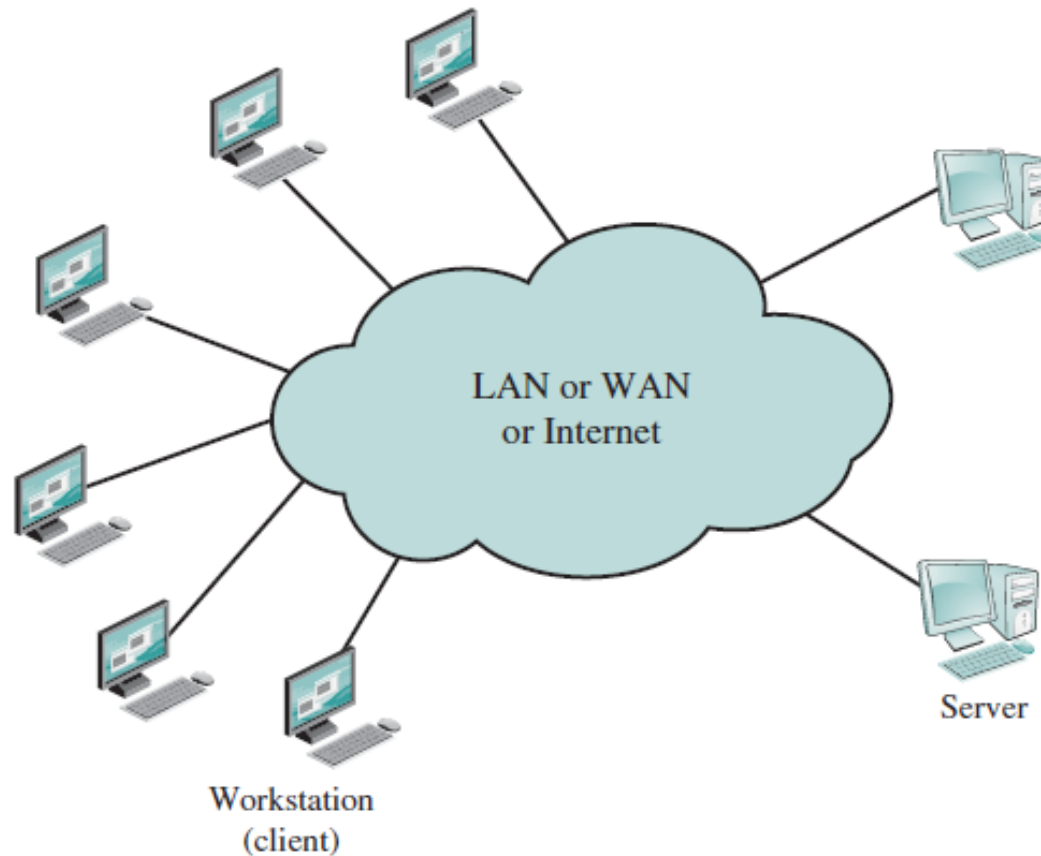


Разпределени системи

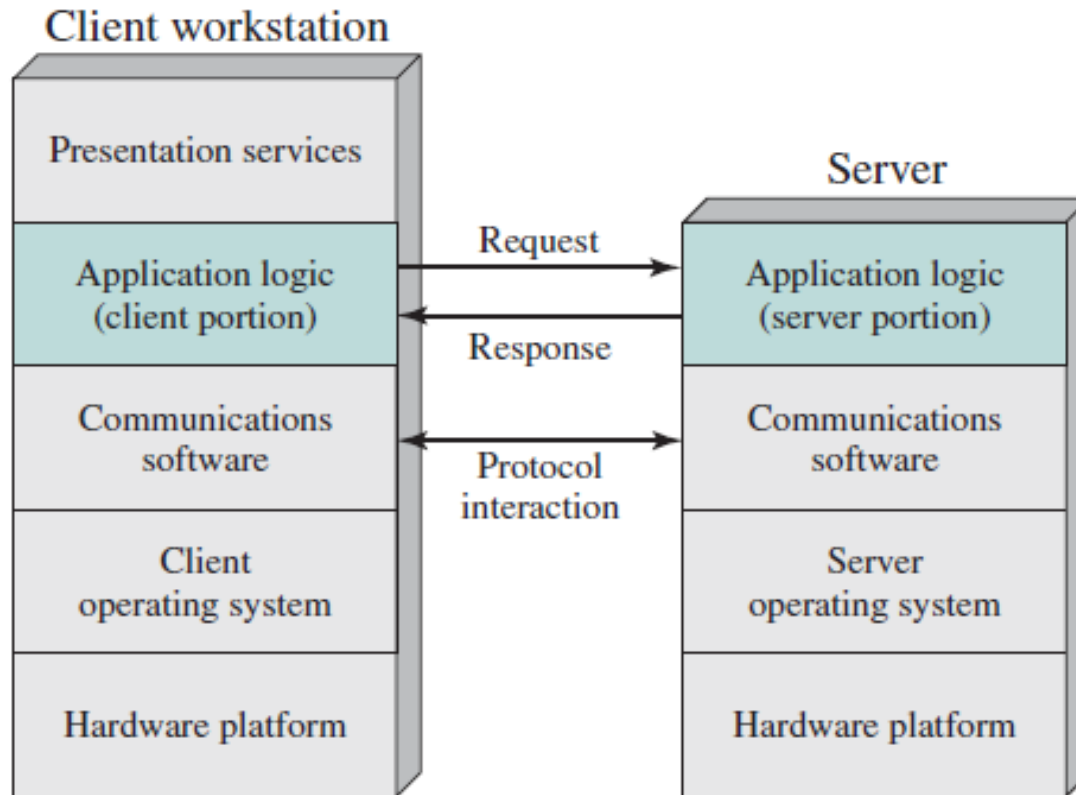
проф. д-р инж. Христо Вълчанов

<http://cs.tu-varna.bg>

Модел клиент-сървър

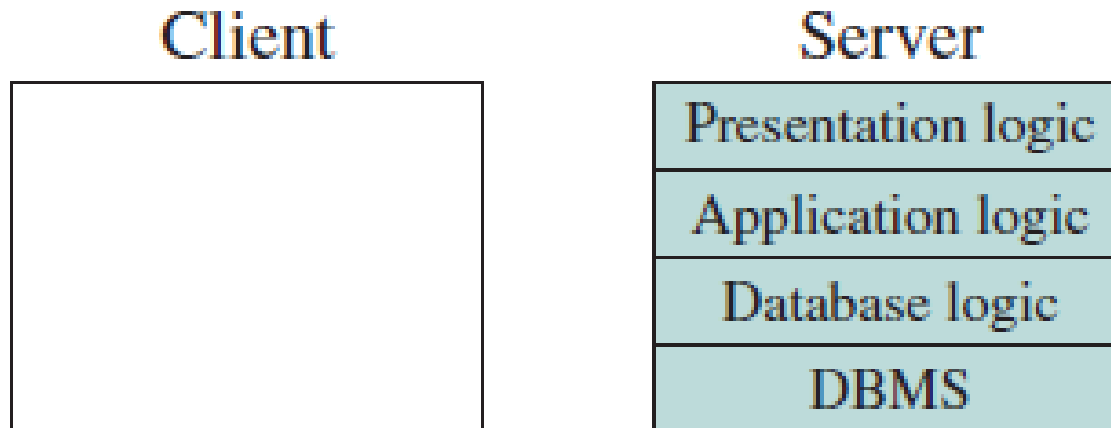


Клиент-сървър приложения



Класове клиент-сървър приложения

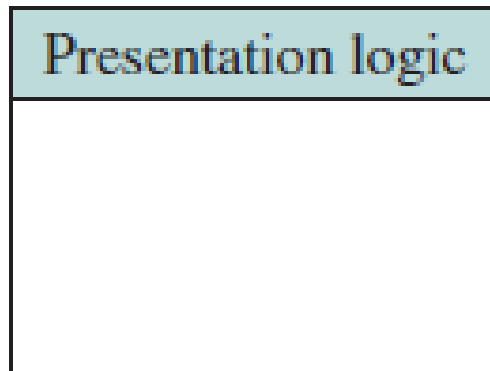
Хост-базирана обработка



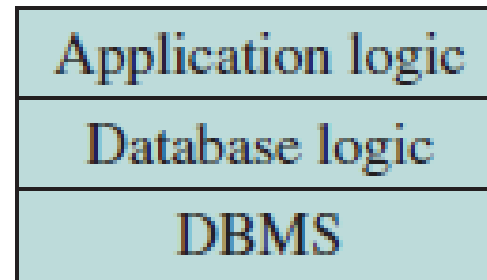
Класове клиент-сървър приложения

Сървър-базирана обработка

Client



