Sieci TCP/IP - cz. 3
Programy sieciowe

Ostatnia modyfikacja: 08.04.2020

Model klient-serwer

- Klient-serwer: (asymetryczna) relacja pomiędzy dwoma komunikującymi się procesami, odzwierciedlająca asymetrię nawiązywania kontaktu:
  - Serwer: proces oczekujący (pasywnie) pod „dobrze znanym adresem” (WNA) na zlecenie klienta
  - Klient: proces inicjujący kontakt z serwerem w celu zlecenia zadania
- Klient jednej relacji może być serwerem innej (=>architektura wielowarstwowa)
- Proces może być serwerem więcej niż jednej usługi
- Typy komunikacji klient-serwer:
  - połączeniowa
  - bezpołączeniowa
- Serwer jednej usługi może być:
  - współbieżny
  - iteracyjny

	Komunikacja bezpołączeniowa	Komunikacja połączeniowa
Serwer iteracyjny	+	
Serwer współbieżny		+

- Jeżeli protokół aplikacji jest stanowy, to określa również po której stronie jest przechowywany stan sesji klient-serwer :
  - przez serwer (np. FTP), bądź
  - przez klienta (np. NFS) – serwer jest „bezstanowy”

Jakie są konsekwencje zawodności klienta, serwera, sieci w każdym z tych przypadków?

Model partnerski (P2P)

- Partnerzy dzielą dostęp do zasobu, pełniąc względem siebie role serwera i klienta
- Aplikacje P2P typowo wykorzystują zdecentralizowane zasoby „obszarów peryferyjnych Internetu” (n.p. komputerów PC) o niestabilnej topologii/dostępności => do funkcjonowania usług potrzeba znacznej/zupełnej decentralizacji zarządzania
- Czysty model P2P (np. Gnutella, Freenet) : bez centralnego zarządzania, partnerzy są równoprawni, usunięcie dowolnego węzła nie zmienia funkcjonalności („no single point of failure”);
- Hybrydowy model P2P (np. Napster): zawiera centralny serwer, który
  - prowadzi katalog partnerów
  - kieruje ruchem danych pomiędzy partnerami
- Zalety P2P, względem modelu klient-serwer:
  - mniejsza podatność na awarię i przeciążenie sieci („load balancing”)
  - dobra skalowalność (?)
- Wady:
  - Nie może spełniać wysokich standardów bezpieczeństwa
  - Nieprzewidywalna dostępność i jakość dostępu
  - Możliwe problemy ze spójnością danych, przechowywanych w wielu kopiach

Kryteria oceny jakości usług

- Dokładność realizacji zamawianej usługi
- Czas odpowiedzi: średni, maksymalny, nierównomierność
- Bezpieczeństwo
- Skalowalność
- Niezawodność, dostępność (HA)
- Inne, np.:
  - Otwartość rozwiązań
  - Możliwość pracy w środowiskach heterogenicznych

Realizacja współbieżnego dostępu do wielu kanałów komunikacji sieciowej

1. wejście/wyjście blokujące (domyślny tryb obsługi) +
   a) wątki robocze – lekkie, brak narzutów na lokalną komunikację/współdzielenie danych (ale mogą być narzuty na synchronizację), awaria wątku prowadzić może do awarii procesu
   b) podprocesy robocze – narzut na tworzenie i komunikację/współdzielenie danych awaria podprocesu nie wpływa w niekontrolowany sposób na inne podprocesy i proces główny.
2. wejście/wyjście zwielokrotnione (synchroniczne multipleksowanie wejścia/wyjścia) – przełączanie aktywności wątku pomiędzy deskryptorami gotowymi do obsługi (nieblokującymi). Omówiono dalej.
3. wejście/wyjście nieblokujące – programowanie zaawansowane
4. wejście/wyjście sterowane sygnałami – trudne i zawodne (*)

.

