana jest flaga mówiąca, iż jest ono aktualnie obsługiwane. Jeżeli natomiast deskryptor ten odpowiada gniazdu do nasłuchiwania na nowe połączenia TCP, to akceptuje to połączenie i zaznacza jako aktualnie nieobsługiwane.

Wątek obsługujący wyjmuje połączenie z kolejki komunikacyjnej i w zależności od typu połączenia zachowuje się następująco:

- dla UDP pobiera datagram z gniazda i jeżeli jest on poprawny (dał się zdeserializować), to przesyła go do modułu ModuleDispatcher, po czym wysyła uzyskaną odpowiedź (za pomocą gniazda UDP).

- dla TCP próbuje dokonać odpowiedniej operacji (odczyt lub zapis — jest to zapisane w strukturze symbolizującej połączenie), przy czym próba ta podejmowana jest tylko raz. Jeżeli nie udało się zapisać/odczytać całego komunikatu, to w strukturze opisującej połączenie zostaje zapisana liczba pobranych/wysłanych bajtów oraz przeczytana część komunikatu (w przypadku odczytu). Następnie połączenie zostaje przekazane do wątku nasłuchującego, który przekaże go znowu do kolejki, gdy operacja wejścia/wyjścia będzie nieblokująca. Jeżeli po odczycie wątek obsługujący stwierdzi, że został odczytany już cały komunikat, to przesyła go do dispatchera, po czym próbuje wysłać odpowiedź według podanego schematu.

Dużym problemem podczas projektowania okazał się mechanizm przekazywania połączeń przez wątki obsługujące do wątku nasłuchującego. W środowisku uniksowym używa się w tym celu zazwyczaj łączy nienazwanych (ang. *pipe*), kolejek FIFO (łączy nazwanych) lub gniazd uniksowych (lokalnych). W tym przypadku nie mogliśmy ich jednak użyć, gdyż Windowsowa wersja O'Camla w ogóle nie wspiera gniazd lokalnych, a funkcja `select` nie działa dla łączy,



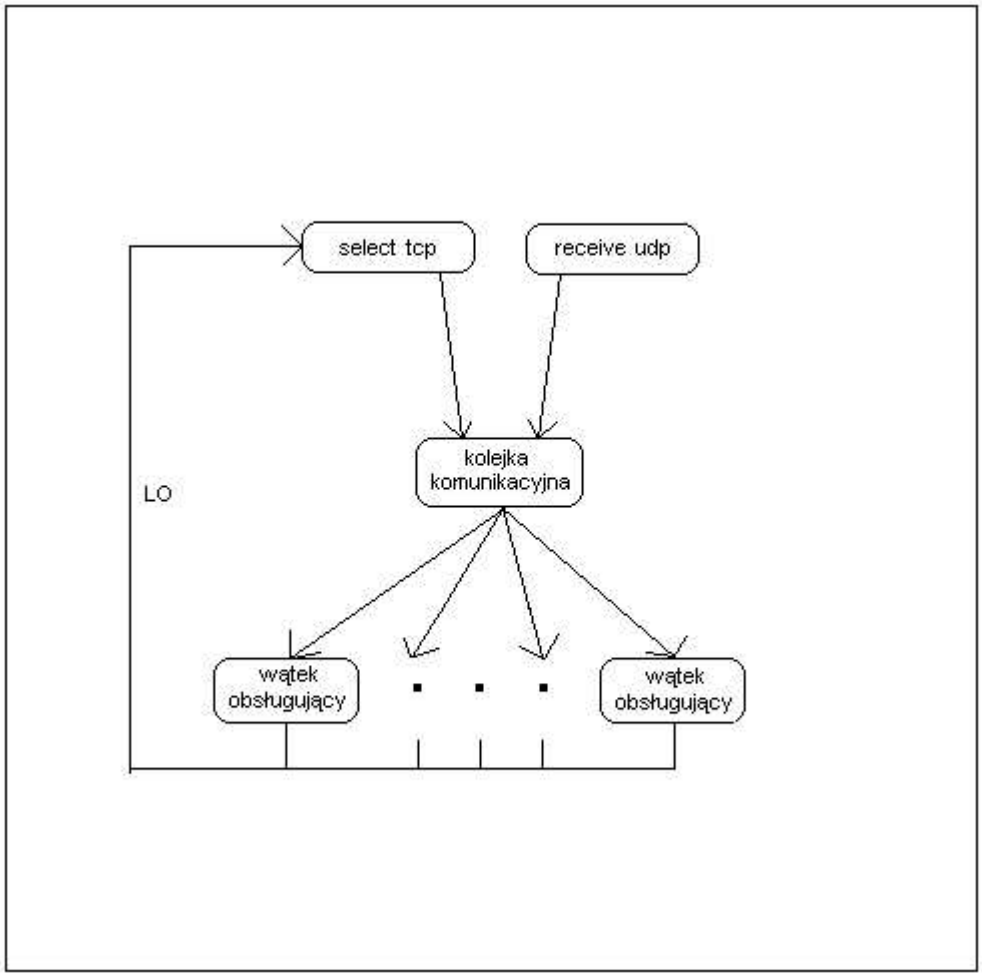
Rysunek 4.2: Diagram pierwotnej wersji modułu Network

a jedynie dla gniazd. Kolejnym pomysłem było użycie mechanizmu komunikacji niezależnego od platformy (kanałów O'Camlowych). Okazało się to również niemożliwe, ponieważ nie ma w O'Camlu mechanizmu pozwalającego wykonywać funkcję `select` równocześnie dla gniazd sieciowych oraz kanałów. Komunikacja za pomocą sygnałów również nie jest wspierana przez windowsową wersję O'Camla (brakuje asychronicznych sygnałów na platformie Microsoft Windows).

Ostatecznie zdecydowaliśmy się na użycie gniazd TCP przywiązanych do lokalnego interfejsu sieciowego (loopback/LO). Potencjalny problem ataku polegającego na próbie połączenia się nieautoryzowanego klienta z tym gniazdem nie stanowi problemu dzięki użyciu funkcji `socketpair`. Funkcja ta przekazuje parę połączonych ze sobą gniazd.

Architektura z jednym wątkiem nasłuchującym jednocześnie na TCP i UDP ma jednak pewną wadę. Gdy większość komunikatów przychodzi przy użyciu protokołu UDP (a tak zazwyczaj jest), powstaje niepotrzebny narzut na wykonywanie co chwila funkcji `select` (wątek nasłuchujący jest zawieszony na funkcji `select`, przychodzi datagram UDP, wątek nasłuchujący wychodzi z funkcji `select`, wykonywana jest funkcja `receive`, wątek nasłuchujący zawisa ponownie na funkcji `select`, przychodzi kolejny datagram UDP itd.). W związku z taką sytuacją zdecydowaliśmy się na rozdzielenie wątku nasłuchującego na dwa inne: jeden dla TCP i jeden dla UDP. Wątek TCP wykonuje funkcję `select` na gnieździe nasłuchującym,

wszystkich otwartych połączeniach TCP, które nie są aktualnie obsługiwane przez wątki obsługujące oraz na połączeniach zwrotnych od wątków obsługujących. Wątek UDP wykonuje tylko funkcję `receive`. Rysunek 4.3 przedstawia ostateczną wersję.



