Środowiska przetwarzania rozproszonego

Dariusz Wawrzyniak

✉ Politechnika Poznańska
Instytut Informatyki
ul. Piotrowo 2 (IIPP, pok. 5)
60-965 Poznań

Dariusz.Wawrzyniak@cs.put.poznan.pl

www.cs.put.poznan.pl/dwawrzyniak

☎ +48 (61) 665 2963

Program przedmiotu

- Wykład (16 godz.):
  - założenia projektowe w budowie systemów rozproszonych
  - transparentność dostępu do zdalnych zasobów (modele komunikacji)
  - skalowalność (replikacja)
- Ćwiczenia laboratoryjne (16 godz.)
  implementacja przykładowych aplikacji rozproszonych z wykorzystaniem wybranych mechanizmów/środowisk:
  - PVM/MPI (wymiana komunikatów)
  - RPC (wywoływanie procedur zdalnych)
  - Java RMI (podejście obiektowe —wywoływanie metod zdalnych)
  - Java Message Service

Sposób zaliczenia przedmiotu

- Wykład (16 godz.)
  egzamin końcowy.
- Ćwiczenia laboratoryjne (16 godz.)
  testy/sprawdziany z zagadnień laboratoryjnych w trakcie ćwiczeń lub w ramach egzaminu końcowego.

Literatura

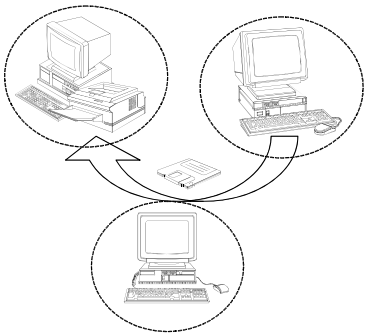
- A. S. Tanenbaum, M. van Steen: Systemy rozproszone, Zasady i paradygmaty. WNT, Warszawa, 2006.
- G. Coulouris, J. Dollimore, T. Kindberg: Systemy rozproszone: podstawy i projektowanie. WNT, Warszawa, 1998.
- A. S. Tanenbaum: Rozproszone systemy operacyjne, PWN, Warszawa, 1997.
- dokumentacja wybranych technologii (PVM/MPI, Sun RPC, Java, Java RMI, JMS, JavaSpaces)

System rozproszony od strony projektowej

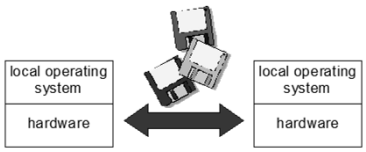
System rozproszony

- Zbiór niezależnych komputerów i zasobów komputerowych zdolnych do kooperacji (np. poprzez sieć komputerową), postrzeganych przez użytkownika jako całościowo spójny system.
- Ogólny cel projektowy systemu rozproszonego: stworzenie przezroczystego, otwartego, elastycznego, wydajnego i skalowalnego mechanizmu współdzielenia zasobów.

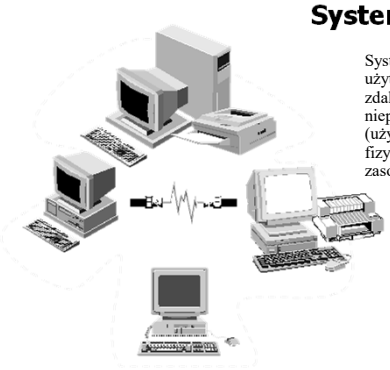
Niezależne komputery



Niezależne komputery — organizacja oprogramowania

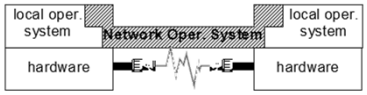


System sieciowy

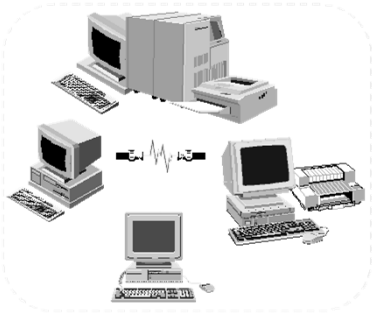


System sieciowy umożliwia użytkownikom dostęp do zdalnych zasobów w sposób nieprzezroczysty (użytkownicy są świadomi fizycznego rozproszenia zasobów)

System sieciowy — organizacja oprogramowania

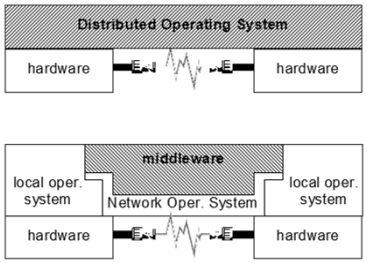


System rozproszony



Rozproszony system operacyjny umożliwia użytkownikom dostęp do zdalnych zasobów w sposób przezroczysty (jest jednolity sposób dostępu do zasobu lokalnego i zdalnego)

System rozproszony — organizacja oprogramowania



Zagadnienia projektowe (1)

- Otwartość, elastyczność — zdolność do rozbudowy (sprzętowej, programowej) i rekonfiguracji, możliwość dodawania nowych usług bez głębokiej ingerencji w istniejące już usługi.
- Skalowalność — możliwość dostosowania systemu do rosnących rozmiarów problemów i wymagań użytkowników.
- Współbieżność — możliwość współbieżnego (równoczesnego) ubiegania się o zasoby i ich użytkowania.

Zagadnienia projektowe (2)

- Tolerowanie uszkodzeń — odporność na awarie przez redundancję (sprzętu, danych) oraz możliwość odtworzenia spójnego stanu po awarii.
- Przezroczystość — ukrywanie przed użytkownikiem (programistą) fizycznego odseparowania składowych, zapewnienie jednolitego dostępu do zasobów lokalnych i zdalnych.
- Wydajność — optymalizacja ruchu w sieci w celu zredukowania negatywnego wpływy stosunkowo wolnej komunikacji sieciowej.

Przezroczystość (1)

- przezroczystość dostępu — ukrywanie różnic w reprezentacji danych, zagwarantowanie jednolitego sposobu dostępu do zasobów, niezależnie od tego, czy są to zasoby lokalne, czy zdalne
- przezroczystość położenia — identyfikacja zasobów niezależna od ich fizycznej lokalizacji (np. usługa nazw)
- przezroczystość migracji — zmiana fizycznej lokalizacji zasobu nie powoduje zmian w sposobie ich identyfikacji i w sposobie dostępu
- przezroczystość relokacji — fizyczna zmiana lokalizacji zasobu może być dokonana w sposób niewidoczny dla aplikacji w czasie realizacji dostępu do niego przez użytkowników

Przezroczystość (2)

- przezroczystość replikacji — utrzymywanie i udostępnianie kilku egzemplarzy tego samego zasobu (kopii) w taki sposób, jak gdyby użytkownik widział i działał tylko na jednym
- przezroczystość awarii — zdolność ukrycia przed użytkownikiem faktu chwilowych nieprawidłowości funkcjonowania
- przezroczystość współbieżności — realizacja współbieżnego dostępu do zasobu w taki sposób, że konkurujące procesy nie przeszkadzają sobie wzajemnie

Otwartość

- standaryzacja reguł dostępu do usług — formalny opis składni (jak skorzystać z usługi) i semantyki (na czym polega realizacja usługi)
- interoperacyjność (interoperability) — zdolność współpracy dwóch różnych systemów, korzystający wzajemnie jedynie ze swoich własnych usług
- przenośność (portablility) — zdolność aplikacji zaprojektowanej w systemie rozproszonym A do działania bez modyfikacji w systemie B

Polityka i mechanizm

- Polityka jest zbiorem reguł dostępu do zasobów i może wynikać z:
  - projektu systemu (ustalona jest na etapie projektowania systemu)
  - wytycznych kierownictwa (ustalana jest na etapie instalacji lub eksploatacji systemu)
  - decyzji indywidualnych użytkowników (podjętych w trakcie eksploatacji systemu)
- Mechanizm jest zbiorem dostępnych środków do wymuszania polityki
  - mechanizm powinien być na tyle uniwersalny, żeby dało się go dostosować do zmian w polityce ➔ **otwartość**
  - w skrajnym przypadku każda zmiany polityki mogłaby wymagać zmiany mechanizmu ➔ **system zamknięty**

Koncepcja dostępu do współdzielonych zasobów

- zarządca zasobu (ang. resource manager) — moduł oprogramowania odpowiedzialny za udostępnianie zasobu
- użytkownik zasobu — moduł (proces) zgłaszający zapotrzebowanie na zasób

Współpraca pomiędzy zarządcą a użytkownikiem

- model komunikatowy — współpraca odbywa się przez komplementarne wykonanie operacji *send* i *receive* odpowiednio przez nadawcę i odbiorcę komunikatu
- model obiektowy — zarządca postrzegany jest jako obiekt o pewnym identyfikatorze, będący w określonym stanie, który zmienia się pod wpływem operacji żądanych przez użytkowników
  - obiekt aktywny — obiekt jest procesem (wielowątkowym), oczekującym na żądania wywołania metod
  - obiekt pasywny — każdy obiekt ma własną przestrzeń adresową (segment), odwzorowywaną na przestrzeń adresową procesu, który się do niego odwołuje

Komunikacja w systemach sieciowych/rozproszonych

- Elementarne mechanizmy komunikacji pomiędzy procesami
- Model komunikacji w systemach rozproszonych

Elementarne mechanizmy komunikacji pomiędzy procesami

- Współdzielenie pamięci
  - podstawą komunikacji jest dostęp do wspólnych danych w pamięci,
  - wymiana informacji sprowadza się do zapisu i odczytu wspólnych danych oraz związanej z tym synchronizacji.
- Przekazywanie komunikatów
  - podstawą komunikacji jest umieszczanie danych w podsystemie komunikacyjnym oraz ich pobieranie z podsystemu komunikacyjnego,
  - wymiana informacji polega na wywoływaniu odpowiednich funkcji w celu wysłania i odbioru komunikatu.

Sieciowa realizacja elementarnych mechanizmów komunikacji

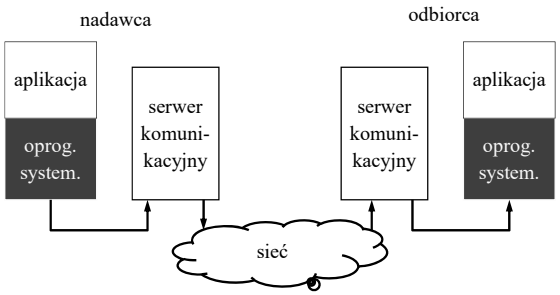
- Współdzielenie pamięci
  - brak możliwości współdzielenia pamięci fizycznej,
  - emulacja pamięci współdzielonej poprzez odpowiednią obsługę błędu strony w systemie pamięci wirtualnej.
- Przekazywanie komunikatów
  - komunikacja asynchroniczna → wymagana gotowość do odbioru danych po stronie adresata komunikatu,
  - schemat komunikacji zgodny z modelem klient-serwer.

