Instrukcja do laboratorium Systemów Operacyjnych

(semestr drugi)

Ćwiczenie dziesiąte

Temat: Gniazda BSD – protokoły internetowe

Opracowanie:

mgr inż. Arkadiusz Chrobot

Wprowadzenie

1. Usługi sieciowe w Linuksie

Gniazda BSD (nazwa pochodzi od wersji BSD Uniksa, istnieją również gniazda TLI pochodzące z Systemu V, ale nie są one używane w Linuksie) stanowią API dla protokołów komunikacyjnych stosowane we wszystkich systemach operacyjnych, które umożliwiają pracę w sieci. W przypadku systemów uniksowych umożliwiają nie tylko pracę w środowisku rozproszonym, ale również lokalną komunikację między procesami stanowiąc uzupełnienie wcześniej omawianych mechanizmów. Dzięki gniazdom można pracować z protokołami należącymi do różnych dziedzin np.: Uniksa, Internetu i Xerox NS. Dodatkowo możliwa jest praca zarówno z protokołami połączeniowymi, jak i bezpołączeniowymi.

2. Różnice między protokołem TCP i UDP

W dziedzinie Internetu aplikacje sieciowe wykorzystują najczęściej jeden z dwóch najpopularniejszych protokołów warstwy transportowej: TCP lub UDP. Za pomocą pierwszego dane są wysyłane w postaci strumienia. Ten protokół jest protokołem połączeniowym, zachowuje kolejność wysyłanych komunikatów po stronie odbiorcy oraz nadzoruje przebieg transmisji dbając o retransmisję zagubionych i zniekształconych pakietów. Nie ma w nim ograniczenia na rozmiar wysyłanych danych. W niektórych zastosowaniach może się jednak okazać zbyt powolny. Można wtedy zamiast niego zastosować protokół UDP. Jest on protokołem bezpołączeniowym. Jednorazowo można za pomocą tego protokołu wysłać dane o wielkości mniejszej od 64KiB. Protokół ten nie zapewnia retransmisji danych, a przypadki zniekształcenia lub zagubienia pakietów należy obsługiwać samodzielnie. Jest on jednak zdecydowanie szybszy od protokołu TCP. Pakiety obu protokołów są „opakowywane” w komunikaty protokołu IP, stąd najczęściej w literaturze pojawiają się nazwy TCP/IP i UDP/IP.

3. Serwery iteracyjne i współbieżne

Aplikacje sieciowe można podzielić na dwie kategorie: klientów i serwery[1]. Zadaniem serwerów jest wykonywanie usług, o które proszą klienty[2]. Obsługa żądań klientów może przebiegać w sposób sekwencyjny (iteracyjny) lub współbieżny. W pierwszym przypadku serwer nawiązuje połączenie z klientem, realizuje jego prośbę, wysyła odpowiedź i wraca do oczekiwania na połączenia z innymi klientami. Podczas realizacji żądania klienta żaden inny klient nie jest w stanie połączyć się z serwerem. Serwer współbieżny po nawiązaniu połączenia z klientem tworzy proces potomny (lub nowy wątek), który obsługuje prośbę klienta, a proces macierzysty oczekuje na połączenia od innych klientów.

4. Struktury danych

Strukturę wysyłanych przez gniazdo danych, czyli protokół wyższego rzędu, osadzony na protokole transmisji określa użytkownik. Aby jednak nawiązać połączenie należy zadeklarować i wypełnić odpowiednie pola zmiennej typu *struct sockaddr_in*.

5. Opis funkcji

W tym podrozdziale opisane zostaną tylko funkcje niezbędne do wykonania większości zadań zawartych w instrukcji. Osoby, które chcą dokładniej zapoznać się z tematyką pisania oprogramowania dla sieci komputerowych powinny skorzystać z innych źródeł, jak np.: klasyczna już książka W. Richarda Stevensa „Programowanie zastosowań sieciowych w systemie Unix".

