STUDIA INFORMATICA 2013
Volume 34 Number 3 (113)

Błażej ADAMCZYK, Hafed ZGHIDI
Politechnika Śląska, Instytut Informatyki

PORÓWNANIE METOD PROGRAMOWANIA GNIAZD SIECIOWYCH I ICH WYDAJNOŚCI W SYSTEMACH ISERIES ORAZ X86

Streszczenie. W ostatnich latach coraz większą popularnością cieszą się klastry i różnego rodzaju systemy komputerowe, złożone z wielu tańszych komputerów. Warto więc zastanowić się, jakie zalety zapewniają zaawansowane architektury serwerowe, takie jak na przykład IBM Power Series, w szczególności w kontekście rozproszonych usług sieciowych. W tym artykule chcemy skupić się na sposobie obsługi żądań sieciowych komputerów IBM iSeries z systemem operacyjnym i5/OS oraz porównać ich wydajność z komputerami opartymi na architekturze x86 i systemie Linux. Chcemy sprawdzić, czym różnią się techniki programowania podstawowych usług sieciowych w tych platformach oraz wpływ architektury procesora i samego systemu operacyjnego na ich wydajność.

Słowa kluczowe: IBM Power Series, iSeries, x86, obsługa żądań sieciowych, przetwarzanie równoległe, programowanie gniazd

COMPARISION OF SOCKET PROGRAMMING AND THEIR EFFICIENCY IN IBM ISERIES AND X86 COMPUTER SYSTEMS

Summary. Clusters and other complex computer systems, containing many cheaper computers, are getting very popular these days. Therefore, it is worth to analyze what advantages bring the advanced server architectures like for example IBM Power Series, especially in the distributed network services context. In this article we would like to focus on the way IBM iSeries computer with i5/OS operating system services network requests and compare its performance with standard x86 computer with Linux system. We want to summarize the differences between methods of programming the network services in both platforms and analyze the influence of processor and operating system on their performance.

Keywords: IBM Power Series, i5, x86, network request servicing, parallel execution, socket programming

1. Wstęp

Platforma IBM Power Series (wcześniej: AS/400, iSeries, i5) jest jedną z najbardziej popularnych platform serwerowych, wykorzystywanych w środowiskach produkcyjnych na świecie. Dzięki wielu specyficznym rozwiązaniom oferuje ona wysoką niezawodność, skalowalność i stabilność dostarczanych usług, zapewniając przy tym w miarę wysoką wydajność. W następnym rozdziale (2) chcielibyśmy częściowo przedstawić zalety serwerowych komputerów IBM i rozwiązania, które je odróżniają od tradycyjnych platform, takich jak x86 czy x86_64. Następnie chcemy skupić się na istotnym kontekście dostarczania usług sieciowych, ponieważ jest to najważniejsza funkcja platform serwerowych. W rozdziale 3 przedstawimy, na co należy zwrócić uwagę przy implementacji usług sieciowych dla systemów i5/OS, jakie są metody programowania tych usług i czym różnią się w porównaniu do tradycyjnych systemów Unix/Linux. Na koniec w rozdziale 4 przedstawimy wyniki eksperymentu przeprowadzonego na podobnej klasy komputerach iSeries i x86. Porównamy wpływ architektury sprzętowej i systemu operacyjnego obu tych platform na wydajność i stabilność oferowanej usługi.

2. Architektura systemu IBM i5

Komputery typu IBM i5 oferują wiele rozwiązań technologicznych odmiennych niż te, zastosowane w popularnych systemach komputerowych. Cechy charakterystyczne systemu to:

- Warstwowa architektura maszyny. Uniezależnia ona użytkownika od sprzętu umożliwiając łatwe przejście do nowszych technologii, bez konieczności ingerencji w już istniejące aplikacje.
- Pamięć jednopoziomowa (ang. *Single Level Store*). Pamięć główna i pamięć dyskowa widziane są jako ciągła przestrzeń. Dostęp do obiektu przechowywanego w systemie odbywa się poprzez mechanizm adresowania, niezależny od fizycznych urządzeń. Oznacza to, że dodatkowa pamięć główna lub dyskowa może być dodana do systemu i używana bez wpływu na istniejące programy.
- Orientacja obiektowa. Wszystkie elementy przechowywane w systemie stanowią obiekty. Uniezależnia to użytkownika od wewnętrznej struktury maszyny i pozwala na ponowne wykorzystanie istniejących elementów.
- Hierarchia mikroprocesorów. Oprócz głównego procesora systemowego, i5/OS ma dużą liczbę innych procesorów.

- System i5/OS całkowicie integruje wszystkie elementy oprogramowania (np. relacyjną bazę danych, oprogramowanie sieciowe i komunikacyjne) wymagane dla większości zastosowań komercyjnych.

W przypadku komputerów IBM i5 pełną niezależność od sprzętu uzyskano w wyniku opracowania interfejsu TIMI (ang. *Technology Independent Machine Interface*). Warstwa ta nie jest związana z żadnym rozwiązaniem sprzętowym, jest warstwą logiczną, a nie fizyczną i dostarcza programiście i jego aplikacjom sposobu widzenia różnych zasobów systemowych, oddzielając spojrzenie na nie od ich fizycznej realizacji. Sprzęt oraz część systemu, która z tym sprzętem musi się komunikować w sposób bezpośredni, są ukryte pod warstwą TIMI. W ten sposób uzyskano zupełną niezależność od fizycznej architektury procesora oraz innych, używanych rozwiązań sprzętowych.

2.1. System operacyjny i5/OS

Wśród współczesnych systemów operacyjnych wyróżnić można dwie tendencje. W pierwszym przypadku system operacyjny dostarcza jedynie podstawę, do której użytkownik musi samodzielnie dołączyć wiele niezależnych, zróżnicowanych komponentów, w celu skompletowania pożądanych funkcji. W drugim przypadku, którego doskonałym przykładem jest i5/OS, całą złożoność współczesnej technologii przetwarzania informacji zaabsorbować ma sam system operacyjny.

W komputerach IBM i5 zintegrowane zostały wszystkie elementy, niezbędne dla środowiska aplikacji komercyjnych: relacyjna baza danych, wspomaganie funkcji komunikacyjnych i sieciowych, mechanizmy bezpieczeństwa i ochrony zasobów i wiele innych. Integracja tych elementów w innych systemach wiąże się z trudnościami wynikającymi ze zróżnicowanego pochodzenia komponentów, niezgodności ich wersji, konserwacji itp. W przeciwieństwie do tej sytuacji, w przypadku i5/OS wszystkie elementy stanowią spójną całość. Użytkownik ma dostęp do wszystkich funkcji systemu poprzez jednolity, przyjazny interfejs języka komend Control Language (CL). i5/OS zapewnia również jednolite środowisko dla rozwoju aplikacji.

Najważniejsze cechy systemu operacyjnego i5/OS to:

- relacyjna baza danych DB2/400, zintegrowana zarówno z systemem operacyjnym, jak i z warstwą sprzętową systemu i5,
- zorientowanie obiektowe,
- rozbudowany system ochrony wszystkich zasobów przed niepowołanym dostępem,
- system zapewniający integralność bazy danych w przypadku awarii,
- bogaty zbiór funkcji komunikacyjnych, umożliwiających korzystanie z wielu protokołów, w tym SNA i TCP/IP,

- obsługa rozproszonego przetwarzania zgodnie ze standardem DCE w sieciach heterogenicznych bardzo wydajne mechanizmy przetwarzania typu klient/serwer,
- zintegrowane zarządzanie i konfiguracja całego systemu poprzez Operations Navigator.

Możliwość centralnego zarządzania systemem i5 zapewnia utworzenie elastycznej i funkcjonalnej struktury, z której skorzystać może każdy użytkownik systemu, niezależnie od tego jakiej wielkości zasobów potrzebuje.

2.2. Podsystemy

System i5 składa się z wielu podsystemów. Każdy podsystem odpowiada za uruchomienie określonego rodzaju zadań. Podsystemy działają niezależnie od siebie. Definiując podsystem można określić go jako pojedyncze, wstępnie zdefiniowane środowisko operacyjne, przez które system koordynuje przepływ pracy i wykorzystanie zasobów. Podsystemy zarządzają zasobami. Charakterystyki czasu działania podsystemu zdefiniowane są w obiekcie zwanym opisem podsystemu. Przykładowe podsystemy IBM i5 to podsystem QINTER, odpowiedzialny za wykonanie operacji interaktywnych, podsystem QBATCH, odpowiedzialny za realizację zadań wsadowych (batch), podsystem QPRINT, odpowiedzialny za wszystkie zadania związane z wydrukami.

