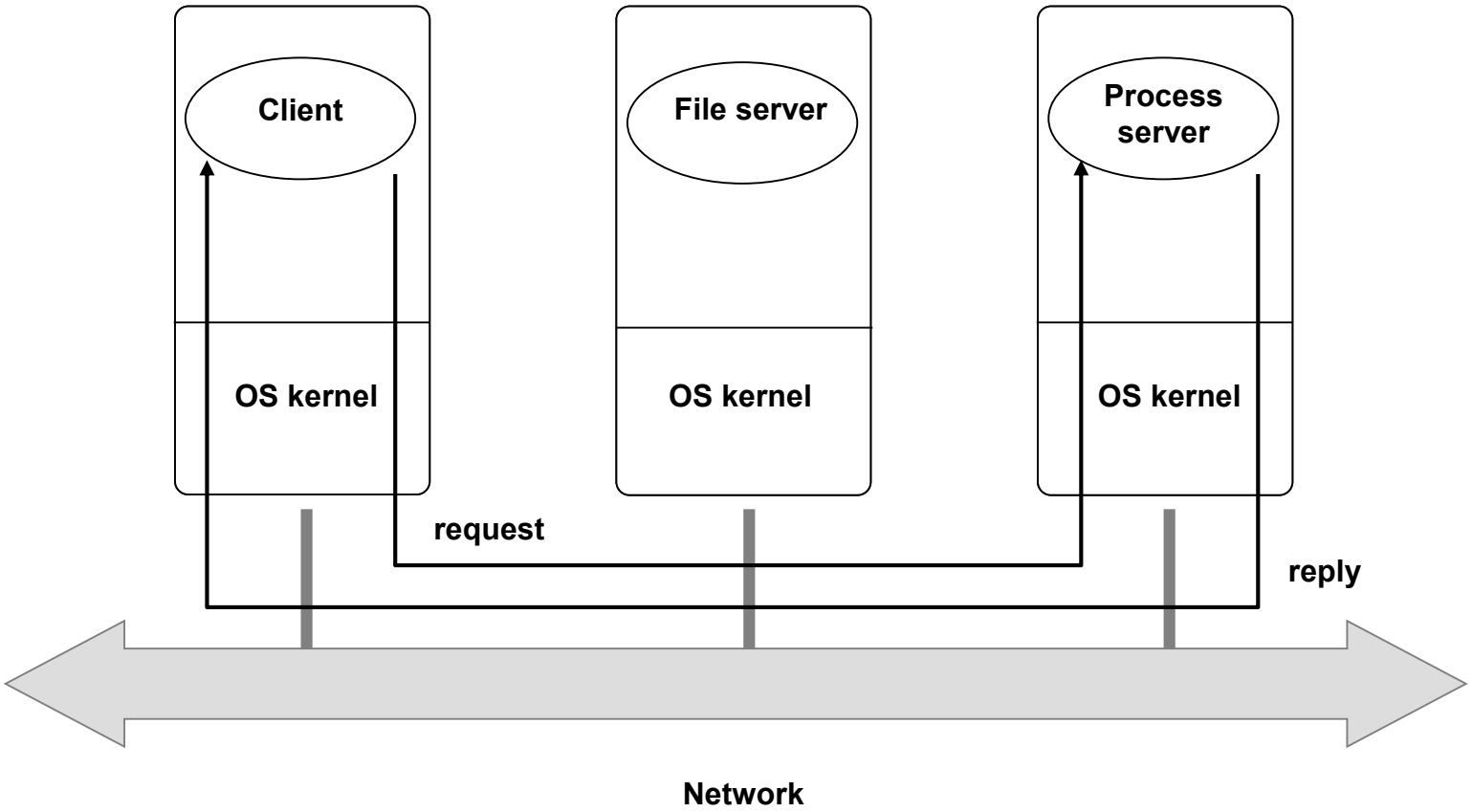
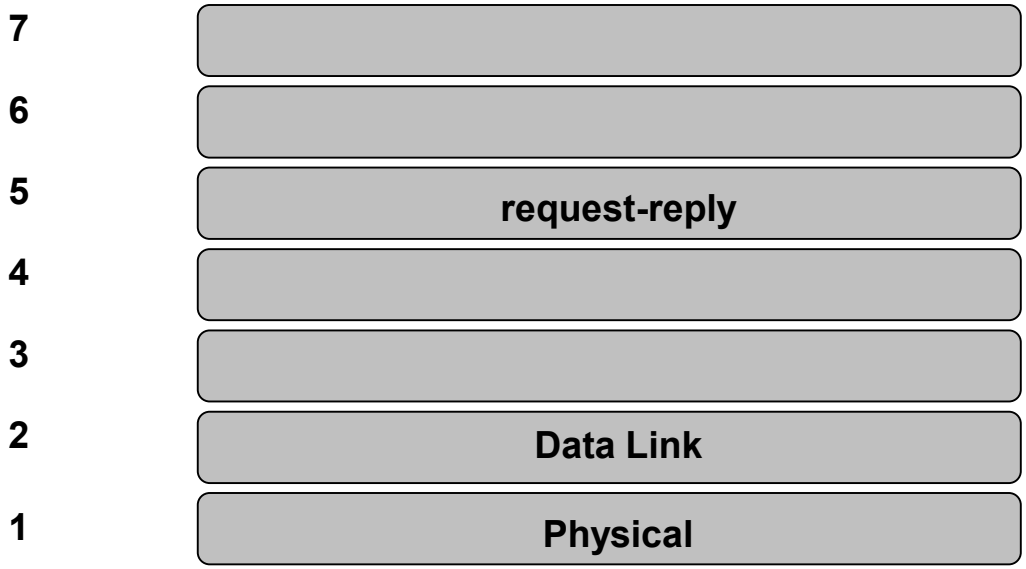
Модел Клиент-Сървър

Модел Клиент-Сървър

- **Структура**: група от комуникиращи процеси, където част от тях (сървъри) предоставя услуги на друга (клиенти);
- **Платформа**: комуникационен модел request/reply;
- **Програмни технологии**: sockets, remote procedure call (RPC), remote method invocation (RMI).


Client
File server
Process server
OS kernel
OS kernel
OS kernel
request
reply
Network

Протоколни слоеве за модела



Системни извиквания

- *send (destination, &message)* – изпраща съобщение от адрес &message към процес с идентификатор destination;
- *receive (address, &message)* – указва получателя (*address*) и адреса на неговото адресно пространство, където полученото съобщение да бъде записано.

Проблеми

- Как да се адресират процесите;
- Блокиране или не на процеса по време на изпълнение на *send()* и *receive()*;
- Буфериране или не на съобщенията при изпращане и получаване;
- Осигуряване или не на надеждност на доставката;
- Архитектура на сървъра – последователен или паралелен;
- Разширяемост.