Tryb blokującego wejścia/wyjścia

Funkcje czytające (`recvfrom`, `read`, `recv`) blokują, powracając gdy:

- wystąpi błąd (przyczyna w `errno`), bądź gdy wykonanie funkcji przerwała obsługa doręczonego sygnału (funkcja zwraca wówczas –1, a `errno==EINTR`)
- w protokole datagramowym: odebrano cały datagram
- w protokole połączeniowym: odebrano co najmniej `SO_RCVLOWAT` bajtów (znacznik dolnego ograniczenia bufora odbiorczego, (`receive low-water mark` , dom.: 1); opcja `SO_RCVLOWAT` protokołu, jest dostępna przez `setsockopt()`

Uwagi:

1. Domyślnie obsługa każdego gniazda jest blokująca.
2. Wznawianie automatyczne przy przerwaniu sygnałem wymaga użycia sygnalizatora `SA_RESTART` przy zlecaniu obsługi sygnału (patrz: `man sigaction`)

Tryb blokującego wejścia/wyjścia – c.d.

Funkcje piszące (`sendto`, `write`, `send`) normalnie blokują, powracając gdy:

- wystąpi błąd (przyczyna w `errno`), bądź gdy wykonanie funkcji przerwała obsługa doręczonego sygnału (funkcja zwraca wówczas –1, a `errno==EINTR`)
- w protokole datagramowym: cały datagram został dodany do kolejki danych wyjściowych warstwy kanałowej
- w protokole połączeniowym: jądro skopiowało wszystkie dane z bufora użytkownika do bufora wysyłkowego (o długości `SO_SNDBUF`); nie wiadomo przy tym, czy druga strona dane otrzymała i pobrała

Funkcja `accept` blokuje, aż:

- pobierze połączenie z kolejki
- wystąpi błąd, bądź gdy wykonanie funkcji przerwała obsługa doręczonego sygnału

Funkcja `connect` blokuje, aż:

- zostanie ustanowione połączenie z serwerem
- wystąpi błąd, próba połączenia się przeterminuje, bądź gdy wykonanie funkcji przerwała obsługa doręczonego sygnału (w tym przypadku łączenie jest kontynuowane „w tle”). Stan połączenia można uzyskać za pomocą funkcji `select()` (deskryptor jest dostępny do zapisu); błąd można pobrać za pomocą `getsockopt()` (opcja gniazda: `SO_ERROR`). Patrz Tutorial 7.

Tryb blokującego wejścia/wyjścia– c.d.

Sygnał `SIGALRM` można wykorzystać do realizacji blokowania ograniczonego w czasie.

Przykład (tcpudpsv.c)

ustanowienie **obsługi** sygnału `SIGALRM`

włączenie budzika (5 sekund)

próba ustanowienia połączenia

Wykrycie przerwania funkcji przez doręczenie sygnału

Obsługa połączenia

```
void ALRMhand(int sig){return;/* trivial handler */
. . .
int newsock, sockfd;
static struct sigaction sa;
sigset_t zeromask;

...
   memset(&act, 0, sizeof(struct sigaction));
   act.sa_handler = ALRMhand;
   if (-1==sigaction(SIGALRM, &act, NULL)){. . . }
   sigemptyset(&zeromask);
…
   alarm(5); /* to interrupt accept()
                        if client disconnected */
   newsock=accept(clisock,. . .);
   if (newsock==-1){
            if(errno==EINTR){/* signal occured */
                        . . .  /* accept() interrupted */
            }
   } else {/* newsock is a data socket of
                        a new connection */
      . . .
   }
   . . .

}
```

Zwielokrotnione wejście/wyjście

Proces blokuje się na funkcji `select()` oczekując na gotowość jednego z wielu deskryptorów (wskazanych za pomocą masek), zamiast na funkcji wejścia/wyjścia (oczekującej na gotowość pojedynczego deskryptora.

```
#include <sys/select.h>
#include <sys/time.h>
int select(int maxfd1,     // największy numer deskryptora+1
     fd_set *rdset, // modyf. maska deskryptorów odczytu
     fd_set *wrset, // modyf. maska deskryptorów zapisu
     fd_set *exset, // modyf. maska deskryptorów z sytuacjami
                            wyjątkowymi (OOB data)
     const struct timeval *timeout // sekundy i mikrosekundy u
               // oczekiwania na gotowość (wartość modyfikowana)
   // dla NULL – możliwe nieograniczone blokowanie
); // -1: błąd lub przerwanie sygnałowe; 0 - timeout;
   // >0: liczba gotowych deskryptorów
FD_ZERO(fd_set *maska); // zeruje maskę
FD_SET(int bit, fd_set *maska); // ustawia wskazany bit maski
FD_CLR(int bit, fd_set *maska); // zeruje wskazany bit maski
FD_ISSET(int bit, fd_set *maska); // testuje wskazany bit maski
```

```
struct timeval{
  long tv_sec;
  long tv_usec;
}
```

