Systemy rozproszone

Procesy

dr inż. Jacek Czerniak

jczerniak@ukw.edu.pl

Plan wykładu:

- Wątki
- Klienci
- Serwery
- Migracja kodu
- Agenci programowi

Procesy w środowiskach rozproszonych

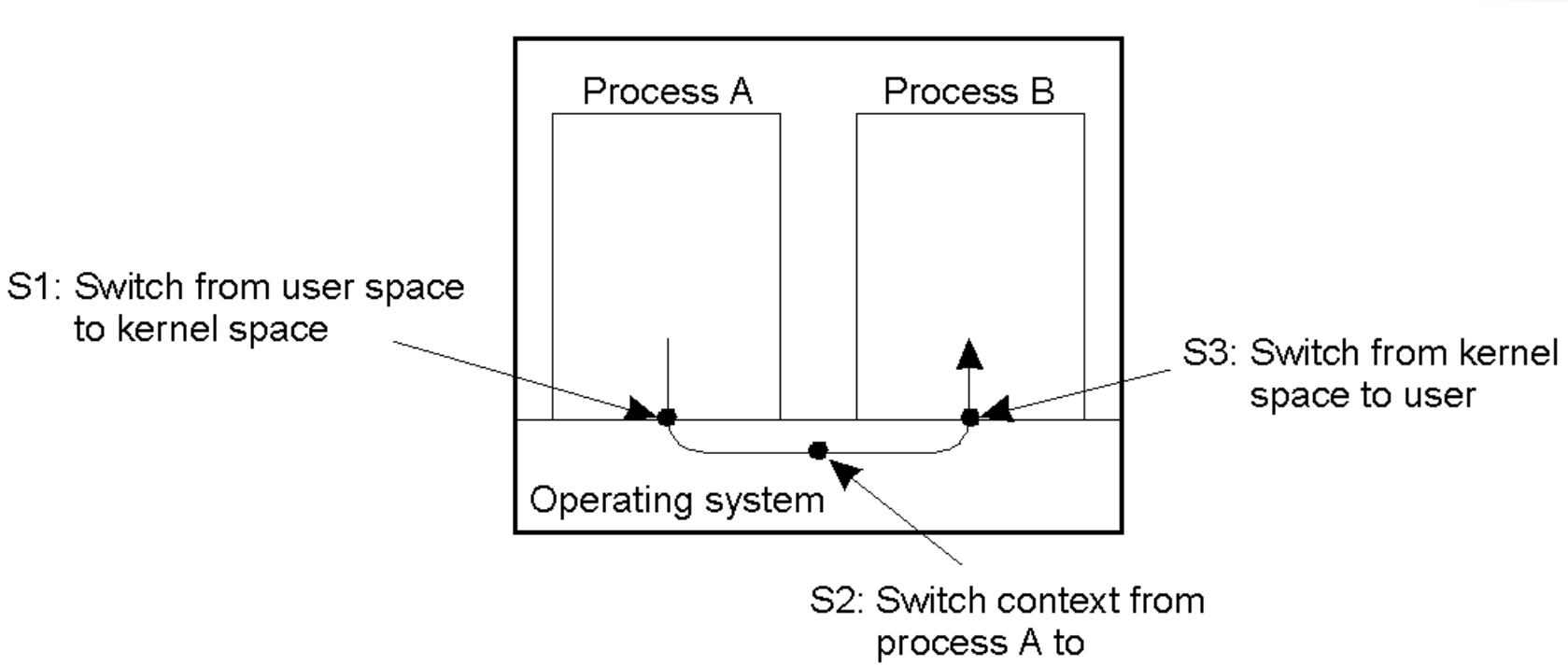
- ***Proces*** –- program w trakcie wykonania w otaczającym środowisku.
  - kluczowe kwestie w scentralizowanych systemach: zarządzanie procesami i scheduling,
  - w systemach rozproszonych, ponadto: techniki wielowątkowe, model klient-serwer, przenoszenie procesów między różnymi maszynami, agenci programowe.
- ***Agenci programowe*** –- zbiór równie ważnych agentów, które wspólnie próbują osiągnąć wspólny cel.
-

Wątki

- Procesy - przeźroczystość współbieżności dość wysokim kosztem.
- tworzenie procesu,
- alokacja pamięci,
- inicjalizacja pamięci,
- kosztowne przełączanie kontekstu CPU.

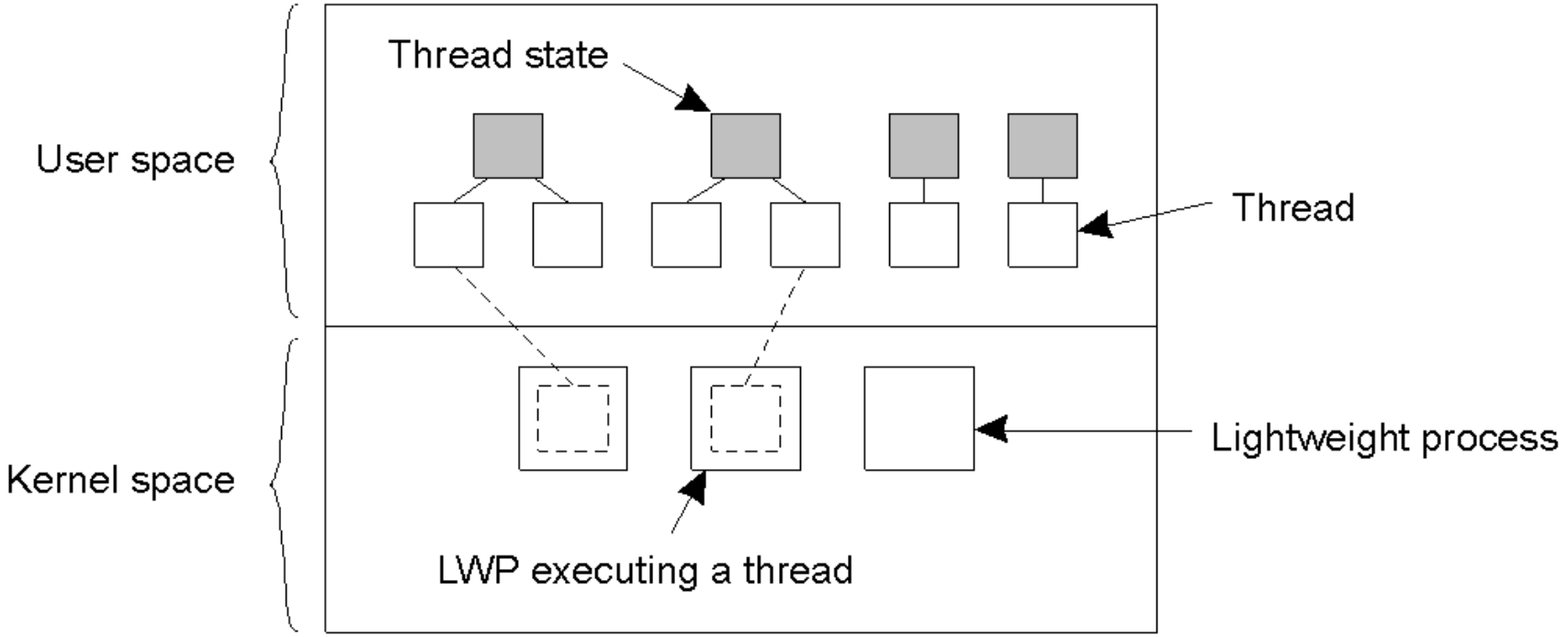
- Kontekst wątku składa się zazwyczaj z kontekstu CPU.
- szybkie przełączanie,
- wykorzystanie równoległości w systemach wieloprocesorowych,
- znacznie szybsze mechanizmy "IPC",
- wiele aplikacji struktury łatwiej ustrukturalizować jako zbiór współpracujących ze sobą wątków.