Model komunikacji w systemach rozproszonych

- Wywoływanie procedur zdalnych (ang. remote procedure call)
- Wywoływanie metod zdalnych (ang. remote method call)
- Komunikacja zorientowana na przysyłanie wiadomości (ang. message-oriented communication)
- Komunikacja strumieniowa
- Komunikacja za pośrednictwem wirtualnej pamięci współdzielonej

Komunikacja zorientowana na przysyłanie wiadomości

Przesyłanie wiadomości — model systemu



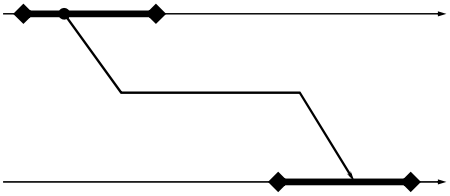
Trwałość komunikacji

- Komunikacja przejściowa (ang. transient communication) — wiadomość jest przekazywana (utrzymywana w podsystemie komunikacyjnym) pod warunkiem, że działa nadawca i odbiorca tej wiadomości.
- Komunikacja nieustanna (ang. persistent communication) — wiadomość jest przechowywana w celu doręczenia do odbiorcy nawet, gdy odbiorca nie działa w danej chwili, a nadawca zakończył działanie po wysłaniu tej wiadomości.

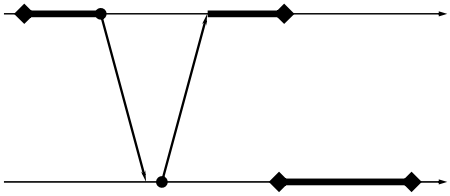
Synchroniczność komunikacji

- Komunikacja synchroniczna — nadawca kontynuuje działanie dopiero, gdy wiadomość znajdzie się w buforze odbiorczym lub zostanie doręczona do adresata.
- Komunikacja asynchroniczna — nadawca kontynuuje działanie zaraz po przekazaniu wiadomości do podsystemu komunikacyjnego.

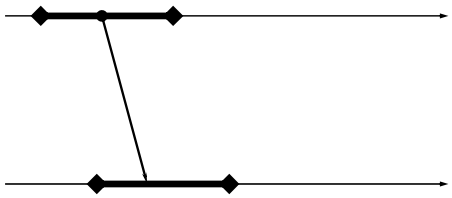
Komunikacja nieustanna asynchroniczna



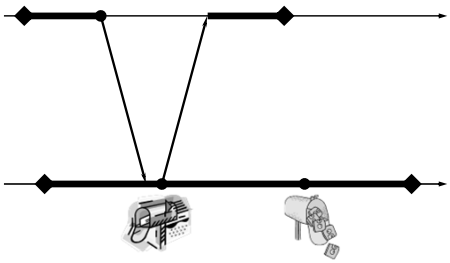
Komunikacja nieustanna synchroniczna



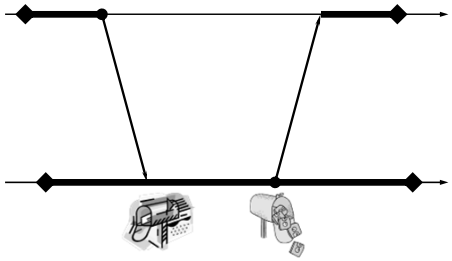
Komunikacja przejściowa asynchroniczna



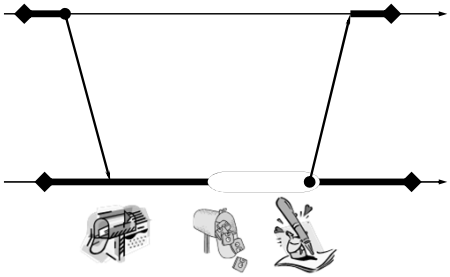
Komunikacja przejściowa synchroniczna z potwierdzeniem dotarcia



Komunikacja przejściowa synchroniczna z potwierdzeniem odebrania



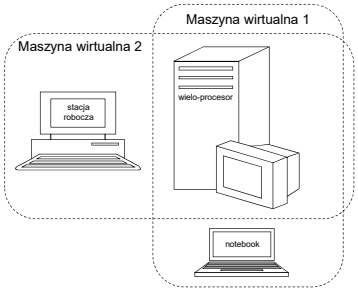
Komunikacja przejściowa synchroniczna z oczekiwaniem na odpowiedź



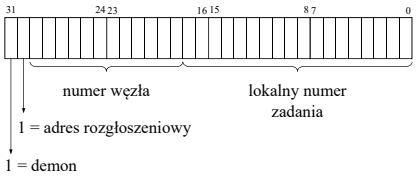
Środowiska wymiany komunikatów

- Identyfikacja procesów — przeźroczystość położenia
  - PVM: unikalny identyfikator procesu (TID)
  - MPI: unikalny numer w grupie procesów
- Mechanizmy komunikacji — przeźroczystość dostępu
  - PVM: adresowanie komunikatów do procesów o podanych identyfikatorach
  - MPI: adresowanie komunikatów do procesów o podanych numerach
- Mechanizm zdalnego uruchamiania zadań
  - PVM: dynamicznie w trakcie działania aplikacji
  - MPI (v.1): statycznie w trakcie uruchamiania aplikacji

PVM — maszyna wirtualna



PVM — budowa identyfikatora zadania (TID)



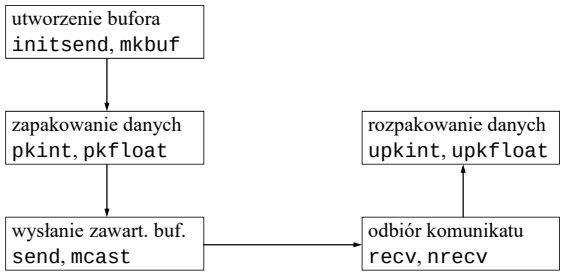
PVM — uruchamianie przetwarzania

- Przygotowanie programów
  - przygotowanie kodów źródłowych w języku C lub Fortran
  - kompilacja i konsolidacja kodów źródłowych
- Skonfigurowanie maszyny wirtualnej
  - wybranie i przygotowanie opisu odpowiednich węzłów
  - uruchomienie demonów na poszczególnych węzłach
  - ewentualna dynamiczna zmiana konfiguracji początkowej
- Uruchomienie zadań
  - uruchomienie procesu na terminalu jednego z węzłów
  - uruchomienie zadania poleceniem `spawn` z konsoli PVM
  - uruchomienie zadania przez inne zadanie

PVM — podstawowe funkcje biblioteczne

- Konfiguracja maszyny wirtualnej: `pvm_addhosts`, `pvm_delhosts`, `pvm_config`, `pvm_tidtohost`
- Obsługa zadań: `pvm_mytid`, `pvm_exit`, `pvm_spawn`, `pvm_kill`, `pvm_task`, `pvm_parent`
- Komunikacja międzyprocesowa
  - obsługa buforów: `pvm_initsend`, `pvm_mkbuf`, `pvm_freebuf`, `pvm_getsbuf`, `pvm_getrbuf`, `pvm_setsbuf`, `pvm_setrbuf`
  - pakowanie danych: `pvm_pk`..., `pvm_upk`...
  - wymiana komunikatów: `pvm_send`, `pvm_mcast`, `pvm_psend`, `pvm_recv`, `pvm_nrecv`, `pvm_probe`, `pvm_trecv`, `pvm_bufinfo`, `pvm_precv`

PVM — schemat wymiany komunikatów



PVM — dynamiczne grupy procesów

- Grupa procesów identyfikowana jest przez nazwę.
- Każde zadanie może w dowolnej chwili dołączyć się do grupy, jak i opuścić grupę.
- Przyłączając się do grupy, zadanie otrzymuje w tej grupie unikalnym numer (jest to numer kolejny, począwszy od 0).
- Zadanie może należeć jednocześnie do wielu grup.
- Zadanie może wysłać komunikat do wszystkich procesów w grupie, nawet jeśli do niej nie należy (grupy otwarte).

PVM — funkcje biblioteczne do obsługi grup procesów

- Dołączanie procesu do grupy (również tworzenie grupy): `pvm_joingroup`
- Odłączanie procesu od grupy (ostatecznie usuwanie grupy): `pvm_lvgroup`
- Identyfikacja procesów w grupie: `pvm_gettid`, `pvm_getinst`
- Informacja o liczbie procesów w grupie: `pvm_gsize`
- Rozgłaszanie (komunikat do wszystkich w grupie): `pvm_bcast`
- Bariera synchronizująca: `pvm_barrier`

MPI — podstawowe cechy

- Model przetwarzania — SPMD (Single Program Multiple Data)
  - wszystkie uruchomione procesy wykonują ten sam program
- Wszystkie procesy uruchamiane są przy rozpoczęciu przetwarzania (w wersji 2.0 można również dynamicznie uruchomić dodatkowe procesy) — `mpirun`, `mpiexec`
- Procesy tworzą grupę, w której numer procesu jest jego identyfikatorem
  - grupa jest częścią tzw. komunikatora
  - istnieje predefiniowany komunikator `MPI_COMM_WORLD` obejmujący wszystkie procesy

MPI — komunikator

- Kontekst komunikacyjny — wirtualny kanał komunikacyjny, umożliwiający odseparowanie komunikatów
  - można odbierać tylko komunikaty przekazywane w ramach tego samego kontekstu
- Grupa procesów — grupa o ustalonym rozmiarze, w ramach której identyfikowane są procesy
  - nadawca i odbiorca identyfikowany jest poprzez numer w danej grupie
- Komunikator jest parametrem każdej funkcji do realizacji wymiany komunikatów

MPI — obsługa komunikatora

- Uzyskanie własnego numeru przez proces: `MPI_Comm_rank`
- Uzyskanie liczby procesów w grupie komunikatora: `MPI_Comm_size`
- Uzyskanie grupy komunikatora: `MPI_Comm_group`
- Utworzenie komunikatora dla określonej grupy procesów: `MPI_Comm_create`

MPI — komunikacja punkt-punkt

- Komunikacja w trybie blokującym: `MPI_Send`, `MPI_Recv`
- Komunikacja natychmiastowa: `MPI_Isend`, `MPI_Irecv`
- Komunikacja synchroniczna: `MPI_Ssend`, `MPI_Issend`
- Komunikacja buforowana: `MPI_Bsend`, `MPI_Ibsend`
- Komunikacja w trybie gotowości: `MPI_Rsend`, `MPI_Irsend`

MPI — przetwarzanie kolektywne

- Kolektywny transfer danych: `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter`, `MPI_Gatherall`, `MPI_Alltoall`
- Obliczenie kolektywne: `MPI_Reduce`, `MPI_Reducescatter`
- Synchronizacja: `MPI_Barrier`

MPI_Alltoall

A1	A2	A3	A4
B1	B2	B3	B4
C1	C2	C3	C4
D1	D2	D3	D4

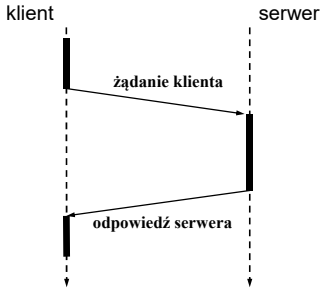
A1	B1	C1	D1
A2	B2	C2	D2
A3	B3	C3	D3
A4	B4	C4	D4