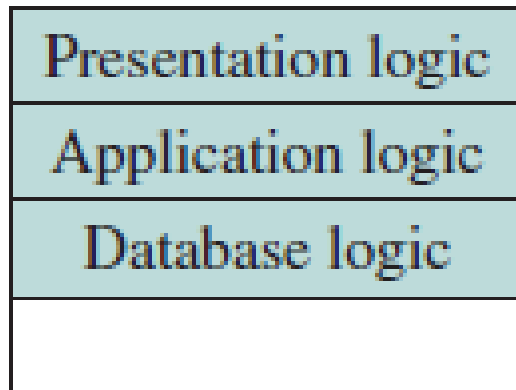
Server



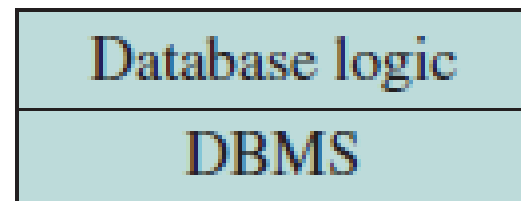
Класове клиент-сървър приложения

Клиент-базирана обработка

Client



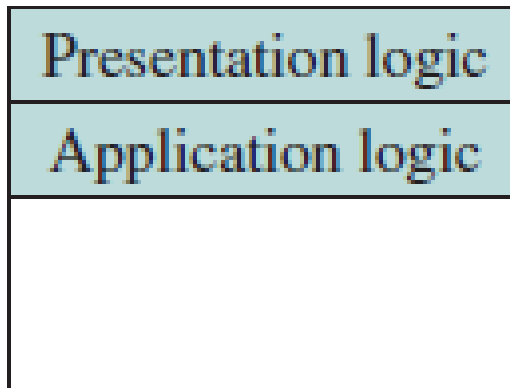
Server



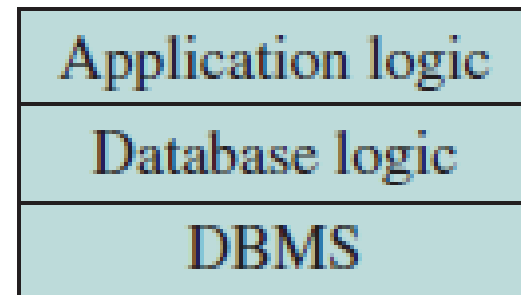
Класове клиент-сървър приложения

Кооперативна обработка

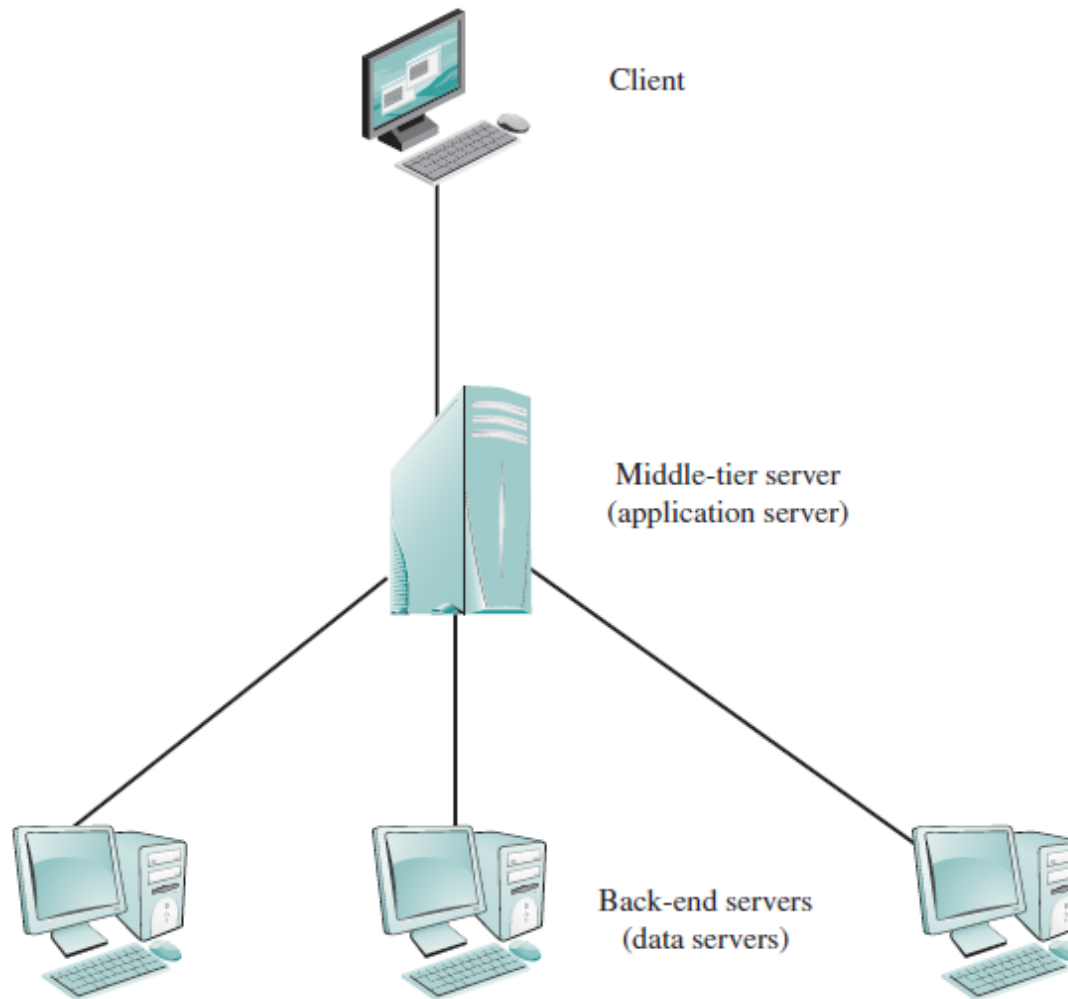
Client



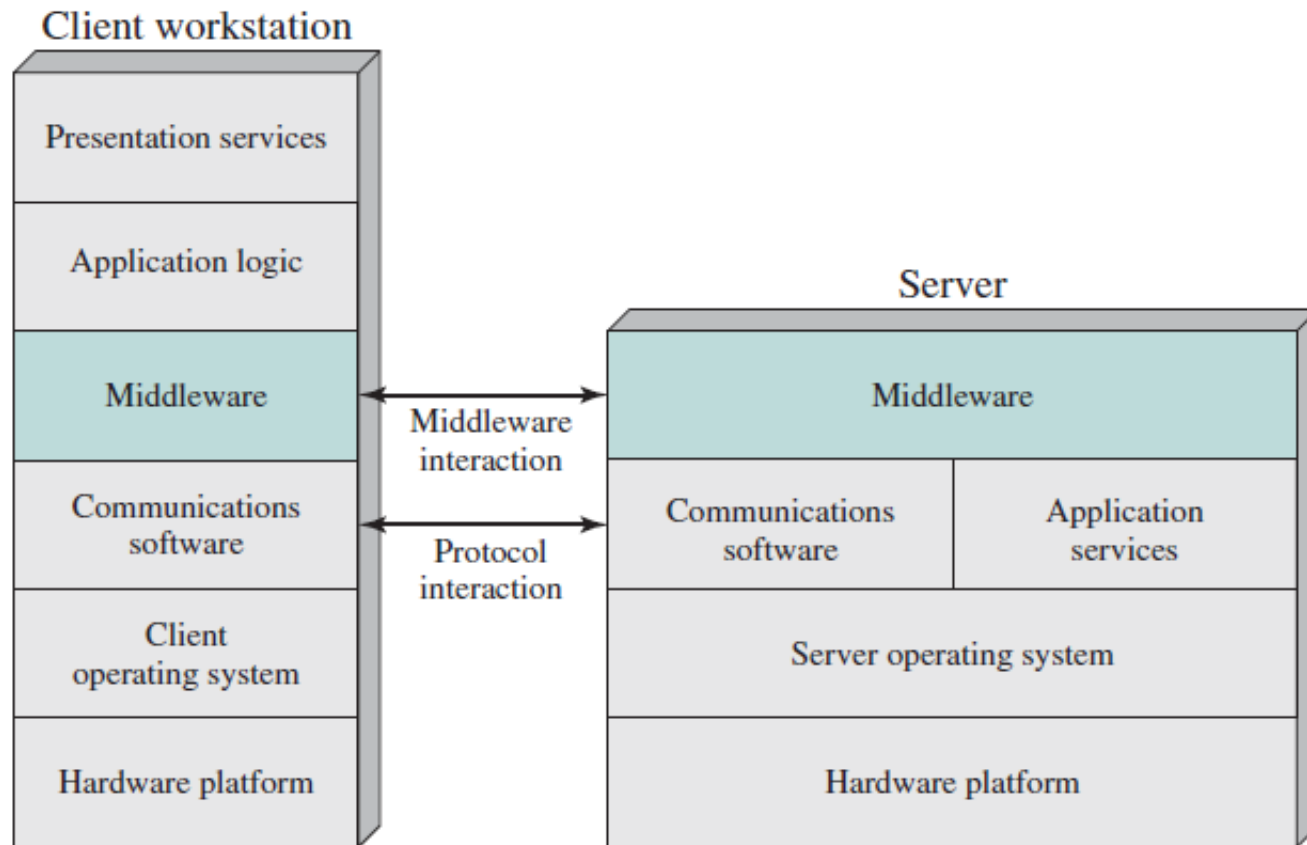
Server



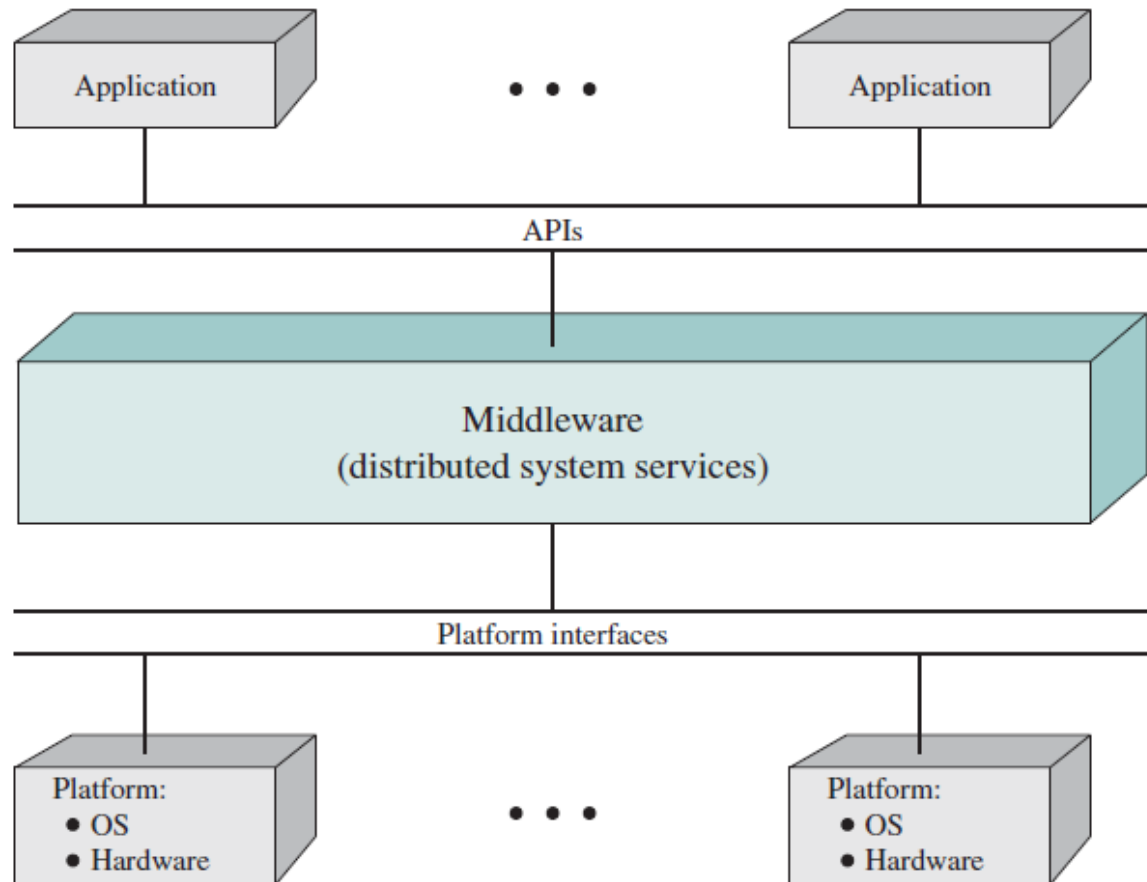
Трислойна клиент-сървър архитектура