Wykorzystanie wątków w systemach nierozproszonych



- Przełączanie kontekstu jako wynik IPC

Implementacja wątków



- Kombinacja lekkich procesów poziomu jądra i wątków poziomu użytkownika

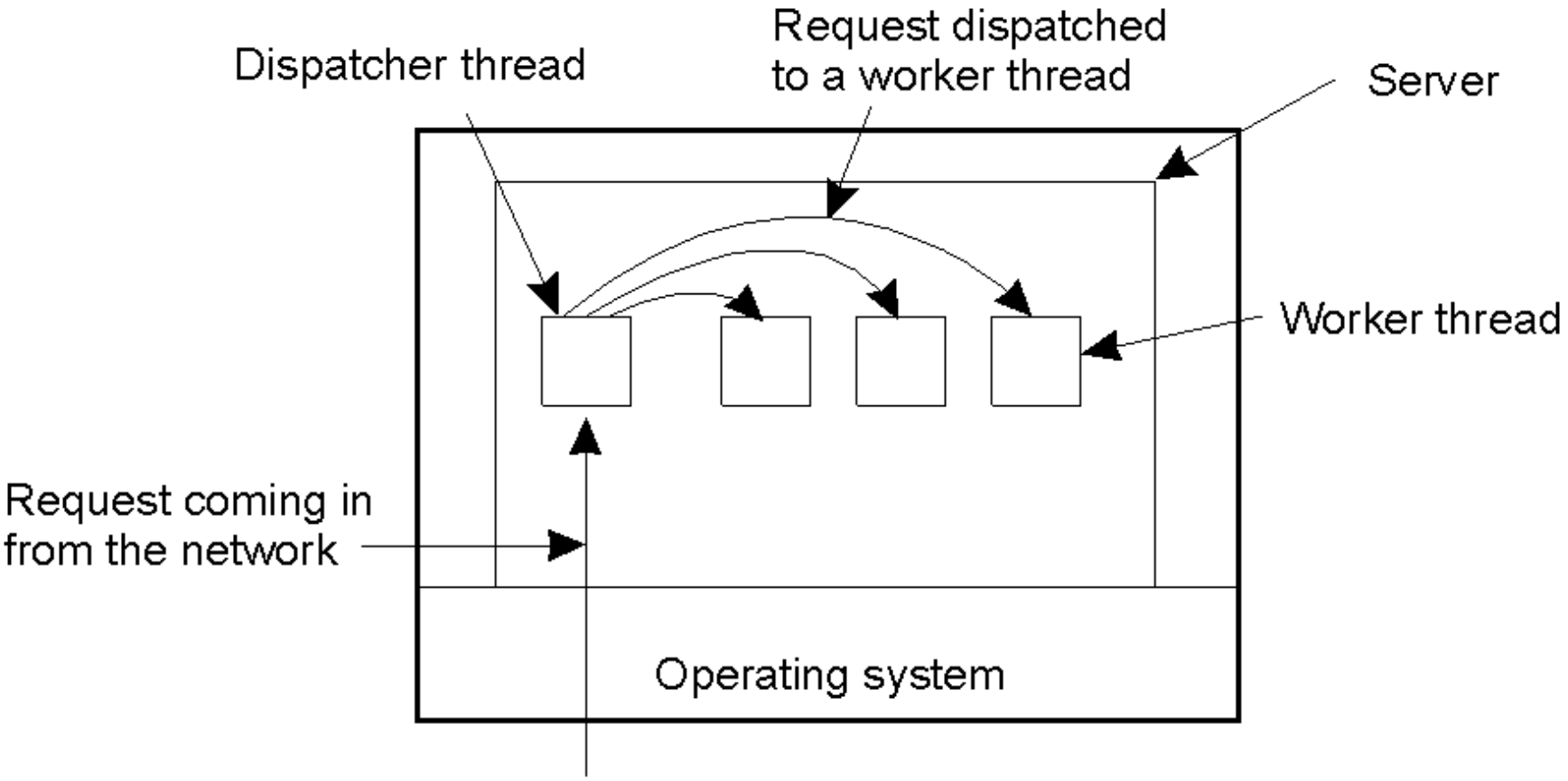
- ***LWP*** – lekkie procesy,
- wykonuje się w kontekście jednego (ciężkiego) procesu,
- Mmożliwe kilka LWP na proces,
- dodatkowy pakiet obsługi wątków poziomu użytkownika,
- muteksy i zmienne warunkowe dla synchronizacji.

Aktywacje planisty

- przełączanie kontekstu realizowane w całości w przestrzeni użytkownika, jawi się LWP jak normalny kodu programu,
- jedynym mankamentem LWP: konieczność stworzenia LWP i niszczenia, które są tak kosztowne, jak wątki poziomu jądra.

- Alternatywne podejście do LWP: ***aktywacje planisty (scheduler'a)***
- jeśli wątek blokuje się na wywołaniu systemowym, jądro nie wykonuje wywołania pakietu obsługi wątków,
- to samo, gdy wątek, że jest odblokowywany,
- zaleta: oszczędność zarządzania LWP przez jądro,
- wada: narusza struktury warstwowych systemów.
-

Serwery wielowątkowe (1)



- Serwer wielowątkowy zorganizowany w modelu dispatcher/worker.

Serwery wielowątkowe (2)

Zakłada się:

- wątki niedostępne,
- strata wydajności ze względu na wielowątkowość niedopuszczalna.

Rozwiązanie: duża *maszyna stanów skończonych* (*finite-state machine)*

- napływające żądania rejestruje się w tablicy,
- gotowa do odbioru następnych komunikatów (żądania lub odpowiedzi),
- wymaga wykorzystania nieblokujących wywołań *send* i *receive*.