Wywoływanie procedur zdalnych

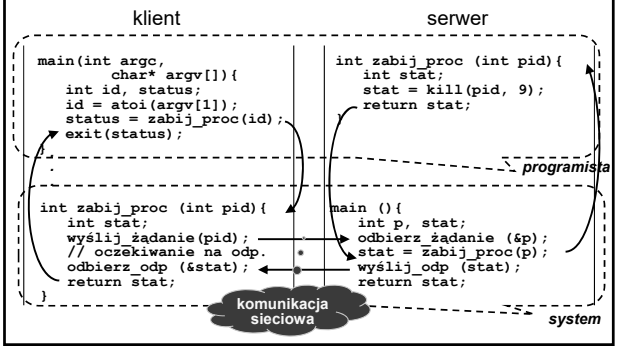
Mechanizm wywołania procedury

```
main(int argc, char* argv[]){
   int id, status;
   id = atoi(argv[1]);
   status = zabij_proc(id);
   exit(status)
}  .
   .
   .
int zabij_proc (int pid){
   int stat;
   stat = kill(pid, 9);
   return stat;
}
```

Schemat interakcji klient-serwer (transakcja komunikatu)



Semantyka wywołania procedury w komunikacji sieciowej



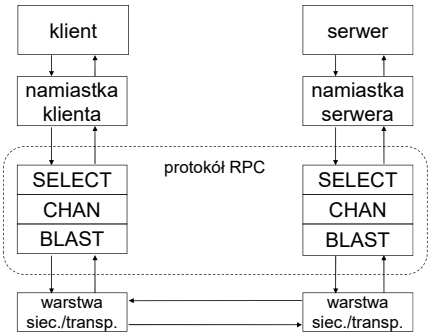
RPC — zagadnienia projektowe

- Przeźroczystość dostępu — ukrycie komunikacji sieciowej przed aplikacją przez odpowiednie opakowanie funkcji komunikacyjnych namiastką klienta oraz serwera.
- Gwarancja wykonania — ukrywanie błędów komunikacyjnych
- Specyfikacja interfejsu — sposób opisu sygnatur procedur zdalnych (nazwy, typy parametrów)
- Obsługa sytuacji wyjątkowych

RPC — przezroczystość dostępu

- Namiastka klienta (ang. client stub) — udostępnienie aplikacji klienckiej procedury lokalnej odpowiedzialnej za przesłanie danych do serwera oraz odebranie wyników
- Namiastka serwera (ang. server stub) — udostępnienie aplikacji po stronie serwera procedury lokalnej odpowiedzialnej za odebranie identyfikatora procedury zdalnej do wywołania, parametrów procedury, a odesłanie wyników lub zgłoszenie wyjątków

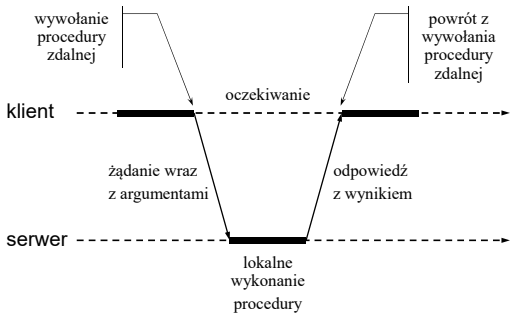
Warstwy protokołu RPC



Interakcja klient-serwer w realizacji wywołania zdalnego

1. Lokalne wywołanie procedury namiastki przez klienta
2. Przygotowanie (upakowanie) danych do wysłania na stronę serwera
3. Wysłanie przygotowanego komunikatu na stronę serwera
4. Odebranie komunikatu przez serwer
5. Rozpakowanie danych
6. Wykonanie procedury zdalnej
7. Przygotowanie (upakowanie) danych z odpowiedzią dla klienta
8. Wysłanie przygotowanego komunikatu na stronę klienta
9. Odebranie komunikatu przez namiastkę klienta
10. Rozpakowanie komunikatu i zwrócenie klientowi wyniku

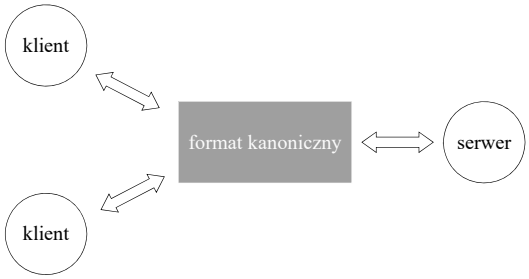
Diagram przestrzenno-czasowy interakcji klient-serwer



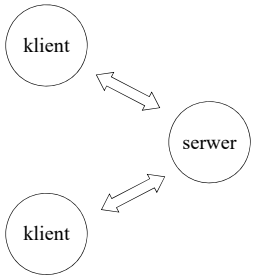
RPC - przekazywanie parametrów

- przekazywanie przez wartość (ang. call-by-value) — problem reprezentacji danych (np. różnice w kodowaniu znaków, różnice w kolejności bajtów, formaty liczb zmiennopozycyjnych)
- przekazywanie przez referencje (ang. call-by-reference) — problem zinterpretowania wartości wskaźnika w innej przestrzeni adresowej
- przekazywanie przez kopiowanie i odtwarzanie (ang. call-by-copy/restore) — utworzenie kopii parametru po stronie procedury i zapis dokonanych tam zmian w procesie klienta po jej zakończeniu (problem opisu struktur danych w celu prawidłowego zidentyfikowania wszystkich składowych)

Reprezentacja kanoniczna



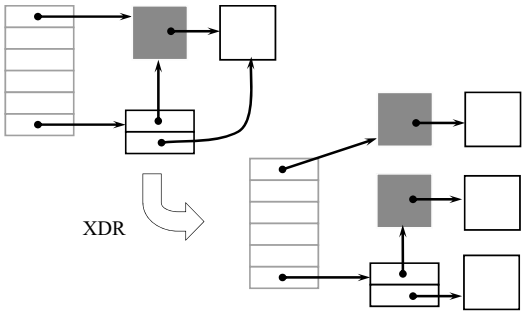
Konwersja bezpośrednia



Przekazywanie referencji przez kopiowanie i odtwarzanie

1. Skopiowanie wskazanej wartości do bufora komunikacyjnego i wysłanie do serwera.
2. Wywołanie po stronie serwera procedury zdalnej ze wskaźnikiem na kopię wartości utworzoną po stronie serwera.
3. Skopiowanie zmodyfikowanej wartości z przestrzeni adresowej serwera do bufora komunikacyjnego i przesłanie z powrotem do klienta.
4. Umieszczenie odebranej wartości w miejscu wskazywanym przez referencję po stronie klienta.

Kopiowanie i odtwarzanie — przykład złożonej struktury danych



RPC — gwarancja wykonania

- Semantyka *ewentualnie* — brak gwarancji, procedura mogła się wykonać lub mogła się nie wykonać.
- Semantyka *co najmniej raz* — po uzyskaniu **odpowiedzi z wynikiem** od serwera klient ma pewność, że wywoływana procedura wykonała się co najmniej raz.
- Semantyka *co najwyżej raz* — po uzyskaniu **odpowiedzi z wynikiem** od serwera klient wie, że wywoływana procedura wykonała się dokładnie raz.

Jak należy zinterpretować przypadek wystąpienia błędu (wyjątku) w wywołaniu procedury zdalnej?

- Semantyka *dokładnie raz* — niemożliwa do uzyskania, jeśli system narażony jest na awarie (np. serwera lub łączy).

Specyfikacja interfejsu

- Opis interfejsu w języku implementacji (np. Ada, Java RMI)
- Opis interfejsu w języku specjalnym, niezależnym od implementacji (CORBA — IDL, Sun RPC — rpcgen)

RPC — zagadnienia realizacyjne

- Przetwarzanie interfejsu
- Wiązanie klienta z serwerem
- Obsługa komunikacji klient-serwer
- Realizacja semantyki błędu
- Problem osieroconych obliczeń

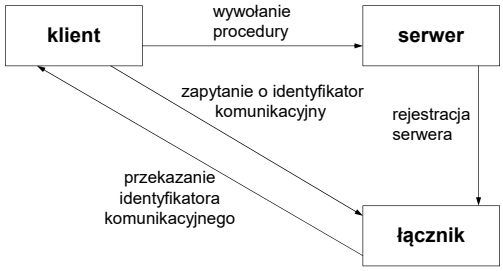
Przetwarzanie interfejsu procedur zdalnych

- Generowanie namiastki klienta
- Generowanie namiastki serwera
- Generowanie przykładowego programu klienta (client sample)
- Generowanie wzorca do implementacji procedur zdalnych (template)
- Generowanie plików do zarządzania kompilacją

Wiązanie klienta z serwerem

- Wiązanie statyczne — klient ma na stałe wprowadzony identyfikator komunikacyjny serwera (np. para: adres IP, nr portu).
- Wiązanie dynamiczne — klient uzyskuje adres serwera za pośrednictwem łącznika (np. portmap, lub rpcbind w Sun RPC).

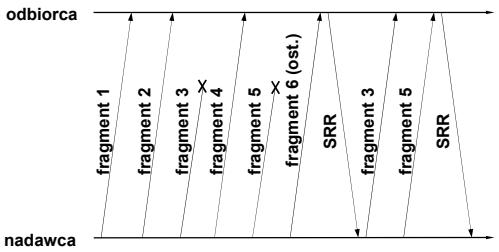
Wiązanie dynamiczne klienta z serwerem



Obsługa komunikacji klient-serwer

- BLAST — realizuje przesyłanie dużych komunikatów poprzez podział na mniejsze części, transmisję poszczególnych części i ponowne złożenie w jeden komunikat po stronie odbiorczej,
- CHAN — synchronizuje wymianę komunikatów z żądaniami wywołania procedur oraz odpowiedziami,
- SELECT — rozdziela i przekazuje komunikaty z żądaniami do odpowiednich procesów.