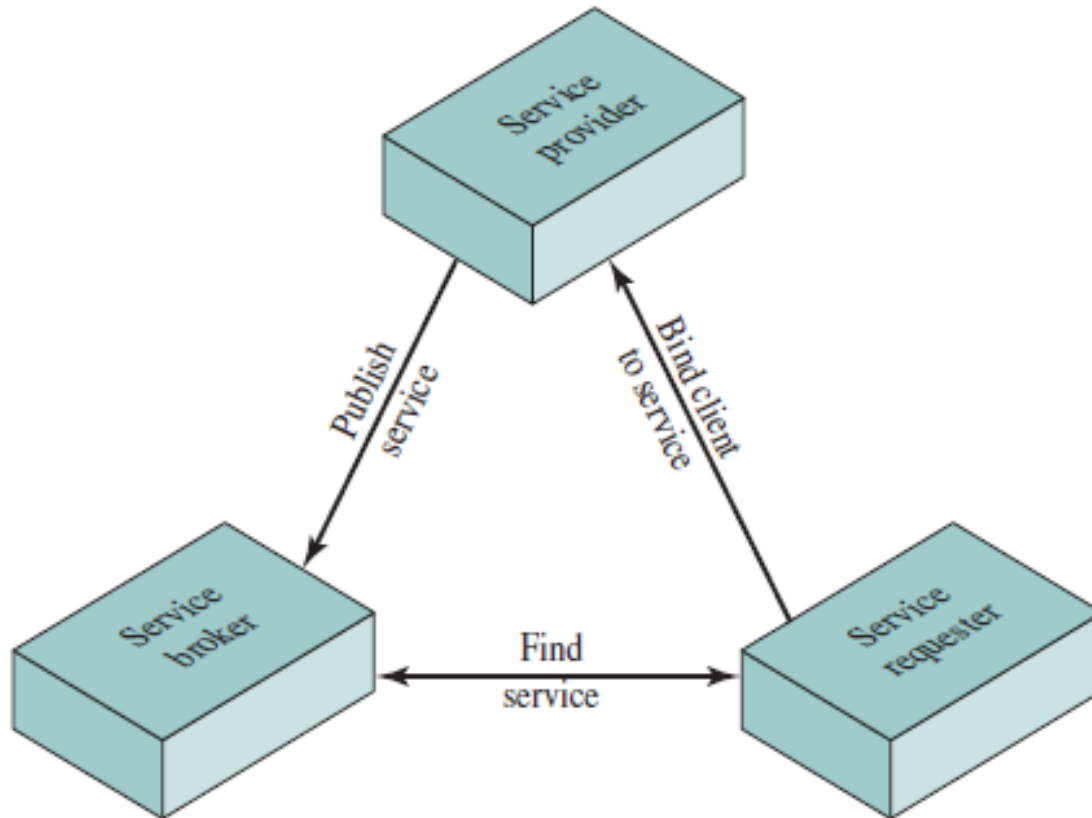
Middleware



Логическо представяне на middleware

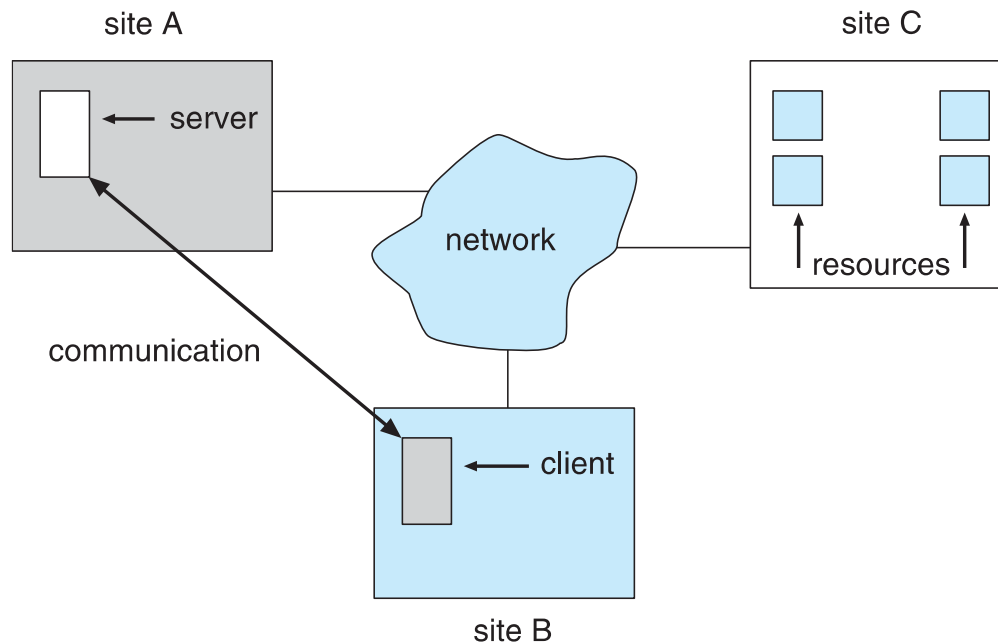


Сървис ориентирана архитектура (SOA)



Разпределена система

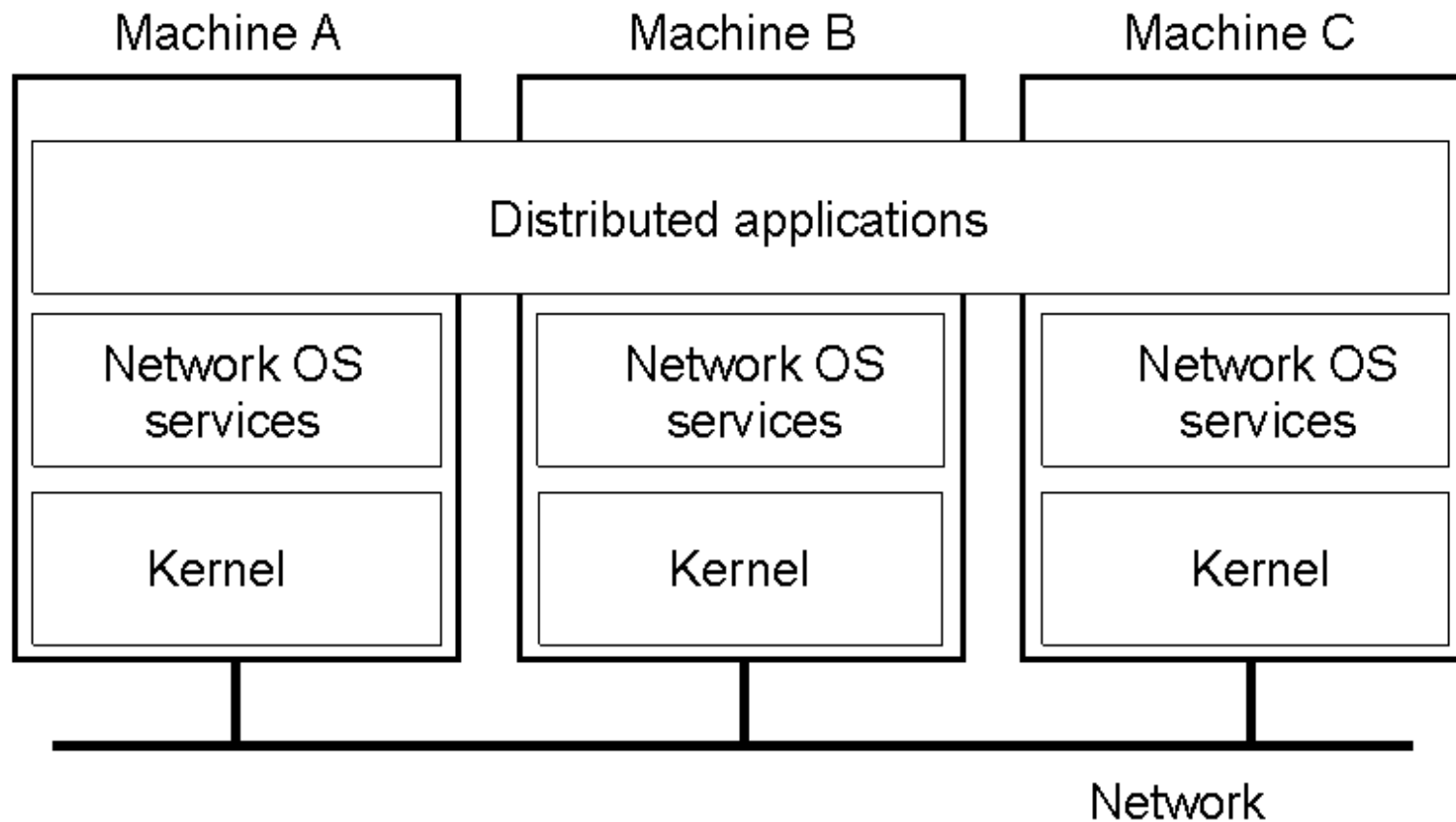
Съвкупност от слабо свързани машини, комуникиращи си през комуникационна мрежа.



Мрежова ОС

- Слабо-свързана система от хетерогенни машини.
- Потребителите знаят за наличието на множество машини
- Достъпът до ресурсите на различни машини е явен:
 - Отдалечено логване (telnet, ssh);
 - Remote Desktop;
 - Прехвърляне на данни от отдалечена на локална машина чрез File Transfer Protocol
- Потребителите трябва да осъществят сесия и да използват мрежови команди

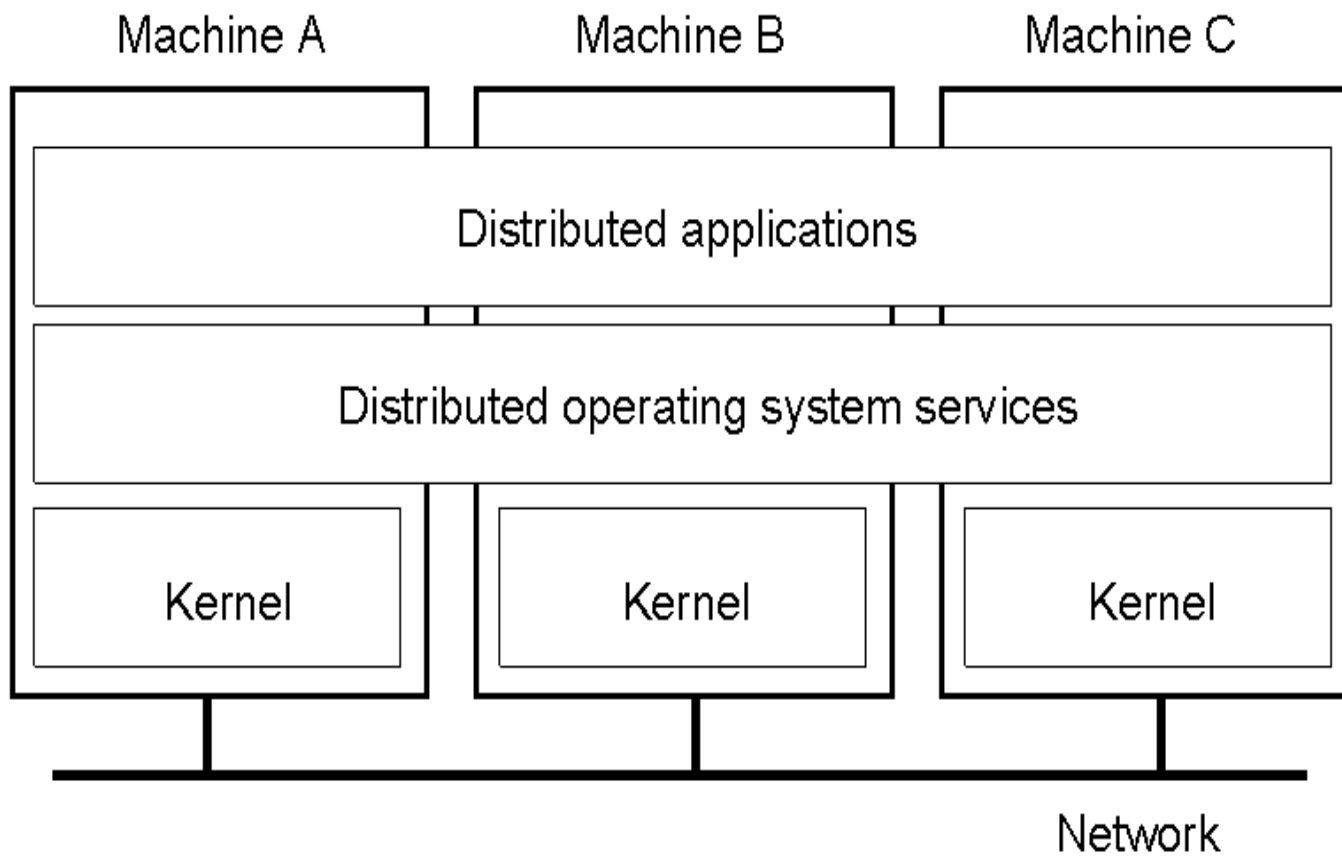
Мрежова ОС



Разпределена ОС

- Силно-свързана система от хомогенни машини
- Потребителите не знаят за наличието на множество машини
- Достъпът до отдалечени ресурси е подобен на този до локални ресурси