Rysunek 4.3: Diagram ostatecznej wersji modułu Network

Braliśmy też pod uwagę rozdzielenie wątków obsługujących ze względu na połączenie TCP i UDP (kolejka komunikacyjna zostałaby również podzielona na dwie). Rozwiązanie takie powoduje jednak, że część zasobów może pozostawać przez dłuższy czas nie używana, ponadto powstaje trudność konfiguracji liczby poszczególnych wątków obsługujących.

Więcej na temat wielowątkowych serwerów TCP można znaleźć w [Stevens, Comer].

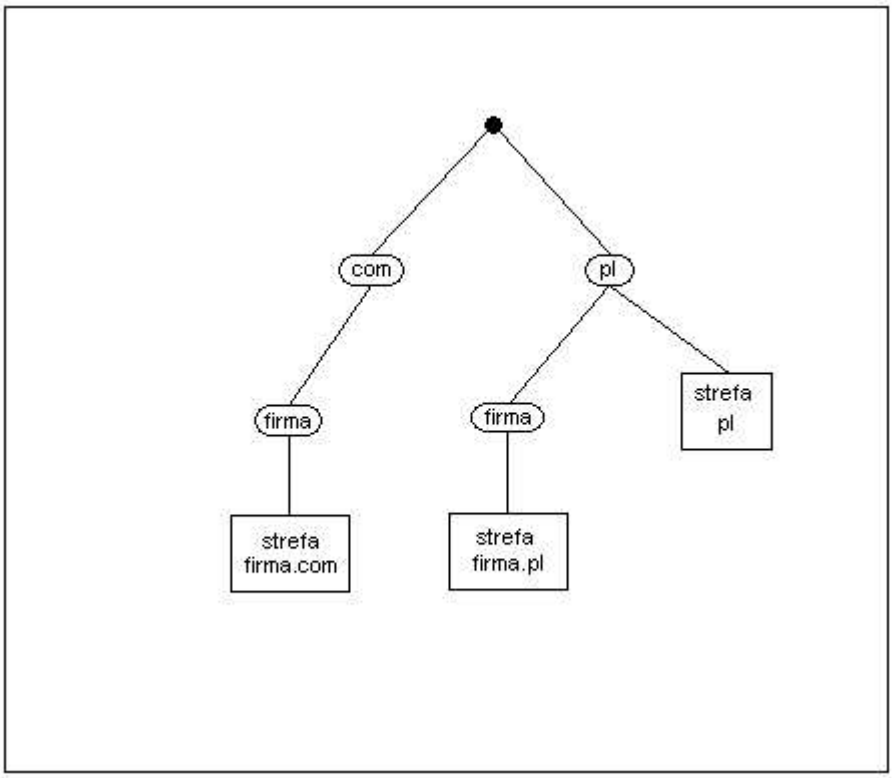
Kolejka komunikacyjna

Kolejka komunikacyjna została zrealizowana poprzez standardową O'Camlową strukturę danych chronioną monitorem działającym według wzorca producenci-konsumenci. Monitor ten ma standardowo dwa wejścia, jedno dla wątku nasłuchującego i jedno dla wątków obsługujących. Wątek obsługujący wchodzi do monitora i jeżeli w kolejce znajdują się jakieś połączenia, to pobiera pierwsze z nich i wychodzi. W przeciwnym przypadku zawiesza się na zmiennej warunkowej. Wątek nasłuchujący wchodzi do monitora, wrzuca połączenie do kolejki, wykonuje `signal` na zmiennej warunkowej i wychodzi z monitora.

4.3.4. Rrdb

Interfejs tego modułu został zaprojektowany tak, aby nie narzucać sposobu przechowywania danych. Nasza implementacja przechowuje je w pamięci, jednak z łatwością można napisać ten moduł tak, aby dane były przechowywane na dysku, czy nawet w relacyjnej bazie danych bądź w katalogu LDAP.

Nasza implementacja została zrealizowana jako "rzadkie" drzewo stref, którego węzły etykietowane są takimi samymi etykietami jak drzewo przestrzeni nazw (np. `edu`, `pl`, `mimuw`). Niektóre z węzłów przechowują dowiązania do stref (rzadkie drzewo oznacza w tym przypadku, że tylko w niektórych węzłach znajdują się strefy). Dowiązania do dzieci danego węzła przechowywane są w tablicy haszującej, której kluczami są właśnie etykiety przestrzeni nazw. Na diagramie 4.4 przedstawione jest drzewo stref dla serwera, który obsługuje domeny "`pl.`" oraz "`firma.com.`".

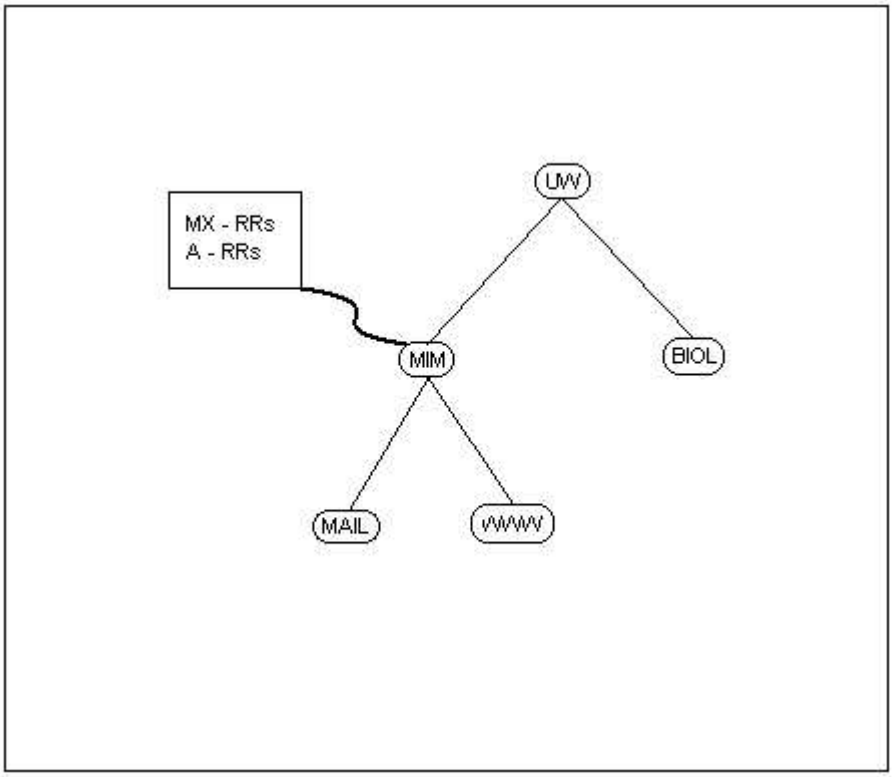


Rysunek 4.4: Diagram przykładowego drzewa stref

Pojedyncza strefa jest zrealizowana również jako drzewo etykietowane takimi samymi etykietami co węzły w drzewie przestrzeni nazw. Pojedynczy węzeł przechowuje tablicę haszującą, w której kluczami są typy rekordów RR, a wartościami kolekcje rekordów danego typu. Przykładowe drzewo przedstawione jest na diagramie 4.5.