BLAST



BLAST — nadawca

1. otrzymanie bloku z w wyższej warstwy stosu protokołów i podział bloku na fragmenty
2. wysłanie kolejno poszczególnych fragmentów do odbiorcy (ze specjalnym oznaczeniem ostatniego fragmentu)
3. ustawienie czasomierza (ang. timer) DONE
4. jeśli dotarł SRR z informacją o brakujących fragmentach, to wysłanie brakujących fragmentów i przejście do pkt. 3
5. jeśli dotarł SRR z informacją o odebraniu wszystkich fragmentów,
   to SUKCES
6. jeśli DONE = 0 // minął czas oczekiwania
   to BŁĄD

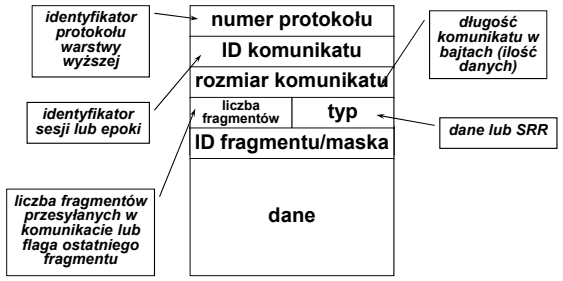
BLAST — odbiorca (1)

1. po odebraniu pierwszego komunikatu z fragmentem bloku danych:
   inicjalizacja struktur danych do przechowywania poszczególnych bloków, umieszczenie odebranego fragmentu w odpowiedniej strukturze
   ustawienie licznika powtórzeń na 0
2. ustawienie czasomierza LAST_FRAG
3. po odebraniu kolejnego (ale nie ostatniego) komunikatu: umieszczenie nadesłanego fragmentu bloku w odpowiedniej strukturze i przejście do pkt. 2

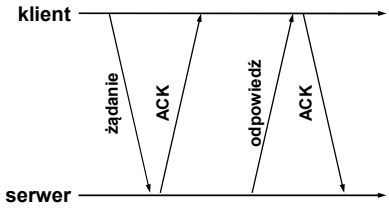
BLAST — odbiorca (2)

4. jeśli LAST_FRAG = 0, to przejście do pkt. 8
5. po odebraniu ostatniego komunikatu umieszczenie nadesłanego fragmentu bloku w odpowiedniej strukturze i sprawdzenie kompletności bloku
6. jeśli blok jest kompletny to wysłanie komunikatu SRR i przekazanie bloku do wyższej warstwy stosu protokołów
7. jeśli blok nie jest kompletny przejście do pkt. 8
8. jeśli licznik powtórzeń jest mniejszy od 3, to wysłanie komunikatu SRR z informacją o brakujących blokach, zwiększenie licznika powtórzeń o 1 i przejście do pkt. 2 w przeciwnym razie rezygnacja z odbioru

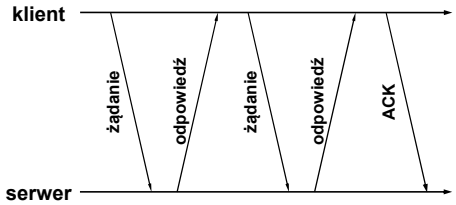
BLAST — format komunikatu



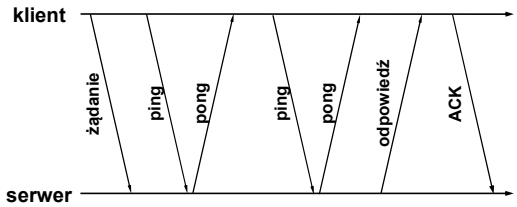
CHAN



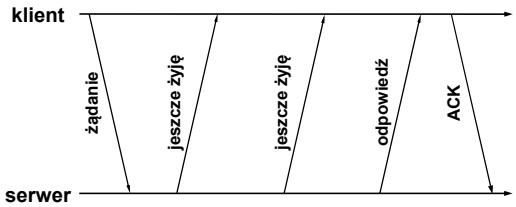
CHAN — domniemane potwierdzenia



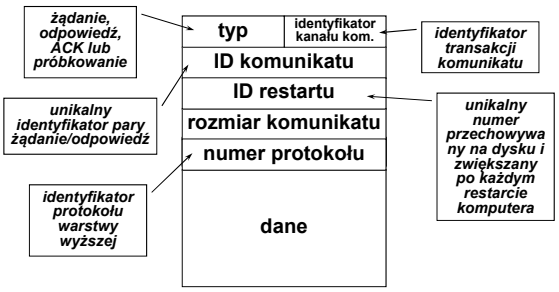
CHAN — próbkowanie serwera



CHAN — „bicie serca”



CHAN — format komunikatu



Realizacja semantyki błędu

- nie można zlokalizować serwera — zgłoszenie wyjątku
- zaginione żądanie — retransmisja żądania po upłynięciu ustalonego czasu oczekiwania
- zaginiona odpowiedź — retransmisja żądania
  - stosowanie procedur idempotentnych
  - numerowanie żądań i retransmisja odpowiedzi
- awaria serwera
  - przed podjęciem realizacji → retransmisja żądania
  - po wykonaniu →zgłoszenie wyjątku
- awaria klienta — osierocenie obliczeń

Usuwanie osieroconych obliczeń (1)

- Eksterminacja — rejestrowanie działań podejmowanych przez klienta na nośniku niewrażliwym na awarie i usuwanie na tej podstawie osieroconych obliczeń po restarcie klienta.
- Reinkarnacja — każdy restart klienta rozpoczyna nową epokę (identyfikowaną przez numer kolejny), po której usuwane są wszystkie obliczenia związane z poprzednią epoką.

Usuwanie osieroconych obliczeń (2)

- Łagodna reinkarnacja — reinkarnacja, w której usuwa się tylko te obliczenia rozpoczęte w starej epoce, dla których nie ma właściciela.
- Wygaśnięcie — przydział określonego czasu *T* serwerowi na wykonanie procedury. Jeśli wykonanie nie zakończy się w czasie *T*, serwer musi uzyskać kolejny przydział, pod warunkiem, że obliczenia nie zostały osierocone. Jeśli klient odczeka czas *T* przy restarcie, osierocone obliczenia same się zakończą.

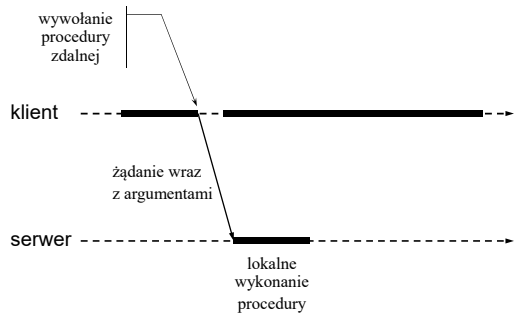
SELECT

- Po stronie klienta: odwzorowanie wywoływanej procedury na jej identyfikator, przekazywany do serwera.
- Po stronie serwera: zlokalizowanie wywoływanej procedury na podstawie identyfikatora.

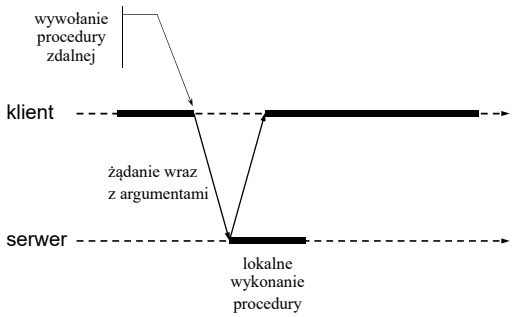
Warianty użycia mechanizmu RPC

- Wywołanie asynchroniczne
- Wywołanie zwrotne

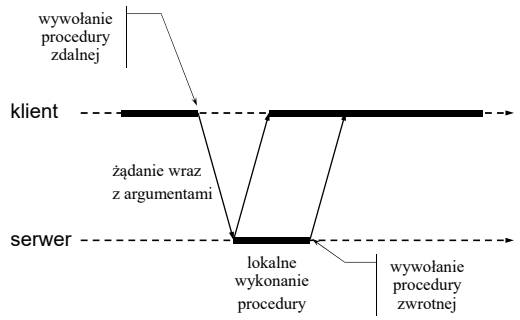
Wywołanie asynchroniczne



Wywołanie asynchroniczne z potwierdzeniem odbioru



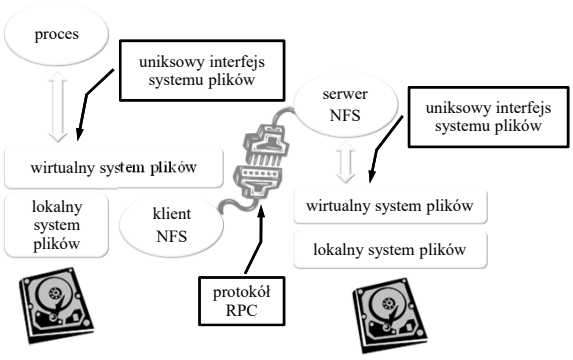
Wywołanie zwrotne



Idempotentność i bezstanowość

Przykład — Network File System

Architektura NFS



Uniksowy interfejs systemu plików

- open (pathname, flags) → fd
- creat (pathname, mode) → fd
- read (fd, buf, count)
- write (fd, buf, count)
- lseek (fd, offset, whence)
- close (fd)
- unlink (pathname)

Interfejs zdalnego dostępu do pliku —protokół RPC

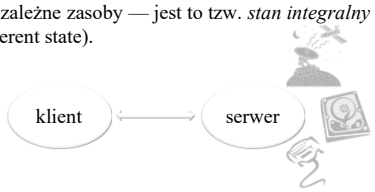
- lookup (dirfh, name) → fh, attr
- create (dirfh, name, attr) → newfh, attr
- read (fh, offset, count) → data, attr
- write (fh, offset, data) → attr
- remove (dirfh, name) → status

Odwzorowanie interfejsu uniksowego na interfejs zdalnego dostępu

- Jak wygląda realizacja uniksowej funkcji *open*?
- Jak wygląda realizacja uniksowej funkcji *creat*?
- Gdzie przechowywane są dane identyfikujące **otwarty** plik?
- Jak wygląda realizacja uniksowej funkcji *read*/*write*?
- Jak wygląda realizacja uniksowej funkcji *lseek*?

Problem opisu stanu systemu

- Stan systemu/obiektu z perspektywy klienta postrzegany jest poprzez odpowiedzi serwera — jest to tzw. *stan postrzegany* (ang. observable state).
- Stan systemu po stronie serwera może być współtworzony przez niezależne zasoby — jest to tzw. *stan integralny* (ang. inherent state).



Idempotentność procedur

- Procedura/metoda/operacja zdalna jest idempotentna
  - w sensie postrzegania (ang. observably idempotent), jeśli jej wywołanie z określonymi wartościami parametrów zawsze daje taki sam wynik (wynik w sposób jednoznaczny jest zdeterminowany wartościami parametrów)
  - w sensie integralnym (ang. inherently idempotent), jeśli jej wywołanie z określonymi wartościami parametrów i przy określonym stanie zasobów zawsze daje taki sam wynik (wynik w sposób jednoznaczny jest zdeterminowany wartościami parametrów oraz stanem niezależnych zasobów)

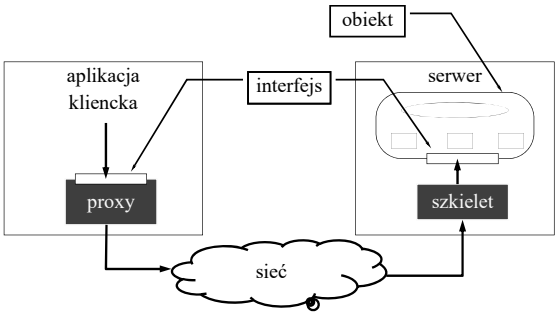
Bezstanowość

- Serwer/obiekt jest bezstanowy (ang. stateless)
  - w sensie postrzegania, jeśli wszystkie jego procedury/metody/operacje są idempotentne w sensie postrzegania.
  - w sensie integralnym, jeśli wszystkie jego procedury/metody/operacje są idempotentne w sensie integralnym.