Co wnoszą wątki:

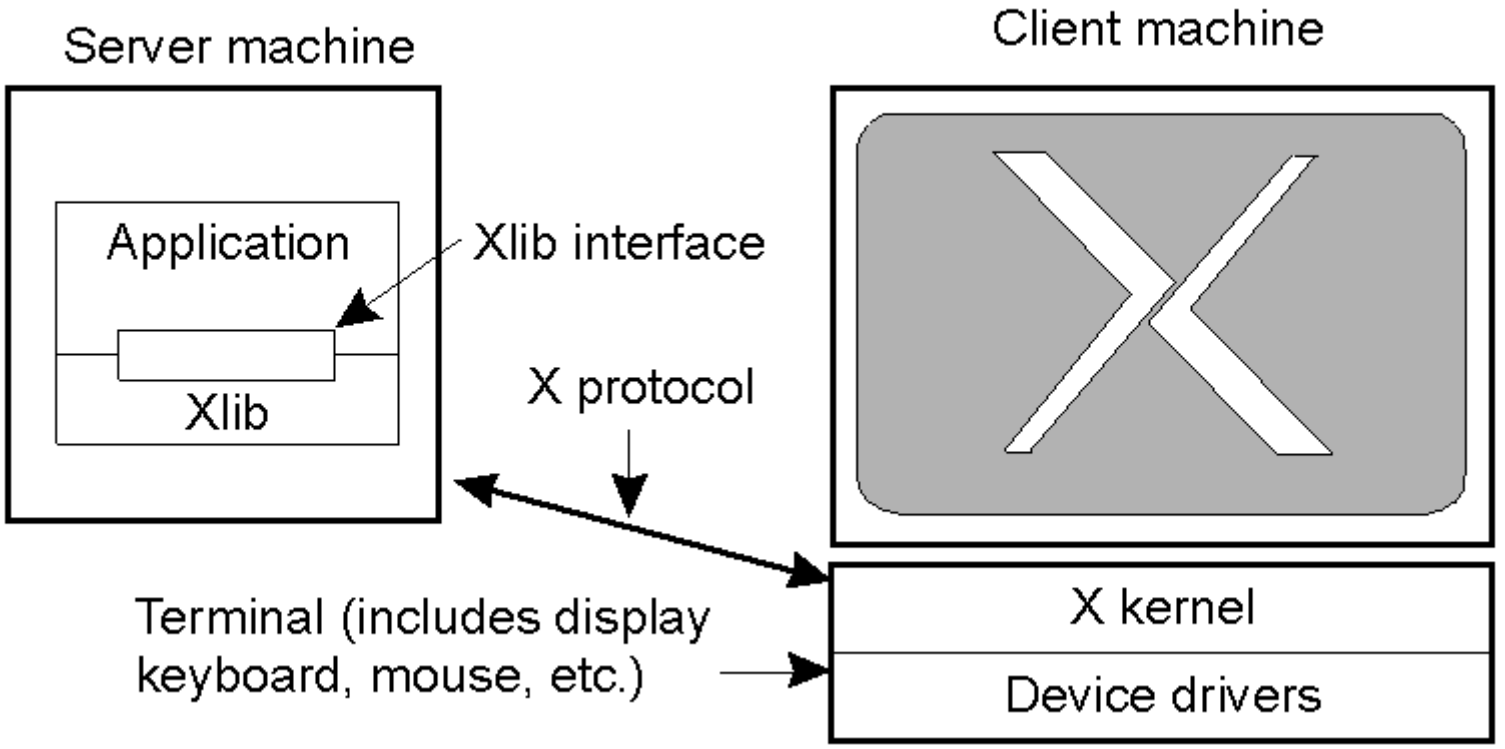
- podtrzymują ideę procesów sekwencyjnych gdzie wywołania systemowe są blokujące, ale osiąga się równoległość.

Serwery wielowątkowe (3)

Model	Characterystyka
Wątki	Równoległość, blokujące wywołania systemowe
Procesy jednowątkowe	Brak równoległości, blokujące wywołania systemowe
Maszyna stanów skończonych	Równoległość, nieblokujące wywołania systemowe

- Trzy sposoby budowy serwera

System X-Window



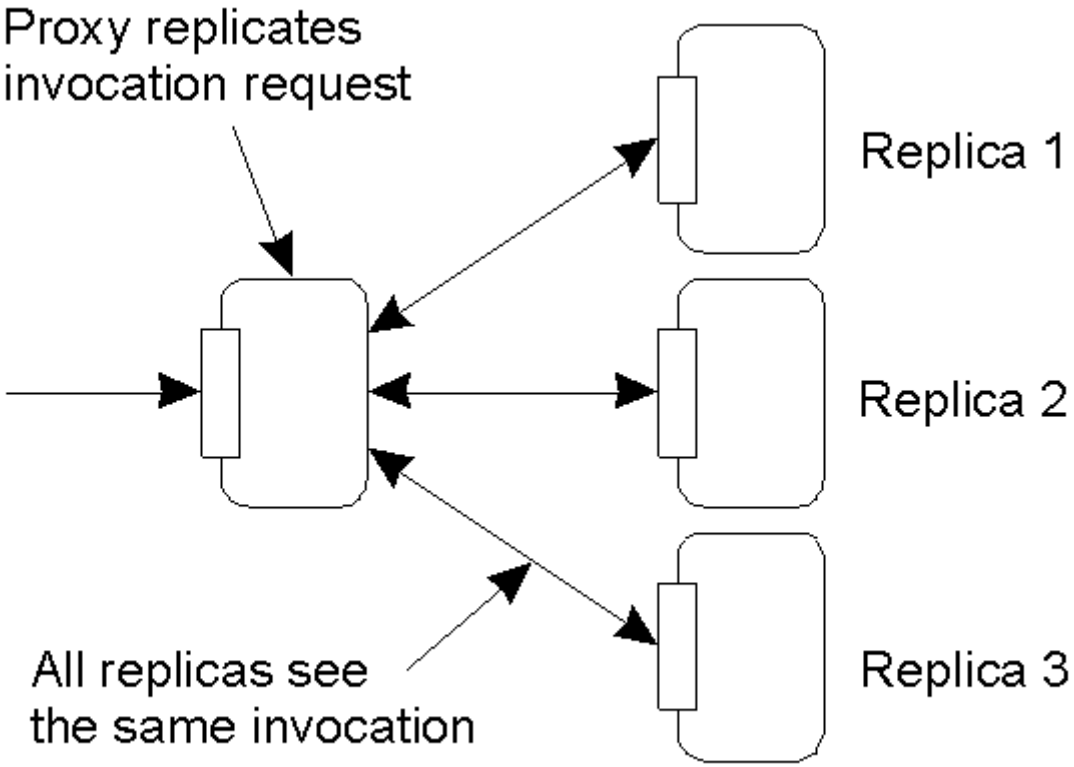
- Organizacja Systemu X Window

Oprogramowanie klienckie dla przezroczystości rozproszenia (1)

- Elementy rozproszenia ze wsparciem ze strony klienta:
- przezroczystość dostępu,
- przezroczystość lokalizacji,
- przezroczystość migracji,
- przezroczystość przemieszczania,
- przezroczystość awarii.

- Bez wsparcia ze strony klienta:
- przezroczystość współbieżności (specjalne pośrednie serwery – ***monitory transakcyjne***),
- przezroczystość persystencji często całkowicie obsługiwana na serwerze.