Strefa zabezpieczona jest monitorem działającym według wzorca jeden pisarz – wielu czytelników. Wszelkie zmiany strefy wykonywane są jednak nie na oryginalnej strukturze strefy, lecz na jej kopii. Po wprowadzeniu zmian do kopii, oryginalna wersja zostaje podmieniona na kopię, dzięki czemu strefa blokowana jest na bardzo krótki czas (tylko tyle, ile zajmuje podmiana wskaźnika).

Większość operacji na drzewie strefy opiera się na generycznej funkcji wyższego rzędu, która jako parametr przyjmuje funkcję wykonującą konkretną operację na danym węźle.



Rysunek 4.5: Diagram przykładowego drzewa strefy

Inne implementacje kolekcji stref i samej strefy

Wnikliwemu czytelnikowi może nasunąć się pytanie, dlaczego nie zastosowaliśmy tablicy haszującej do przechowywania i wyszukiwania stref. Aby móc odpowiedzieć na to pytanie, należy przypomnieć, że klient pyta się o całą nazwę symboliczną maszyny. Na nazwę maszyny składają się dwie części, nazwa strefy oraz nazwa węzła w strefie (nazwa "`zodiac.mimuw.edu.pl`" składa się z dwóch części: "`mimuw.edu.pl`" jako nazwa strefy oraz "`zodiac`" jako nazwa węzła w tej strefie). Podczas parsowania całej nazwy, serwer nie wie, do którego miejsca jest nazwa strefy, a gdzie zaczyna się nazwa węzła wewnątrz strefy. Ta sama uwaga dotyczy analogicznego pytania, dlaczego pojedyncza strefa nie została zorganizowana jako tablica haszująca. W tym wypadku serwer nie wie, gdzie kończy się drzewo tej strefy i w którym węźle jest wskaźnik do podstrefy (mając strefę "`gegez.com`", dostając pytanie "`host.podstrefa.gegez.com`", na które nie ma odpowiedzi w tablicy haszującej, nie wie, czy jest to spowodowane tym, że węzeł ten znajduje się w innej strefie, czy nie istnieje rekord o danym typie dla tego węzła. Powstaje problem wydajnego znalezienia odpowiedzi na to pytanie).

Zamiast budować skomplikowane drzewa, można by zastosować listy (zarówno do trzymania kolekcji stref, jak i samej strefy). Takie rozwiązanie jest rzeczywiście bardzo proste do zaimplementowania, jednak nie byłoby zbyt wydajne. Przeszukiwanie liniowe takiej listy mogłoby stać się wąskim gardłem w wydajności całego systemu. Możliwość zastosowania tablic haszujących została omówiona w poprzednim paragrafie. Z kolei w miejscach gdzie zostały użyte tablice haszujące można by rozważyć użycie drzew binarnego wyszukiwania. Zostały one jednak użyte tam gdzie kluczami wyszukiwania są napisy, których porównywanie w zestawieniu z porównywaniem liczb całkowitych jest kosztowne.

Rozdział 5

Testy

5.1. Opis środowiska testowego

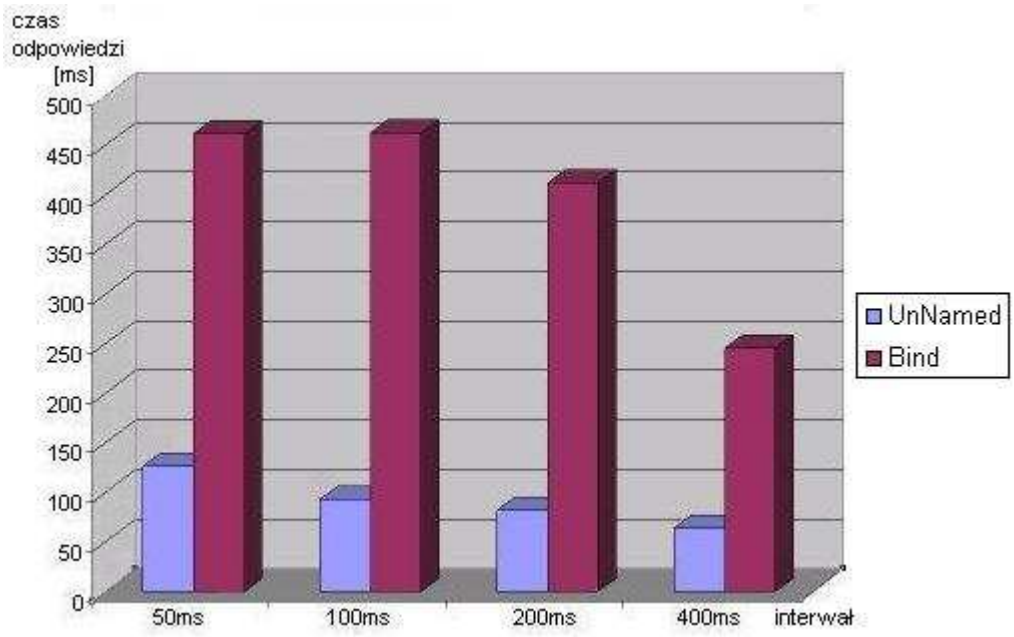
Testy wydajnościowe zostały przeprowadzone na następujących maszynach:

- OLDOAK – procesor K6-2/500MHz, pamięć 448MB, system operacyjny RedHat 9,
- DARKSTORM – procesor Athlon 2000+, pamięć 1GB, system operacyjny RedHat 9

Obie maszyny były połączone siecią Ethernet 100.