Wywoływanie metod zdalnych

Podejście obiektowe do budowy systemów rozproszonych

Wywoływanie metod zdalnych — model systemu



Istota podejścia obiektowego

- Obiekt jest jednostką integrującą w sobie dane (stan obiektu) oraz odpowiednio zdefiniowane operacje (metody)
- Jedyna forma dostępu do danych polega na wywoływaniu metod wyspecyfikowanych w publicznym interfejsie obiektu
- Obiekt może implementować wiele interfejsów
- Ten sam interfejs może być implementowany przez wiele obiektów

Podejście obiektowe do budowy systemów rozproszonych

- Kluczowe dla mechanizmu wywoływania metod zdalnych jest oddzielenie definicji interfejsu (specyfikacji) od jego implementacji w obiekcie
- W językach definicji interfejsu (IDL) interfejs traktowany jest jak typ danych
- W języku implementacji każdy obiekt implementujący dany interfejs jest instancją tego typu

Przykład opisu interfejsu w podejściu obiektowym (CORBA IDL)

```
interface Konto {
     float stan();
     float wplac(in float value);
     float pobierz(in float value);
};

interface Bank {
     Konto otworzKonto(in int numer);
     float zamknijKonto(in Konto k);
};
```

Klasyfikacja obiektów ze względu na sposób implementacji

- Obiekt zdalny (ang. remote object) — stan obiektu utrzymywany jest przez jeden serwer (wszystkie dane obiektu znajdują się na jednym serwerze)
- Obiekt rozproszony (ang. distributed object) — stan obiektu przechowywany jest przez więcej niż jeden serwer (poszczególne serwery przechowują odpowiednie dane, stanowiące fragmenty tego samego obiektu)
- Z punktu widzenie klienta zarówno obiekt zdalny jak i obiekt rozproszony stanowi pewną całość odpowiednio identyfikowaną

Klasyfikacja ze względu na dostępność informacji o obiektach

- Obiekt dostępny w czasie kompilacji (ang. compile-time object) — informacja o obiekcie (jego typ) znana jest i dostępna w odpowiedniej formie w programie klienta na etapie tworzenia oprogramowania
- Obiekt dostępny w czasie wykonania (ang. runtime object) — informacja o obiekcie dostępna jest dopiero w czasie działania aplikacji

Klasyfikacja obiektu ze względu na trwałość

- Obiekt trwały (ang. persistent object) — obiekt istnieje nawet wówczas, gdy nie ma serwera, w przestrzeni adresowej, którego mógłby być przechowywany stan tego obiektu
- Obiekt przejściowy (ang. transient object) — obiekt istnieje tak długo, jak długo działa serwer utrzymujący jego stan

Referencja do obiektu

- Referencja jest identyfikatorem, wykorzystywanym do wskazania obiektu, na którym ma zostać wykonana operacja
- W celu wywołania metody obiektu rezydującego na innej maszynie potrzebna jest zdalna referencja
- Referencja może być przekazywana jako parametr wywołania metody lub zwrócona jako wynik wykonania metody

Implementacja zdalnej referencji

- Referencja jest wartością, której struktura wewnętrzna nie jest w żaden sposób interpretowana przez aplikację
- Zdalna referencja musi zawierać wystarczająco dużo informacji, żeby dowiązać obiekt w programie klienta
- Dowiązanie obiektu oznacza uzyskanie informacji, niezbędnej do wywołania metody obiektu (uzyskanie proxy)

Dowiązanie obiektu

- Dowiązanie jawne (ang. explicit binding) — dowiązanie wymaga wywołania odpowiedniej funkcji (np. bind), po którym może dopiero nastąpić wywołanie metody
- Dowiązanie domyślne (ang. implicit binding) — wywołanie metody za pośrednictwem referencji skutkuje ustanowieniem dowiązanie do obiektu w procesie klienta

Wywoływanie metod

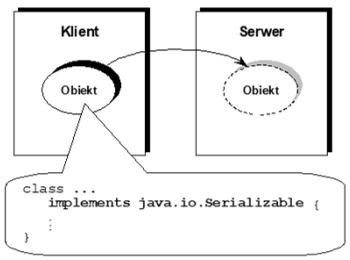
- Statyczne wywoływanie metod — wywołanie metody za pośrednictwem proxy, wygenerowanego na podstawie definicji interfejsu
- Dynamiczne wywoływanie metod — „skomponowanie" informacji niezbędnych do wywołania w czasie działania systemu, np.:

```
invoke(ref_obj, id_metody,
       param_wej, param_wyj);
```

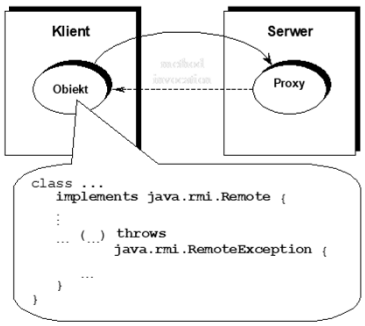
Przekazywanie obiektów jako parametrów

- Przekazywanie przez wartość — do zdalnej metody przekazywana jest kopia obiektu, który jest parametrem rzeczywistym wywołania.
- Przekazywanie przez referencję (zmienną) — do zdalnej metody przekazywana jest referencja (zdalna) do obiektu, który jest parametrem rzeczywistym.
- Przekazywanie przez kopiowanie i odtwarzanie — do zdalnej metody przekazywana jest kopia obiektu, a po zakończeniu zdalnej metody następuje aktualizacja obiektu po stronie wywołania, stosownie do zmian dokonanych w ramach wykonania metody.

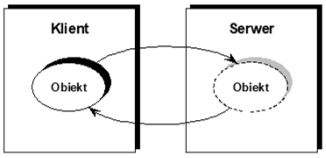
Przekazywanie przez wartość — przykład w Java RMI



Przekazywanie przez referencję — przykład w Java RMI



Przekazywanie przez kopiowanie i odtwarzanie



Message Oriented Middleware

Systemy kolejkowania komunikatów

Cechy MOM

- Uniezależnienie funkcjonowania składników aplikacji od dostępności informacji o interfejsach innych składników
- Uniezależnienie funkcjonowania warstwy komunikacyjnej (kanału komunikacyjnego) od działania (obecności) komunikujących się procesów
- Łatwość wdrożenia komunikacji asynchronicznej

Koncepcja komunikacji oparta na kolejkowaniu

- Organizacja warstwy komunikacyjnej w postaci systemu kolejek realizowanych w oparciu o zasoby pamięci, w tym pamięci dyskowej (gwarancja trwałości na wypadek awarii)
- Udostępnienie mechanizmów komunikacji polegających na:
  - umieszczeniu komunikatów w kolejkach,
  - pobieraniu komunikatów z kolejek

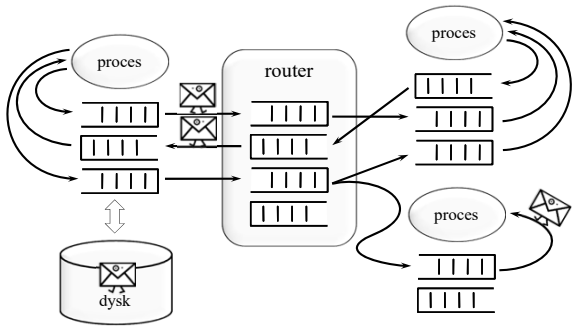
Paradygmat publish/subscribe

- Udostępnienie mechanizmu komunikacji opartego na identyfikacji wiadomości, a nie adresu nadawcy/odbiorcy
- Komunikujące się strony nie znają się wzajemnie
- Strona publikująca (nadawca) udostępnia treść związaną z określonym tematem (ang. topic)
- Środowisko komunikacyjne (usługa) przekazuje treść udostępnionych wiadomości odbiorcom (subskrybentom), którzy zarejestrowali (zapisali) się na dany temat.

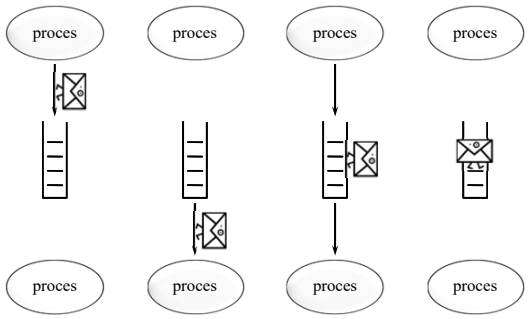
Model systemu kolejkowania komunikatów — podstawowe pojęcia

- Wiadomość (ang. message) — porcja danych (najczęściej z dodatkowymi własnościami) składowana w kolejce
- Kolejka (ang. queue) — miejsce przechowywania wiadomości (komunikatów)
- Proces (ang. process) — element aplikacji, zlecający operacje na wiadomościach w kolejce
- Zarządca zbioru kolejek — moduł na danym węźle, odpowiedzialny za wykonywanie operacji na kolejkach (np. tworzenie, usuwanie, lokalizowanie kolejek, ustawianie atrybutów kolejek itp.)

Model systemu kolejkowania komunikatów — funkcjonowanie



Warianty komunikacji



Przykłady rozwiązań typu MOM

- IBM MQSeries (WebSphere MQ, XMS — Message Service Client)
- Microsoft Message Queuing (MSMQ)
- Java Message Service (JMS)
- Apache ActiveMQ
- Oracle Advanced Queueing
- Sun Java System Message Queue (OpenMQ — wersja open source)
- RabbitMQ
- ZeroMQ
- Usługa IceStorm w systemie ICE

MOM API

rodzaj operacji	IBM MQSeries	MSMQ
tworzenie kolejki		MQCreateQueue
usuwanie kolejki		MQDeleteQueue
otwieranie kolejki	MQopen	MQOpenQueue
zamykanie kolejki	MQclose	MQCloseQueue
wysyłanie wiadomości	MQput	MQSendMessage
odbieranie wiadomości	MQget	MQReceiveMessage

Java Message Service

na podstawie slajdów Cezarego Sobańca

Historia JMS

- Opracowany w 1998
- Pierwotny cel: dostęp do istniejących systemów kolejkowania wiadomości (tzw. MOM — Message Oriented Middleware, np. IBM MQSeries)
- Integralna część Java EE od wersji 1.3

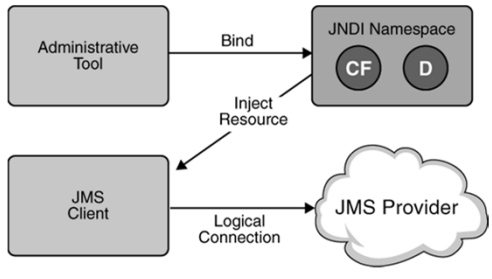
Komponenty JMS

- Dostawca JMS (ang. JMS provider) — implementacja interfejsów JMS, administracja, sterowanie
- Klienci JMS — aplikacje i komponenty wysyłające i odbierające komunikaty
- Wiadomości — obiekty do przenoszenia informacji
- Obiekty zarządzania (ang. administered objects) – prekonfigurowane obiekty na potrzeby zarządzania:
  - cele (ang. destinations)
  - fabryki połączeń (ang. connection factories)