Разпределена ОС



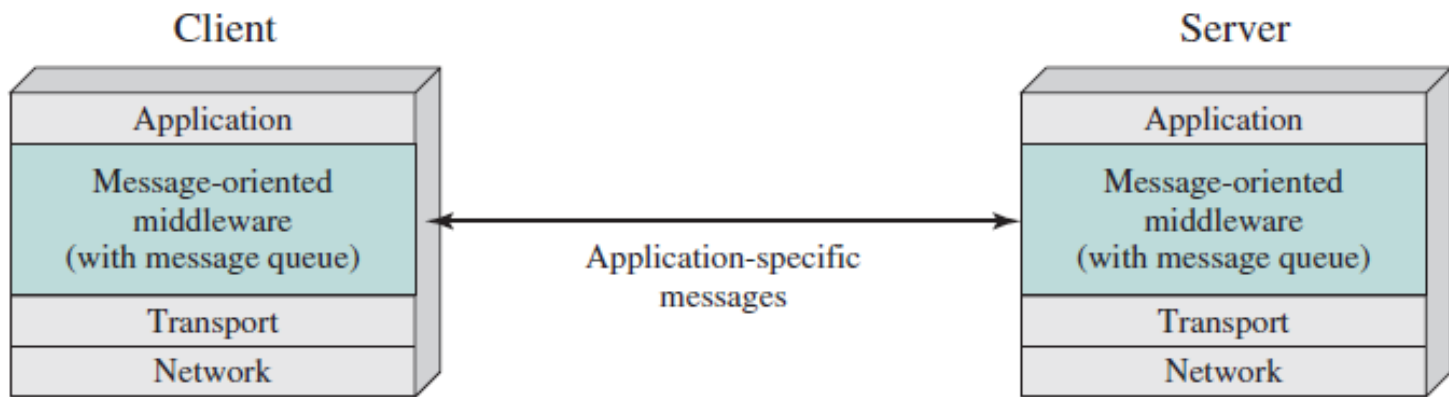
Разпределена ОС

- Мигриране на данни – прехвърляне на цял файл или на тази част, която е необходима за задачата
- Мигриране на изчисленията – прехвърляне на изчисления вместо данни
- Мигриране на процеси – изпълнение на цял процес или част от него на различни места

Разпределена ОС - цели

- Прозрачност
- Гъвкавост
- Надеждност
- Производителност
- Разширяемост

Обмен на съобщения



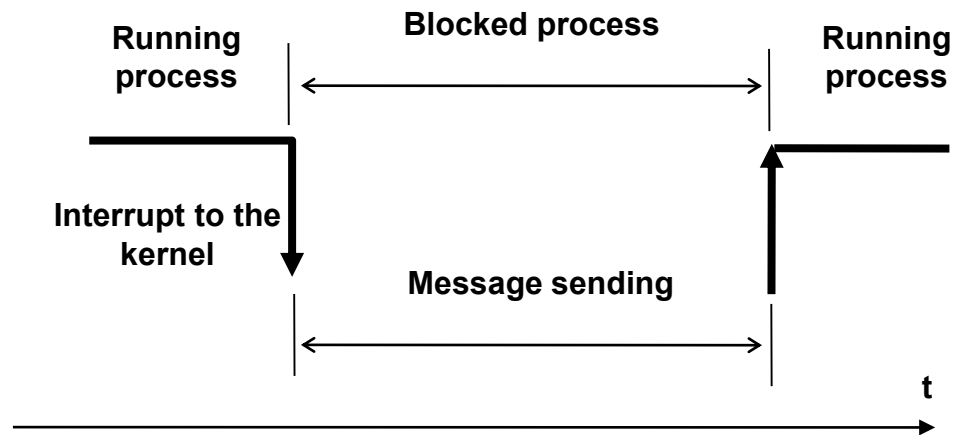
- `send()`
- `receive()`

Комуникация

- Блокиращи извиквания
- Неблокиращи извиквания

Синхронна комуникация

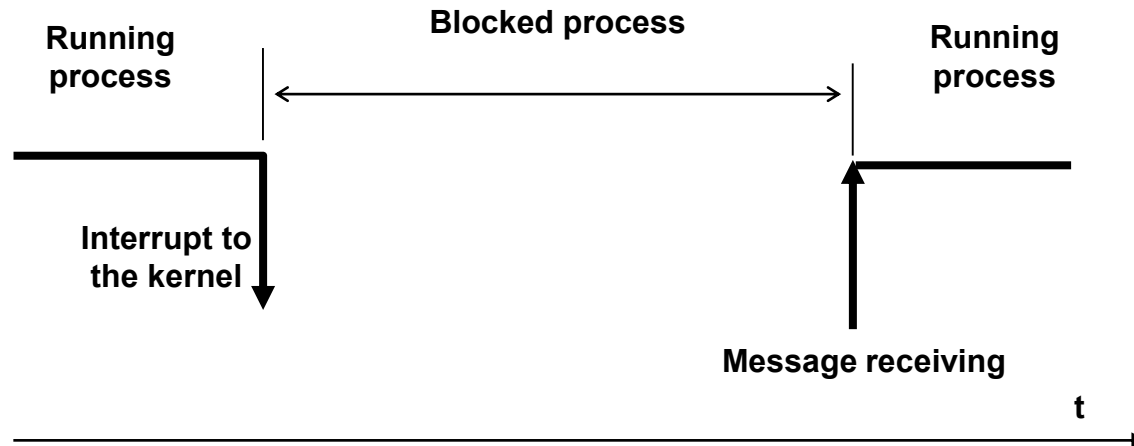
Блокиращо извикване *send()*



Процесът ще се блокира до завършване изпращането на цялото съобщение.