Oprogramowanie klienckie dla przezroczystości rozproszenia(2)

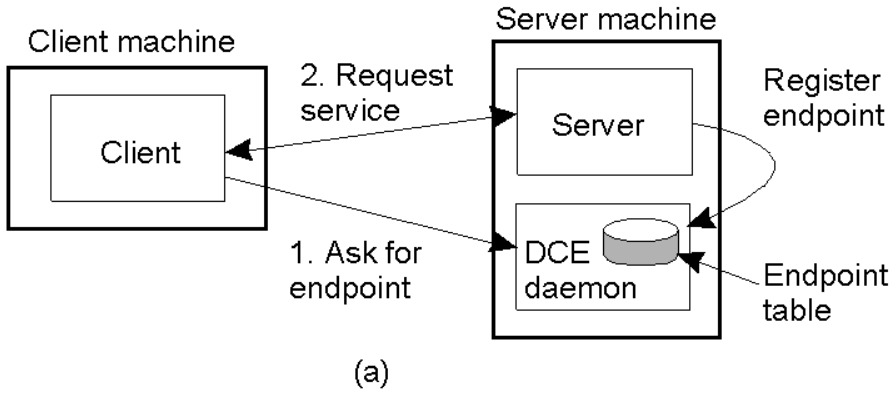


- Podejście do przezroczystej replikacji zdalnego obiektu z wykorzystaniem rozwiązania po stronie klienta

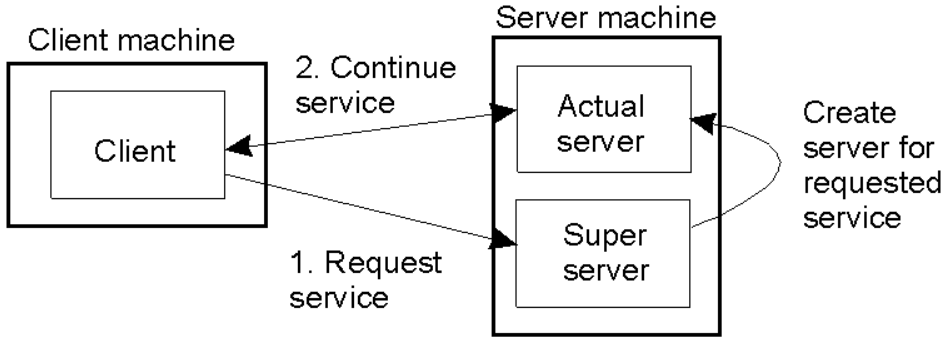
Serwery: zagadnienia projektowe (1)

- typy organizacji serwerów:
  - ***serwer iteracyjny*** – sam obsługuje żądania,
  - ***serwer współbieżny*** – przekazuje żądania do osobnego wątku lub procesu,
- endpoint-y, porty:
  - dobrze znane usługi, określone przez IANA (Internet Assigned Numbers Authority),
- super-serwery,
- dane *out-of-band* z:
  - osobnym endpointem sterującym,
  - pilne dane w TCP
- serwery bezstanowe a stanowe,
- równorzędne procedury,
- serwery internetowe i „ciasteczka”

Serwery: zagadnienia projektowe (2)



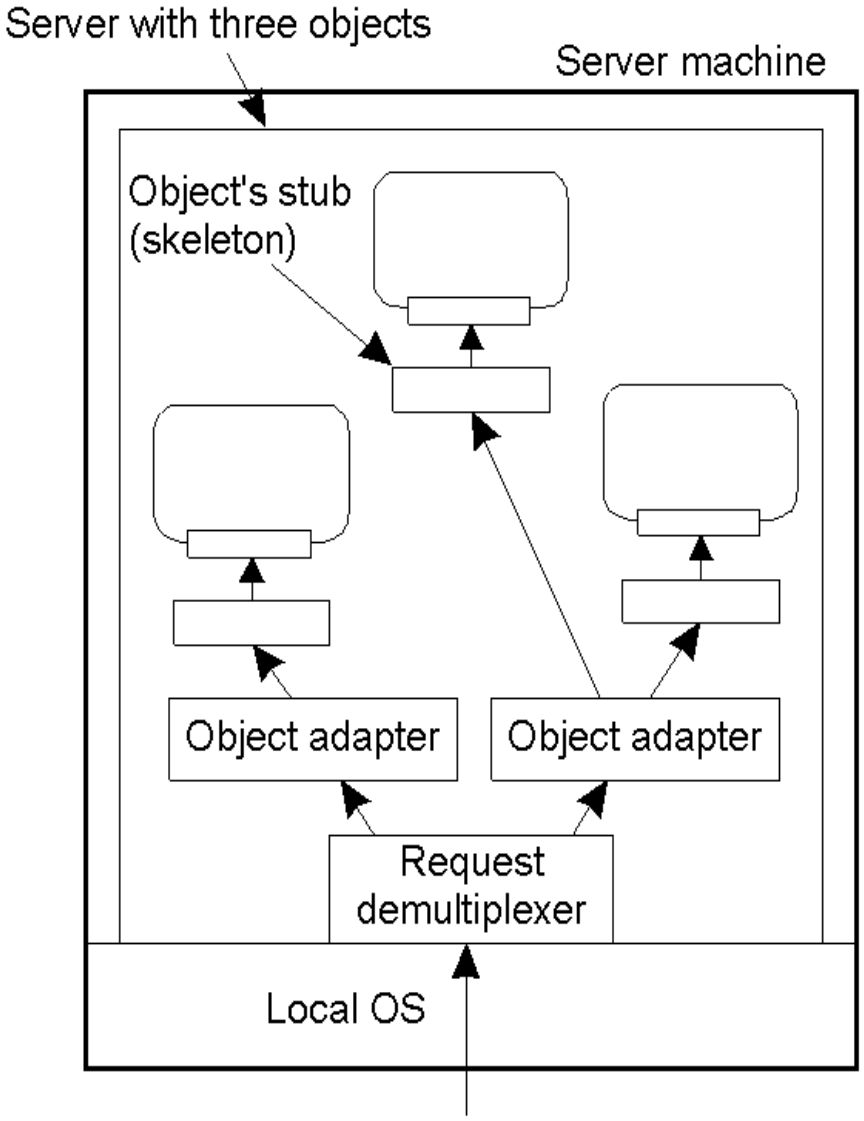
(a)



(b)

a) Wiązanie klient-serwer z użyciem demona jak w DCE
b) Wiązanie klient-serwer z użyciem super-serwera jak w UNIX-ie

Serwery obiektów - Adapter



Organizacja serwera obiektów ze wsparciem dla różnych polityk aktywacji.

Serwery obiektów - Adapter

Polityka aktywacji:

- decyzja jak wywołać obiekt,
- obiekt musi być najpierw umieszczony w przestrzeni adresowej serwera (tj., najpierw aktywowany, następnie wywoływany)
- adaptery obiektów nieświadome specyficznych interfejsów obiektów, które kontrolują,
- żądania nie są przekazywane od razu do obiektów,
- adapter dostarcza hands żądanie wywołania do stub'u obiektu po stronie serwera,

Przykład oczekiwanej operacji adaptera obiektów implementowanej przez każdy skeleton specyficzny dla obiektu:

- **invoke**(unsigned **in_size**, char **in_args**[], unsigned* **out_size**, char* **out_args**[])

Rejestracja obiektu:

- przekazuje wskaźnik do specyficznej dla obiektu implementacji procedury *invoke*,
- zwraca id obiektu względem adaptera.