Funkcjonalność JMS

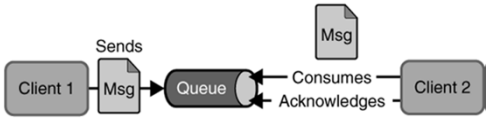
- Nieustanna, niezawodna, asynchroniczna komunikacja międzyprocesowa
- Transakcyjna interakcja z dostawcą JMS
- Modele komunikacji (messaging domains)
  - punkt-punkt (ang. point-to-point)
  - subskrypcji (ang. publish/subscribe)

Architektura JMS



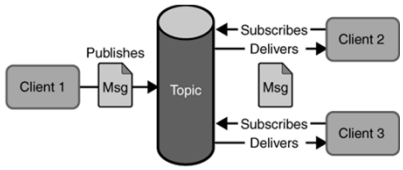
Model punkt-punkt

- Wysyłanie i odbiór poprzez dostęp do kolejki
- Wiadomości pozostają w kolejce do czasu odbioru lub przedawnienia
- Każda wiadomość ma 1 konsumenta
- Brak ograniczeń czasowych



Model subskrypcji

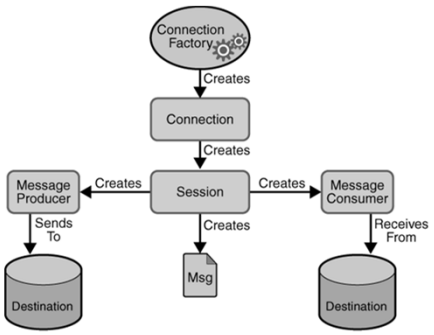
- Każda wiadomość może mieć wielu konsumentów
- Wiadomości są dostarczane do wszystkich aktualnych subskrybentów a następnie są usuwane
- Nie można odczytać wiadomości sprzed subskrypcji
- Trwała subskrypcja (ang. durable subscription) – odbiór wiadomości z okresu nieaktywności klienta



Model odbioru (konsumpcji) wiadomości

- Konsumpcja synchroniczna – blokująca metoda receive() z (ewentualnym) ograniczeniem czasowym
- Konsumpcja asynchroniczna – odbiornik wiadomości (ang. message listener) – asynchroniczne wywołanie metody onMessage()

Model programistyczny JMS



Fabryki połączeń

- Ogólna: ConnectionFactory (interfejs bazowy dla poniższych)
- Dla kolejek: QueueConnectionFactory
- Dla tematów: TopicConnectionFactory

Destinations

- *queue* w przypadku komunikacji punkt-punkt
- *topic* w przypadku modelu subskrypcji

```
@Resource(mappedName="jms/Queue")
private static Queue queue;

@Resource(mappedName="jms/Topic")
private static Topic topic;
```

Połączenie (ang. connection)

- Reprezentacja wirtualnego połączenia z dostawcą JMS
- Połączenie jest potrzebne do zainicjowania sesji
- Połączenie musi być jawnie zamknięte na końcu aplikacji (zwolnienie zasobów, m.in. otwartych sesji)
- Rozpoczęcie odbioru wiadomości: metoda start()
- Wstrzymanie odbioru wiadomości: metoda stop()

```
Connection connection =
  connFactory.createConnection();
connection.start();
...
connection.close();
```

Sesje

- Jednowątkowy kontekst do tworzenia i odbioru wiadomości
- Sesje tworzą:
  - wiadomości
  - producentów (nadawców) wiadomości
  - konsumentów (odbiorców) wiadomości

```
Session session = connection.createSession(false,
  Session.AUTO_ACKNOWLEDGE);
```

- Pierwszy argument wskazuje czy ma być tworzona transakcja

Producenci wiadomości

```
@Resource(mappedName="jms/Queue")
private static Queue queue;

MessageProducer producer =
session.createProducer(queue);
producer.send(message);
```

- Wysyłanie do dowolnej kolejki:

```
MessageProducer producer =
session.createProducer(null);
producer.send(queue, message);
```

Odbiorcy wiadomości

```
@Resource(mappedName="jms/Queue")
private static Queue queue;

MessageConsumer consumer = session.createConsumer(queue);
Message m1 = consumer.receive();
Message m2 = consumer.receive(1000);
```

- Message listeners:

```
MessageListener myListener = new AListener();
consumer.setMessageListener(myListener);
```

- Listener ma metodę onMessage(Message). Musi obsługiwać wszystkie wyjątki.

Filtry wiadomości

- Filtr jest wyrażeniem zapisywanym jak warunki w SQL92
  - typ = 'wyniki' and id = '120'
- Wyrażenie odwołuje się do właściwości wiadomości
- Filtr może być parametrem tworzenia konsumenta wiadomości

Wiadomości

- Wiadomości składają się z
  - nagłówka
  - listy właściwości
  - ciała
- Standardowe właściwości (przechowywane w nagłówku)
  - identyfikator JMSMessageID
  - odbiorca JMSDestination
  - znacznik czasowy JMSTimestamp
  - priorytet JMSPriority
  - typ JMSType

Zawartość wiadomości (body)

- TextMessage – wiadomość tekstowa (np. dok. XML)
- MapMessage – zbiór par nazwa-wartość (String i typ prymitywny)
- BytesMessage – nieinterpretowany strumień bajtów
- StreamMessage – strumień wartości prymitywnych typów
- ObjectMessage – serializowalny obiekt Javy
- Message – pusta zawartość

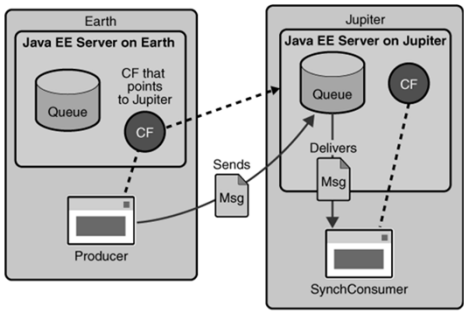
```
TextMessage message = session.createTextMessage();
message.setText("Ala ma kota");
producer.send(message)
...
Message m = consumer.receive();
if (m instanceof TextMessage) { ... m.getText(); }
```

Przeglądanie zawartości kolejki

- Interface QueueBrowser
- Możliwość zastosowania filtru
- Nie można przeglądać topic'ów (wiadomości znikają)

```
QueueBrowser browser =
  session.createBrowser(queue);
Enumeration msgs = browser.getEnumeration();
while (msgs.hasMoreElements()) {
  Message m = (Message)msgs.nextElement();
  ...
}
```

Przesyłanie wiadomości między systemami



Niezawodność komunikacji

- Potwierdzenia wiadomości
- Trwałość komunikatów – awarie dostawców JMS
- Priorytety komunikatów
- Czas życia komunikatów
- Tymczasowe kolejki

Potwierdzenia

- Po odbiorze wiadomości
- Po przetworzeniu wiadomości
- Po odbiorze potwierdzenia
- Wiadomości są automatycznie potwierdzane po zakończeniu transakcji
- Wycofanie transakcji → ponowne dostarczenie

- AUTO_ACKNOWLEDGE – po odbiorze
- CLIENT_ACKNOWLEDGE – jawne potwierdzenie wszystkich odebranych wiadomości w ramach sesji
- DUPS_OK_ACKOWLEDGE – leniwe potwierdzanie z możliwością powstawania duplikatów

Potwierdzenie

- Wiadomości niepotwierdzone przed końcem sesji są dostarczane ponownie (przy kolejnym połączeniu)
- Przetrzymywanie niepotwierdzonych wiadomości dla trwałych subskrypcji

Potwierdzenie tylko przetworzonych wiadomości:

- asynchroniczny odbiór i tryb AUTO_ACKNOWLEDGE
- odbiór synchroniczny i tryb CLIENT_ACKNOWLEDGE
- odbiór synchroniczny w trybie AUTO_ACKNOWLEDGE powoduje natychmiastowe potwierdzanie (przed przetworzeniem)

Trwałość wiadomości

- Tryb PERSISTENT – każda wiadomość jest rejestrowana w pamięci trwałej (tryb domyślny)
- Tryb NON_PERSISTENT – wiadomość może zostać utracona w przypadku awarii dostawcy JMS (większa wydajność)

- Własność trwałości może być ustawiana dla producenta wiadomości lub dla pojedynczej wiadomości

```
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
producer.send(msg, DeliveryMode.NON_PERSISTENT,3,10000);
```

Priorytety wiadomości

- Priorytet może być ustawiany dla producenta wiadomości lub dla pojedynczej wiadomości
- 0 – najniższy priorytet, 9 – najwyższy, domyślnie 4
- Priorytet określa preferencje i nie decyduje o bezwzględnej kolejności dostarczania

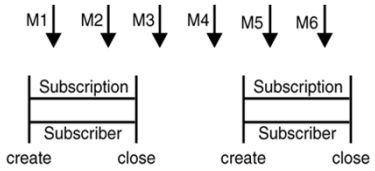
```
producer.setPriority(5);
producer.send(msg,
DeliveryMode.NON_PERSISTENT,5,10000);
```

Przedawnianie wiadomości

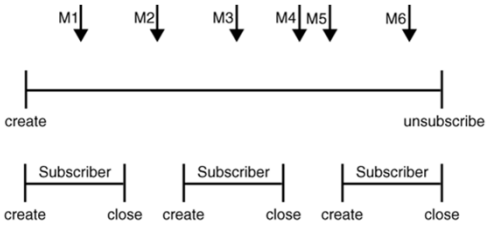
- Czas życia może być ustawiany dla producenta wiadomości lub dla pojedynczej wiadomości
- Domyślnie wiadomości nie ulegają przedawnieniu
- Czas życia 0 oznacza brak przedawniania

```
producer.setTimeToLive(10000);
producer.send(msg,
DeliveryMode.NON_PERSISTENT,5,10000);
```

Trwałe subskrypcje



Trwałe subskrypcje (2)



Trwałe subskrypcje (3)

- Identyfikacja subskrypcji
  - identyfikator klienta
  - topic
  - nazwa subskrypcji

```
MessageConsumer topicSubscriber =
       session.createDurableSubscriber(myTopic,
"MySub");
...
topicSubscriber.close();
...
session.unsubscribe("MySub");
```

Lokalne transakcje

- Grupowanie operacji wysyłania/odbioru w transakcji
- Metody Session.commit() i Session.rollback()
- Zatwierdzenie oznacza wysłanie wyprodukowanych wiadomości i potwierdzenie odebranych
- Wycofanie oznacza usunięcie wyprodukowanych wiadomości i ponowne dostarczenie wiadomości odebranych (z pominięciem przedawnionych)
- Uwaga na zakleszczenia: wysłanie następuje po zatwierdzeniu

Kolejki tymczasowe

- Utworzenie kolejki tymczasowej
  `TemporaryQueue tq = session.createTemporaryQueue();`
- Dołączenie kolejki tymczasowej do komunikatu
  `msg.setJMSReplyTo(tq);`
- Uzyskiwanie dostępu do kolejki tymczasowej (po odebraniu komunikatu)
  `tq = msg.getJMSReplyTo();`