`poll*(` jest funkcją o zastosowaniach podobnych `select()`; umożliwia ona oczekiwanie na zdarzenia związane ze wskazanymi deskryptorami (w tym gotowość do odczytu, zapisu).

Zwielokrotnione wejście/wyjście – c.d.

Przykład. Szkic wykorzystania `select` do pseudo-współbieżnej obsługi deskryptora `fd1` (odczyt) i `fd2` (zapis).

```
fd_set   rdmask,   wrmask;
int fd1=..., fd2=..., numfds;
struct timeval timeout={5,0};   /* 5 sekund oczekiwania */
while(1){
  FD_ZERO(&rdmask); FD_SET(fd1,&rdmask);
  FD_ZERO(&wrmask);  FD_SET(fd2,&wrmask);
  numfds=(fd1>fd2?fd1:fd2);
  if ((numfds=select(numfds+1,&rdmask,&wrmask,NULL,&timeout))<0){
                perror("select failed ");
                if(errno==EINTR) continue; else exit(1);
  }
  if (numfds==0)
        continue; /* przekroczony czas oczekiwania */
  if (FD_ISSET(fd1,&rdmask)) { /* obsługa deskryptora fd1 */
     ..................
  }
  if (FD_ISSET(fd2,&wrmask)) {/* obsługa deskryptora fd2 */
     ..................
  }
}/* while(1) */
```

Zwielokrotnione wejście/wyjście – c.d.

Warunki gotowości deskryptora **do odczytu**:

- w buforze odbiorczym jest co najmniej `SO_RCVLOWAT` bajtów (UDP, TCP, dom.:1)
- została zamknięta część czytająca połączenia (jest „do przeczytania” EOF)
- dla gniazda nasłuchującego jest co najmniej jedno nawiązane połączenie
- wystąpił nieobsłużony błąd dotyczący gniazda (`so_error`, dostępny przez wywołanie `getsockopt` z opcją `SO_ERROR`); wywołanie `read` ustawia `error=so_error` oraz `so_error=0`, jeśli brak danych wejściowych – w przeciwnym przypadku jest normalny odczyt, a `so_error` zachowuje wartość.

Warunki gotowości deskryptora **do zapisu**:

- w buforze wysyłkowym jest co najmniej `SO_SNDLOWAT` bajtów wolnych (UDP, TCP, typ. dom. 2048)
- istnieje nieobsłużony błąd dotyczący gniazda (`so_error`, dostępny przez wywołanie `getsockopt` z opcją `SO_ERROR`); wywołanie `write` powraca z błędem `error=so_error` oraz ustawiane jest `so_error=0`

Warunki gotowości deskryptora **do odbioru powiadomienia o danych pilnych**:

- są dane pozapasmowe (OOB, dane pilne) dla gniazda
- gniazdo pozostaje w pozycji znacznika danych pozapasmowych

Zwielokrotnione wejście/wyjście – c.d.