Адрес на сървъра(1)

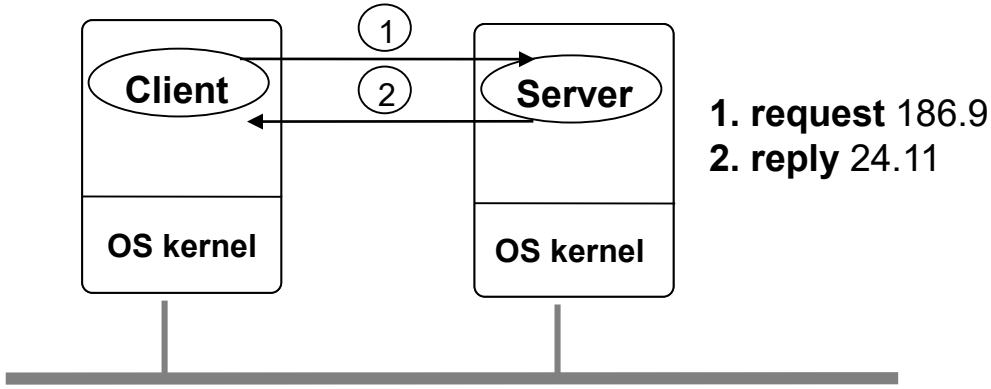
Основна адресна схема: ***<machine>.<process>***

Проблем: Как да се намери сървъра?

Решения:

- ***Предварително зададен адрес***

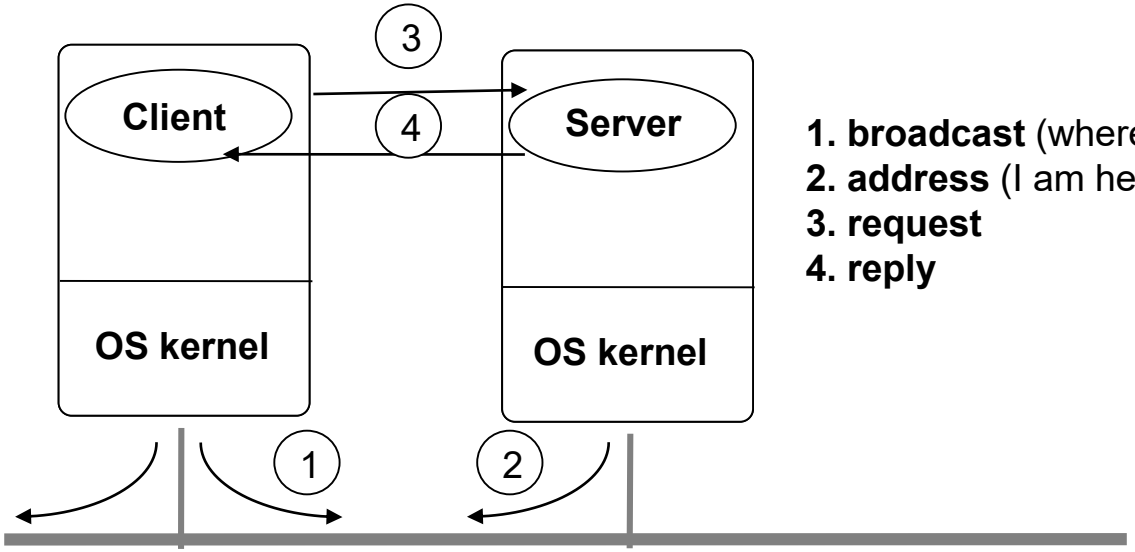
  Идентификаторът на машината трябва да е предварително известен.



Недостатък – адресът е записан твърдо в кода. Няма гъвкавост.

Адрес на сървъра (2)

- ***Използване на broadcast за намиране на сървъра***

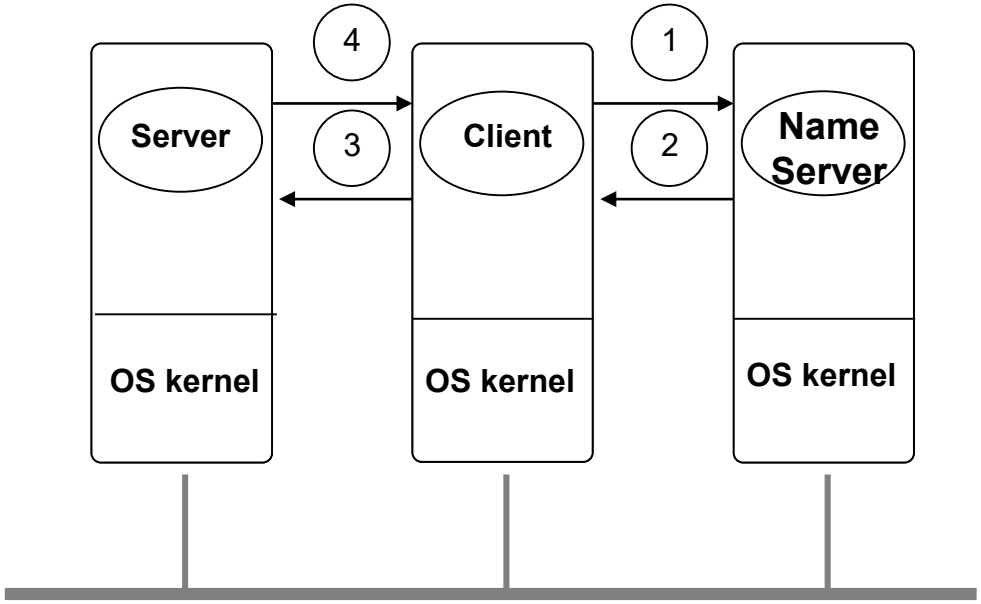


1. **broadcast** (where is the server?)
2. **address** (I am here)
3. **request**
4. **reply**

<u>Недостатък</u> – допълнителен broadcast трафик

Адрес на сървъра (3)

- ***Получаване на адреса от name server***



1. lookup for service
2. name server reply
3. request
4. reply

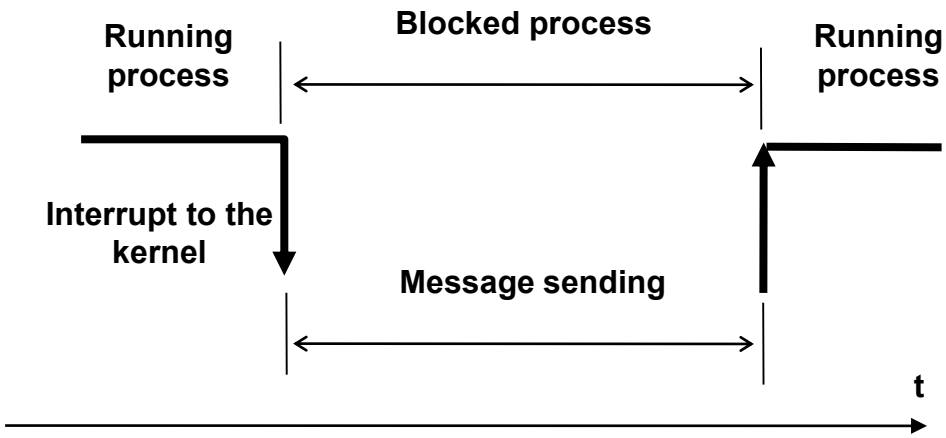
Гъвкавост, но изисква допълнителен централизиран компонент – name server.