2.3. Partycjonowanie logiczne

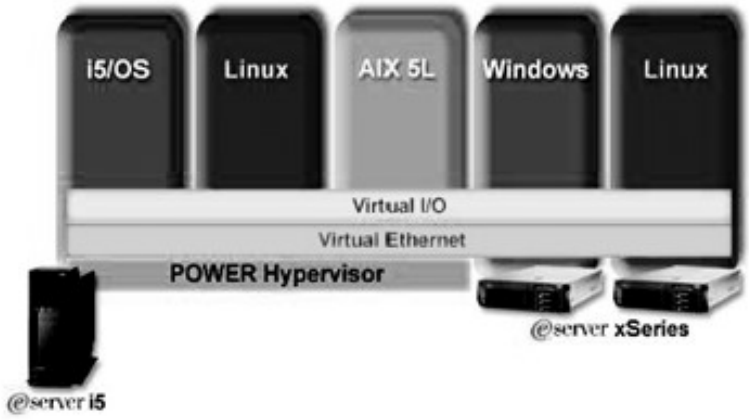
Nowoczesne systemy komputerowe pozwalają na zastosowanie partycjonowania logicznego (ang. *Logical Partitioning* LPAR). Polega to na tworzeniu w jednym systemie kilku partycji zarządzanych jednym procesorem. Każda partycja może pracować pod kontrolą innego systemu operacyjnego. System IBM i5 pozwala na zainstalowanie i zarządzanie partycjami logicznymi, pracującymi pod kontrolą następujących systemów operacyjnych: i5/OS, Linux, Windows, AIX. Do realizacji takich rozwiązań konieczne jest wykorzystanie kolejnej ważnej technologii, jaką jest wirtualizacja zasobów.

Wirtualizacja to proces prezentowania zasobów obliczeniowych w sposób ułatwiający czerpanie z nich korzyści przez aplikacje i użytkowników, zamiast w sposób wynikający z ich fizycznej konstrukcji i geograficznego umiejscowienia. Wirtualizacja zapewnia logiczny zamiast fizycznego obraz mocy obliczeniowej, przestrzeni dyskowej i przepustowości sieci.

Rys. 1. Wirtualizacja w systemach i5
Fig. 1. Virtualization in i5 systems

Ta technologia umożliwia integrację różnych systemów komputerowych w jeden. Pozwala to na zaoszczędzania wielu kosztów i niedogodności związanych z utrzymywaniem i administrowaniem kilku serwerów.

Technologia wirtualizacji procesora pozwala na utworzenie do 10 partycji na procesor. Każda partycja wymaga więc do funkcjonowania min. 0,1 procesora. Zastosowanie LPAR i wirtualizacja dają dużo korzyści. Te technologie pozwalają na lepsze wykorzystanie zasobów komputera, redukując koszty związane z użytkowaniem systemu informatycznego poprzez zwiększenie utylizacji zasobów, a także upraszczają procedury zarządzania infrastrukturą informatyczną.



Rys. 2. Wirtualizacja w systemach i5
Fig. 2. Virtualization in i5 systems

2.4. Wirtualizacja Ethernet na i5

Serwery i5 obsługują do 4094 wirtualnych segmentów sieci Ethernet. Moduł Virtual Partition Manager obsługuje do 4 wirtualnych segmentów Ethernet o szybkości transmisji 1Gbit dla każdego (w trybie pracy full duplex).

Większość systemów komputerowych zapewnia transmisję danych o maksymalnym rozmiarze ramki 1496 bajtów. W przypadku komunikacji pomiędzy partycjami, serwery i5 zapewniają transmisję danych o maksymalnym rozmiarze ramki 8996 bajtów. Zapewnia to szybszy transfer danych i lepszą wydajność systemu.

3. Usługi sieciowe w i5/OS

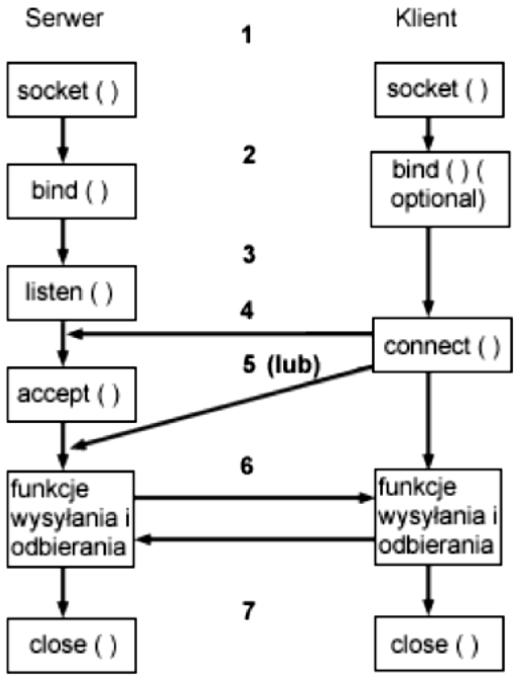
Każda aplikacja sieciowa komunikuje się z innymi komputerami wykorzystując interfejs udostępniony przez system operacyjny. Ten ostatni z kolei, używa sterowników karty sieciowej, aby nawiązać fizyczną komunikację. Aby można było łatwo programować aplikacje sieciowe działające w różnych warunkach, konieczna jest warstwa niezależna od sprzętu i systemu operacyjnego. Tak w ramach systemu Berkeley Software Distribution (BSD) powstał *de facto* standard określający taką warstwę – BSD socket API.

Berkeley sockets jest to implementacja opierająca się na systemie BSD i języku programowania C. Bardzo szybko jednak stała się standardem, który został wdrożony zarówno w innych systemach operacyjnych jak i innych językach programowania. Z czasem zbiór reguł dotyczących gniazd sieciowych ewoluował do standardów, takich jak m.in.: UNIX 98, następnie POSIX, a nawet UNIX 3 (The Single UNIX Specification, Version 3) – więcej można znaleźć w [2], [3].

System operacyjny IBM i5/OS również wspiera interfejs gniazd (w wersjach BSD 4.3 oraz BSD 4.4/UNIX 98). W następnych podrozdziałach 3.1 i 3.2 opisane zostaną podstawowe metody programowania gniazd z wykorzystaniem tych interfejsów oraz w skrócie zostaną omówione różnice pomiędzy implementacją gniazd w systemach BSD i systemie i5/OS.

3.1. Gniazda sieciowe

Tak jak już wspomniano, niezależna warstwa pozwala programistom na uruchomienie tej samej funkcjonalności w różnych środowiskach, bez konieczności głębokich zmian kodu. Samo API gniazd ze względu na swoją strukturę definiuje sposób, w jaki aplikacja sieciowa powinna funkcjonować (przykład kolejności wykonań podstawowych funkcji gniazd jest przedstawiony na rys. 3). W tym artykule skupimy się na części serwerowej, ponieważ jej praca ma kluczowy wpływ na wydajność usługi. Aplikacja serwerowa to ta, która ma za zadanie obsługiwać wiele żądań przychodzących od różnych aplikacji klienckich. Stąd też problemy z aplikacjami sieciowymi związane są zazwyczaj z aplikacją serwerową.



Rys. 3. Typowy sposób działania aplikacji wykorzystujących gniazda sieciowe. Źródło [1]
Fig. 3. Typical network sockets application construction. Source [1]

Wszystkie funkcje opisane są w specyfikacji POSIX (Portable Operating System Interface for Unix – [3]) oraz SUS (Single Unix Specification – [2]). Typowa aplikacja serwerowa wykonuje następujące czynności:

- Tworzy gniazdo sieciowe wykonując funkcję *socket()*. Ta zwraca deskryptor gniazda, który następnie jest wykorzystywany przez kolejne funkcje.
- Przypisuje gniazdu unikatową nazwę (adres IP i port w przypadku TCP i UDP) za pomocą funkcji *bind()*.
- Rozpoczyna nasłuchiwanie i sygnalizuje gotowość do akceptacji nowych połączeń za pomocą funkcji *listen()*.
- Dla kolejnych połączeń:
  - akceptuje połączenie klienta wykonując funkcję *accept()*,
  - nawiązuje faktyczną komunikację z klietem za pomocą funkcji wysyłania i odbierania danych – *send(), recv(), read(), write()* itp.,
  - po zakończeniu komunikacji następuje zakończenie połączenia funkcją *close()*.

Wymienić można trzy najbardziej popularne metody programowania gniazd sieciowych w aplikacji serwerowej: iteracyjna, wielowątkowa – *accept* w głównym wątku, wielowątkowa – *accept* w wątkach pochodnych. Różnice wynikają ze sposobu wywołania fukncji *accept*, a wybór zależy głównie od zastosowania.

Metoda iteracyjna polega na przetwarzaniu kolejnych żądań klientów sekwencyjnie jeden po drugim. Takie rozwiązanie jest proste w implementacji i może być wykorzystane w przypadku bardzo prostych aplikacji.

Jak wiadomo, obecnie serwery usług muszą pozwolić na obsługę dużej liczby klientów jednocześnie. W takim przypadku metoda iteracyjna nie sprawdzi się – należy tutaj wykorzystać wielozadaniowość systemu operacyjnego i przetwarzać żądania klientów w sposób równoległy (metody przetwarzania równoległego w skrócie omówione są w rozdziale 3.3). Aplikacja po uruchomieniu w głównym wątku wykonuje funkcje *socket(), bind(), listen()*, a następnie oczekuje na kolejne żądania od klientów za pomocą funkcji *accept*, wywoływanej w pętli. Po akceptacji połączenia główny wątek nie wykonuje faktycznej komunikacji tylko uruchamia równoległy wątek/proces i jemu zleca dalsze przetwarzanie połączenia. Główny wątek/proces może w tym czasie obsłużyć kolejne połączenia.