Синхронна комуникация

Блокиращо извикване *receive()*



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

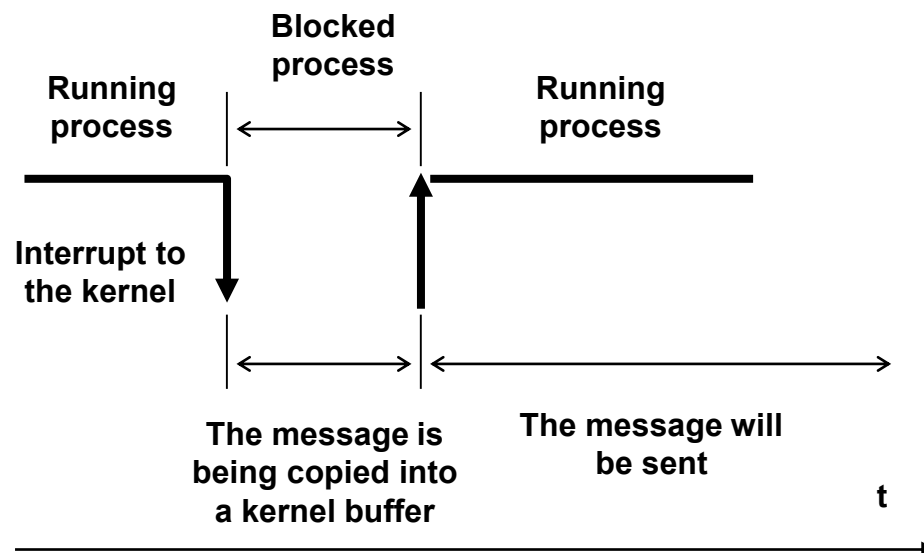
Синхронна комуникация

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

Неблокиращо извикване *send()*



Асинхронна комуникация

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

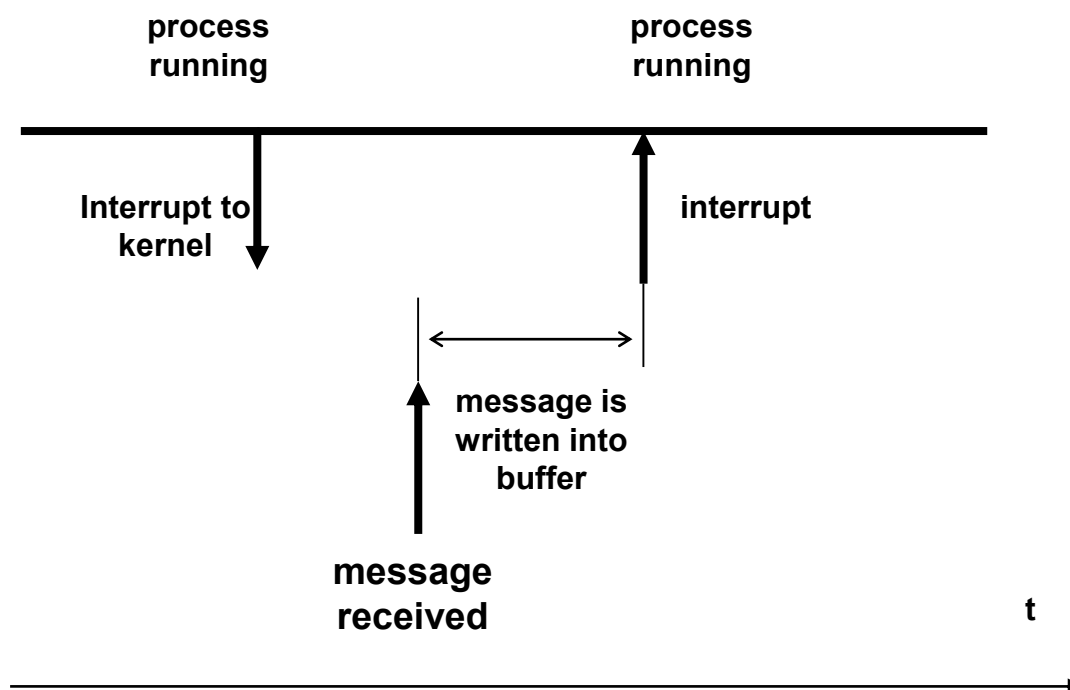
Асинхронна комуникация

Решение 2: 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

Неблокиращ receive()



Асинхронна комуникация

Проблем: Как процесът ще разбере ако съобщението е било вече записано в буфера на процеса?

Решение:

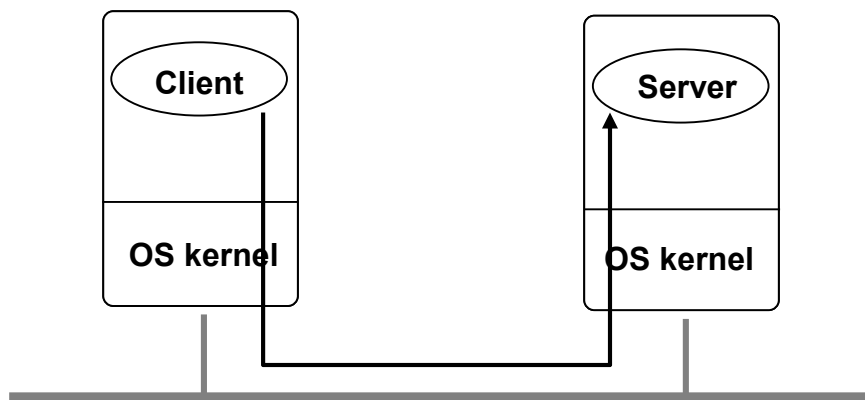
- С прекъсване към процеса след запис в буфера;
- Процесът периодично пита ядрото (polling) – допълнителни средства;
- При извикване на *receive()*, ядрото веднага дава съобщението или връща код за грешка, ако съобщението още не е получено.

Буфериране на съобщенията

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

Решение:

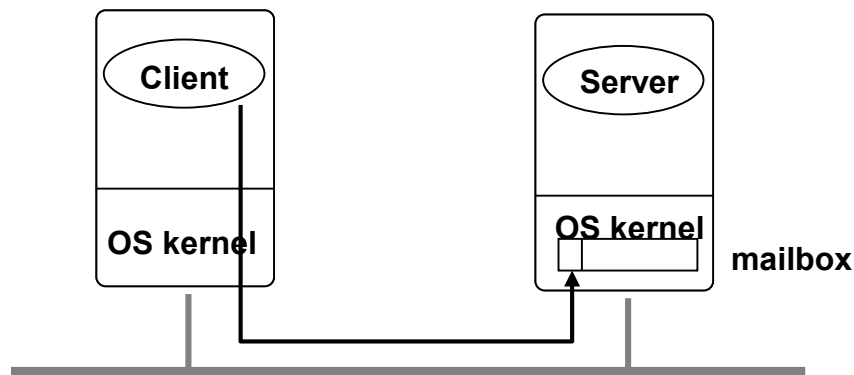
- Отхвърляне на съобщението, надявайки се, че клиентът ще го прати отново след като сървърът успее с *receive()*;
- Съхраняване на получените съобщения в ядрото за период чрез таймер. Изисква време и заемане на памет



Буфериране на съобщенията

С буфефиране – 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.



Remote Procedure Call

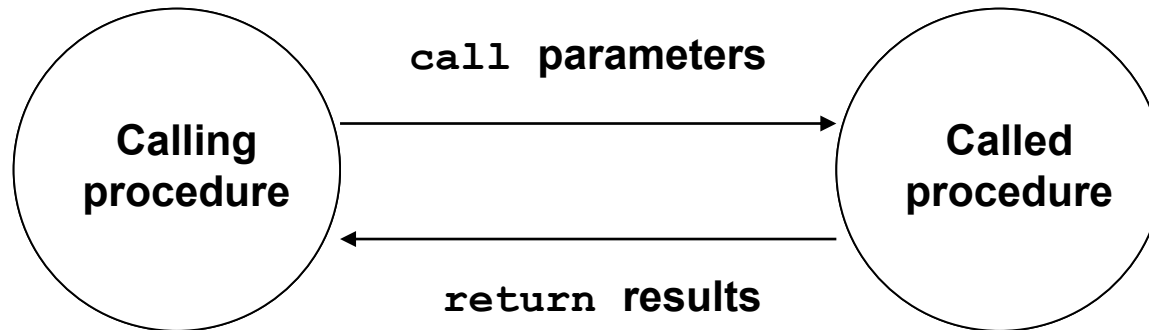
Основна идея:

Процес от машина А извиква процедура от машина В. Извикващият процес се блокира докато процедурата се изпълни на другата машина и резултатите се върнат обратно.