Комуникации

- Блокиращи извиквания;
- Неблокиращи извиквания.

Синхронна комуникация (1)

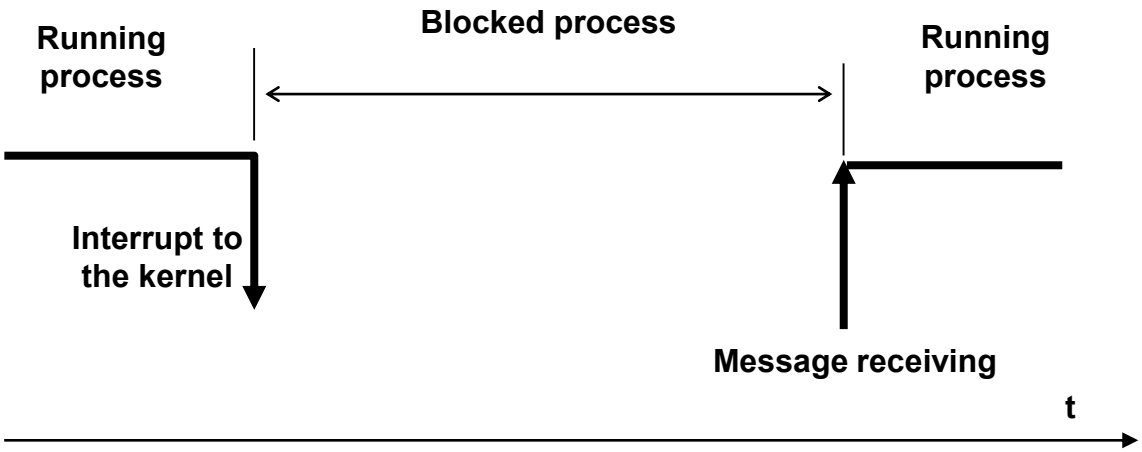
- Блокиращо извикване send()



Процесът ще се блокира до завършване изпращането на цялото съобщение.

Синхронна комуникация (2)

- Блокиращо извикване receive()



Процесът ще се блокира докато съобщението се получи и запише в дадения от процеса буфер.

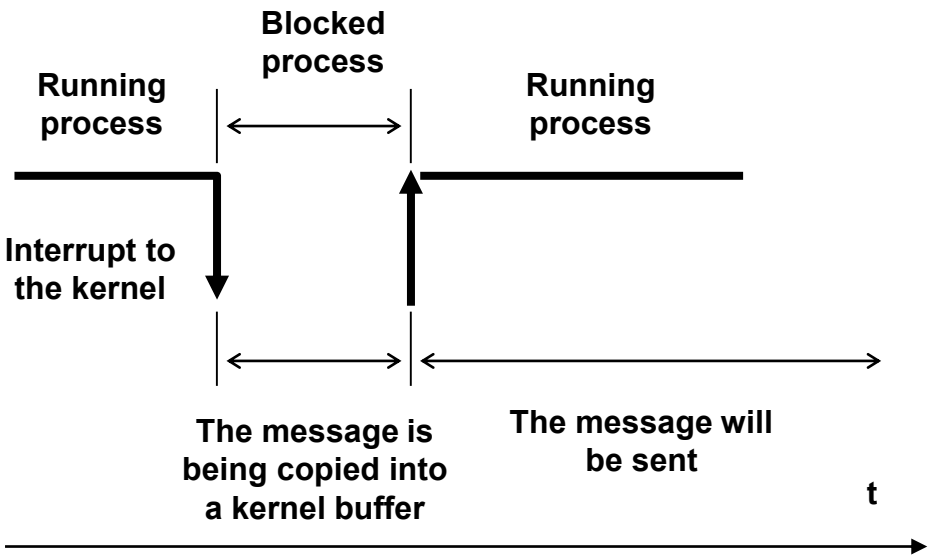
Синхронна комуникация (3)

Блокиращите примитиви са:

- Лесни за реализация;
- Не изискват буфериране в ядрото на ОС;
- Не позволяват паралелна работа на процесите по време на изпращане и получаване на съобщение.

Асинхронна комуникация (1)

Неблокиращо извикване *send()*



Асинхронна комуникация (2)

Проблем: Как процесът ще разбере кога съобщението е било изпратено?

Решение 1: Копиране на съобщението в буфер на ядрото

- Съобщението се копира в буфера;
- Незабавно след копирането, процесът може да продължи изпълнението си;
- Ако е налице претоварена комуникация може да се получи презаписване на буфера и загуба на съобщение или ниска производителност.

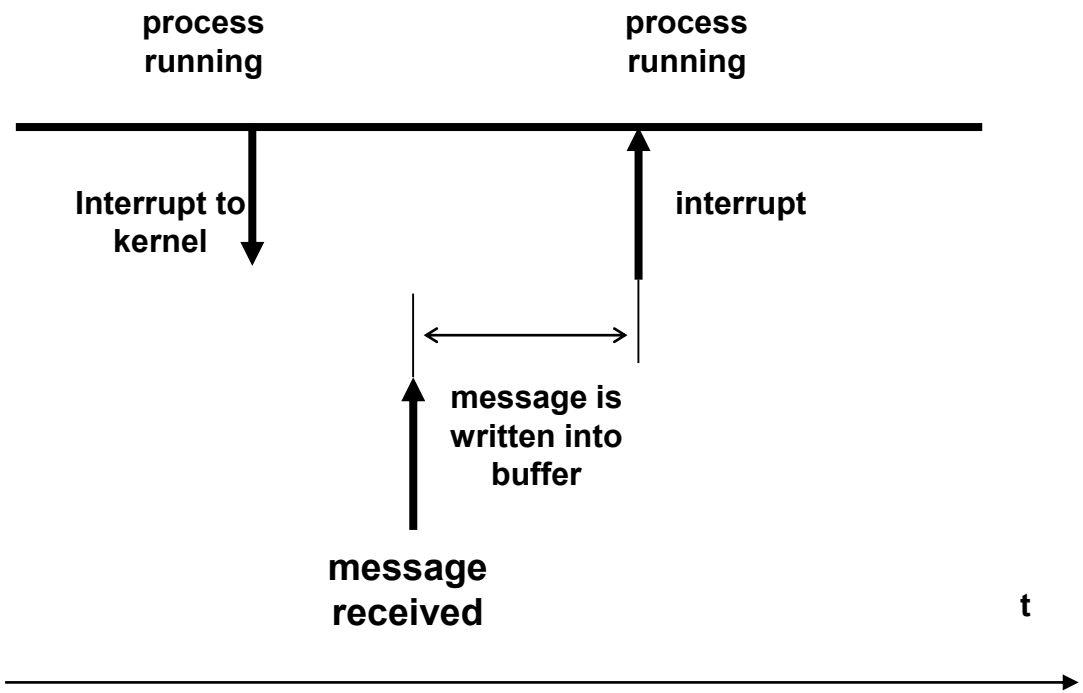
Асинхронна комуникация (3)

<u>Решение 2</u>: ОС генерира прекъсване към процеса веднага след като съобщението е било изпратено

- Съобщението се изпраща директно от буфера на процеса без да се копира в буфер на ядрото;
- След изпращане на съобщението ядрото генерира прекъсване към процеса. Буферът е свободен за последващо използване;
- Висока степен на паралелизъм - висока производителност;
- Сложно програмиране.