Wielozadaniowość można również wykorzystać wcześniej przed wykonaniem funkcji accept. W takim przypadku główny wątek/proces po wykonaniu funkcji *socket(), bind()* i *listen()* tworzy z góry zdefiniowaną liczbę wątków/procesów (zwanych *workers*), które następnie w pętli, w sposób iteracyjny przetwarzają kolejne połączenia wykonując funkcję *accept()*, a następnie całą komunikację.

3.2. Kompatybilność implementacji systemu i5/OS z BSD

Jak już wcześniej wspomniano, system i5/OS spełnia specyfikację gniazd sieciowych. Ze względu na specyfikę systemu i5/OS, istnieją jednak różnice, które należy mieć na uwadze programując aplikację sieciową, mającą działać w tych systemach. Do najistotniejszych różnic należą:

- Pliki konfiguracyjne zdefiniowane w BSD mają swoje odpowiedniki w plikach wbudowanej bazy danych systemu i5/OS:
  - QUSRSYS/QATOCHOST odpowiada plikowi /etc/hosts
  - QUSRSYS/QATOCPN odpowiada plikowi /etc/networks
  - QUSRSYS/QATOCPP odpowiada plikowi /etc/protocols
  - QUSRSYS/QATOCPS odpowiada plikowi /etc/services
- Plik /etc/resolv.conf został zastąpiony ustawieniami konfiguracyjnymi (patrz [4] i komenda *CHGTCPDMN*).
- Klienci w systemach BSD mogą wywołać funkcję *bind()* dla gniazd typu *AF_UNIX* po nawiązaniu połączenia. W systemach i5/OS nie jest to możliwe (funkcja *bind()* zwraca błąd).
- Implementacja gniazd i5/OS obsługuje flagę *SO_LINGER*, która zmienia domyślne zachowanie funkcji *close()*. Warto o tym pamiętać, jako że nie wszystkie systemy BSD obsługują tę opcję.

- W systemach BSD wykonanie funkcji *connect()* na połączonym gnieździe (w trybie bezpołączeniowym), wykorzystując niepoprawny adres lub długość adresu powoduje rozłączenie istniejącego gniazda. W implementacji i5/OS gniazdo pozostaje połączone. Poprawne rozłączenie gniazda w trybie bezpołączeniowym należy wykonać wywołując funkcję *connect()* z zerową długością adresu.
- W systemach BSD wywołanie funkcji *listen()* z parametrem *backlog* mniejszym od zera nie powoduje błędu. W systemie i5/OS podanie wartości ujemnej powoduje błąd, natomiast podanie wartości wyższej od stałej *SOMAXCONN* powoduje, że wartość *backlog* dla danego gniazda jest równa *SOMAXCONN*.

Wszystkie różnice pomiędzy implementacją gniazd w i5/OS, a w systemach BSD można znaleźć w [1]. Dokument ten opisuje również metody programowania gniazd i przedstawia przykłady aplikacji – serwera i klienta.

3.3. Uzyskanie równoległego przetwarzania

Jak już wspomniano w rozdziale 3.1, uzyskanie dużej wydajności i możliwości obsługi wielu klientów w tym samym czasie jest możliwe dzięki wielozadaniowości systemu operacyjnego. Funkcjonalność taka nie jest jednak już częścią specyfikacji gniazd i jest ściśle związana z systemem operacyjnym. Dzisiejsza technologia dysponuje maszynami wieloprocesorowymi, z których każdy procesor może mieć kilka rdzeni, dzięki czemu przetwarzanie równoległe nabiera pełnego znaczenia (wcześniej systemy uzyskiwały abstrakcję wielozadaniowości za pomocą szybkiego przełączania zadań – tzw. *task switching*). Warto więc – w szczególności w przypadku aplikacji serwerowych, mających obsługiwać dziesiątki, setki a nawet tysiące klientów – wykorzystać zalety przetwarzania równoległego.

W obecnych systemach operacyjnych najczęściej wyróżnia się dwa rodzaje wielozadaniowości: wątki i procesy. Wątki są tworzone w obrębie jednego procesu i pozwalają na dzielenie pamięci pomiędzy sobą. Nie tworzą one kolejnych instancji procesu w systemie operacyjnym, co sprawia, że zazwyczaj są szybsze. Ich wadą natomiast jest konieczność kontroli dostępu do tzw. krytycznych obszarów pamięci (ang. *critical section*). Popularną implementacją systemu wielowątkowości jest biblioteka *pthreads* (POSIX threads – [5]), która podobnie do BSD sockets, szybko stała się swego rodzaju standardem i została wdrożona w większości obecnych systemów operacyjnych. Biblioteka *pthreads* jest również wspierana przez system i5/OS i pozwala na szybkie tworzenie aplikacji wielowątkowych.

Drugim, bardzo popularnym rodzajem wielozadaniowości jest tworzenie osobnych procesów za pomocą funkcji systemu operacyjnego. Każde kolejne zadanie jest widziane w systemie jako osobny proces i w związku z tym ma swoją przestrzeń adresową i swój kod wyko-

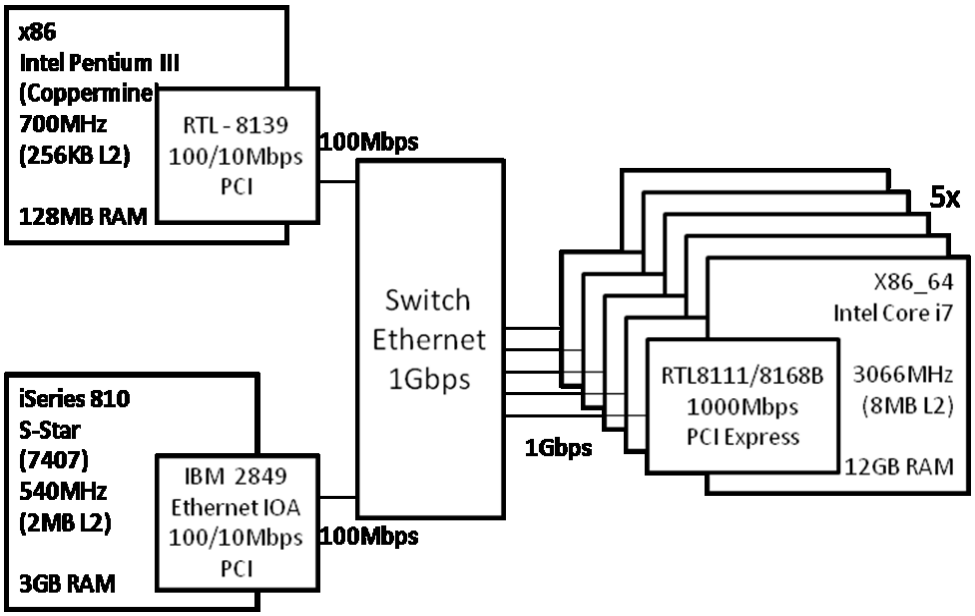
nywalny. Takie rozwiązanie pozwala na znacznie większą izolację i stabilność kosztem wydajności. Wymiana informacji pomiędzy zadaniami jest bardziej skomplikowana niż w przypadku wątków, ponieważ nie mamy do czynienia ze wspólną przestrzenią adresową i taka wymiana musi korzystać z dostępnych mechanizmów *IPC* (Inter Process Communication) systemu operacyjnego. System i5/OS nie oferuje w standardzie znanej z systemów Linux funkcji *fork()*, która pozwala na szybkie stworzenie kopii obecnego procesu i wznowienie wykonywania tuż za funkcją *fork()* (patrz [6]). Programista ma jednak do dyspozycji funkcję *spawn()* [7], która pozwala na więcej, ale jest znacznie mniej wydajna. Funkcja ta pozwala na uruchomienie dowolnego programu dostępnego w systemie z odpowiednimi parametrami. Dodatkowo, tworzonemu procesowi możemy przekazać otwarte wcześniej deskryptory plików i gniazd sieciowych, dzięki czemu proces może z nich dalej korzystać.

4. Eksperyment

Mając do dyspozycji lokalny serwer iSeries postanowiliśmy zbadać jak w praktyce wygląda obsługa gniazd w takich serwerach. Chcieliśmy zbadać jak zachowuje się implementacja gniazd w systemach i5/OS i porównać wyniki z systemem Linux w architekturze x86. W tym celu napisaliśmy aplikację serwerową z wykorzystaniem gniazd sieciowych i dostosowaliśmy ją do obu systemów. Następnie wykonaliśmy testy wielu połączeń z innych komputerów w sieci lokalnej, monitorując najbardziej istotne czasy wykonywania fragmentów aplikacji serwerowej. Dzięki temu w łatwy sposób można porównać zachowanie obu systemów zarówno w obsłudze gniazd sieciowych, jak i wielozadaniowości.