- *socket*() – funkcja ta zwraca deskryptor gniazda poprzez które będzie odbywała się komunikacja między stacjami roboczymi w sieci. Można o niej myśleć jako o funkcji open przeznaczonej dla urządzeń sieciowych. Pierwszy pobierany przez nią argument oznacza rodzinę protokołów (PF_INET dla protokołów Internetu), drugi rodzaj gniazda (połączeniowe - SOCK_STREAM, bezpołączeniowe - SOCK_DGRAM), natomiast ostatni

1 W terminologii uniksowej nazywane demonami.

2 To nie literówka. W języku polskim wprowadzono dwie odmiany liczby mnogiej wyrazu „klient", aby odróżnić ludzi od programów.

określa którego konkretnie protokołu będziemy używać (w wypadku protokołów internetowych jest zawsze równy 0). Funkcja wykorzystywana jest zarówno przez oprogramowanie serwera jak i klienta. Szczegóły: man socket.

- funkcja *bind*() – nadaje gniazdu nazwę. Zazwyczaj wywołuje ją serwer przed rozpoczęciem komunikacji z klientem, ale może jej również użyć klient celem zarezerwowania lub sprawdzenia adresu. Jako argumenty funkcja pobiera deskryptor gniazda, strukturę zawierającą adres komputera (patrz: man unix, man 7 ip), oraz rozmiar tej struktury. Szczegóły: man 2 bind.

- funkcja *connect*() – jest wykorzystywana tylko przez klienta i służy do ustanowienia połączenia z serwerem w protokole połączeniowym. Pobiera jako parametry: deskryptor gniazda, strukturę z adresem serwera i rozmiar tej struktury. Szczegóły: man connect

- funkcja *listen*() – używana jest przez serwer pracujący z protokołem połączeniowym do zgłoszenia że będzie nasłuchiwał żądań połączenia. Jeśli odbierze takie żądanie umieści je w kolejce. Funkcja listen() przyjmuje dwa argumenty: deskryptor gniazda oraz liczbę żądań, które system może umieścić w kolejce zanim zostaną one zaakceptowane (szczegóły: man tcp). Szczegóły: man listen.

- funkcja *accept*() – jest wywoływana przez oprogramowanie serwera pracującego z protokołem połączeniowym. Służy do przyjmowania połączeń. Wymaga trzech argumentów. Pierwszym jest deskryptor gniazda, drugim wskaźnik do struktury do której będzie zapisany adres klienta, a trzecim rozmiar tej struktury. Funkcja pobiera pierwsze żądanie z kolejki i tworzy dla niego gniazdo, o takich samych właściwościach jak gniazdo, do którego nadeszło żądanie. Jeśli kolejka jest pusta, to accept() blokuje się do momentu, aż pojawi się w niej jakieś żądanie. W serwerach współbieżnych gniazdo, którego deskryptor zwraca accept jest obsługiwane przez proces potomny (lub nowy wątek). Szczegóły: man 2 accept.

- Funkcje *read*(), *write*() – służą do odbierania i wysyłania danych w protokole połączeniowym. Działają one trochę inaczej niż w przypadku plików. Jeśli przez gniazdo połączeniowe są wysyłane dane o rozmiarze przekraczającym rozmiar bufora, to wprawdzie są wysyłane jako jeden strumień, ale mogą ulec segmentacji. Oznacza to, że funkcja read() może odebrać mniej danych, niż określiliśmy to w jej wywołaniu. Nie jest to

błąd, należy po prostu powtórzyć jej działanie.

- Funkcja *close*() - służy do zamykania gniazda po zakończeniu komunikacji, niezależnie od tego jakim protokołem się posługujemy.

- Funkcja *sendto*() – służy do wysyłania informacji przez gniazdo zarówno w protokole bezpołączeniowym jak i połączeniowym, choć częściej jest stosowana w tym pierwszym. Przyjmuje sześć argumentów wywołania: deskryptor, wskaźnik na bufor wysyłanych danych, rozmiar bufora, flagę (najczęściej 0), wskaźnik na strukturę w której zapisany jest adres przeznaczenia oraz rozmiar tej struktury. Zwraca liczbę przesłanych bajtów. Szczegóły: man sendto.