Асинхронна комуникация (4)

Неблокиращ *receive()*



Асинхронна комуникация (5)

Проблем: Как да се разбере ако съобщението е вече било записано в буфера на процеса?

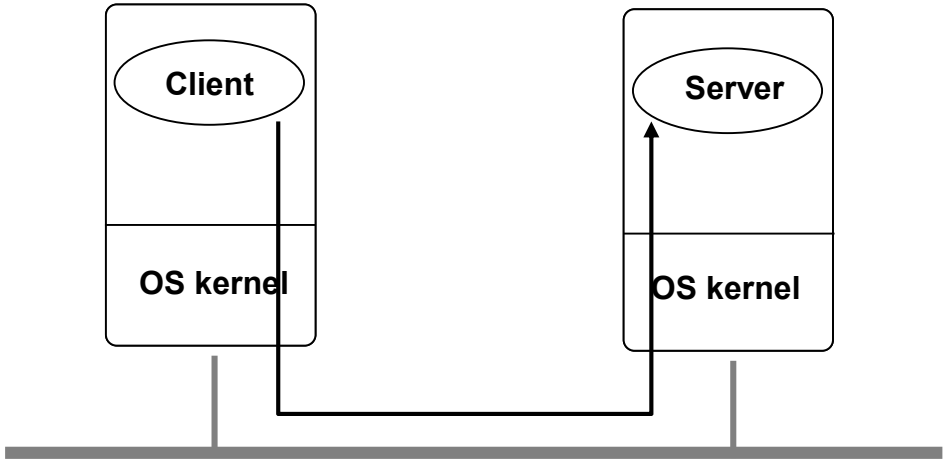
- С прекъсване на процеса след записа в буфера;
- Процесът периодично пита ядрото (pooling) – допълнителни средства;
- При извикване на *receive()*, ядрото веднага дава съобщението или връща код за грешка ако съобщението още не е получено.

Буфериране на съобщения

Без буфефиране – Възможен е проблем, ако сървърът изпълни *receive()* преди клиентът да изпълни *send()* – няма да се знае съобщението от кого е.

Решение:

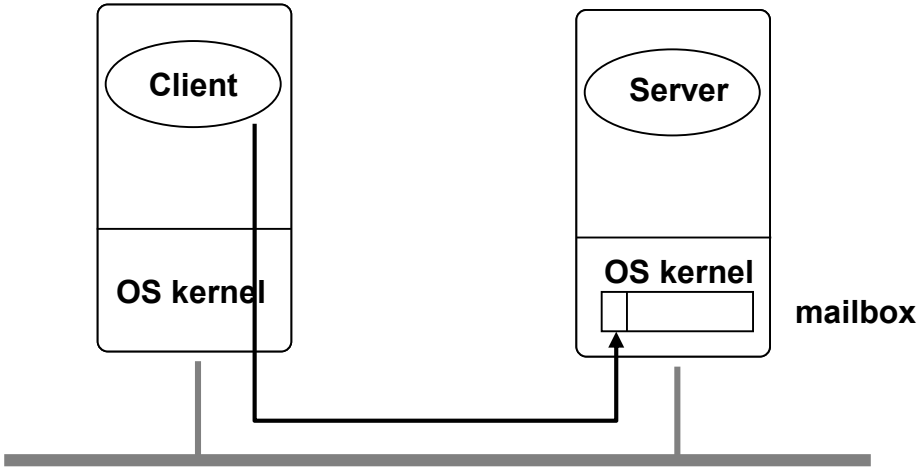
- Отхвърляне на съобщението, надявайки се, че клиентът ще го прати отново след като сървърът успее с *receive()*;
- Съхраняване на получените съобщения в ядрото за период чрез таймер. Изисква време и заемане на памет.



Буфериране на съобщения

С буфериране – чакащият съобщения процес изисква от ядрото да заеме специлен буфер – “mailbox”.

- Получените за процеса съобщения се запазват в този буфер, независимо дали процесът е изпълнил *receive()* ;
- За всеки *receive*() процесът взема първото съобщение от mailbox. Ако е празен, процесът се блокира;
- Може да настъпи препълване на буфера. Ако няма място, изпращащият процес се блокира, докато се получи потвърждение за получено съобщение.



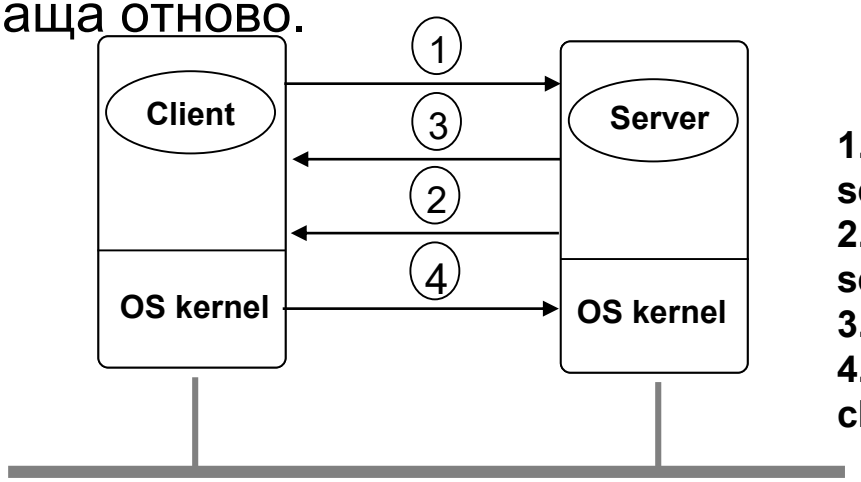
Надеждност

Възможна е загуба на съобщения при комуникацията.