Adapter obiektu (2)

```
/* Definitions needed by caller of adapter and adapter */
#define TRUE
#define MAX_DATA 65536

/* Definition of general message format */
struct message {
    long  source                /* senders identity */
    long  object_id;            /* identifier for the requested object */
    long  method_id;            /* identifier for the requested method  */
    unsigned  size;             /* total bytes in list of parameters */
    char  **data;               /* parameters as sequence of bytes */
};

/* General definition of operation to be called at skeleton of object */
typedef void (*METHOD_CALL)(unsigned, char* unsigned*, char**);

long register_object (METHOD_CALL call);      /* register an object  */
void unrigester_object (long object_id);      /* unregister an object */
void invoke_adapter (message *request);       /* call the adapter */
```

- Plik *header.h* używany przez adapter i dowolny program wywołujący ten adapter.

Adapter obiektu (3)

```
typedef struct thread THREAD;                    /* hidden definition of a thread */

thread *CREATE_THREAD (void (*body)(long tid), long thread_id);
/* Create a thread by giving a pointer to a function that defines the actual  */
/* behavior of the thread, along with a thread identifier  */

void get_msg (unsigned *size, char **data);
void put_msg(THREAD *receiver, unsigned size, char **data);
/* Calling get_msg blocks the thread until of a message has been put into its  */
/* associated buffer.  Putting a message in a thread's buffer is a nonblocking */
/* operation. */
```

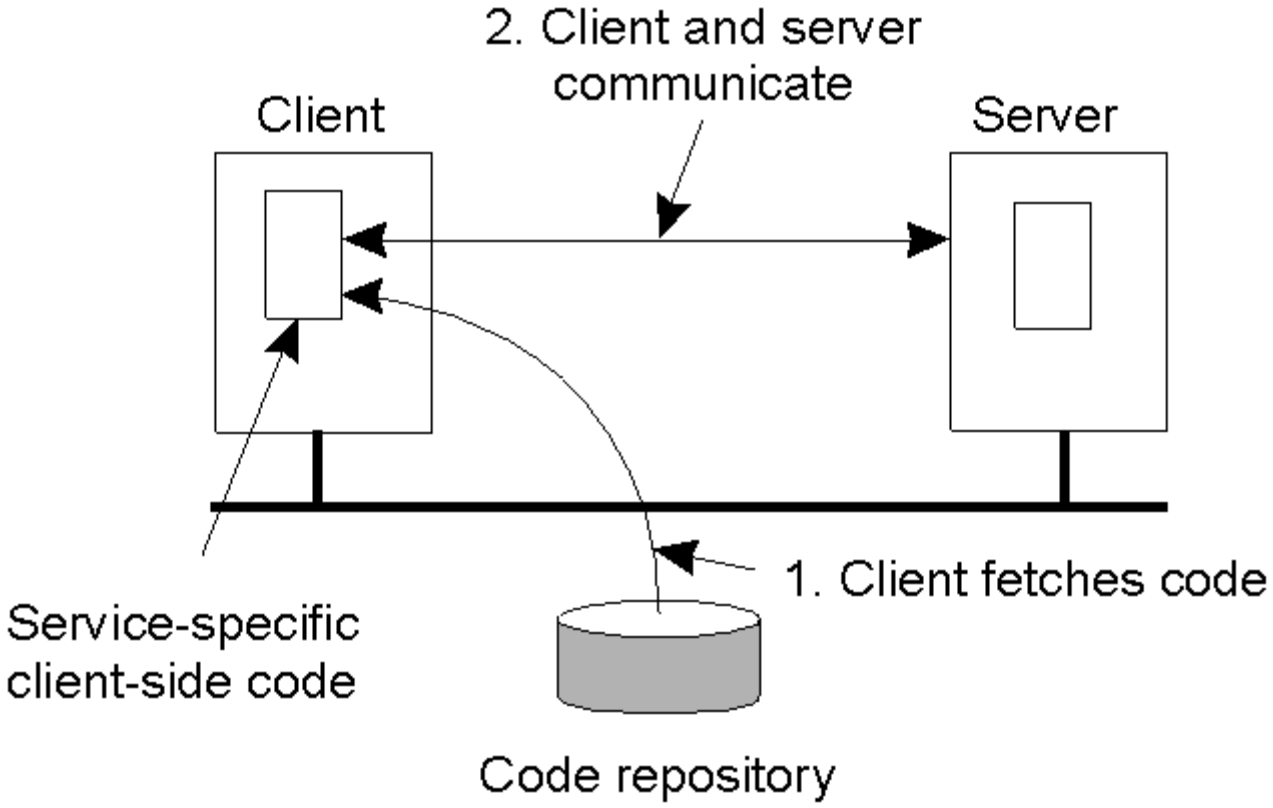
- Plik *thread.h* wykorzystywany przez adapter za pomocą wątków.

Adapter obiektu (4)

- Główna część adaptera który implementuje politykę *wątek-na-obiekt*.

```
#include <header.h>
#include <thread.h>
#define MAX_OBJECTS     100
#define NULL            0
#define ANY             -1

METHOD_CALL invoke[MAX_OBJECTS];    /* array of pointers to stubs   */
THREAD *root;                       /* demultiplexer thread         */
THREAD *thread[MAX_OBJECTS];        /* one thread per object        */

void thread_per_object(long object_id) {
    message     *req, *res;         /* request/response message     */
    unsigned    size;               /* size of messages             */
    char        **results;          /* array with all results       */

    while(TRUE) {
        get_msg(&size, (char*) &req);   /* block for invocation request */

        /* Pass request to the appropriate stub. The stub is assumed to */
        /* allocate memory for storing the results.                     */
        (invoke[object_id]*)(req→size, req→data, &size, results);

        res = malloc(sizeof(message)+size); /* create response message  */
        res→object_id = object_id;          /* identify object         */
        res→method_id = req.method_id;      /* identify method         */
        res→size = size;                    /* set size of invocation results*/
        memcpy(res→data, results, size);    /* copy results into response */
        put_msg(root, sizeof(res), res);    /* append response to buffer */
        free(req);                          /* free memory of request   */
        free(*results);                     /* free memory of results   */
    }
}

void invoke_adapter(long oid, message *request) {
    put_msg(thread[oid], sizeof(request), request);
}
```

Przyczyny migracji kodu



- Zasada dynamicznego konfigurowania klienta do komunikacji z serwerem. Klient najpierw ściąga potrzebne oprogramowanie, i następnie wywołuje serwer.