5.2. Testy porównawcze z serwerem BIND w wersji 9.2.1

5.2.1. Test 1



Rysunek 5.1: Czas odpowiedzi (w milisekundach) testowanych serwerów na zapytanie typu AXFR przy bombardowaniu zapytaniami typu AXFR

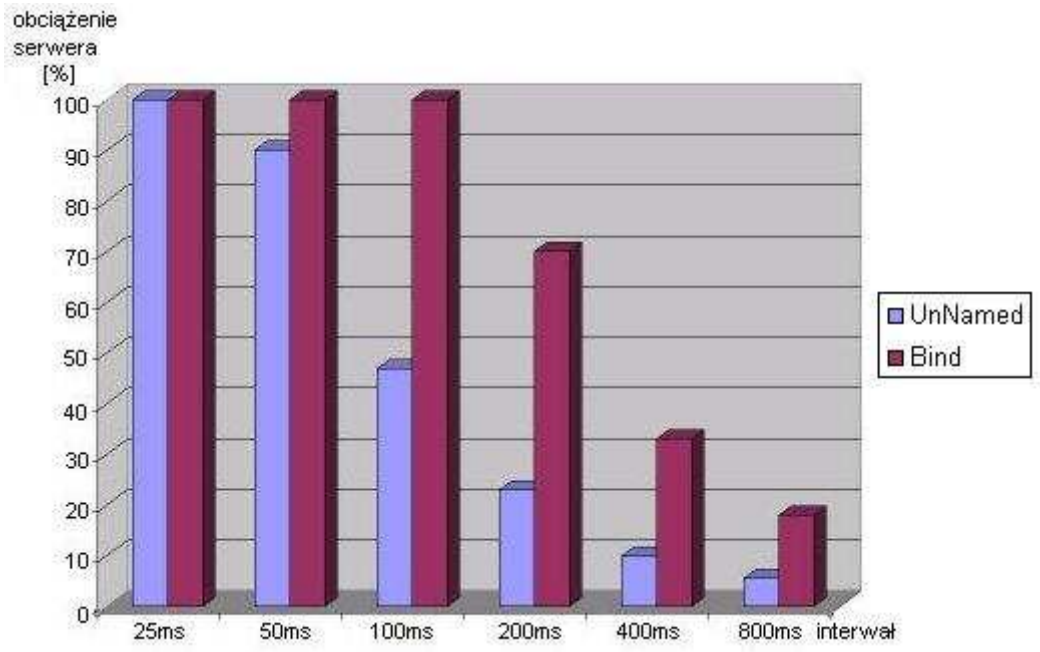
Test polegał na pomiarze czasu odpowiedzi na zapytanie typu AXFR (o strefę `mimuw.edu.pl`). Jednocześnie serwer był bombardowany innymi, niezależnymi zapytaniami typu AXFR (o strefę `mimuw.edu.pl`) co pewien interwał czasu. Wyniki testu podano w zależności od interwału pomiędzy zapytaniami bombardującymi. W pierwszej fazie testu na maszynie OLDOAK został uruchomiony BIND, a zapytania zadawano z maszyny DARKSTORM. Druga faza wyglądała analogicznie z tą różnicą, że na maszynie OLDOAK uruchomiony był serwer UnNamed. Dla każdej wartości interwału pomiar wykonano sto razy, a na wykresie przedstawiono wartość uśrednioną. Odchylenia standardowe dla poszczególnych interwałów przedstawiono w tabelce:

	50ms	100ms	200ms	400ms
UnNamed	32,78ms	19,5ms	12ms	7,67ms
BIND	32,8ms	35,54ms	29,6ms	32,75ms

Konfiguracja obydwu serwerów (BIND oraz UnNamed) była analogiczna – obsługiwały one dwadzieścia małych stref oraz jedną większą `mimuw.edu.pl`.

Jak widać na diagramie 5.1 UnNamed wypadł tu znacznie lepiej. Był do czterech razy szybszy.

5.2.2. Test 2



Rysunek 5.2: Obciążenie procesora wyrażone w procentach przy bombardowaniu zapytaniami typu AXFR

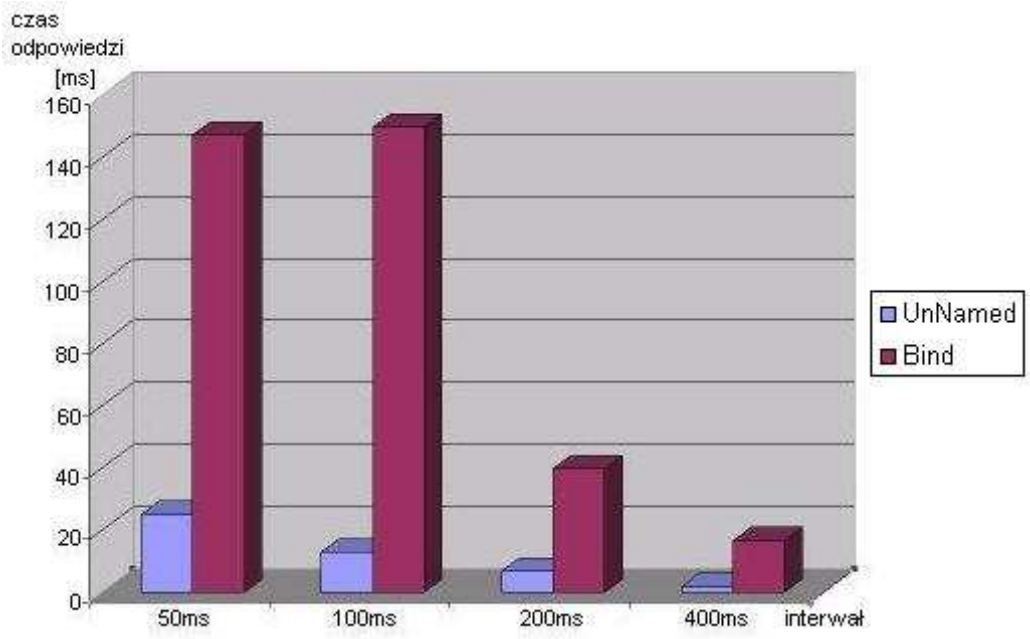
Test polegał na pomiarze średniego obciążenia procesora przez trzydzieści sekund podczas bombardowania serwera zapytaniami typu AXFR (o strefę `mimuw.edu.pl`) w zależności od interwału pomiędzy tymi zapytaniami bombardującymi. W pierwszej fazie testu na maszynie OLDOAK został uruchomiony BIND, a zapytania zadawano z maszyny DARKSTORM. Druga faza wyglądała analogicznie z tą różnicą, że na maszynie OLDOAK uruchomiony był serwer UnNamed. Konfiguracja serwerów i stref była identyczna jak w teście numer 1. Dla

każdej wartości interwału pomiar wykonano dziesięć razy, a na wykresie przedstawiono wartość uśrednioną. Odchylenia standardowe dla poszczególnych interwałów przedstawiono w tabelce (jednostką są punkty procentowe, w skrócie pp.):

	25ms	50ms	100ms	200ms	400ms	800ms
UnNamed	0pp.	1.2pp.	2.4pp.	3,1pp.	3,2pp.	1,1pp.
BIND	0pp.	0.7pp.	0pp.	3,8pp.	2,4pp.	2,2pp.