<u>Решения</u>*:*

- Семантиката на модела се дефинира като ненадеждна. Надеждността се гарантира от приложенията;

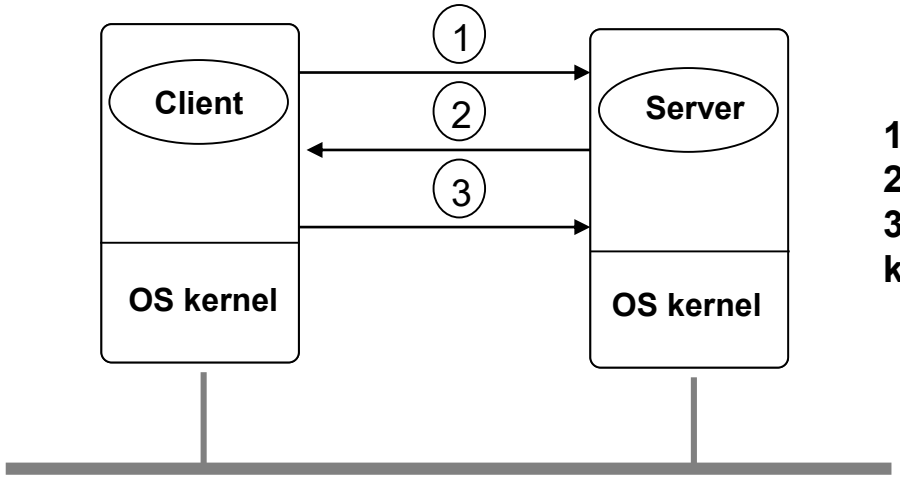
- Ядрото на получателя изпраща потвърждения за всяко получено съобщение. Ако няма потвърждение за определен период от време, съобщението се счита загубено и се препраща отново.



1. request from client to server
2. acknowledge from server kernel
3. reply from server
4. acknowledge from client kernel

Надеждност (2)

- Отговорът от сървъра може да използва като потвърждение за получаване на съобщението.



1. request from client
2. reply from server
3. acknowledge from client kernel

- Използване на надеждна комуникация чрез специален надежден протокол (TCP).

Пример за клиент-сървър

За реализация на система клиент-сървър е необходимо да се дефинира:

- Множеството от функции на сървъра;
- Множеството от резултати, връщани на клиент;
- Форматът на приложните съобщения;
- Алгоритмите на клиента и сървъра.

Пример (обща част)

“header.h”

```
/* common definitions */
#define  MAX_PATH     255     /* max length of the filename */
#define  BUF_SIZE     1024    /* data length */
#define  FILE_SERVER  125     /* network address of the server */
/* allowable operations */
#define  CREATE       1       /* create a file */
#define  READ         2       /* read from a file */
#define  WRITE        3       /* write to a file */
#define  DELETE       4       /* delete a file */
/* error codes */
#define  OK           0       /* the operation is successful */
#define  BAD_OPCODE   -1      /* unknown operation */
#define  BAD_PARAM    -2      /* wrong parameters */
#define  E_IO         -3      /* IO error */
```

Пример (2)
(обща част)

/* message format */

```
struct message {
    long  source;          /* source identifier */
    long  dest;            /* destination identifier */
    long  opcode;          /* operation */
    long  count;           /* number of data bytes transferred */
    long  offset;          /* file pointer*/
    long  extra1;          /* additional field */
    long  extra2;          /* additional field */
    long  result;          /* operation result */
    char name [MAX_PATH];          /* file name */
    char data [BUF_SIZE];          /* data read/written*/
};
```

Пример (3) (сървър)

```
#include <header.h>
void  main (void)
{
    struct message m1, m2;          /* input/output messages */
    int  r;                         /* result code */
    while (1) {
        receive (FILE_SERVER, &m1);     /* wait for a message */
        switch (m1,opcode) {
            case CREATE:        r = do_creat(&m1, &m2);         break;
            case READ:          r = do_read(&m1, &m2);          break;
            case WRITE:         r = do_write(&m1, &m2);         break;
            case DELETE:        r = do_delete(&m1, &m2);        break;
            default:            r = BAD_OPCODE;
        }
        m2.result = r;
        send (m1.source, &m2);
    }
}
```

Пример (4) (клиент)

```
#include <header.h>
int  copy (char *src, char *dst);          /* function for file copying */
{
    struct message  m1;                    /* message buffer */
    long   position;                       /* current file pointer */
    long client = 215;                     /* client network address*/
    initialize ();                         /* ready to esxecute */
    position = 0;
    do  {
                        /* read a data block from the file */

        m1.opcode = READ;          /* read operation */
        m1.offset = position;      /* current file pointer */
        m1.count= BUF_SIZE;        /* number of bytes for reading */
        strcopy( &m1.name, src);   /* write the file na e into the message */
        send (FILE_SERVER, &m1);          /* send the message*/
        receive (client, &m1);            /* wait for an answer */
```

Пример (5) (клиент)

```
/* write received data into a copy of the file */

    m1.opcode = WRITE;      /* write operation */
    m1.offset = position;   /* current file pointer */
    m1.count = m1.result;   /* number of bytes for writing */
    strcopy (&m1.name, dst; /* write file name into the message */
    send (FILE_SERVER, &m1);      /* send the message */
    receive (client, &m1);        /* wait for an answer*/
    position  += m1.result;
  } while (m1.result > 0 );

  return (m1.result  >=  0  ?  OK  : m1.result); /*returns OK or ERROR */
}
```

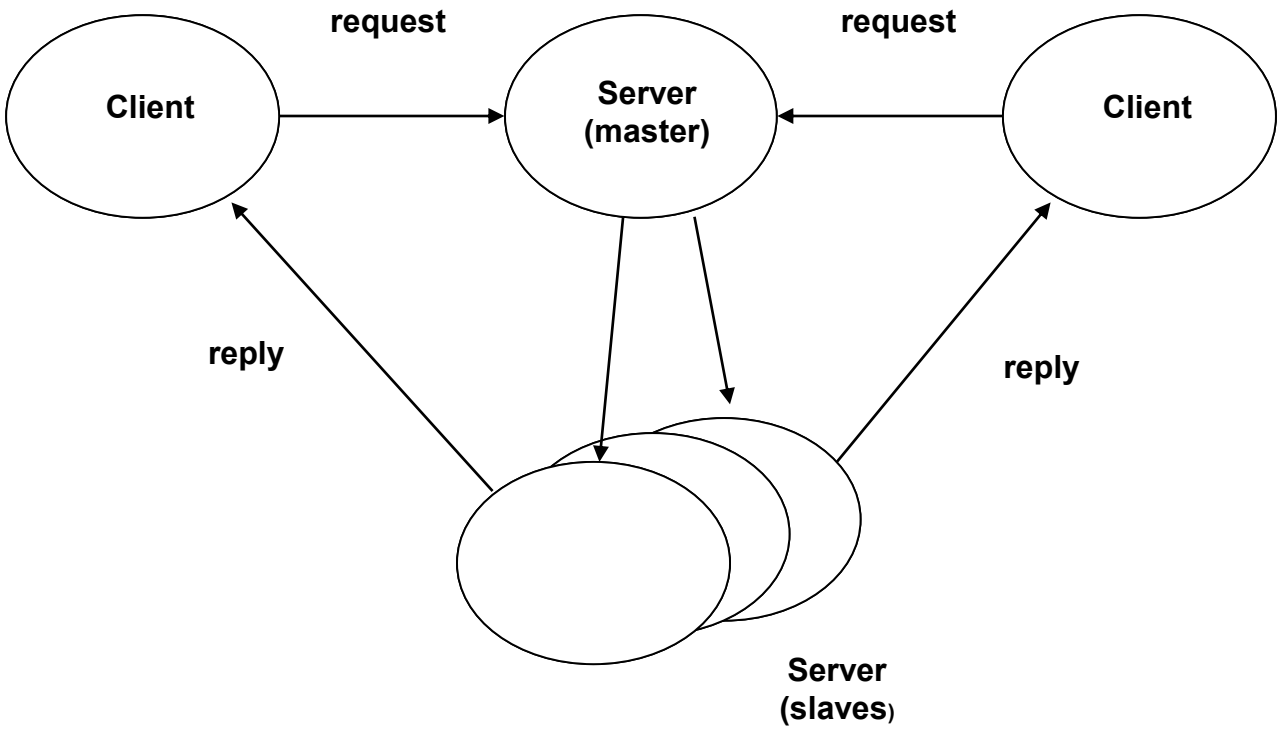
Архитектура на сървър

Сървърът трябва да е безкраен процес, обслужващ много клиенти.