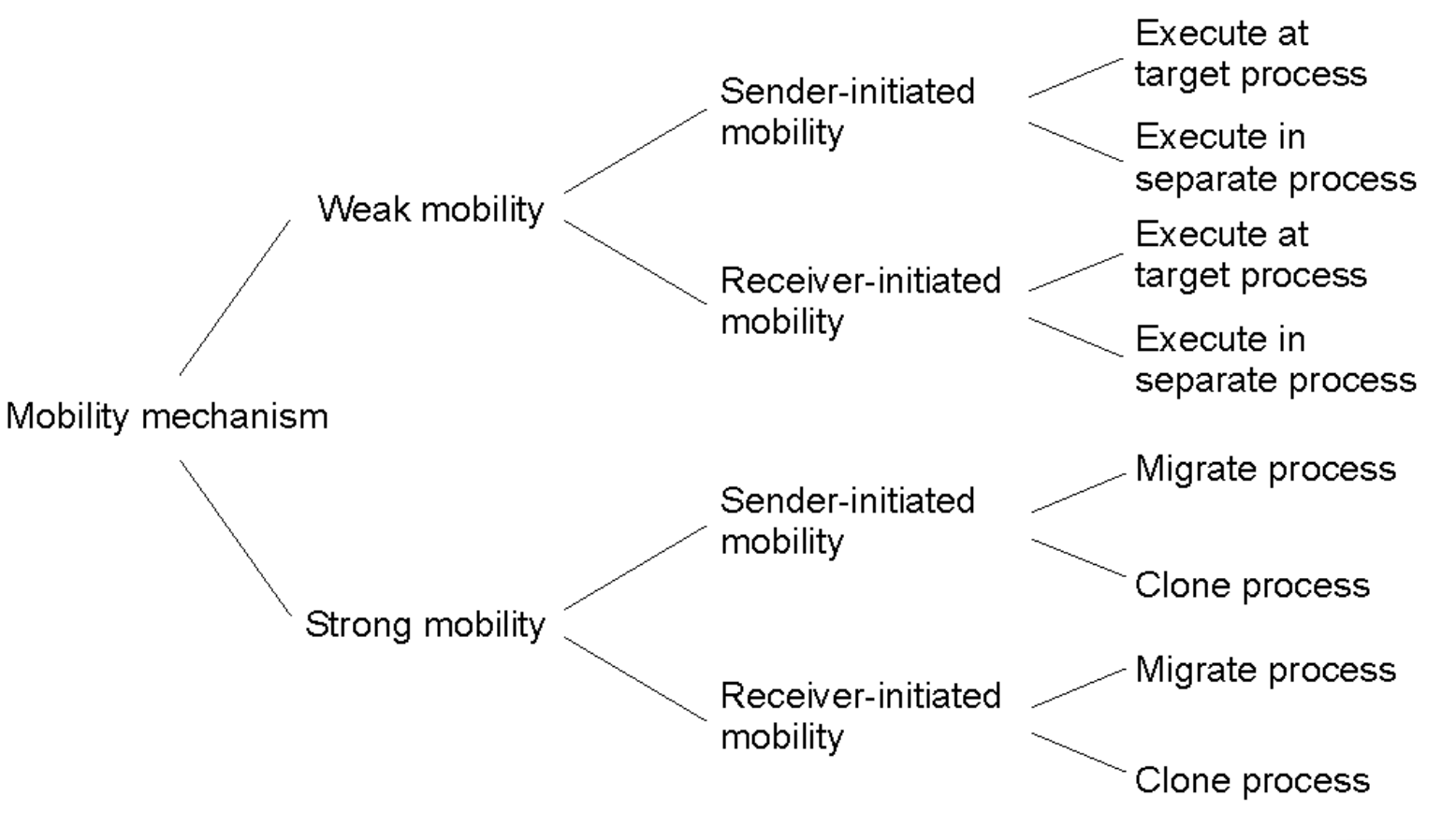
Migracja kodu

- Platforma: proces składa się z trzech segmentów:
  - segment kodu,
  - segment zasobów (referencje do zewnętrznych zasobów),
  - segment wykonania (bieżący stan wykonania).

  Wiązanie z zasobami:
  - przez id – np. URL,
  - przez wartość– gdy program bazuje na standardowych bibliotekach (C, Java),
  - przez typ – referencje do lokalnych urządzeń np.. monitory, drukarki.

- ***Słaba (weak) mobilność*** – możliwy do przeniesienia segment kodu z częścią inicjalizacyjną,
- ***Silna (strong) mobilność*** – mogą być przeniesione segmenty kodu i wykonania