4.1. Scenariusz testowy

Na rys. 4 przedstawiono scenariusz wykonanego eksperymentu. Komputery podobnej klasy – iSeries i x86 zostały podłączone do przełącznika sieciowego. Na tych komputerach uruchomiliśmy program serwerowy pisany w *C++* z wykorzystaniem gniazd sieciowych. Program serwerowy dla każdego kolejnego akceptowanego połączenia tworzy kolejne zadanie systemu operacyjnego (wątek lub proces), który dalej obsługuje samo połączenie, natomiast zadanie główne jest odpowiedzialne tylko za akceptację kolejnych połączeń i tworzenie nowych zadań. Taka struktura pozwala na pomiar czasów w różnych momentach pracy aplikacji serwerowej i sprawdzenie czasu trwania kluczowych elementów. Dzięki temu można stwierdzić, czym różni się implementacja gniazd w obu platformach i jakie są zalety każdej z nich.



Rys. 4. Scenariusz wykonanego eksperymentu
Fig. 4. Experiment scenario

Serwer iSeries wykorzystany do eksperymentu to model 810 z procesorem 540 MHz, 2 MB L2 cache (typ 7407), natomiast komputer x86 jest podobnej klasy – procesor 700 MHz, 256 KB L2 cache (Coppermine). Oba komputery są wyposażone w podobnej klasy karty sieciowe PCI o szybkości łącza 100 Mbps. Niestety ze względu na brak bezpośredniego dostępu do nowszego komputera iSeries eksperyment musiał zostać przeprowadzony na starszych maszynach.

Na komputerze iSeries uruchomiony był system operacyjny i5/OS w wersji V5R4. Warto tu jednak zaznaczyć, że w momencie eksperymentu komputer iSeries miał uruchomione również inne usługi (jak na przykład serwer aplikacji, ftp itp.) oraz partycje logiczne: Susi Linux, Windows 2003 Serwer i druga instancja i5/OS na dodatkowej macierzy dyskowej. Na komputerze x86 uruchomiony został system Linux Gentoo 2.6.38-r6 zupełnie nieobciążony, bez jakichkolwiek innych uruchomionych usług czy procesów.

Eksperyment miał na celu zbadanie, w jaki sposób każda z platform przetwarza dużą liczbę przychodzących połączeń. Każdy z pięciu komputerów „klientów” (patrz rys. 4 po prawej) generował po 200 wątków, z których każdy wysyłał żądanie połączenia TCP z serwerem, co daje łącznie 1000 prawie jednoczesnych prób nawiązania połączenia na warstwie TCP/IP. Należy tutaj jasno zaznaczyć, że liczba równocześnie otwartych połączeń może być znacznie większa w obu systemach, natomiast liczba równoległych żądań nowych połączeń może sprawiać problem dla aplikacji serwerowej z powodu potrzeby tworzenia kolejnych wątków dla każdego nowego połączenia. Stąd też tak wiele ataków sieciowych typu Denial of Service, które bazują na nawiązywaniu nowych połączeń TCP (patrz [8]).

Po nawiązaniu połączenia wysyłana została do serwera informacja o długości 1000 B, a następnie serwer po poprawnym odebraniu wysyła odpowiedź potwierdzającą poprawne odebranie danych. Rozmiar komunikatu nie ma większego znaczenia na wyniki eksperymentu, ponieważ najbardziej interesujący jest sam proces obsługi połączenia, a nie transmisja danych. Po tej krótkiej wymianie komunikatów połączenie jest zamykane i wątek/proces kończy pracę. Aplikacja uruchomiona na serwerze cały czas monitoruje następujące parametry:

- **Czas wykonywania samej funkcji *accept()*** – główny wątek/proces po zaakceptowaniu nowego połączenia tworzy nowe zadanie do jego obsługi, a następnie ponownie wywołuje funkcję *accept*, oczekując na kolejne żądanie połączenia. Ten pomiar pozwala jasno stwierdzić jak szybko system operacyjny i stos protokołów sieciowych przetwarzają kolejne żądania połączenia. Wiedząc, że bufor połączeń oczekujących nie jest pusty można zmierzyć czas, jaki jest potrzebny zanim aplikacja otrzyma informację o nowym żądaniu.
- **Czas wykonywania głównej pętli programu** – czas pomiędzy kolejnymi wywołaniami funkcji *accept*. Ten pomiar pozwala określić, z jaką częstotliwością główny wątek/proces aplikacji jest w stanie przyjmować kolejne połączenia.
- **Czas tworzenia nowego wątku/procesu** – czas wykonywania funkcji *pthread_create()* bądź *fork()/spawn()*. Ten pomiar wraz z następnym pozwalają na określenie jak działa implementacja wielozadaniowości testowanego systemu operacyjnego i jak długo system tworzy nowe wątki/procesy.
- **Czas tworzenia nowego wątku/procesu** – czas pomiędzy żądaniem a faktycznym uruchomieniem nowego wątku/procesu.
- **Czas transmisji danych** – czas, w jakim nowy wątek/proces odbiera i wysyła dane.
- **Pełny czas obsługi połączenia** – czas od wywołania funkcji *accept* do zakończenia transmisji. Ten pomiar pozwala na sprawdzenie całościowej wydajności aplikacji serwerowej na danej platformie.

Wszystkie pomiary czasu wykorzystywały tę samą funkcję języka C++ *gettimeofday* dostępną na obu platformach, pozwalającą na monitorowanie czasu ze stosunkowo wysoką dokładnością (około 10 μs – przy wyłączonych mechanizmach aktualizacji czasu, takich jak ntpd).

Eksperyment był powtarzany dwukrotnie dla każdej z omawianych platform – najpierw testowany był wariant aplikacji serwerowej wykorzystujący wątki (bibliotekę *pthreads*) dla obsługi połączeń, w drugim przypadku wielozadaniowość osiągana była przez tworzenie kolejnych procesów systemu operacyjnego (funkcja *spawn()* w systemie i5/OS oraz funkcja *fork()* w systemie Linux).

4.2. Ograniczenia platform i parametry

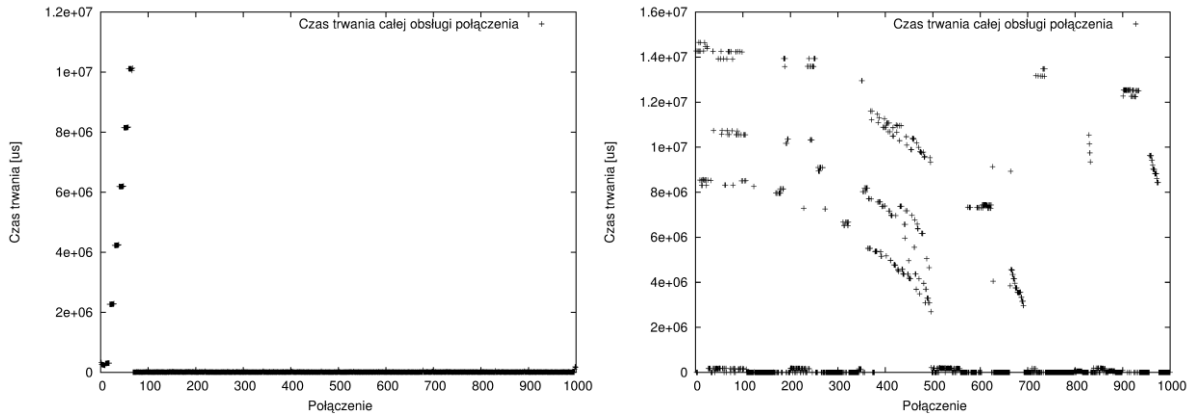
Głównym celem eksperymentu było porównanie wydajności każdej z platform i sposobu przetwarzania nowych połączeń TCP przez system operacyjny. Specyfikacja gniazd określa, że wszystkie przychodzące żądania nowego połączenia powinny być buforowane do czasu zaakceptowania ich (funkcją *accept()*) przez aplikację. Rozmiar tego buforu jest określany przez parametr *BACKLOG* funkcji *listen()*, natomiast jego maksymalna wartość przez zmienną systemową *SOMAXCONN*. W systemie i5/OS zmienna ta nie może zostać zmieniona przez użytkownika i jej domyślna wartość to 512 połączeń. W systemach Linux zmienną *SOMAXCONN* można modyfikować poprzez narzędzie *sysctl*, a domyślna wartość w testowanym systemie to 128. W celu porównania obu systemów, w obu platformach ustawiliśmy rozmiar bufora aplikacji serwerowej na 512. Częstotliwość połączeń przychodzących jest znacznie większa niż szybkość obsługi tych połączeń w obu platformach, jako że żądania połączeń generowane są na pięciu osobnych, dużo nowszych komputerach. W związku z tym, kolejka nowych połączeń wypełnia się od samego początku eksperymentu, a mierzone czasy odpowiadają czasom, w jakim sam system operacyjny przetwarza kolejne połączenia, a nie oczekiwaniu na nadejście nowych żądań. Warunek przepełniającego się bufora *BACKLOG* jest konieczny, aby można było poprawnie porównywać otrzymane wyniki obu platform.