Jak widać z diagramu 5.2 UnNamed potrzebuje znacznie mniejszej mocy obliczeniowej niż BIND. Interwał 100ms między zapytaniami okazał się maksymalnym obciążeniem jakie mógł znieść BIND, podczas gdy UnNamed miał jeszcze sporo zapasu.

5.2.3. Test 3



Rysunek 5.3: Czas odpowiedzi testowanych serwerów na zapytanie typu A przy bombardowaniu zapytaniami typu AXFR

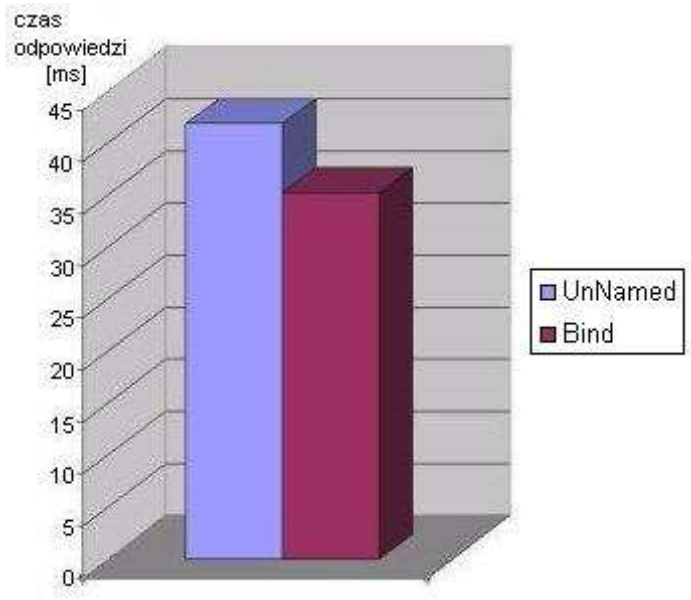
Test ten był podobny do testu numer 1 z tą różnicą, że mierzono czas odpowiedzi na na zapytanie typu A (o adres maszyny `cnametst1.mimuw.edu.pl`) zamiast na zapytanie typu AXFR. Podobnie jak w teście numer 1, dla każdej wartości interwału pomiar wykonano sto razy, a na wykresie przedstawiono wartość uśrednioną. Odchylenia standardowe dla poszczególnych interwałów przedstawiono w tabelce:

	50ms	100ms	200ms	400ms
UnNamed	5,36ms	9,06ms	7ms	2,34ms
BIND	30,34ms	29,52ms	28,98ms	20,93ms

Diagram 5.3 wygląda podobnie do diagramu 5.1 – znacząca przewaga serwera UnNamed.

5.2.4. Test 4

Test polegał na pomiarze czasu odpowiedzi na zapytanie typu AXFR (o strefę `mimuw.edu.pl`) przy jednoczesnym bombardowaniu serwera zapytaniami typu A (o adres maszyny



Rysunek 5.4: Czas odpowiedzi testowanych serwerów na zapytanie typu AXFR przy bombardowaniu zapytaniami typu A

`cnametst1.mimuw.edu.pl`) bez opóźnienia pomiędzy tymi zapytaniami bombardującymi. Tym razem serwery (najpierw BIND, a potem UnNamed) były uruchomione na maszynie DARKSTORM, a zapytania zadawane były z maszyny OLDOAK. Dla każdego serwera pomiar wykonano sto razy, a na wykresie przedstawiono wartość uśrednioną. Odchylenia standardowe wynosiły odpowiednio 1,55ms dla serwera UnNamed i 0,79ms dla serwera BIND. Konfiguracja serwerów i stref była identyczna jak w teście numer 1.

Na diagramie 5.4 widać, że w tym teście wyniki są już bardziej porównywalne. Tym razem BIND okazał się nieznacznie szybszy (około 15%).

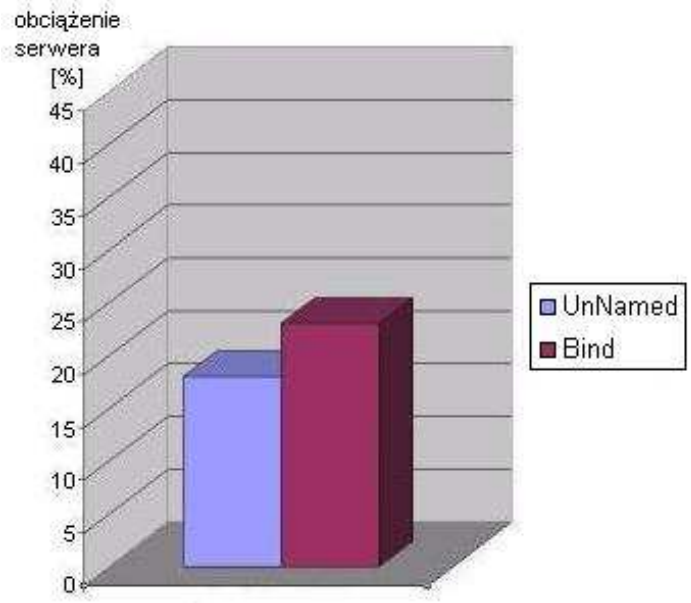
5.2.5. Test 5

Test polegał na pomiarze średniego obciążenia procesora przez trzydzieści sekund podczas bombardowania serwera zapytaniami typu A (o adres maszyny `cnametst1.mimuw.edu.pl`) bez interwału pomiędzy tymi zapytaniami bombardującymi. I tym razem serwery (najpierw BIND, a potem UnNamed) były uruchomione na maszynie DARKSTORM, a zapytania zadawane były z maszyny OLDOAK. Dla każdego serwera pomiar wykonano sto razy, a na wykresie przedstawiono wartość uśrednioną. Odchylenia standardowe wynosiły odpowiednio 0,9 punktu procentowego dla serwera UnNamed i 1,1 punktu procentowego dla serwera BIND. Konfiguracja serwerów i stref była identyczna jak w teście numer 1.

Diagram 5.5 pokazuje, że wbrew oczekiwaniom po wynikach testu numer 4, UnNamed zużywał nieco mniej czasu procesora niż BIND.