Modele migracji kodu



- Alternatywy migracji kodu

Migracja, a lokalne zasoby

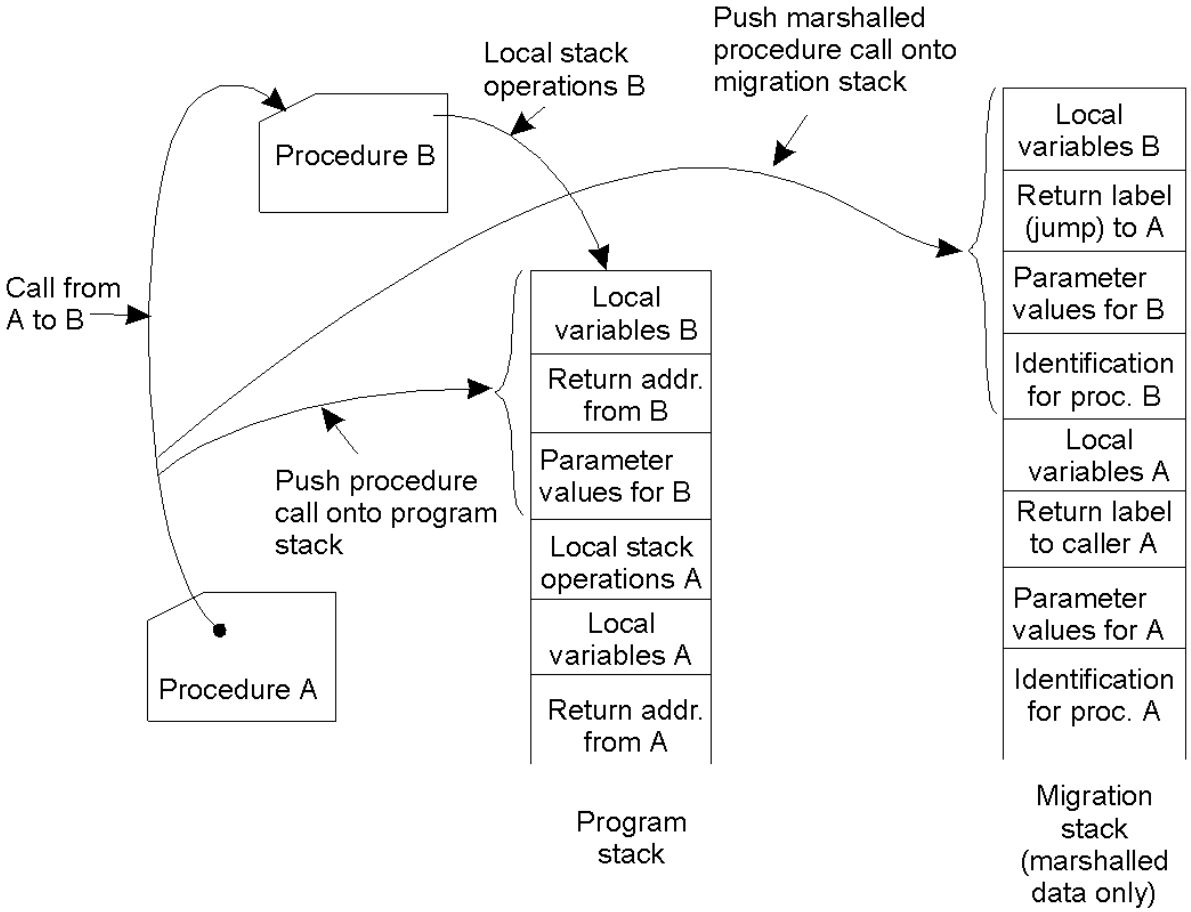
Wiązanie zasobów do maszyn

Wiązanie procesu do zasobu	niezobowiązujący	umocowany	zafiksowany
przez id	MV (lub GR)	GR (lub MV)	GR
przez wartość	CP (lub MV, GR)	GR (lub CP)	GR
przez typ	RB (lub GR, CP)	RB (lub GR, CP)	RB (lub GR)

- Akcje podejmowane w stosunku do referencji do lokalnych zasobów przy migracji kodu na inną maszynę.

GR – ustanowić globalną referencję w skali systemu,
MV – przenieść zasób,
CP – kopiować wartość zasobu,
RB – ponownie powiązać proces do lokalnie dostępnych zasobów.

Migracja w systemach heterogenicznych



- Zasada zachowania stosu migracji do wspierania migracji segmentu wykonania w środowisku heterogenicznym. Migracja kodu ograniczona do specificznych punktów wykonania.
- **Stos migracji** (migration stack**)** – kopia stosu programu niezależna od platformy sprzętowej.

Przegląd migracji kodu w D'Agents (1)

```
proc factorial n {
    if ($n ≤ 1) { return 1; }                    # fac(1) = 1
   expr $n * [ factorial [expr $n – 1] ]         # fac(n) = n * fac(n – 1)
}
set number …          # tells which factorial to compute
set machine …         # identify the target machine

agent_submit $machine –procs factorial –vars number –script {factorial $number }

agent_receive …       # receive the results (left unspecified for simplicity)
```

- Prosty przykład agenta Tel w D'Agents wysyłającego skrypt do zdalnej maszyny (źródło [gray.r95])

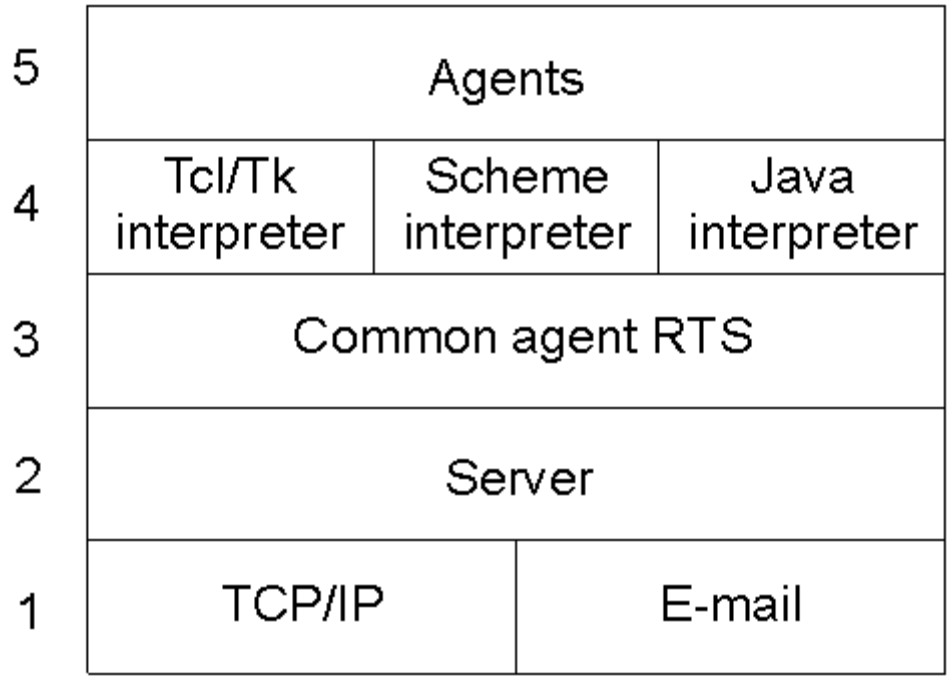
Przegląd migracji kodu w D'Agents (2)

all_users $machines

```
proc all_users machines {
    set list ""                     # Create an initially empty list
    foreach m $machines {           # Consider all hosts in the set of given machines
        agent_jump $m               # Jump to each host
        set users [exec who]        # Execute the who command
        append list $users          # Append the results to the list
    }
    return $list                    # Return the complete list when done
}

set machines …                      # Initialize the set of machines to jump to
set this_machine                    # Set to the host that starts the agent

# Create a migrating agent by submitting the script to this machine, from where
# it will jump to all the others in  $machines.

agent_submit $this_machine –procs all_users
                                    -vars  machines
                                    -script { all_users $machines }

agent_receive …                     #receive the results  (left unspecified for simplicity)
```

- Przykład agenta Tel w D'Agents migrującego na różne maszyny, gdzie wykonuje komendę UNIX-a *who* (źródło [gray.r95])

Zagadnienia implementacyjne (1)



- Architektura systemu D'Agents

Zagadnienia implementacyjne (2)

Status	Opis
Globalne zmienne interpretera	Zmienne potrzebne interpreterowi agenta
Globalne zmienne systemowe	Kodu powrotu, kody błędów, komunikaty o błędach, itd.
Globalne zmienne programowe	Globalne zmienne użytkownika w programie
Definicje procedur	Definicje skryptów do wykonania przez agenta
Stos komend	Stos komend aktualnie wykonywanych
Stos ramek wywołania	Stos zarejestrowanych aktywacji, po jednej na każdą wykonywaną komendę