- Funkcja *recvfrom*() – służy do odbioru danych z gniazda zarówno w protokole bezpołączeniowym jaki i połączeniowym, choć częściej jest stosowana w tym pierwszym. Liczba i znaczenie argumentów jest podobne jak w przypadku funkcji sendto(). Różnica polega na tym, że do bufora na dane są zapisywane odebrane informacje, a w przedostatnim argumencie zapisywany jest adres strony połączenia, która te informacje nadała. Szósty argument jest wskaźnikiem na zmienną i w niej jest zapisywana wielkość odebranej struktury adresu. Funkcja ta domyślnie blokuje swoje działanie w oczekiwaniu na dane, jeśli nie zostały jeszcze wysłane.

- Funkcje zmiany porządku bajtów – porządek bajtów w sieci (big-endian) może się różnić od tego który jest używany w komputerze podłączonym do sieci. Aby sobie z tą różnica poradzić stworzono funkcje, które dokonują odpowiedniej konwersji. Przy realizacji zadań przydatne będą: *htons*() (host to network short) – do przeliczania numeru portu, *htonl*() (host to network long) do przeliczania adresu serwera, który można domyślnie określić jako INADDR_ANY oraz funkcja *inet_addr*(), która przekształca adres w notacji kropkowej (będący ciągiem bajtów) do 32-bitowego adresu internetowego. Szczegóły: man inet_addr, man htons, man htonl

- Funkcja *select()* - służy do oczekiwania na zmianę stanu pewnej liczby deskryptorów plików lub gniazd. Przyjmuje pięć argumentów: pierwszy argument jest ogólna liczbą sprawdzanych deskryptorów, trzy środkowe są trzema zbiorami deskryptorów, a ostatni jest strukturą typu *struct timeval*. Pierwszy zbiór zawiera deskryptory, badane pod względem gotowości do odczytu, drugi deskryptory badane pod względem gotowości do zapisu, a trzeci deskryptory badane na ewentualność pojawienia się wyjątków. Zbiory te obsługiwane są z użyciem makr FD_CLR, FD_SET, FD_ZERO

i FD_ISSET. Pierwsze usuwa podany deskryptor ze zbioru, drugie dodaje do zbioru, trzecie zeruje cały zbiór, a czwarte sprawdza, czy deskryptor należy do zbioru i jest wykorzystywane do sprawdzenia, czy zmienił się stan deskryptora. Ostatni parametr funkcji select() pozwala określić czas po jakim przerwane zostanie badanie gniazd, jeśli ich stan się nie zmieni. W Linuksie wartość tego pola jest dodatkowo modyfikowana, jeśli pojawi się zmiana stanu któregoś z deskryptorów. Wartość parametru określa wówczas ile czasu upłynęło od wywołania select() do pojawienia się tej zmiany. Funkcja zwraca -1 w przypadku błedu, 0 jeśli upłynął czas oczekiwania i nie pojawiło się żadne zdarzenie (żaden deskryptor nie zmienił stanu) oraz wartość większą od zera oznaczającą ile deskryptorów, spośród badanych, zmieniło stan. Bardziej rozbudowaną funkcją, zbliżoną w działaniu do select() jest *pselect*(). Pozwala ona z większą precyzją określić czas zakończenia jej działania, przy czym nigdy nie modyfikuje parametru, który określa ten czas. Dodatkowo pozwala określić, które sygnały podczas jej działania zostaną zablokowane. Szczegóły: man select, man pselect.