- Funkcja **pselect** do zwielokrotniania wejścia/wyjścia w standardzie POSIX

```
#include <sys/select.h>
#include <time.h>
int pselect(int maxfd1,  // największy numer deskryptora+1
      fd_set *rdset,  // modyf. maska deskryptorów odczytu
      fd_set *wrset,  // modyf. maska deskryptorów zapisu
      fd_set *exset,  // modyf. maska deskryptorów syt. wyj.
      const struct timespec *timeout  // sekundy i nanosekundy czasu oczekiwania
           // (wartość modyfikowana); dla timeout==NULL blokowanie nieskończone
      const sigset_t *sigmask  // maska sygnałów blokowanych w czasie czekania
);  // -1: błąd lub przerwanie sygnałowe; 0 - timeout;
    // >0: liczba gotowych deskryptorów
```

```
struct timespec{
   long tv_sec;
   long tv_nsec;
}
```

Poza różnicą w precyzji 5-go argumentu (czas oczekiwania) w porównaniu z `select`, wywołanie:

```
pselect(nfds,&rmask,&wmask,&emask,&tmout,&smask);
```

jest odpowiednikiem niepodzielnego wykonania następujących funkcji:

```
sigset_t orgmask;
sigprocmask(SIG_SETMASK, &sigmask,&orgmask);
ready=select(nfds,&rmask,&wmask,&emask,&tmout);
sigprocmask(SIG_SETMASK,&orgmask,NULL);
```

connect() + pselect()

Przykład. Nawiązywanie połączenia z zabezpieczeniem przed konsekwencjami przedwczesnego wyjścia `connect()`, spowodowanego przez asynchroniczną obsługę sygnału (patrz Tutorial 7):

```
int connect_socket(char *name){
   struct sockaddr_un addr;
   int socketfd;
   socketfd = make_socket(name,&addr);
   if(connect(socketfd,(struct sockaddr*) &addr,SUN_LEN(&addr)) < 0){
      if(errno!=EINTR) ERR("connect");
      else {
         fd_set wfds ;
         int status;
         socklen_t size = sizeof(int);
         FD_ZERO(&wfds);FD_SET(socketfd, &wfds);
         if(TEMP_FAILURE_RETRY(select(socketfd+1,NULL,&wfds,NULL,NULL))<0)
            ERR("select");
         if(getsockopt(socketfd,SOL_SOCKET,SO_ERROR,&status,&size)<0)
            ERR("getsockopt");
         if(0!=status) ERR("connect");
      }
  }
  return socketfd;
}
```

Tryb nieblokującego wejścia/wyjścia

Dwa sposoby ustawienia nieblokującego trybu obsługi deskryptora

```
// Pierwszy sposób
int flag=1;
if (ioctl(fd, FIONBIO, &flag) == -1)
{
    perror("ioctl FIONBIO");
  ..................
}
```

```
// Drugi sposób
int oldflag, newflag;
if ((oldflag=fcntl(fd, F_GETFL))<0)
  {
   perror("fcntl F_GETGL");
  ..............
}
newflag=oldflag | FNDELAY;
if (fcntl(fd, F_SETFL, newflag)<0)
  {
   perror("fcntl F_SETFL");
  ........
}
```

Przy ustawieniu deskryptora w tryb obsługi bez blokowania takie funkcje, jak

`accept, read, recv, recvfrom, send, sendto, write`

wykonują bez blokowania swoją czynność (niekoniecznie w pełni), albo kolejkują akcję i powracają z -1, ustawiając zmienną globalną `errno` na `EWOULDBLOCK` bądź `EAGAIN`. Wywołanie funkcji jest zwykle ponawiane, często po sprawdzeniu (funkcją `select()`), czy deskryptor jest już gotowy.

Konsekwencje użycia tego trybu:

- Bardzo dobre wykorzystanie aktualnej przepustowości kanałów komunikacji
- Komplikacja zarządzania buforami wejścia/wyjścia - ze względu na możliwe cząstkowe wysłania i odbiory danych oraz konieczność synchronicznego sprawdzania aktywności deskryptorów (np. funkcją **select/pselect**)

Tryb nieblokującego wejścia/wyjścia

Funkcja `connect` może rozpocząć procedurę nawiązywania połączenia i powrócić z -1, ustawiając zmienną globalną `errno` na `EINPROGRESS`. Łączenie jest kontynuowane bez potrzeby ponawiania wywołania funkcji (asynchronicznie). Nawiązanie połączenia można wykryć np. za pomocą funkcji `select` (deskryptor gotowy do zapisu), a błąd łączenia – za pomocą funkcji `getsockopt` (opcja gniazda: `SOCKERR`).