Aby móc w pełni przeprowadzić eksperyment z wykorzystaniem wątków na platformie x86 w systemie Linux, konieczna była modyfikacja jeszcze jednej zmiennej systemowej - rozmiaru stosu dla nowych wątków. W przypadku tworzenia bardzo dużej liczby nowych wątków, system operacyjny musi dla każdego z nich zarezerwować miejsce dla stosu we wspólnej przestrzeni adresowej, której w końcu może zabraknąć. W takim przypadku system zwraca błąd. Można jednak zmienić domyślny rozmiar stosu dla nowych wątków tak, by można ich było stworzyć znacznie więcej w obrębie jednego procesu za pomocą polecenia *ulimit -s*. Domyślnie rozmiar stosu wynosi 8192KB. W przestrzeni adresowej systemu 32-bitowego można zarezerwować tylko około 524 takich bloków pamięci, a co za tym idzie w przypadku więcej niż 524 wątków aplikacja zakończy działanie błędem.

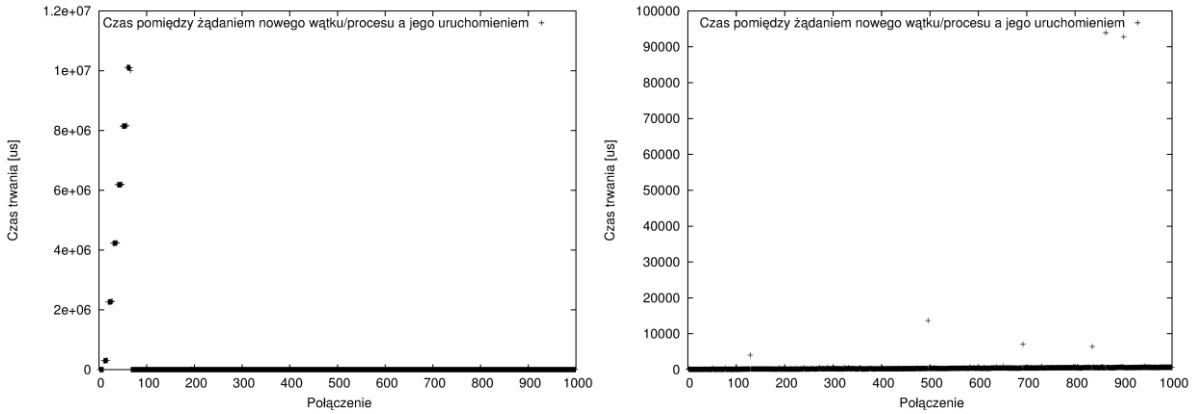
Poza powyższym parametrem, pośrednio wpływającym na maksymalną liczbę wątków, występuje również osobny parametr, bezpośrednio definiujący to maksimum, a jest to zmienna *threads-max* (można ją zmienić narzędziem konfiguracji systemowej *sysctl*).

W systemach i5/OS z kolei jest parametr, który definiuje maksymalną liczbę procesów w obrębie jednego podsystemu. Parametr ten dla podsystemu *QBATCH* (w którym była uruchamiana aplikacja serwerowa) był ustawiony na wartość **NOMAX*, co oznacza brak maksimum. Należy mieć to jednak na uwadze uruchamiając aplikację tworzącą nowe podprocesy w systemach i5/OS. Parametr ten można zmienić za pomocą polecenia *CHGSBSD*.

4.3. Wyniki – wątki



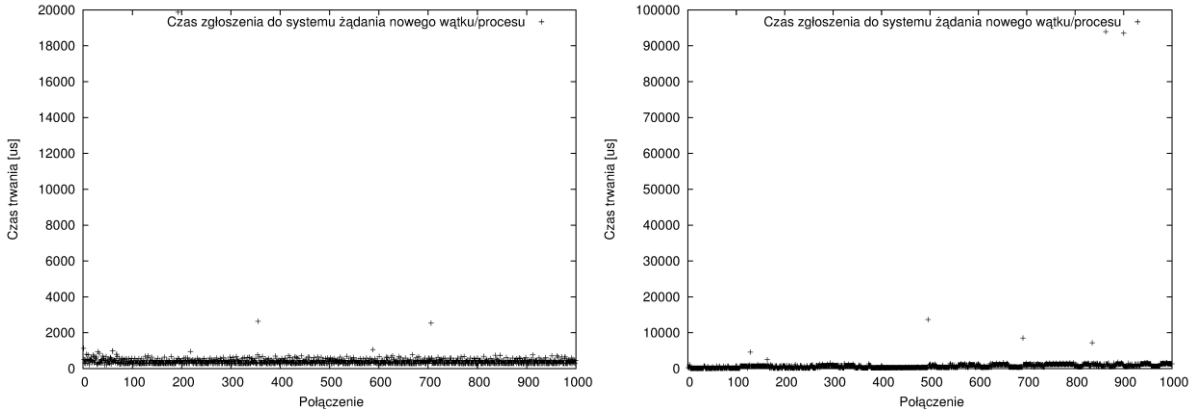
Rys. 5. Wykresy łącznego czasu obsługi połączenia. Z lewej system iSeries, z prawej system x86
Fig. 5. Whole connection handling time. Left side – iSeries, right side – x86



Rys. 6. Wykresy czasu pomiędzy żądaniem nowego wątku a jego uruchomieniem. Z lewej system iSeries, z prawej system x86
Fig. 6. Time between new thread request and its actual execution. Left side – iSeries, right side – x86

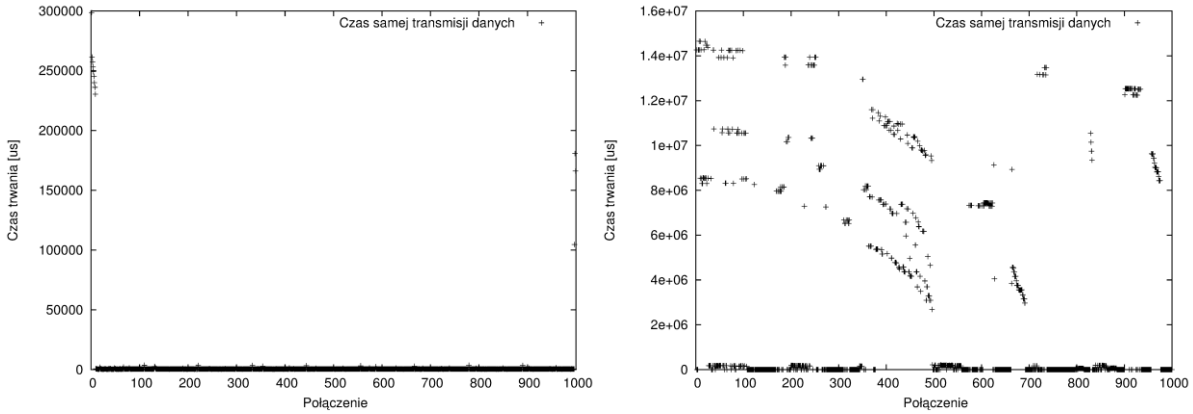
Na rys. 5-10 przedstawiono porównanie wyników eksperymentu dla wielowątkowej aplikacji serwerowej. Na rys. 5 przedstawiono łączny czas trwania obsługi kolejnych połączeń. Widać na nim wyraźnie, że aplikacja uruchomiona na serwerze iSeries działa szybciej i znacznie bardziej stabilnie. Jest jednak pewien specyficzny efekt, który pokazuje, że tworzenie kolejnych wątków w systemie i5/OS na początku nie działa najbardziej wydajnie. Na rys. 6 wyraźnie widać, że tworzenie nowych wątków zajmuje stopniowo coraz więcej czasu. Po stworzeniu określonej liczby wątków efekt ten znika i kolejne wątki tworzone są bardzo szybko. Przyczyną tego efektu może być implementacja mechanizmu puli wątków systemu oraz mechanizmy cache systemu operacyjnego. Problem ten można rozwiązać stosując aplikację serwerową, która na początku inicjuje określoną pulę wątków, a następnie wątki te przetwarzają kolejne żądania. Taka aplikacja powinna również działać lepiej w systemach

x86, natomiast dla porównania obu platform lepiej było wykorzystać wariant bez puli – z dynamicznym tworzeniem wątków/procesów.