5.2.6. Test 6

Test polegał na pomiarze czasu odpowiedzi na zapytanie typu A (o adres maszyny `satano.z666.ee`) w zależności od liczby stref obsługiwanych przez serwer. Test jak zwykle przeprowadzono osobno na serwera BIND i UnNamed uruchomionych na maszynie OLDOAK. Klientem był DARKSTORM. Wszystkie obsługiwane strefy składały się z kilku rekordów. Obydwa ser-



Rysunek 5.5: Obciążenie testowanych serwerów przy bombardowaniu zapytaniami typu A

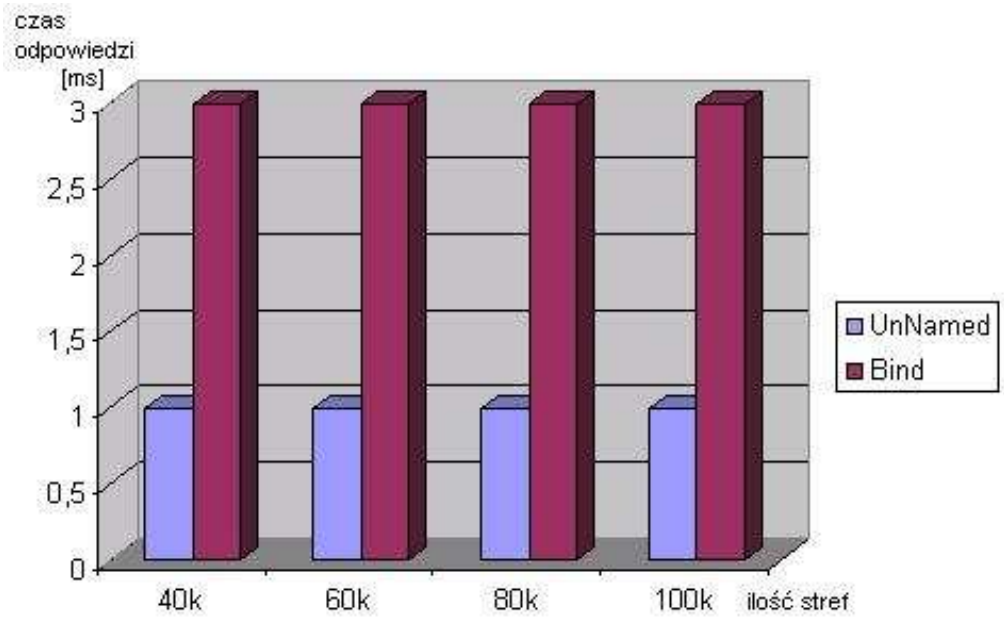
wery używały wyłącznie pamięci fizycznej (nie używały pamięci wirtualnej) w granicach stu tysięcy stref. Dla każdej liczby stref pomiar wykonano sto razy, a na wykresie przedstawiono wartość uśrednioną. Odchylenia standardowe wyniosły 0ms dla obu serwerów.

Jak wynika z diagramu 5.6, dopóki serwery używały wyłącznie pamięci fizycznej, dopóty czas odpowiedzi był niezależny od liczby stref. UnNamed okazał się tu ponad trzy razy szybszy.

5.2.7. Podsumowanie testów

Przeprowadzone testy pokazują, że UnNamed ma bardzo dobrze zaprojektowaną obsługę zarówno żądań TCP, jak i żądań mieszanych (TCP i UDP). BIND natomiast zoptymalizowany jest jedynie pod kątem żądań UDP.

Widać również, iż UnNamed używa lepiej zaprojektowanych struktur danych do przechowywania rekordów RR, co uwidacznia się przy dużej ilości obsługiwanych danych.



Rysunek 5.6: Czas odpowiedzi na zapytanie typu A w zależności od liczby stref obsługiwanych przez testowane serwery

Podsumowanie

Przedstawiona w pracy implementacja serwera DNS UnNamed przekroczyła nasze najśmielsze oczekiwania jeśli chodzi o wydajność. W świetle faktu, że O'Caml jest średnio trzy razy wolniejszy niż C, a UnNamed okazał się do pięć razy szybszy niż BIND, osiągnęliśmy wydajność do piętnastu razy większą niż oczekiwana. Biorąc pod uwagę łatwość rozszerzalności i pielęgnacji naszej implementacji oraz wysoką stabilność uzyskaną dzięki specyfice języka O'Caml, uważamy, że jest duża szansa, iż nasz serwer będzie w przyszłości szeroko używany w Internecie. Oczywiście będzie to wymagać zaimplementowania kilku powszechnie używanych funkcjonalności, takich jak perspektywy (zależność odpowiedzi od adresu IP klienta) czy DNSSEC-bis.

Jednym z celów naszej pracy było sprawdzenie przydatności języka O'Caml w implementacji systemów rozproszonych. Na podstawie testów i naszych doświadczeń możemy śmiało stwierdzić, iż jest to dobry język do tego typu zastosowań. Na opinię tę największy wpływ miały przenośność kodu między systemami operacyjnymi, wydajność, bezpieczeństwo oraz łatwość kodowania.

Podana tabelka zawiera ogólne porównanie cech opisywanych wcześniej serwerów:

cecha	UnNamed	BIND	djbdns
wydajność	wysoka	wysoka	wysoka
stabilność	wysoka	średnia	wysoka
bezpieczeństwo	wysokie	niskie	wysokie
modularność	wysoka	średnia	wysoka
rozszerzalność	wysoka	niska	niska
liczba rozszerzeń	mała	duża	średnia
dokumentacja kodu	dobra	słaba	średnia

Dodatek A

Zawartość płyty CD

Katalog `mgr` zawiera elektroniczną wersję tej pracy w formacie pdf oraz źródła Latex wraz ze wszystkimi plikami pomocniczymi (rysunkami, stylami itd).

W Katalogu `doc/tech` znajduje się dokumentacja techniczna w formacie html, natomiast w `doc/rfc` są dokumenty RFC będące specyfikacją DNS.

Kod źródłowy serwera UnNamed znajduje się w katalogu `src`. Aby skompilować program należy poprostu wywołać program make w tym katalogu. W podkatalogu `perfTests` znajduje się kod programów generujących zapytania bombardujące użytych podczas testów. Katalog `test` zawiera pliki związane z testami: konfiguracje serwerów, wyniki, skrypty, pliki strefowe itp.

Dodatek B

Dokumentacja techniczna

Module `Start` : This is the start module of the server.

It parses the command line, initializes data bases, the network server, the timer and other modules.

Module `Axfr` : this module implements traditional non-incremental zone transfer to client (response to AXFR query)

```
val findAnswer : Msg.msg -> Msg.question -> Unix.sockaddr -> Msg.msg
```

finds answer to axfr request. The AXFR question must be the only question in given request - otherwise the AXFR question is ignored.

Module `InXfr` : This module is a helping module to Xfrer.

It handles messages connected with zone transfers from other server to us by calling appropriate routines from Xfrer.

```
val handle : Msg.msg -> Unix.sockaddr -> Msg.msg option
```

handles the message from peer. Eventually returns the message that should be sent.

```
val registerSoaCallback :
  (Msg.msg -> Unix.sockaddr -> Msg.msg option) -> unit
```

used by Xfrer to register function routine that should be called when answer to SOA request arrives.

```
val registerUdpIxfrCallback :
  (Msg.msg -> Unix.sockaddr -> Msg.msg option) -> unit
```

used by Xfrer to register function routine that should be called when answer to IXFR request arrives by UDP.

```
val registerTcpIxfrCallback :
  (Msg.msg -> Unix.sockaddr -> Msg.msg option) -> unit
```

used by Xfrer to register function routine that should be called when answer to IXFR request arrives by TCP.

```
val registerAxfrCallback :
  (Msg.msg -> Unix.sockaddr -> Msg.msg option) -> unit
```

used by Xfrer to register function routine that should be called when answer to AXFR request arrives.