Основна цел: прозрачност

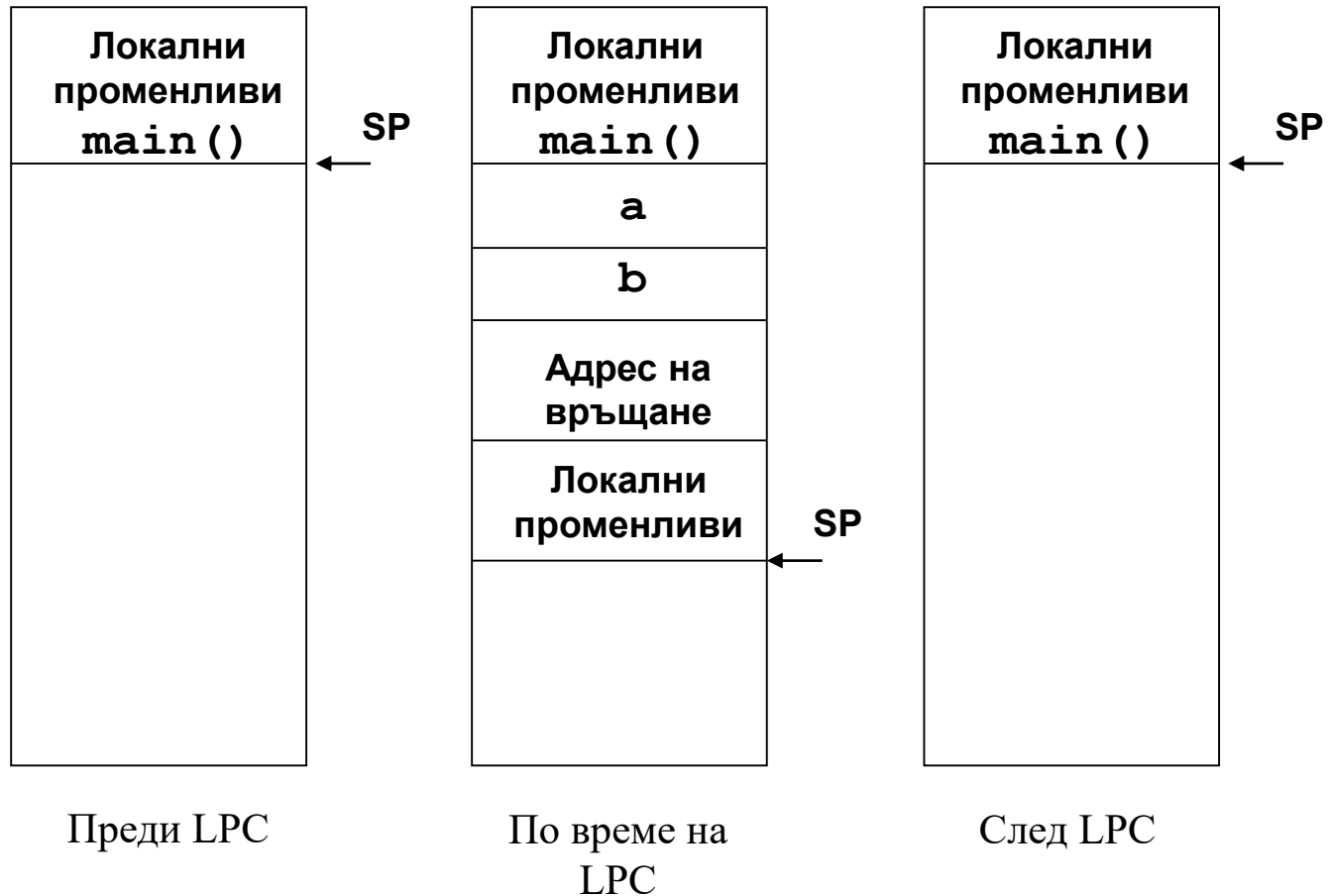
Локално извикване на процедура (LPC)

Процес изпълнява процедура в неговото адресно пространство.



Предаване на параметри при LPC

```
s = null ( a, b );
```

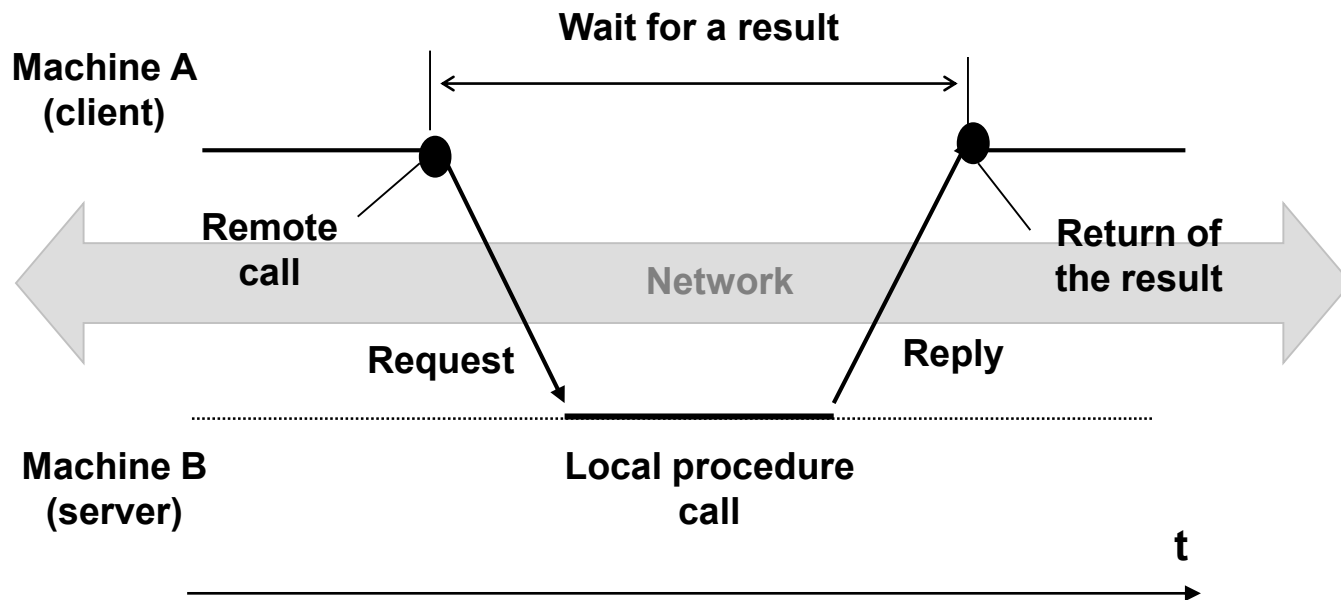


Предаване на параметри при LPC

- ***Call-by-value*** (по стойност) - Променливите се копират в стека. Модифицирането им от извиканата процедура не променя стойността на оригиналните променливи в извикващата процедура;
- ***Call-by-reference*** (по адрес) - В стека се предават само адресите на променливите. Модифицирането им от извиканата процедура води до промяна на стойността на оригиналните променливи в извикващата процедура;
- ***Call-by-copy/restore*** (копиране/възстановяване) – Комбинация от предходните методи.

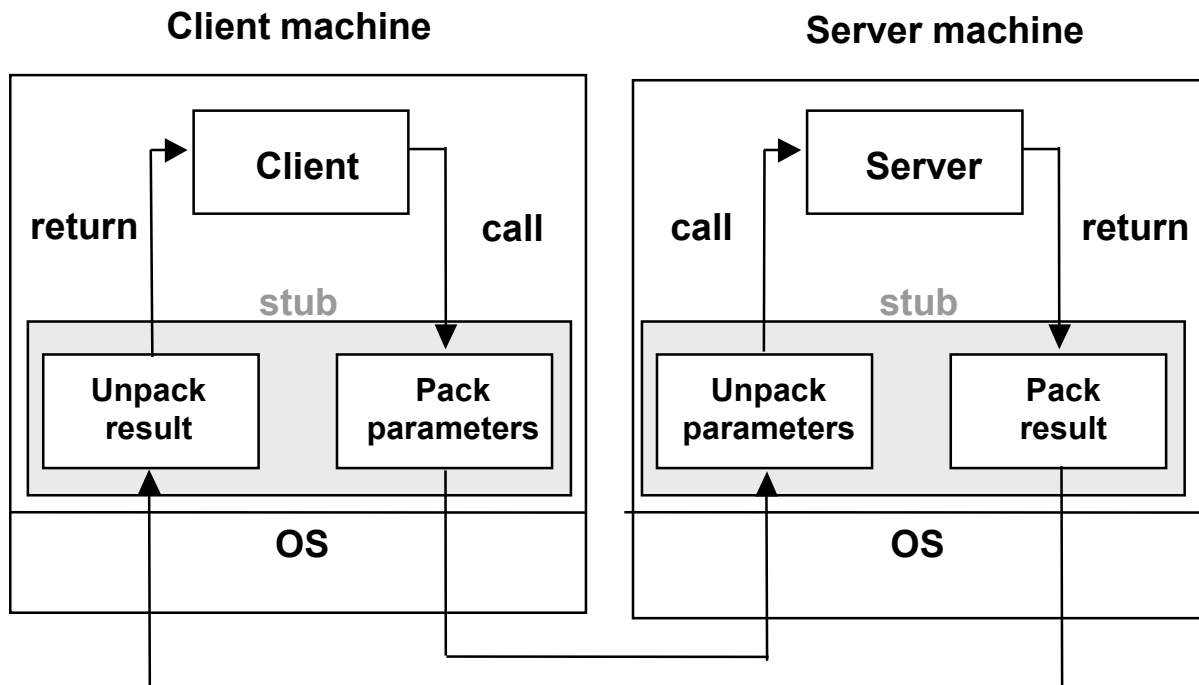
Remote Procedure Call

- Извикващият и извикваният процес са независими един от друг;
- Няма общ стек;
- Използва се комуникационен модел *request-reply*;
- Използват се съобщения между процесите.