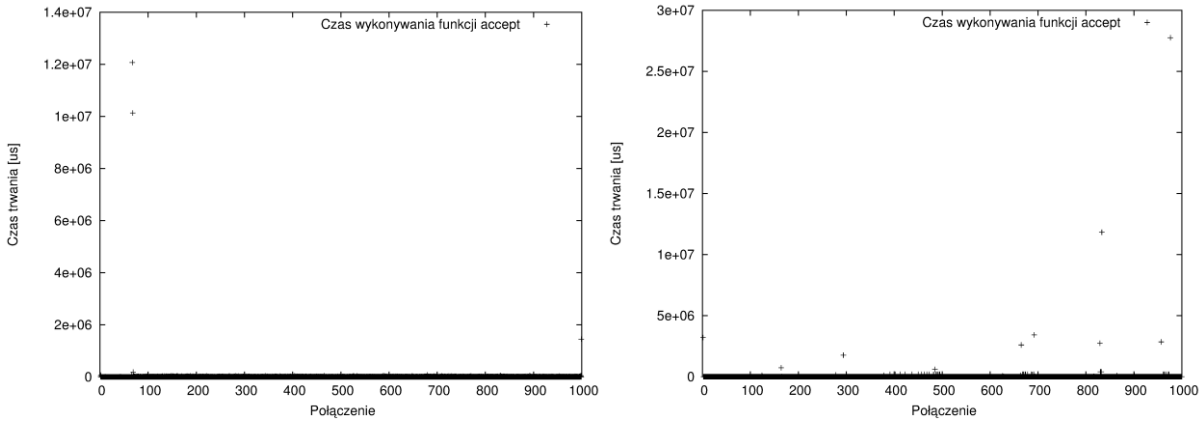
Rys. 7. Wykresy czasu zgłaszania żądania nowego wątku. Z lewej system iSeries, z prawej system x86
Fig. 7. Time of sending request for new thread. Left side – iSeries, right side – x86

Rysunek 7 przedstawia czas wykonania funkcji *pthread_create()* w obu systemach. Jak widać na ogół żądanie nowego wątku w systemie Linux działa znacznie szybciej, jednak czasami zdarzają się znaczne opóźnienia (w tym przypadku nawet rzędu 100000 μs). W systemie i5/OS tworzenie nowego wątku średnio zajmuje więcej czasu, natomiast jest znacznie bardziej stabilne (patrz tabela 1).

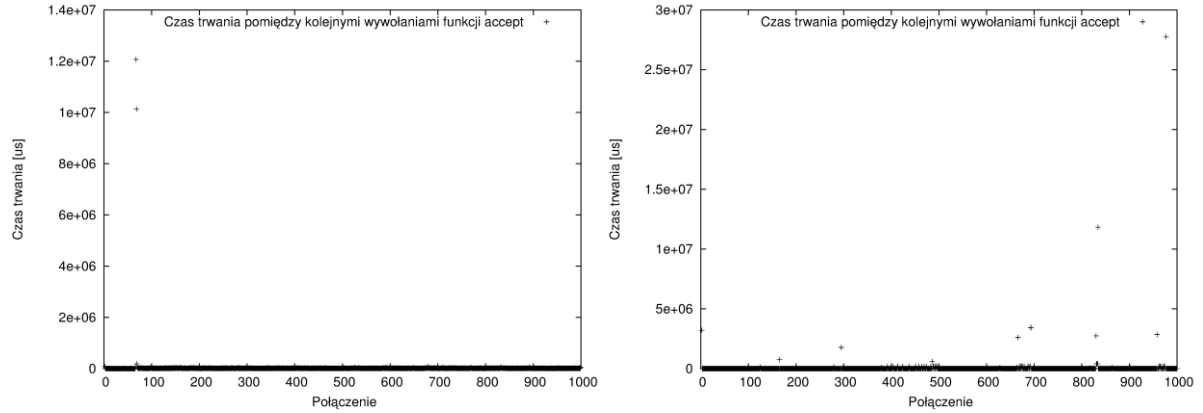


Rys. 8. Wykresy czasu transmisji danych. Z lewej system iSeries, z prawej system x86
Fig. 8. Transmission time. Left side – iSeries, right side – x86

Na rys. 8 widać, że system iSeries równomiernie uruchamia wątki i przetwarza kolejne połączenia. Z kolei, system Linux na platformie x86 często opóźnia zakończenie obsługi połączenia. Przyczyną takiego zachowania może być obsługa innych wątków przez procesor, co powoduje, że dany wątek jest długo nieobsługiwany. Dodatkowo, stos protokołów systemu operacyjnego może mieć znaczący wpływ na szybkość przetwarzania połączeń. Jak widać w tabeli 1, statystyczny czas obsługi połączenia w przypadku serwera i5 jest ponad 10-krotnie niższy niż dla komputera x86.



Rys. 9. Wykresy czasu wykonywania funkcji accept. Z lewej system iSeries, z prawej system x86
Fig. 9. Accept function execution time. Left side – iSeries, right side – x86



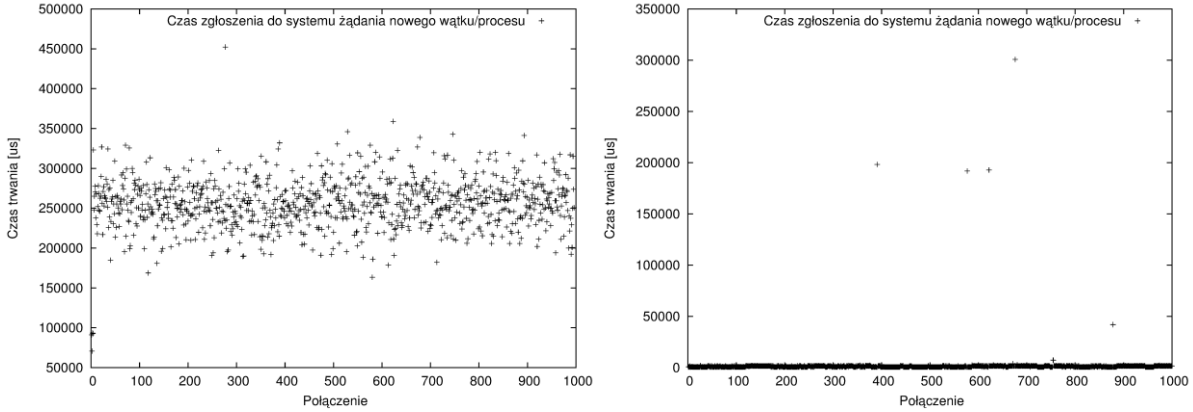
Rys. 10. Wykresy czasu trwania głównej pętli programu. Z lewej system iSeries, z prawej system x86
Fig. 10. Main loop execution time. Left side – iSeries, right side – x86

Na uwagę tutaj zasługuje również fakt, że system iSeries był znacznie bardziej obciążony w momencie testu. Procesory obu maszyn są porównywalnej klasy, natomiast iSeries ma znacząco więcej pamięci RAM. Mogłoby to mieć znaczenie w przypadku transmisji dużej ilości danych. W tym eksperymencie z premedytacją jednak w ramach każdego połączenia przesyłanych jest tylko 1000 B, co oznacza, że nawet gdyby wszystkie 1000 połączeń było obsługiwanych w tym samym momencie, to łączny rozmiar zajmowanej pamięci operacyjnej byłby nieco większy niż 1 MB, co wciąż jest wartością bardzo małą, nawet w przypadku komputera x86, gdzie zainstalowano 128 MB pamięci RAM. W momencie uruchomienia aplikacji serwerowej na platformie x86 prawie połowa dostępnej pamięci RAM była wolna – co potwierdza, że ograniczona pamięć nie powinna mieć wpływu na wynik eksperymentu.

4.4. Wyniki – procesy

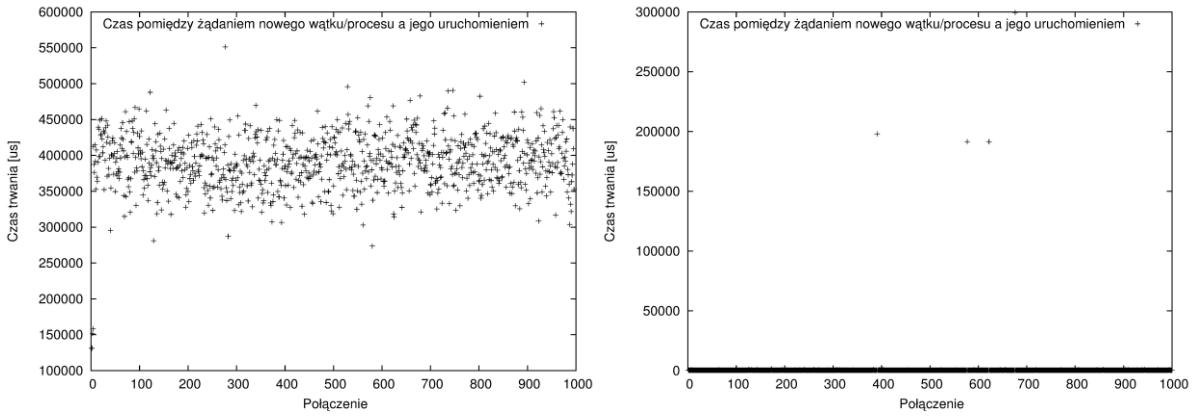
Tworzenie nowych procesów w przypadku systemu iSeries jest uzyskane z wykorzystaniem funkcji *spawn()*, która nie działa równie wydajnie jak funkcja *fork()*, dostępna w systemach Linux. Jak widać na rys. 11, 12 w przypadku procesów wydajność systemu i5/OS dra-

stycznie spada. Zaskakująco, bardzo długi czas zgłaszania żądania nowego procesu ma również negatywny wpływ na pobieranie kolejnego żądania połączenia w głównym procesie – co w teorii powinno skutkować przyspieszeniem działania funkcji *accept()*. Oczywiście wydaje się, że przyczyną takiego zachowania jest funkcjonalność cache systemu operacyjnego i tak długie odstępy pomiędzy kolejnymi wywołaniami funkcji *accept()* powodują efekt zupełnie odwrotny do oczekiwanego.