Funkcja `accept` może zablokować wątek, gdy deskryptor (w trybie blokującym) zostanie wykryty przez `select` jako gotowy do czytania, ale przed wywołaniem `accept` nastąpi zerwanie połączenia przez klienta.

Można temu zapobiec używając deskryptora w trybie nieblokującym.

Po wykryciu przez `select`, że deskryptor gotowy do czytania należy wykonać próbę wykonania `accept()` w trybie nieblokującym, jak w kodzie poniżej (Tutorial 7):

```
int add_new_client(int sfd){/* sfd: listening nonblocking mode  socket */
   int nfd;
   if((nfd=TEMP_FAILURE_RETRY(accept(sfd,NULL,NULL)))<0) {
      if(EAGAIN==errno||EWOULDBLOCK==errno) return -1;
      ERR("accept");
   }
   return nfd;
}
```

Wejście/wyjście sterowane sygnałami (*)

Można polecić jądru, aby generowało sygnał `SIGIO` przy gotowości deskryptora. Funkcja obsługi sygnału musi rozpoznać przyczynę sygnału (i deskryptor) oraz inicjować (bądź przeprowadzić) obsługę.

Przykład. Ustawienie trybu asynchronicznego obsługi deskryptora `fd`

```
//// Pierwszy sposób ////
int        flag = -getpid();
if (ioctl(fd, SIOCSPGRP, &flag) == -1)
   {
   perror("ioctl SIOCSOGRP");
   ,,,,,,,,,,,,,,,
}
flag=1;
if (ioctl(fd, FIOASYNC, &flag) == -1) {
   perror("ioctl FIOASYNC");
   ..............
}
```

```
//// Drugi sposób ////

if (fcntl(fd, F_SETOWN, getpid())<0){
   perror("fcntl F_SETOWN");
   ..................
}
if (fcntl(fd, F_SETFL, O_ASYNC)<0) {
   perror("fcntl F_SETFL O_ASYNC");
   ............
}
```

Uwagi:

- Przed ustawieniem asynchronicznego trybu obsługi gniazda należy ustanowić procedurę sygnału `SIGIO`.
- Dla gniazd TCP tryb asynchroniczny jest prawie bezużyteczny (zbyt częste generowanie sygnału i brak informacji o przyczynie)
- Domyślnie jądro pamięta wystąpienie jednego (zaległego) sygnału danego typu (=> możliwe gubienie sygnałów).

Uruchamianie serwerów

- Procesy serwerów TCP/IP zazwyczaj są działają w systemie jako demony.
- Demon – proces uruchomiony w tle, który nie podlega sterowaniu z żadnego terminala. W systemach Linux jest funkcja **daemon()**, która realizuje przejście procesu w stan „demoniczny”.
- Tradycyjne uruchamianie procesów-demonów (Unix BSD, SysV etc):
  - Za pomocą skryptów powłoki, wykonywanych przy inicjowaniu systemu (np. w `/etc/rc`, `/etc/rc.local`)
  - Przez serwer nadrzędny (super-serwer), uruchamiany przy inicjowaniu systemu, na żądanie klienta
  - Przez demon typu `cron`, uruchamiany przy inicjowaniu systemu. Zamówienia w postaci plików konfiguracji (patrz `crontab(5)`).
  - Z terminala użytkownika (do celów testowych)

Uwaga: brak terminala sterującego powoduje, że do rejestrowania komunikatów demona stosuje się funkcje biblioteki systemowej `openlog`,`syslog`, itp. , konfigurowane za pomocą pliku `/etc/syslog.conf (patrz syslog(3C),syslog.conf(4)).`

Od pewnego czasu popularność zdobywa **systemd** – patrz np. **man systemd(1), systemd.unit(5).** Oprogramowanie to realizuje m.in. funkcjonalność tradycyjnego demona **init** (PID=1) oraz część funkcjonalności tradycyjnych programów-demonów (co upraszcza pisanie demonów); umożliwia też zarządzanie procesami - demonami.