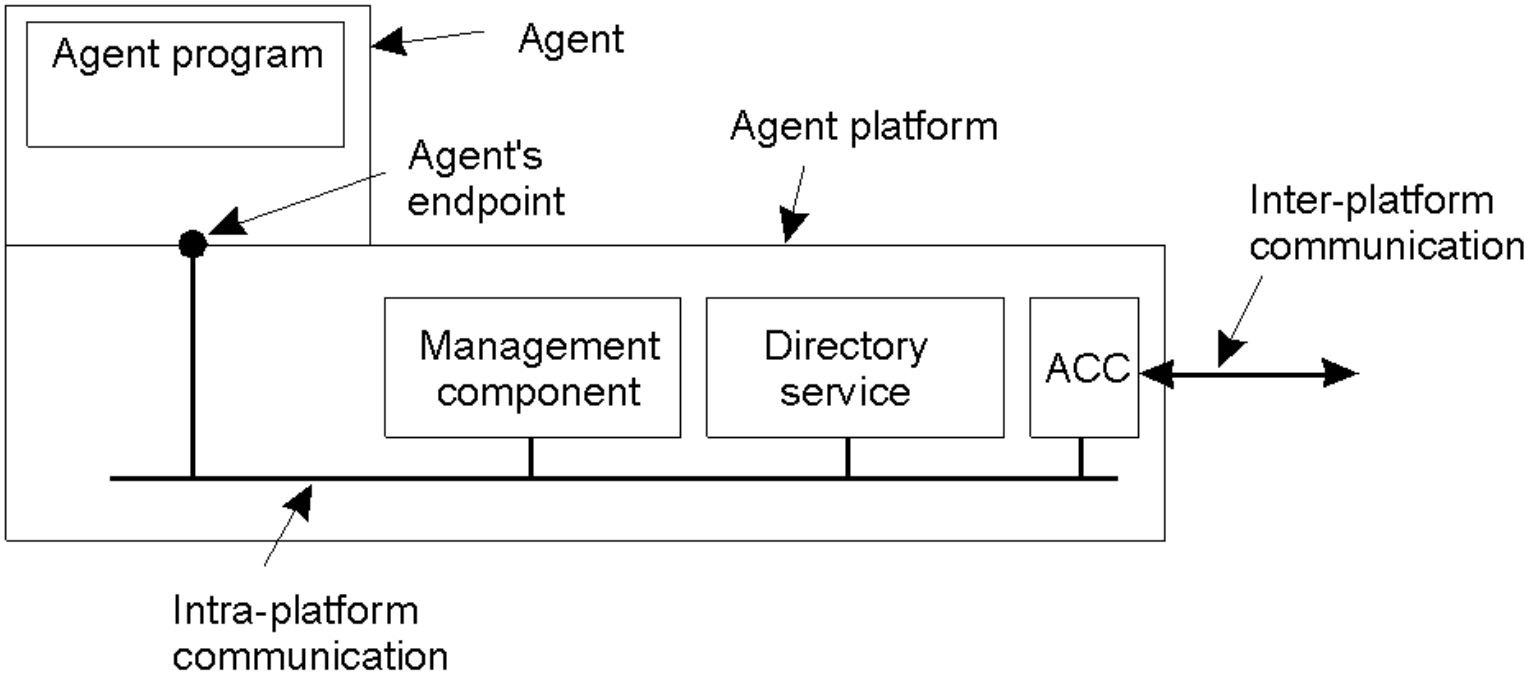
- Składowe stanu agenta w D'Agents.

Agenci programowe w systemach rozproszonych

Własność	Wspólna dla wszystkich agentów?	Opis
Autonomiczność	tak	Może działać samodzielnie
Reaktywność	tak	Odpowiada szybko na zmiany w środowisku
Proaktywność	tak	Uruchamia czynności które wpływają na otoczenie
Kommunikacyjność	tak	Może wymieniać informacje z użytkownikami oraz innymi agentami
Ciagłość	nie	Ma stosunkowo długi żywot
Mobilność	nie	Może migrować z jednego systemu na drugi
Adaptacyjność	nie	Zdolność do uczenia

- Ważne własności różnych typów agentów

Technologia agentowa



- Ogólny model platformy agentowej (źródło [fipa98-mgt]).

Języki komunikacji międzyagentowej (1)

Przeznaczenie komunikatu	Opis	Zawartość komunikatu
INFORM	Informuje że dana propozycja jest prawdziwa	Propozycja
QUERY-IF	Pyta czy dana propozycja jest prawdziwa	Propozycja
QUERY-REF	Zapytanie o dany obiekt	Wyrażenie
CFP	Pyta o wniosek	Zależne od wniosku
PROPOSE	Dostarcza wniosek	Wniosek
ACCEPT-PROPOSAL	Mówi że dany wniosek jest przyjęty	ID wniosku
REJECT-PROPOSAL	Mówi że dany wniosek jest odrzucony	ID wniosku
REQUEST	Prosi o wykonanie danej akcji	Specyfikacja akcji
SUBSCRIBE	Subskrypcja na zasób informacyjny	Referencja do zasobu

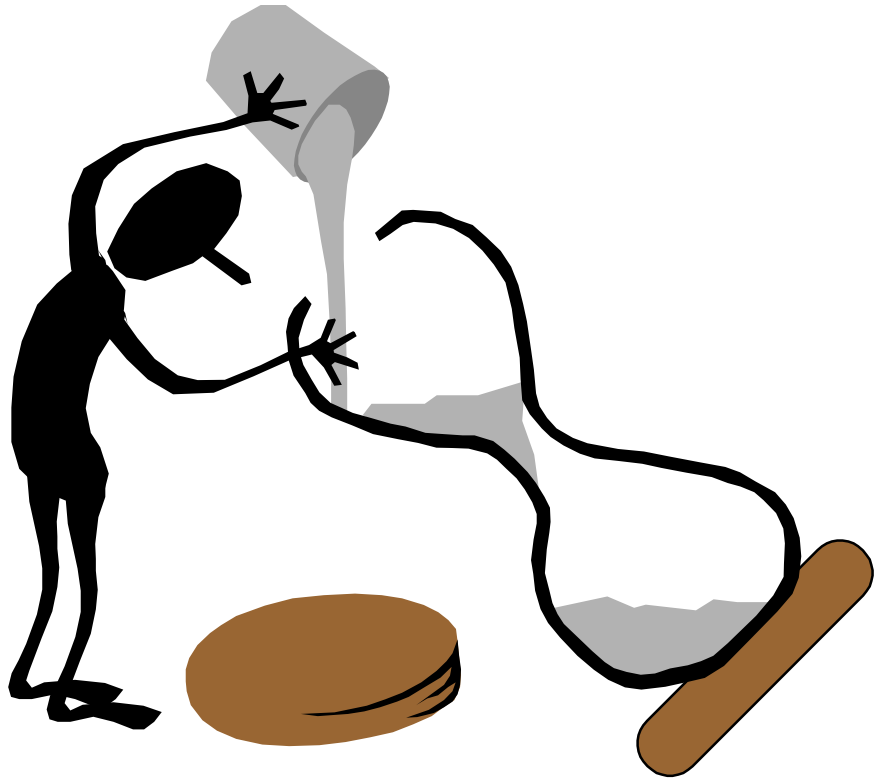
- Przykłady różnych typów komunikatów w FIPA-ACL [fipa98-acl], na które się składają przeznaczenie komunikatu message oraz opis zawartości komunikatu

Języki komunikacji międzyagentowej (2)

Pole	Wartość
Przeznaczenie	INFORM
Nadawca	max@http://fanclub-beatrix.royalty-spotters.nl:7239
Odbiorca	elke@iiop://royalty-watcher.uk:5623
Język	Prolog
Ontologia	genealogy
Zawartość	female(beatrix),parent(beatrix,juliana,bernhard)

- Prosty przykład komunikatu FIPA-ACL przesyłanego między dwoma agentami używającymi Prolog do opisu informacji genealogicznej.

dziękuję za uwagę



dr inż. Jacek Czerniak

jczerniak@ukw.edu.pl