Rys. 11. Wykresy czasu zgłaszania żądania nowego wątku. Z lewej system iSeries, z prawej system x86

Fig. 11. Time of sending request for new process. Left side – iSeries, right side – x86



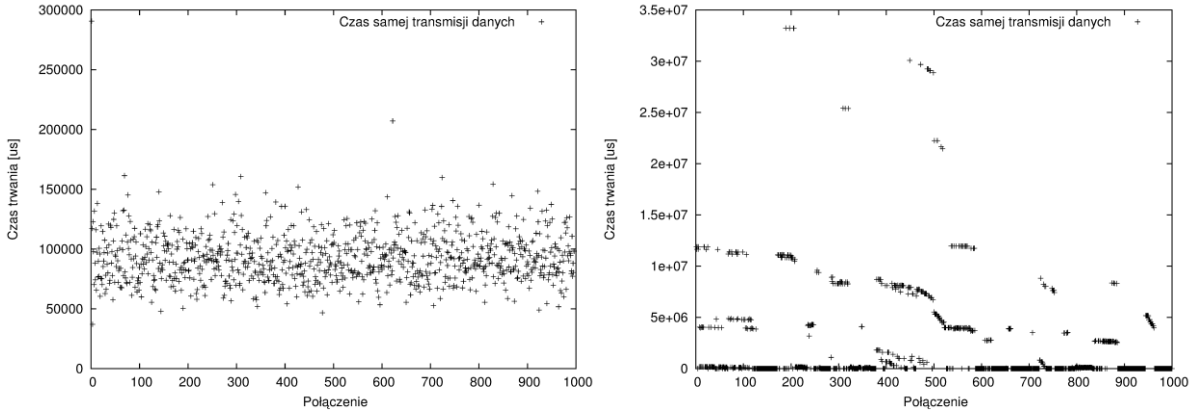
Rys. 12. Wykresy czasu pomiędzy żądaniem nowego wątku a jego uruchomieniem. Z lewej system iSeries, z prawej system x86

Fig. 12. Time between new process request and its actual execution. Left side – iSeries, right side – x86

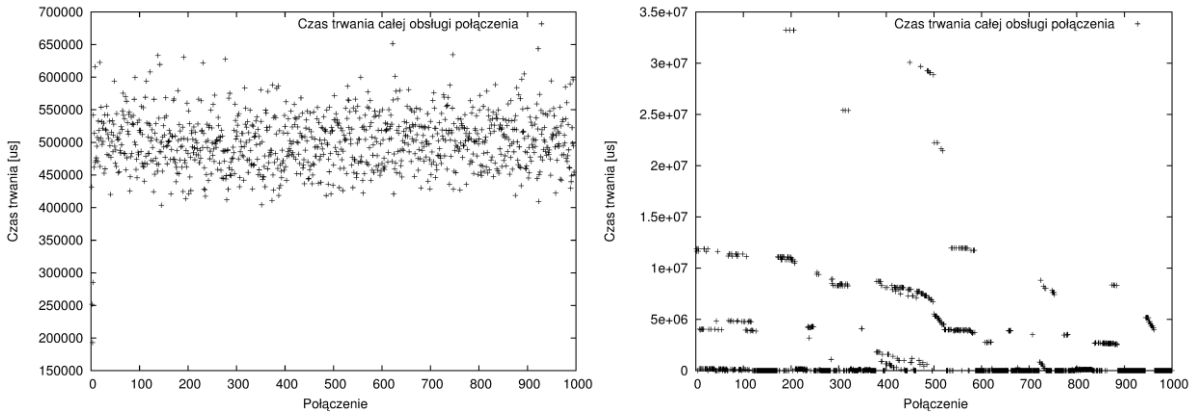
Analizując problem jeszcze dokładniej można zauważyć, że kilka pierwszych procesów jest tworzonych znacznie szybciej niż wszystkie następujące. Świadczy to o tym, że system operacyjny mało optymalnie zarządza mechanizmem tworzenia nowych procesów. Wpływ na tak złe zachowanie mogą mieć też inne procesy, działające na serwerze podczas przeprowadzania eksperymentu.

Dodatkowo, ciekawy jest fakt (co widać na rys. 13 i 14), że sama transmisja danych – mimo iż jest niższa od transmisji dla platformy x86, jest dłuższa niż w przypadku użycia

wątków. Przyczyną tego może być dłuższy odstęp pomiędzy akceptacją połączenia a faktyczną transmisją danych, co może mieć wpływ na szybkość odpowiedzi aplikacji klienckiej, ale równie możliwą przyczyną może być fakt, że system i5/OS znacznie słabiej zarządza przełączaniem procesów niż wątków. Efekt zupełnie odwrotny można zaobserwować w przypadku platformy x86 – tam transmisja danych jest szybsza w przypadku procesów.

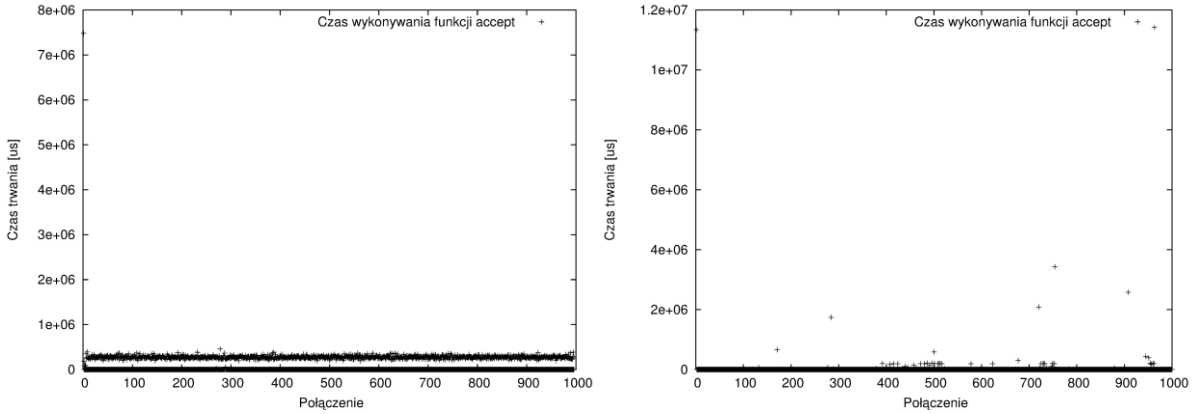


Rys. 13. Wykresy czasu transmisji danych. Z lewej system iSeries, z prawej system x86
Fig. 13. Transmission time. Left side – iSeries, right side – x86

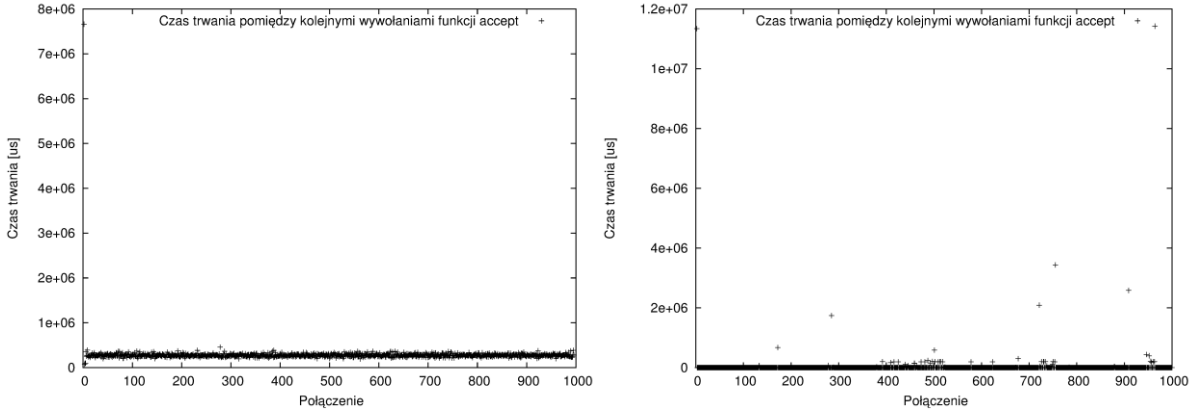


Rys. 14. Wykresy łącznego czasu obsługi połączenia. Z lewej system iSeries, z prawej system x86
Fig. 14. Whole connection handling time. Left side – iSeries, right side – x86

Na rys. 15 i 16 przedstawione zostały wykresy czasu wykonywania funkcji *accept()* oraz czasu trwania pętli wątku głównego i są one odpowiednikami wykresów przedstawionych na rys. 9 oraz 10. Kolejny ciekawy efekt widoczny jest na rys. 15. Ze szczegółowej analizy wyników (co widać również na wykresie) można zaobserwować, że dokładnie co drugie wywołanie funkcji *accept*(), trwa dłużej w przypadku systemu iSeries. Efekt ten powstaje tylko w implementacji dla wielu procesów, gdzie występuje wspomniany wcześniej problem. Można więc wnioskować, że oba problemy są związane.