Module `Ixfr` : This module implements incremental zone transfer to client (response to IXFR question)

```
val findAnswer :
  Msg.msg -> Msg.msg -> Msg.question -> Unix.sockaddr -> Msg.msg
```

finds answer to ixfr request. The IXFR question must be the only question in given request - otherwise the IXFR question is ignored.

Module `Misc` : This module contains miscellaneous low-level functions used across the project

```
val sub2 : string -> int -> int -> string
```

substring from index to index.

```
val remove_all : ('a, 'b) Hashtbl.t -> 'a -> unit
```

removes all bindings of a given key from the given hashtable.

```
val int64_of_bytes : string -> Int64.t
```

deserialization of int64.

```
val int64_to_bytes : Int64.t -> string
```

serialization of int64.

```
val int_of_2bytes : string -> int
```

deserialization of short int.

```
exception ImplExn
val inet_addr_to_int : Unix.inet_addr -> int64
```

converts ip address to int64.

```
type  addrMatch = {
  addr : int64 ;
  mask : int64 ;
}
```

ip network address

```
val addrMatch : Unix.inet_addr -> int -> addrMatch
```

constructor of addrMatch type. The second argument is the number of bits in netmask.

Module `ModuleDispatcher` : This module is a high-level wrapper for finding answers to requests.

It dispatches requests to the proper handling modules.

```
type  connType =
  | INC
  | OUTC
```

The type of flag indicating whether the connection was initiated by us or by peer.

```
val findAnswerTcp :
  string -> int -> connType -> Unix.sockaddr -> string option
```

Finds answer to tcp request.

```
val findAnswerUdp :
  string -> int -> connType -> Unix.sockaddr -> string option
```

Finds answer to udp request.

Module `Msg` : This module defines data structures representing dns messages.

It also defines routines to serialization and deserialization of this messages.

```
type  hdr_opcode_type =
  | QUERY
  | IQUERY
  | STATUS
  | UNKNOWN_OPCODE_TYPE
```

Variant for header opcode types.

```
type  hdr = {
  id : int ;
  qr : bool ;
  opcode : hdr_opcode_type ;
  aa : bool ;
  tc : bool ;
  rd : bool ;
  ra : bool ;
  z : int ;
  rcode : int ;
  qdcount : int ;
  ancount : int ;
  nscount : int ;
  arcount : int ;
}
```

Header type.

```
type  question = {
  question_name : Rr.dName ;
  question_type : Rr.rrType ;
  question_class : Rr.rrClass ;
}
```

Question type.

```
type  msg = {
  header : hdr ;
  question : question list ;
  answer : Rr.rr list ;
  authority : Rr.rr list ;
  additional : Rr.rr list ;
}
```

Message type.

`val udpMaxSize : int`

maximum size of udp message.

`val tcpMaxSize : int`

maximum size of tcp message.

`exception DeserializationExp of hdr option`

raised when deserialization fails. If at least id of peer message was deserialized it carries the header with failure notice that should be sent to peer.

`val deserialize : string -> int -> msg`

deserializes bit stream of length given in second argument to dns message.

`val serializeTcp : msg -> string`

serializes message to be sent by tcp.

`val serializeUdp : msg -> string`

serializes message to be sent by udp.

Module `Network` : This module implements multithreaded tcp/udp server.

`val start : Unix.inet_addr -> int -> int -> unit`

starts the server on given address with given port and given number of handling threads.

Module `Notify` : This module handles notify messages by calling appropriate routine from Xfrer.

`val handle : Msg.msg -> Unix.sockaddr -> Msg.msg option`
handles the request from peer. eventualy returns the message that should be sent.

`val registerCallback : (Msg.msg -> Unix.sockaddr -> Msg.msg option) -> unit`
used by Xfrer to register function routine that should be called when request arrives.

Module `PrioQueue` : This module implements the priority queue

`type 'a queue`
The type of 'a queue.

`val init : int -> 'a queue`
creates the new empty queue with a capacity given in argument.

`val put : 'a queue -> Int64.t -> 'a -> unit`
puts a new element to queue with a given priority.

`val get : 'a queue -> Int64.t * 'a`
gets (without removing) the top element from the queue.

`val remove : 'a queue -> unit`
removes the top element from the queue.

`exception EmptyQueue`
raised when get or remove function was invoked on an empty queue.

Module `Rrdb` : This module implements internal data base of resource records and it's routines

`type zoneT`
handle for dns zone.

`type zoneCopyT`
handle for zone copy (needed for zone modifications).

`val initDB : unit -> unit`
initializes internal structures of db. Must be called before any other function.

`val destroyDB : unit -> unit`

releases internal resources. After call of this function one should not call any functions except initDB to rebuild internal structures.

```
exception ZoneAlreadyExists
```

```
exception CNameExists of Rr.rr
```

raised when No RR found, but CName found.

```
exception StarExistsRrs of Rr.rr list
```

raised when '*' found.

```
val createZone : Rr.dName -> Rr.rrClass -> unit
val createAndGetZone : Rr.dName -> Rr.rrClass -> zoneT
val getZoneName : zoneT -> Rr.dName
val getZoneClass : zoneT -> Rr.rrClass
val unRegisterReader : zoneT -> unit
val registerReader : zoneT -> unit
```

checks if zone is not locked before reading and guarantees consitents of retrived data between call of register and unregister.

```
exception FatalSyncError of string
```

thrown by sync function for example if an IO error occured. String is a description of the error.

```
val sync : zoneT -> unit
```

synchronizes zone with its persistent storage.

```
exception NoMatchingZone
```

raised by getZone function when there is no zone of given class that is the prefix of given dName.

```
val getZone : Rr.rrClass -> Rr.dName -> zoneT
```

returns zone that is the longest prefix of given dName of all supported zones.

Below functions should be used only on locked zone!

```
val getZoneCopy : zoneT -> zoneCopyT
```

returns the copy of the given zone. All modifications should be done on that copy which then replaces actual version (by calling replace).

```
val clearZone : zoneCopyT -> unit
```

removes all rrs from the zone.

```
exception ZoneMismatch
```

raised when dName given as arg to remove put get or exists function does not much to the given zone. Raised also by 'put' and 'remove' functions.

```
val put : zoneCopyT -> Rr.rr -> unit
exception NoSuchRr
```

raised by remove if there is no such rr in the given zone.