В зависимост от едновременно обслужваните клиенти сървърът е:

- **Последователен**. Заявките постъпват в опашка и се обслужват една по една. Ниска производителност;
- **Паралелен**. Обслужват се много клиенти едновременно.
  - Мултипроцесни сървъри. Включва главен процес и много подчинени. Висока ефикасност на обслужване на клиентите. **Но**: значителни системни разходи за създаване, планиране и завършване на процеси. Производителността може да се подобри с използване на предварително създаден пул от процеси (pre-spawned pool).
  - Многонишкови сървъри.

Архитектура на паралелен сървър



Взаимодействие на сървъра

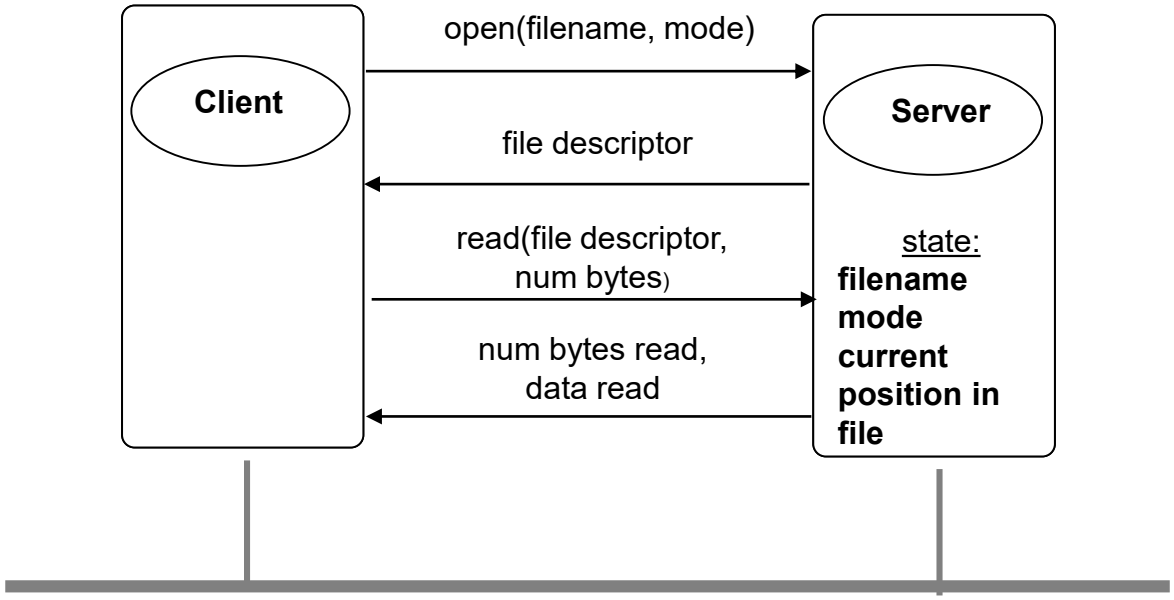
Сървър може да обслужва заявките на клиентите:

- Самостоятелно;
- Взаимодействайки с други сървъри. Ако не може да обслужи клиента (например, липса на данни), той може да комуникира с други сървъри изпращайки заявки. Пример: DNS.

Съхраняване на информацията

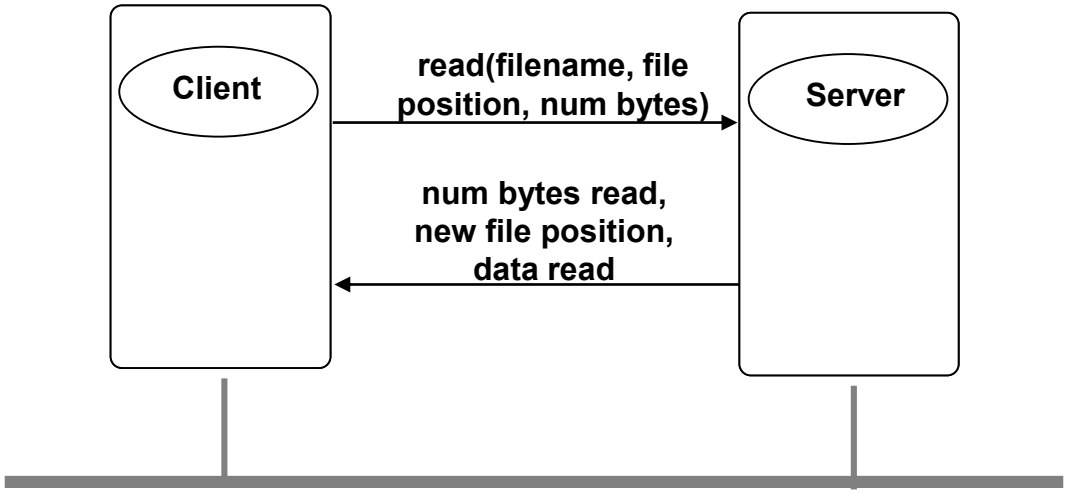
В зависимост от това каква информация за клиента съхраняват, сървърите са:

- **Stateful**. Съхранява се състоянието на обслужването на заявката.



Съхраняване на информацията

- **Stateless.** Не се съхранява информация. За всяка заявка клиентът трябва да изпраща необходимата за обслужването й информация.



Stateful срещу stateless

Stateful сървърите са:

- Повече продуктивни;
- Лесно програмиране;
- При отказ на сървъра, информацията за клиента се губи. Не могат да се възстановят прекъснатите транзакции;
- При отказ на клиента, сървърът ще продължава да съхранява ненужна информация.