Rys. 15. Wykresy czasu wykonywania funkcji accept. Z lewej system iSeries, z prawej system x86
Fig. 15. Accept function execution time. Left side – iSeries, right side – x86



Rys. 16. Wykresy czasu trwania głównej pętli programu. Z lewej system iSeries, z prawej system x86
Fig. 16. Main loop execution time. Left side – iSeries, right side – x86

W przypadku obu systemów operacyjnych wielozadaniowość, osiągnięta za pomocą tworzenia kolejnych procesów, jest mniej wydajna. W przypadku iSeries efekt ten jest wyjątkowo silny. Mimo to, w przypadku systemu x86 średni czas żądania stworzenia nowego procesu (patrz tabela 1) jest ponad 20-krotnie większy, a średni czas pomiędzy żądaniem a stworzeniem nowego procesu około 2-krotnie dłuższy. Ten efekt jest jednak usprawiedliwiony faktem, że system operacyjny musi wykonać całą inicjalizację procesu wraz z alokacją pamięci oraz zduplikowaniem referencji do otwartych deskryptorów. To wszystko powoduje, że wielowątkowość jest znacznie bardziej wydajnym rozwiązaniem, niezależnie od platformy.

Obsługiwanie nowych połączeń poprzez procesy nie jest więc dobrym rozwiązaniem. Ogólnie wielozadaniowość oparta na procesach jest przydatna w sytuacji, kiedy wymagana jest duża izolacja pomiędzy zadaniami. W przypadku połączeń sieciowych koszt związany ze stworzeniem nowego procesu może być zbyt duży.

Jeżeli jednak wykorzystanie wielowątkowości nie jest możliwe, zarówno w przypadku systemu iSeries, jak i x86 problem może być rozwiązany poprzez stworzenie puli działających już procesów. Jak to już zostało przedstawione wcześniej, takie rozwiązanie może

znacznie poprawić działanie aplikacji. Jednym z celów tego eksperymentu jest jednak porównanie wydajności dynamicznego tworzenia nowych procesów/wątków, ponieważ cześć aplikacji już istniejących korzysta właśnie z takiego rozwiązania.

Tabela 1

Wyniki eksperymentu. Średni czas ± przedział ufności dla α= 0,95

Pomiar	Pthreads x86	Pthreads i5	Fork x86	Spawn i5
Czas wykonywania funkcji accept	33,237 ms ± 30,371 ms	18,946 ms ± 15,524 ms	21,233 ms ± 16,598 ms	141,027 ms ± 9,469 ms
Czas trwania głównej pętli programu	66,508 ms ± 60,838 ms	36,455 ms ± 30,953 ms	42,488 ms ± 33,234 ms	282,196 ms ± 14,722 ms
Czas żądania nowego wątku / procesu	99,802 µs ± 261,68 µs	418,603 µs ± 39,533 µs	2,071 ms ± 884,947 µs	255,974 ms ± 1,94 ms
Czas pomiędzy żądaniem a uruchomieniem wątku / procesu	567,395 µs ± 260,324 µs	293,331 ms ± 88,117 ms	1,28 ms ± 880,283 µs	390,654 ms ± 2,401 ms
Czas transmisji danych	3,257 s ± 288,892 ms	3,283 ms ± 1,57 ms	2,682 s ± 307,777 ms	93,017 ms ± 1,291 ms
Czas łącznej obsługi połączenia	3,258 s ± 288,88 ms	314,302 ms ± 90,149 ms	2,684 s ± 307,766 ms	500,926 ms ± 2,65 ms

5. Podsumowanie

Artykuł miał na celu przedstawienie metod programowania gniazd sieciowych na różnych platformach sprzętowych i w różnych systemach operacyjnych. Przedstawia różnice pomiędzy implementacją gniazd w systemach i5/OS i systemach Linux. Wydaje się, że niezależnie od platformy dla przetwarzania wielozadaniowego warto korzystać ze stworzonej wcześniej puli wątków/procesów. Dodatkowo widać, że zarówno w systemach iSeries, jak i x86 korzystanie z wielowątkowości może okazać się bardziej wydajne.

Eksperyment wykazał jednoznacznie, że rzeczywiście komputer iSeries z wykorzystaniem wielowątkowości lepiej radzi sobie z obsługą wielu nowych połączeń, pomimo innych procesów działających w tle. Dodatkowo, system iSeries często zachowuje się bardziej stabilnie i przewidywalnie od systemu Linux na komputerze x86. Z kolei, implementacja aplikacji serwera z wielozadaniowością opierającą się na procesach okazała się działać zaskakująco mało wydajnie na serwerze iSeries i w tym przypadku lepiej zachowuje się platforma x86. Mogą mieć na to wpływ różne czynniki np.: wykorzystanie funkcji *spawn()* w miejsce brakującej funkcji *fork()*, inne procesy działające w systemie, nieoptymalne zarządzanie tworzeniem nowych procesów w systemach i5/OS.

Podsumowując można stwierdzić, że standaryzacja interfejsu gniazd sieciowych jest bardzo istotnym elementem obecnie działających systemów operacyjnych. Programiści mogą w stosunkowo łatwy sposób tworzyć oprogramowanie, działające na różnych platformach

i w różnych systemach operacyjnych. Warto jednak pamiętać o różnicach implementacyjnych i specyficznych parametrach tych platform. Należy również pamiętać, że wielozadaniowość nie jest ujęta w standardzie gniazd sieciowych, a większość aplikacji serwerowych ściśle od niej zależy. Jak widać na przykładzie platformy iSeries różne metody równoległego przetwarzania mogą mieć drastyczny wpływ na wydajność aplikacji.

BIBLIOGRAFIA

1. International Business Machines Corporation: iSeries Socket Programming, IBM Corp., USA 2002.
2. The Open Group: The {Single UNIX Specification} version 3, 2007. http://www.unix.org/version3/ 24.12.2012.
3. The Open Group: Base Specifications Issue 7, IEEE Std 1003.1-2008; 2008. http://pubs.opengroup.org/onlinepubs/9699919799/ 24.12.2012.
4. International Business Machines Corporaton: System i: Networking TCP/IP setup, IBM Corp., USA 2010.
5. The Open Group: Base Specifications Issue 6, Threads, IEEE Std 1003.1-2004 (POSIX: 2004). http://www.opengroup.org/onlinepubs/000095399/basedefs/pthread.h.html 24.12.2012.
6. The Open Group: Base Specifications Issue 6, Unistd – fork, IEEE Std 1003.1-2004 (POSIX:2004). http://pubs.opengroup.org/onlinepubs/009695399/functions/fork.html 24.12.2012.
7. International Business Machines Corporation: Centrum Informacyjne iSeries, Wersja 5 Wydanie 4; IBM Corp. http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/apis /spawn.htm; 24.12.2012.
8. Schuba C. L., Krsul I. V., Kuhn M. G., Spafford E. H., Sundaram A., Zamboni D.: Analysis of a denial of service attack on TCP; Proceedings of IEEE Symposium on Security and Privacy, 1997, pp. 208-223.

Wpłynęło do Redakcji 13 lutego 2013 r.

Abstract

This article presents the available methods for programming network sockets in different platforms and operating systems. It highlights the differences between the socket implementa-

tion in i5/OS and Linux operating systems. It appears, that for both platforms it is better to prepare a pool of worker threads/processes to service network request in the future because otherwise the dynamic creation of those may cause a performance penalty. Moreover, it seems that the use of threads can be more efficient for both x86 and iSeries platforms.

The experiment showed, that the iSeries platform using threads works better handling many simultaneous connection requests. Additionally, the iSeries system often performed much more stable and predictable than x86 architecture with Linux OS. On the other hand, the process based multitasking i5/OS server implementation seems to be very inefficient and in this case (multithreading using processes) the x86 platform shown better performance. This may be caused by several factors e.g. the use of *spawn()* function instead of missing in i5/OS *fork()*, other processes working in background (the x86 had a clean installation of Linux without any unnecessary services but the iSeries platform had many background jobs running while performing tests) or not optimal mechanism for creating new processes in i5/OS.

Concluding, the standardization of network sockets interface is a very important part of most currently used operating systems. Programmers can easily create software which can work on multiple platforms. Nevertheless, we have to remember about specific parameters as well as the implementation differences among platforms and operating systems. Moreover, network sockets servers usually strongly depend on multitasking functionalities which are not a part of network socket standard. Thus, it is essential to use a proper multitasking method for the implementation because, as it can be seen on the iSeries example, different methods may have a significant impact on the application performance.

Adresy

Błażej ADAMCZYK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, blazej.adamczyk@polsl.pl
Hafed ZGHIDI: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, hafed.zghidi@polsl.pl