6. Kolejność wywołań funkcji

Serwer w protokole połączeniowym wywołuje opisane wyżej funkcje w następującej kolejności:

socket() -> bind() -> listen() -> accept() -> {read(), write()} -> close()

natomiast klient w następującej:

socket() -> connect() -> {write(), read()} -> close()

Serwer w protokole bezpołączeniowym wywołuje funkcje w następującej kolejności:

socket() -> bind() -> {recvfrom(), sendto()} -> close()

natomiast klient w następującej:

socket() -> {sendto(), recvfrom()} -> close()

Zadania

1. Napisz programy do przesyłania prostych komunikatów między dwoma komputerami z użyciem protokołu TCP/IP.

2. Napisz programy, które prześlą plik o rozmiarze większym od 1 MiB między dwoma komputerami, z użyciem protokołu TCP/IP. Sprawdź, co się stanie, jeśli plik będzie wysyłany w jednym komunikacie.

3. Napisz programy, które będą realizował polecenie zawarte w zadaniu pierwszym, ale w oparciu o protokół UDP/IP.

4. Napisz programy przesyłające plik o wielkości przekraczającej 1 MiB między dwoma komputerami, przy użyciu protokołu bezpołączeniowego.

5. Uzupełnij programy z trzeciego zadania o wykrywanie i retransmisję zagubionych pakietów. Wskazówka: można wykorzystać obsługę sygnałów, w szczególności sygnał SIGALRM.

6. Protokół UDP/IP nie gwarantuje, że komunikaty dotrą do odbiorcy w kolejności, w jakiej zostały nadane. Napisz programy, które będą same implementowały taką funkcję.

7. Stwórz serwer współbieżny, który będzie obsługiwał połączenia od wielu klientów, również napisanych przez Ciebie – mogą przesyłać np. losowe liczby do serwera, który będzie je wyświetlał na ekranie. Połączenia powinny być obsługiwane przez procesy potomne. Aby uniknąć tworzenia procesów zombie, proces macierzysty powinien ignorować sygnały o zakończeniu procesów potomnych. Użyj protokołu połączeniowego.

8. Wykonaj polecenie z zadania siódmego, używając tym razem wątków zamiast procesów.

9. Stwórz serwer iteracyjny, o takiej samej funkcji jak serwer w zadaniu siódmym.

Skorzystaj z funkcji select().

10. W praktyce dosyć często tworzy się serwery, które mają charakterystykę pośrednią między współbieżnym a iteracyjnym. Taki serwer utrzymuje pewną stałą liczbę wątków, które są odpowiedzialne za obsługę połączeń. Konieczność nawiązania nowego połączenia sprawdza przy pomocy funkcji select(). Po nawiązaniu komunikacji jej obsługę powierza pierwszemu wątkowi z puli, który nie jest zajęty obsługiwaniem innego połączenia. Napisz taki serwer i klientów, którzy będą do niego wysyłać komunikaty tekstowe, będące kolejnymi wierszami plików tekstowych.

11. Napisz programy, które będą podawały czasy przesyłania kolejnych pakietów przez sieć. Wielkość pakietu będzie określał użytkownik jako argument wywołania programu klienckiego. Użyj protokołu UDP/IP.

12. Napisz programy, które będą podawały czasy przesyłania kolejnych pakietów przez sieć. Wielkość pakietu będzie określał użytkownik jako argument wywołania programu klienckiego. Użyj protokołu TCP/IP.

13. Stwórz programy, które będą tworzyły strukturę farmer-worker. Farmer będzie rozsyłał liczby naturalne z przedziału od 2 do 302 do procesów typu worker, z których każdy będzie sprawdzał, czy otrzymana przez niego liczba jest pierwsza i odsyłał wynik do farmera. Farmer powinien współpracować z trzema procesami tego typu. Użyj protokołu bezpołączeniowego.

14. Napisz programy, o strukturze klient – serwer, które pozwolą użytkownikom wysyłać do siebie wiadomości asynchronicznie (tak jak e-mail). Serwer będzie jedynym programem działającym w trybie ciągłym. Do jego zadań będzie należało rejestrowanie nowych użytkowników, wysyłanie wiadomości do adresata, jeśli nawiąże połączenie, odbieranie wiadomości do innych użytkowników. Klient po uruchomieniu powinien wysłać serwerowi nazwę użytkownika, następnie odebrać wiadomości, które są dla niego przeznaczone i umożliwić mu wysłanie komunikatów do innych użytkowników.