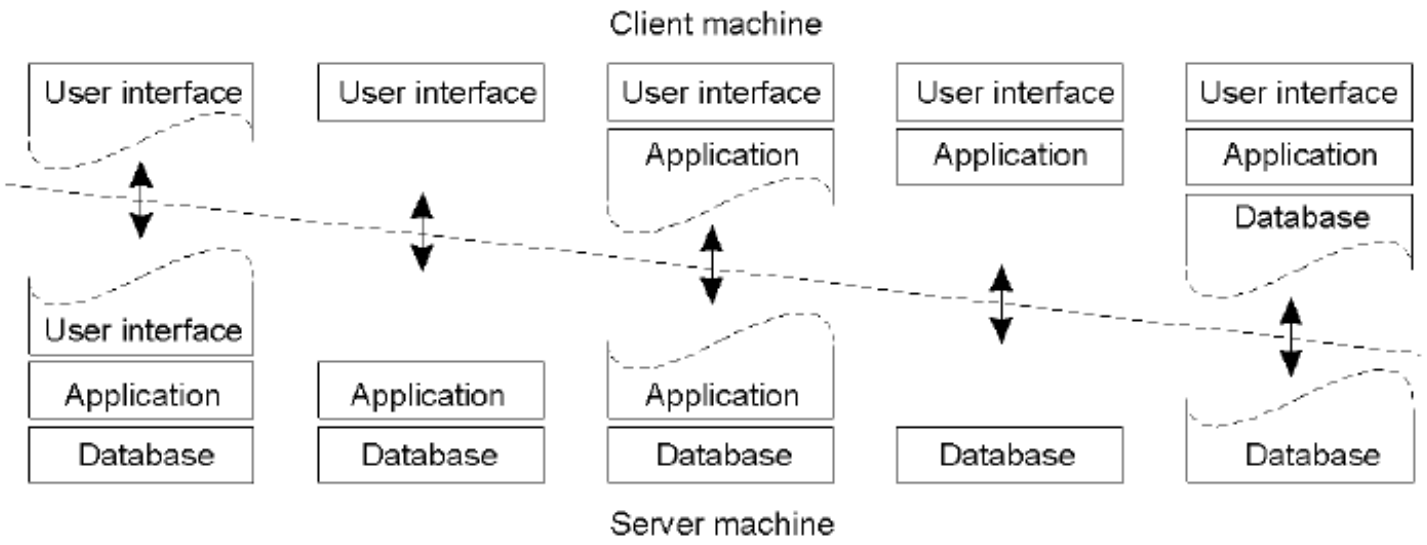
Stateless сървърите:

- Предоставят по-добри възможност за възстановяване след отказ.

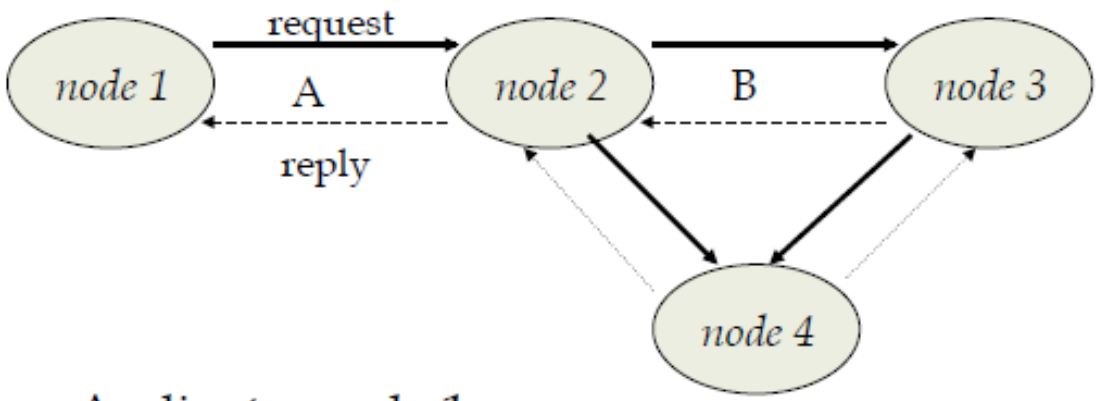
Разширяване на възможностите и капацитета на сървърите

- Пренасяне на сървъра на по-мощна машина;
- Реплициране на сървърите
- Използване на разпределени данни/алгоритми;
- Връщане на код за резултат;
- Кеширане.

Многослойни архитектури



Многослойни архитектури (2)