ZeroMQ

- Mechanizm komunikacji ze zintegrowanym kolejkowaniem komunikatów (brak brokera)
- Interfejs wzorowany na gniazdach BSD
- Interfejs dostępny dla między innymi dla języka C, C++, C#, Java, Python, Ruby, Ada
- Zdefiniowane schematy komunikacji (REQ-REP, PUB-SUB, PUSH-PULL)
- Brak struktury komunikatu (komunikat jest sekwencją bajtów, których typ/struktura zdefiniowana jest na poziomie aplikacji)

Ogólny schemat komunikacji

Proces 1

1. utworzenie kontekstu
2. utworzenie gniazda
3. połączenie gniazda
4. wysłanie komunikatu
5. inne operacje komunikacyjne
6. zamknięcie gniazda
7. usunięcie kontekstu

Proces 2

1. utworzenie kontekstu
2. utworzenie gniazda
3. związanie gniazda
4. odbiór komunikatu
5. inne operacje komunikacyjne
6. zamknięcie gniazda
7. usunięcie kontekstu

Kontekst

- Kontekst jest zbiorem gniazd.
- Tworzenie kontekstu
  `void *context = zmq_ctx_new ();`
- Zalecenie: 1 kontekst na proces.
- Usuwanie kontekstu
  `zmq_ctx_destroy (context);`
  lub
  `zmq_ctx_term (context);`
  - zakończenie wszystkich operacji blokujących
  - oczekiwanie na zakończenie wysyłania komunikatów
  - oczekiwanie na zamknięcie wszystkich gniazd

Tworzenie gniazda

- Wywołanie:
  `void *zmq_socket (void *context, int type);`
- Podstawowe typu gniazd:
  - REQ i REP (naprzemienne wysyłanie i odbiór)
  - PUB i SUB (publikowanie do wielu subskrybentów)
  - PUSH i PULL (przetwarzanie potokowe)

Wiązanie i łączenie gniazd

- Wiązanie gniazda:
  ```
  void * gniazdo = zmq_socket (…, …);
  zmq_bind (gniazdo, "tcp://*:5556");
  ```
- Łączenie gniazd:
  ```
  void * gniazdo = zmq_socket (…, …);
  zmq_connect (gniazdo, "tcp://*:5556");
  ```
- Łączyć można tylko „kompatybilne” gniazda.

Komunikacja

- Wysyłanie wiadomości:
```
int zmq_send (void *socket, void *buf,
size_t len, int flags);
int zmq_sendmsg (void *socket,
zmq_msg_t *msg, int flags);
```
- Odbiór wiadomości:
```
int zmq_recv (void *socket, void *buf,
size_t len, int flags);
int zmq_recvmsg (void *socket,
zmq_msg_t *msg, int flags);
```

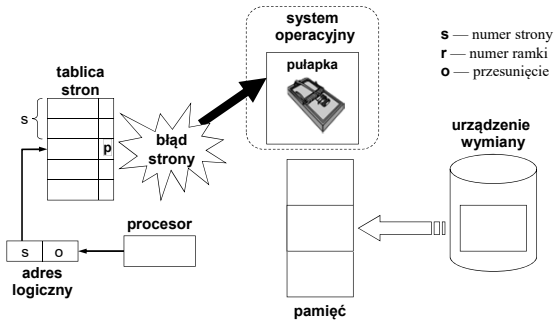
Rozproszona pamięć współdzielona (ang. Distributed Shared Memory)

DSM — podstawowe cechy

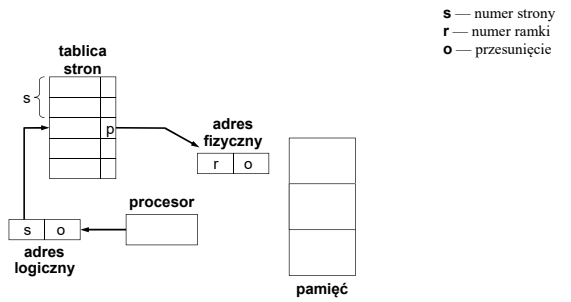
Cel: dostarczenie wspólnej wirtualnej przestrzeni adresowej, dostępnej (potencjalnie) dla wszystkich węzłów systemu

Zalety:
- wygodny dla programisty paradygmat programowania równoległego,
- dostępność wirtualnej przestrzeni adresowej obejmującej pamięci fizyczne wszystkich węzłów,
- możliwość uruchamiania w środowisku rozproszonym programów równoległych zaprojektowanych dla środowiska wieloprocesorowego z pamięcią współdzieloną

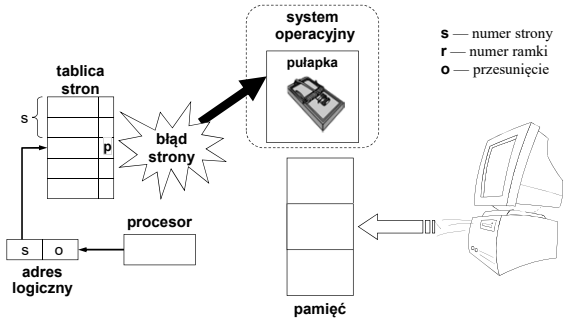
Obsługa błędu strony w systemie pamięci wirtualnej



Obsługa błędu strony w systemie pamięci wirtualnej



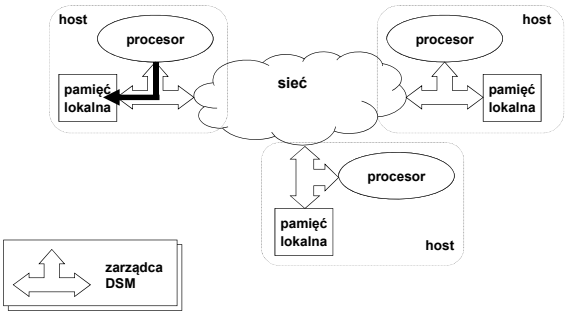
Obsługa błędu strony w systemie wirtualnej pamięci rozproszonej



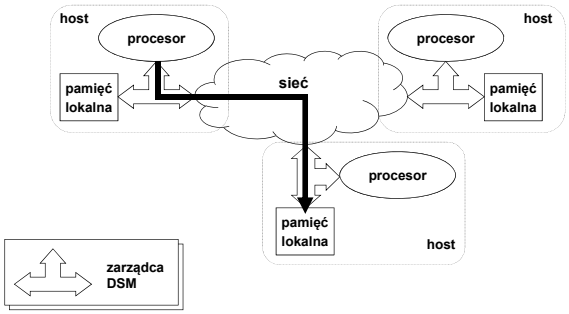
Koncepcje dostępu do danych

- ***Dostęp zdalny*** — każdy dostęp do współdzielonego obiektu, zlokalizowanego fizycznie w pamięci lokalnej innego węzła, odbywa się przez sieć.
- ***Relokacja*** — możliwa jest zmiana fizycznej lokalizacji współdzielonego obiektu, czyli umieszczenie go w pamięci lokalnej węzła, w którym pojawiło się żądanie dostępu.
- ***Replikacja*** — obiekt logiczny może być jednocześnie zlokalizowany fizycznie w pamięci lokalnej wielu węzłów, co umożliwia równoległy dostęp do tego obiektu w wielu węzłach.

Dostęp zdalny



Dostęp zdalny

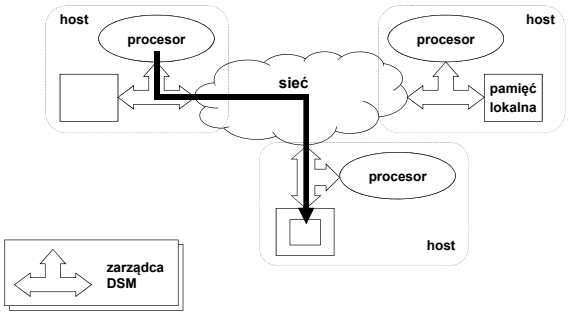


Zdalny dostęp — charakterystyka

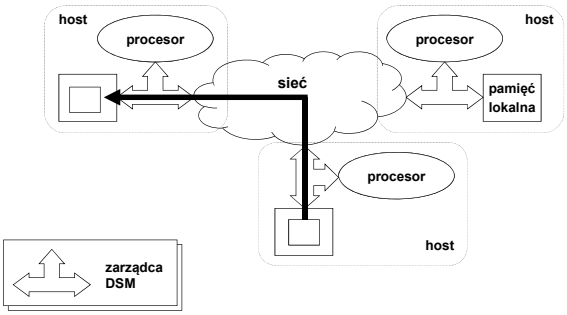
- Stosunkowo prosta realizacja

- Problem efektywności — duży czas dostępu do danych w zdalnych węzłach

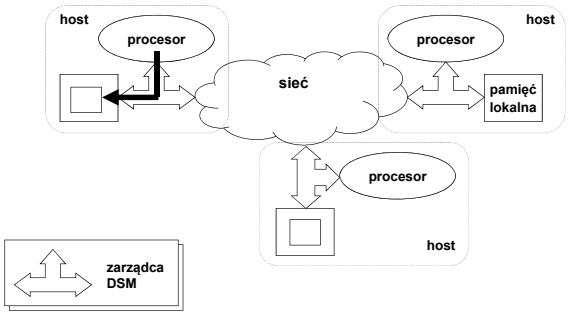
Relokacja



Relokacja



Relokacja

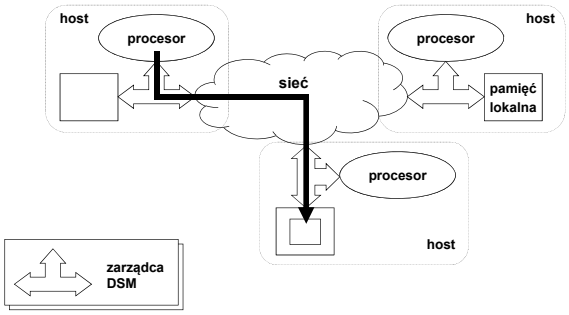


Relokacja — charakterystyka

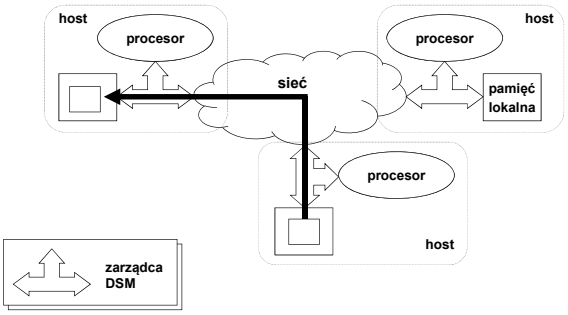
- Problem lokalizacji
- Problem rozmiaru i struktury jednostki podlegającej relokacji

- Problem migotania (ang. trashing, ping-pong effect)