`val remove : zoneCopyT -> Rr.dName -> Rr.rrType -> Rr.rdata -> unit`

raises ZoneMismatch when end of zone is encountered.

`val replaceZone : zoneCopyT -> unit`

locks the zone, replaces current version of the zone with the given copy and unlocks the zone.

Below functions should be used only after registering reader!

`val exists : zoneCopyT -> Rr.dName -> Rr.rrType -> Rr.rdata -> bool`

`exception EndOfZone of Rr.rr list`

raised by get when end of zone was encountered during search for some node in some zone. The rr list should contain NS rrs for subzone.

`val get : zoneT -> Rr.dName -> Rr.rrType -> Rr.rr list`

`val getAll : zoneT -> Rr.rr list`

gets all but soa

Shortcuts for getZone and then long form of the function

Module Rr : This module defines data structeres representing Resoruce Records

```
type  int64 = Int64.t
type  label = string
```

type of labels in domain name.

```
type  dName = label list
```

type of domain names.

from root to leaf. ex: `"org"; "ocaml"; "www"`

`val dName_of_string : string -> dName`

conversion from string to dName.

`val dName_to_string : dName -> string`

conversion from dName to string.

```
type  rdataCname = {
  cname : dName ;
}
```

type of cname rdata.

```
type  rdataHinfo = {
  cpu : string ;
  os : string ;
}
```

type of hinfo rdata.

```
type  rdataMx = {
  preference : int ;
  exchange : dName ;
}
```

type of mx rdata.

```
type  rdataNs = {
  nsdname : dName ;
}
```

type of ns rdata.

```
type  rdataSoa = {
  mname : dName ;
  rname : dName ;
  serial : int64 ;
  refresh : int64 ;
  retry : int64 ;
  expire : int64 ;
  minimum : int64 ;
}
```

type of soa rdata.

```
type  rdataPtr = {
  ptrdname : dName ;
}
```

type of ptr rdata.

```
type  rdataTxt = {
  txt_data : string ;
}
```

type of txt rdata.

```
type  rdataA = {
  address : Unix.inet_addr ;
}
```

type of a rdata.

```
type  rdata =
  | DATA_CNAME of rdataCname
  | DATA_HINFO of rdataHinfo
  | DATA_MX of rdataMx
  | DATA_NS of rdataNs
  | DATA_SOA of rdataSoa
  | DATA_PTR of rdataPtr
  | DATA_TXT of rdataTxt
  | DATA_A of rdataA
```

```
  | DATA_UNKNOWN
type  rrType =
  | TYPE_A
  | TYPE_NS
  | TYPE_CNAME
  | TYPE_SOA
  | TYPE_PTR
  | TYPE_HINFO
  | TYPE_MX
  | TYPE_TXT
  | TYPE_AXFR
  | TYPE_IXFR
  | TYPE_MAILB
  | TYPE_MAILA
  | TYPE_UNKNOWN
  | TYPE_ALL
```

type of rr type.

```
type  rrClass =
  | CLASS_IN
  | CLASS_CS
  | CLASS_CH
  | CLASS_HS
  | CLASS_ALL
  | CLASS_UNKNOWN
```

type of rr class.

```
type  rr = {
  rrName : dName ;
  rrType : rrType ;
  rrClass : rrClass ;
  rrTtl : int64 ;
  rdata : rdata ;
}
```

and finally the type resource records.

Module `StdQuery` : This module implements the basic functionality of dns that is finding a record of a given type, name and class

```
val findAnswer : Msg.msg -> Msg.question -> Unix.sockaddr -> Msg.msg
```

finds answer to all sort of "ordinary" questions like A, MX, NS and so on.

Module `Timer` : This module implements task scheduler.

```
val start : int -> unit
```

starts the timer thread with given size of task queue.

```
val addTask : int64 -> (unit -> unit) -> unit
```

adds a task to be run at given unix time.

Module `UpdateHistory` : This module implements internal data base of zones' revision changes

```
type  rrSetT
type  zoneUpdateT = {
  oldSoaSN : Rr.int64 ;
  newSoaSN : Rr.int64 ;
  removedRrs : rrSetT ;
  addedRrs : rrSetT ;
}
val init : unit -> unit
val destroy : unit -> unit
val createZone : Rr.rrClass -> Rr.dName -> unit
exception ZoneAlreadyExists
val addUpdate : Rr.rrClass -> Rr.dName -> zoneUpdateT -> unit
```

thread safe (one writer, many readers).

```
val getCombinedUpdate :
  Rr.rrClass -> Rr.dName -> Rr.int64 -> Rr.int64 -> zoneUpdateT
```

thread safe (one writer, many readers). Returns update of zone given by dName and rrClass, from serial number of SOA given by second arg, to serial number of SOA given by the third arg, combined of all updates between these values.

```
val getUpdateList :
  Rr.rrClass ->
Rr.dName -> Rr.int64 -> Rr.int64 -> zoneUpdateT list
```

thread safe (one writer, many readers). Returns update of zone given by dName and rrClass, from serial number of SOA given by second arg, to serial number of SOA given by the third arg.

```
exception NoMatchingZone
exception NoData
```

raised by getUpdate for example if we do not know all updates between requested SOA serial numbers.

Module `Update` : This module implements dynamic updates to zones (DNS UPDATE)

```
val findAnswer :
  Msg.msg -> Msg.msg -> Msg.question -> Unix.sockaddr -> Msg.msg
```

finds answer to update request.

Module `Xfrer` : This module implements zone transfers from other server to us.

There is also a handling module InXfr that calls appropriate routines from this module.

```
val init : unit -> unit
```

puts apropriate tasks to timer and initializes handling module.

Bibliografia

[rfc1996] P. Vixie, *A Mechanism for Prompt Notification of Zone Changes (DNS NOTIFY)*, 1996

[rfc2181] R. Elz, R. Bush, *Clarifications to the DNS Specification*, 1997

[rfc1034] P. Mockapetris, *Domain names – concepts and facilities*, 1987

[rfc1035] P. Mockapetris, *Domain names – implementation and specification*, 1987

[rfc2136] P. Vixie, S. Thomson, Y. Rekhter, J. Bound, *Dynamic Updates in the Domain Name System (DNS UPDATE)*, 1997

[rfc1995] M. Ohta, *Incremental Zone Transfer in DNS*, 1996

[BIND] Oficjalna strona WWW serwera BIND, `http://cr.yp.to/djbdns.html`

[djbdns] Oficjalna strona WWW serwera djbdns, `http://www.isc.org/index.pl?/sw/bind/`

[Comer] D. E. Comer, *Sieci komputerowe i intersieci*, WNT, 2001

[OCaml] X. Leroy, *The Objective Caml system*, INRIA, 2004

[Stevens] R. Stevens, *Unix. Programowanie usług sieciowych*, WNT, 2001

[Cormen] Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, *Wprowadzenie do algorytmów*, WNT, 1998