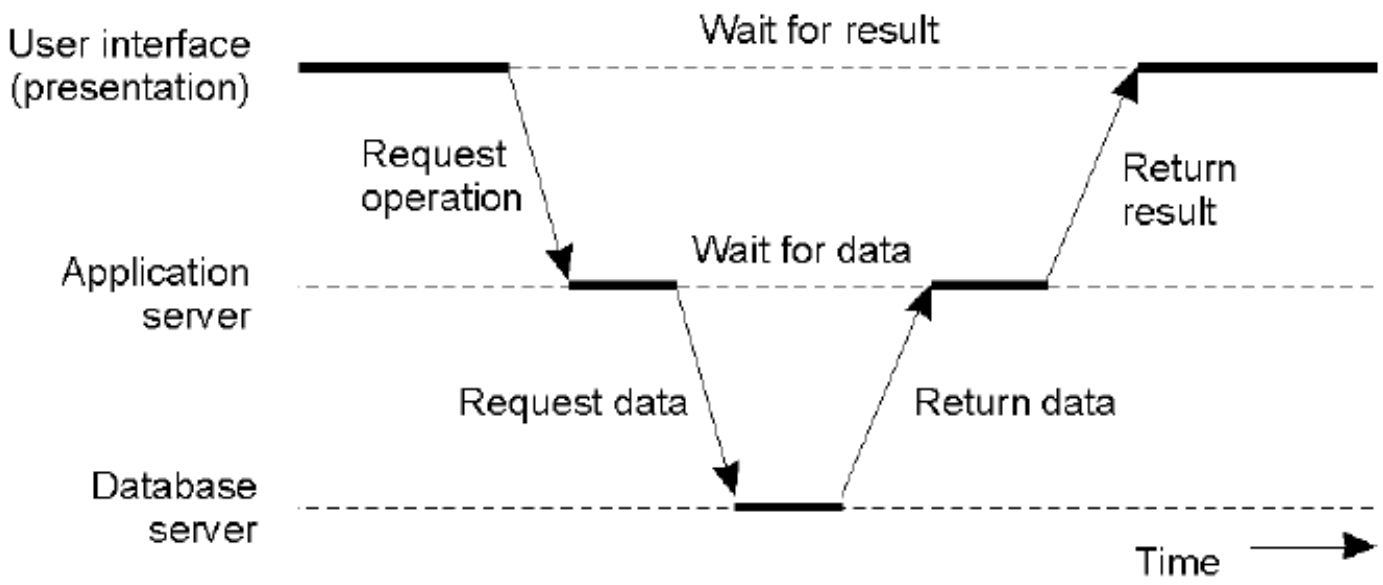


A client: node 1
server: node 2

B client: node 2
server: node 3

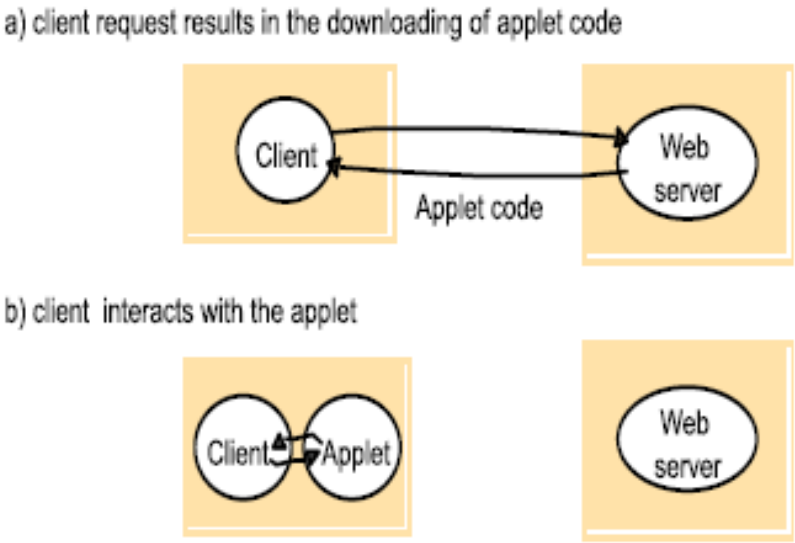
the concept is related
to communication
not to nodes

Многослойни архитектури (3)



Варианти на модела клиент-сървър

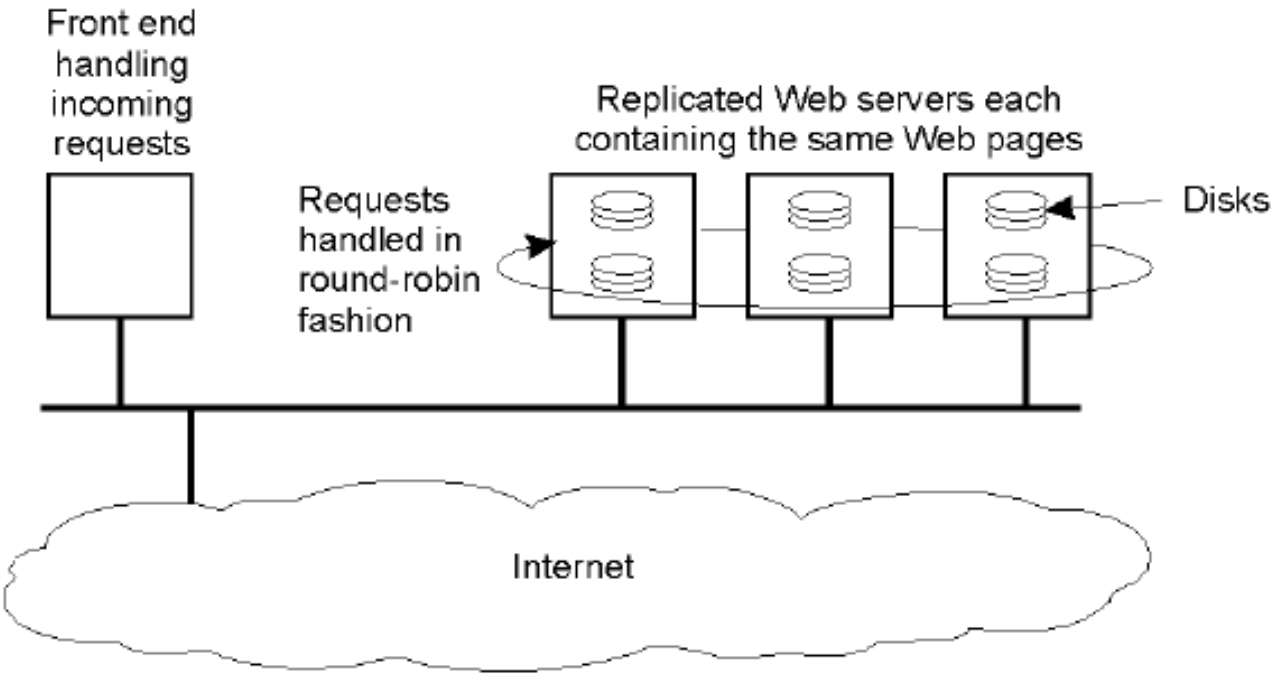
- Мобилен код
  - Изтегля се от клиента и се изпълнява
  - Push service – инициаторът е сървъра
- Мобилни агенти
  - Код и данни, които се придвижват
  - Необходима е платформа



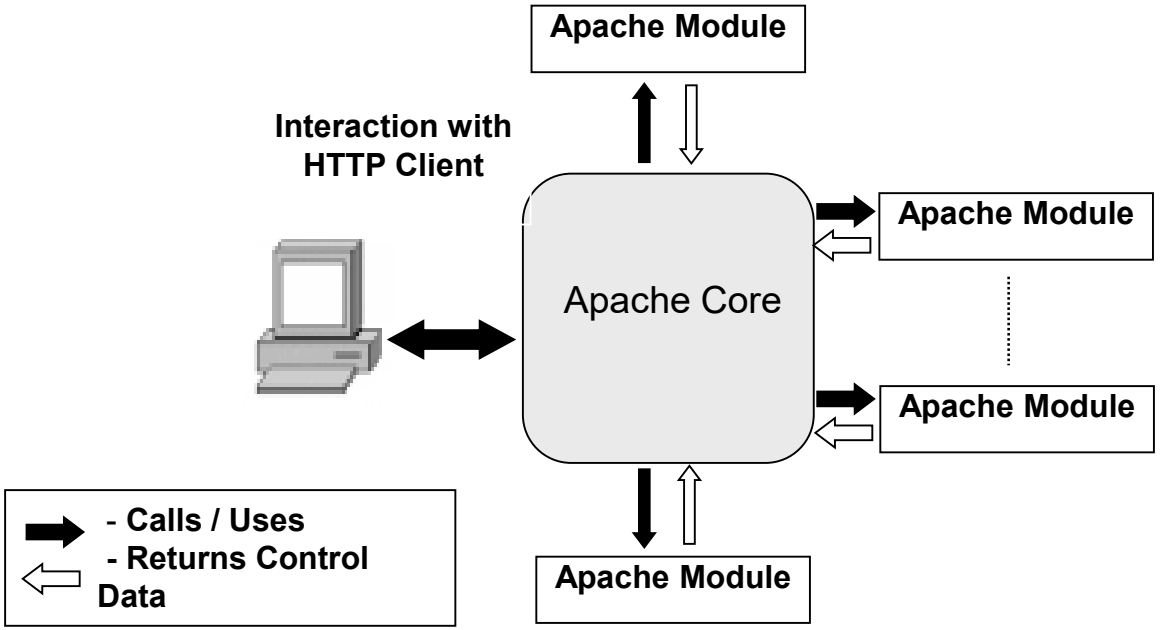
Варианти на модела клиент-сървър

- Мрежови компютри
  - Бездискови станции
  - Необходимо е изтегляне на код и данни
- Тънки агенти
  - Потребителския интерфейс се предоставя от PC
  - Сървърът изпълнява изчисленията (X-windows)

Съвременна архитектура – хоризонтално разпределение на web услуги



Пример – Apache server



Въпроси?