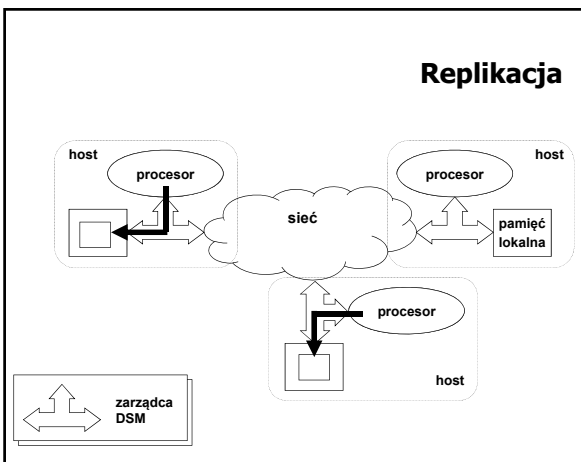
Replikacja



Replikacja



Replikacja



Replikacja — charakterystyka

- Problem lokalizacji

- Problem rozmiaru i struktury jednostki podlegającej replikacji

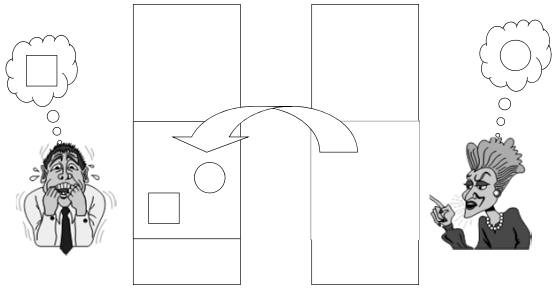
- Problem migotania (ang. trashing, ping-pong effect)

- Problem spójności kopii (replik)

Struktura jednostki podlegającej replikacji lub relokacji

- ***Strona*** — fizyczne połączenie kilku odrębnych obiektów logicznych w jedną jednostkę udostępnianą jako całość przez DSM (problem fałszywego współdzielenia).
- ***Pojedyncza zmienna*** — duży jednostkowy koszt relokacji i utrzymywania spójności.
- ***Obiekt*** — możliwość optymalizacji w strategii utrzymywania spójności w związku ze ściśle określonym sposobem dostępu (poprzez metody).

Fałszywe współdzielenie



Problem spójności replik — protokół koherencji

- Protokół ***unieważniania*** danych (ang. invalidation protocol) — niespójne repliki są usuwane z pamięci lokalnej.
- Protokół ***aktualizacji*** danych (ang. update protocol) — niespójne repliki są aktualizowane.

Problem lokalizacji — system IVY

- ***statyczny scentralizowany*** mechanizm lokalizacji stron
- ***statyczny rozproszony*** mechanizm lokalizacji stron
- ***dynamiczny*** mechanizm lokalizacji stron

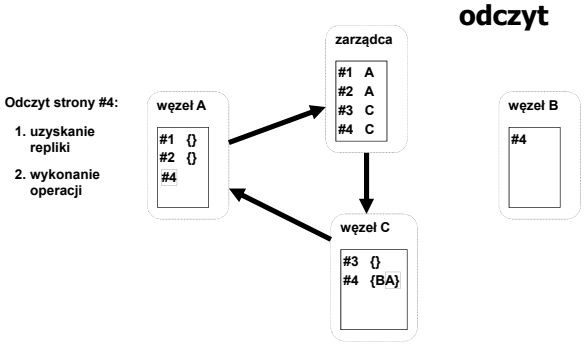
System IVY — podstawowe pojęcia i struktury danych (1)

- właściciel strony — węzeł, na którym była wykonywana ostatnia operacja zapisu danej strony
- zbiór kopii (copyset) — zawiera identyfikatory węzłów posiadających kopię strony (przechowywana przez właściciela strony)

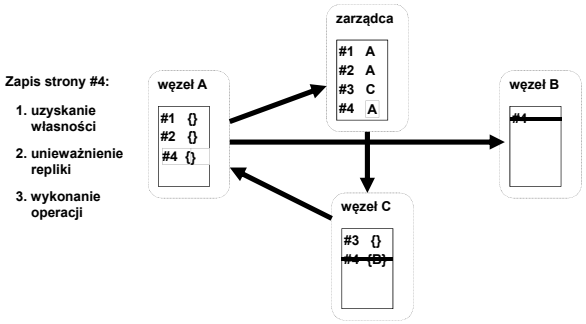
System IVY — podstawowe pojęcia i struktury danych (2)

- zarządca (w podejściu statycznym) — węzeł, który przechowuje dane o właścicielach poszczególnych stron
- tablica właścicieli stron — dla każdej strony zawiera identyfikator jej właściciela (przechowywany przez zarządców)
- prawdopodobny właściciel (w podejściu dynamicznym) — tablica zawierająca dla każdej strony w systemie identyfikator węzła, o którym wiadomo, że był kiedyś (lub jeszcze jest) właścicielem danej strony (przechowywana przez każdy węzeł)

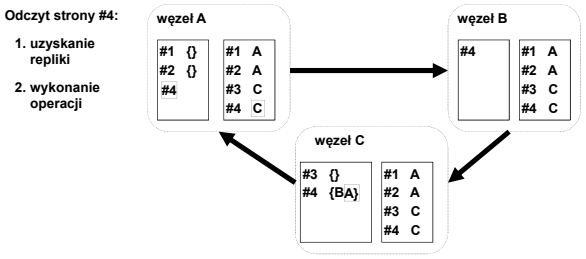
Statyczny scentralizowany mechanizm lokalizacji stron — odczyt



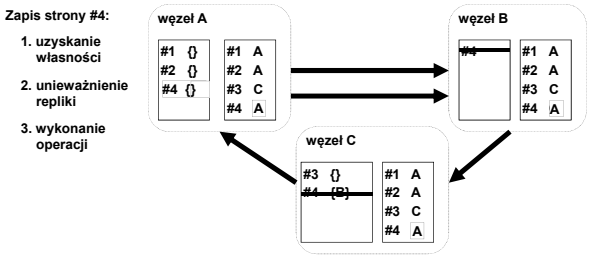
Statyczny scentralizowany mechanizm lokalizacji stron — zapis



Dynamiczny mechanizm lokalizacji stron — odczyt



Dynamiczny mechanizm lokalizacji stron — zapis



Przestrzeń krotek

Linda — ogólna koncepcja

- Mechanizm komunikacji międzyprocesowej zaproponowany przez Davida Gelerntera w 1985
- Luźne powiązanie komunikujących się procesów (nie muszą znać się wzajemnie, nie muszą działać jednocześnie — komunikacja nieustanna)
- Asocjacyjna identyfikacja komunikatów (w odróżnieniu do kolejkowania) we współdzielonej przestrzeni
- Współczesne implementacje:
  - JavaSpaces — Sun Microsystems (w ramach technologii Jini, projekt przejęty przez ASF — Apache River), komercyjna implementacja dostarczana przez Gigaspaces
  - TSpaces — IBM
  - Tuples On The Air (TOTA),
  - L2imbo

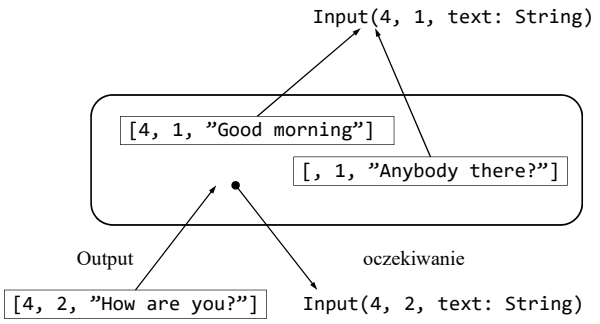
Linda — podstawowe pojęcia

- Krotka — uporządkowana kolekcja danych określonych typów — atrybutów, przy czym atrybuty mogą (ale nie muszą) mieć nadaną konkretną wartość
- Przestrzeń krotek — wspólne miejsce dostępne dla kooperujących procesów, gdzie gromadzone są krotki
- Interfejs dostępu do przestrzeni krotek
  - Output — umieszczanie krotki w przestrzeni
  - Input — pobieranie krotki z przestrzeni
  - Read — odczytywanie krotki bez pobierania (odczytana krotka w dalszym ciągu pozostaje w przestrzeni)
  - Try_Input, Try_Read — nieblokujące wersji Input i Read

Linda — realizacja operacji dostępu

- Krotka staje się dostępna w przestrzeni po wykonaniu operacji Output, której parametrami są wartości atrybutów, np.:
  `Output(4, 1, "Good morning")`
- Operacja Input powoduje pobranie (usunięcie z przestrzeni) krotki, której wartości atrybutów są zgodne z parametrami operacji, np.:
  `Input(4, 1, text: String)`
  Wartości pozostałych atrybutów zostaną nadane zgodnie z zawartością pobranej krotki.
- Jeśli krotka została umieszczona w przestrzeni bez podania wartości któregoś z atrybutów (np.
  `Output(, 1, "Anybody there?")`
  może ona zostać pobrana przez Input z dowolną wartością tego atrybutu lub pominięciem tej wartości.

Linda — operacje na przestrzeni krotek



Linda — specyfikacja komunikacji

- Przestrzeń krotek jest zbiorem (nie kolejką) — krotki nie są uporządkowane i mogą być odbierane w innej kolejności niż były umieszczane.
- W realizacji operacji dostępu gwarantowana jest ogólnie rozumiana żywotność:
  - Jeśli procesy czekają (w operacji Input) na krotkę, to przy odpowiednio dużej liczbie umieszczonych krotek z oczekiwanymi wartościami atrybutów każdy w końcu ją otrzyma.
  - Jeśli krotka jest w przestrzeni, to po odpowiednio dużej liczbie operacji Input (ze zgodnymi parametrami) zostanie w końcu odczytana.

JavaSpaces (przestrzeń obiektów)

- Odpowiednikiem krotki jest obiekt klasy implementującej interfejs `Entry`.
- Zmienne instancji obiektu-krotki muszą być publiczne.
- Obiekt-krotka ma ustalony czas życia w przestrzeni (czas wynajmowania przestrzeni — lease time).
- Czas życia można zwiększać nawet po umieszczeniu obiektu-krotki w przestrzeni.
- Operacje na przestrzeni krotek realizowane są poprzez metody `write`, `take`, `read`, `takeIfExists`, `readIfExists` interfejsu `JavaSpace`.

Przykład definicji klasy dla obiektu-krotki

```
public class Message implements Entry {
   public Integer id;
   public Integer num;
   public String text;

   public Message () {}
   public Message (Integer i, Integer n, String t){
      id = i;
      num = n;
      text = t;
   }
}
```

Przykład umieszczania krotki w przestrzeni

```
JavaSpace space = getSpace();
msg = new Message ( new Integer(4),
                    new Integer(counter++),
                    ”Good morning”);
Lease l = space.write(msg, null,
6*60*60*1000);
```

Przykład pobierania krotki z przestrzeni

```
JavaSpace space = getSpace();
template = new Message ( new Integer(4),
                         null,
                         null);
Message msg = (Message)space.read( template,
                       null, 60*60*1000);
```