RPC комуникационен интерфейс

- Базиран е на библиотечни функции – **стъбове** (*stubs*);
- Клиентът и сървърът комуникират чрез два стъба, по един за всеки от тях.

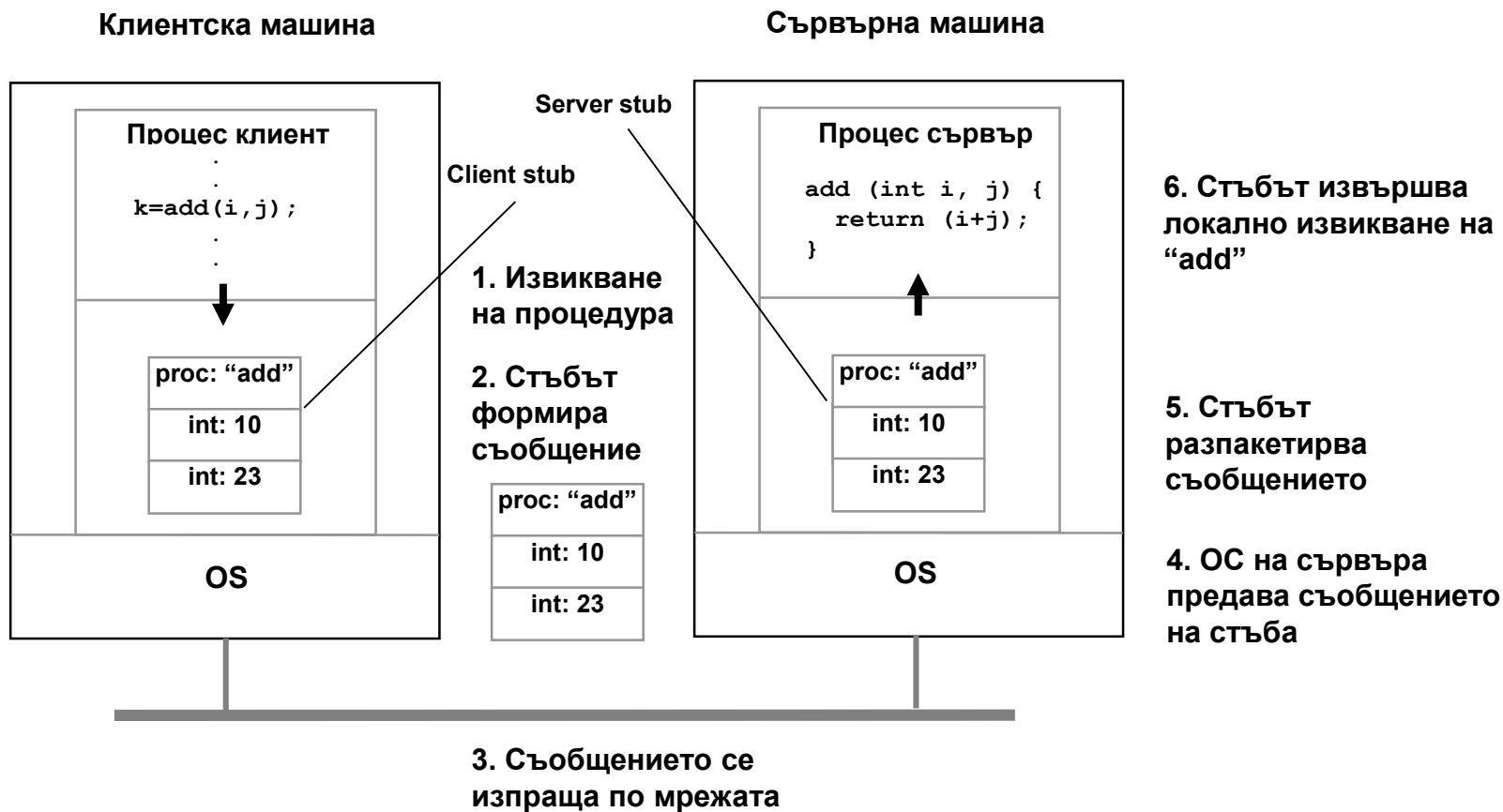


RPC алгоритъм

1. Клиентската процедура извиква клиентския стъб;
2. Клиентския стъб подготвя съобщения и изпълнява системно извикване към локалната ОС;
3. Клиентската ОС изпраща съобщение към сървърната ОС;
4. Отдалечената ОС предава съобщението на сървърния стъб;
5. Сървърният стъб разпакетирва параметрите и извиква сървъра;
6. Сървърният процес обработва данните и връща резултата към сървърния стъб;
7. Сървърният стъб пакетира резултата в съобщение и изпълнява системно извикване към локалната ОС;
8. Сървърната ОС изпраща съобщението към клиентската ОС;
9. Клиентската ОС предава съобщението на клиентския стъб;
10. Клиентският стъб разпакетирва резултата и връща изпълнението към клиентската програма.

RPC пример

```
k = add ( 10 , 23 );
```



Пакетиране на параметри при RPC

Пакетирането на параметрите в съобщение – ***parameter marshaling.***

Две важни задачи за решаване:

- Различни формати на данните;
- Изпращане на указатели.

Формати на данни при RPC

Проблеми:

- Различни архитектури на клиента и сървъра;
- Различни символни таблици;
- Различни числови формати;
- Различно подреждане на байтовете.

Подреждане на байтовете

192	168	0	20
-----	-----	---	----

Big-endian

192	A
168	A+1
0	A+2
20	A+3

192	168	0	20
-----	-----	---	----

Little-endian

20	A
0	A+1
168	A+2
192	A+3

Решаване на проблема

Използване на унифицирана канонична форма за представяне на данните – *External Data Representation (XDR)*:

- Машинно независим формат;
- Big-endian подреждане на данните;
- IEEE представяне на числата с плаваща запетая;
- ASCII представяне на символите.

Представянето в канонична форма – ***serialization***
Обратното представяне – ***deserialization***

RPC и указатели

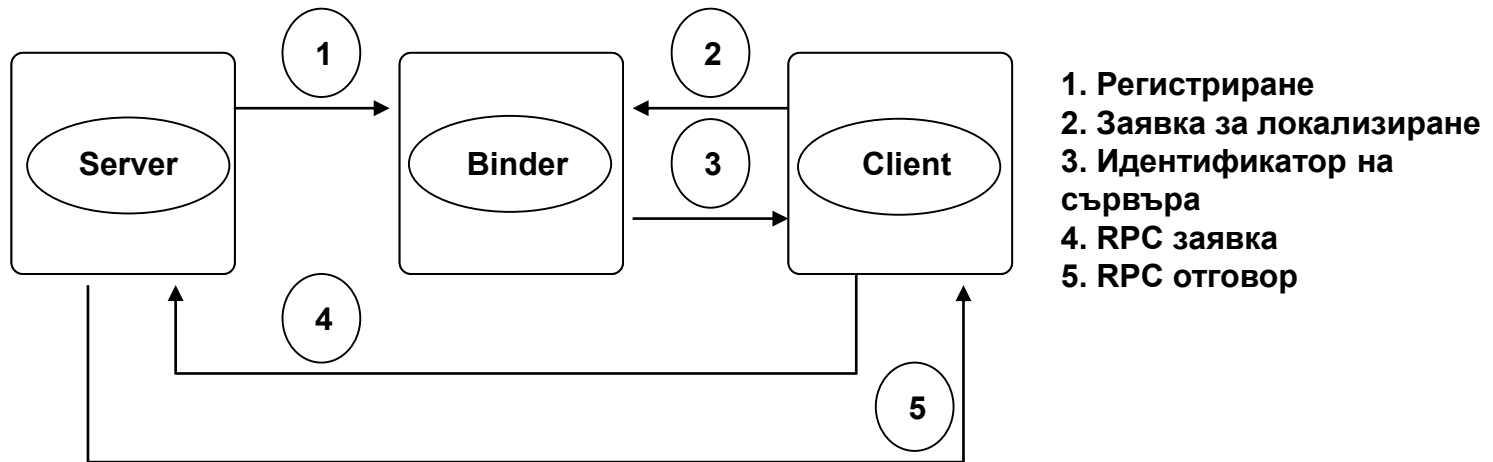
Проблем: указателите са валидни само в локалното адресно пространство на процеса.

Решения:

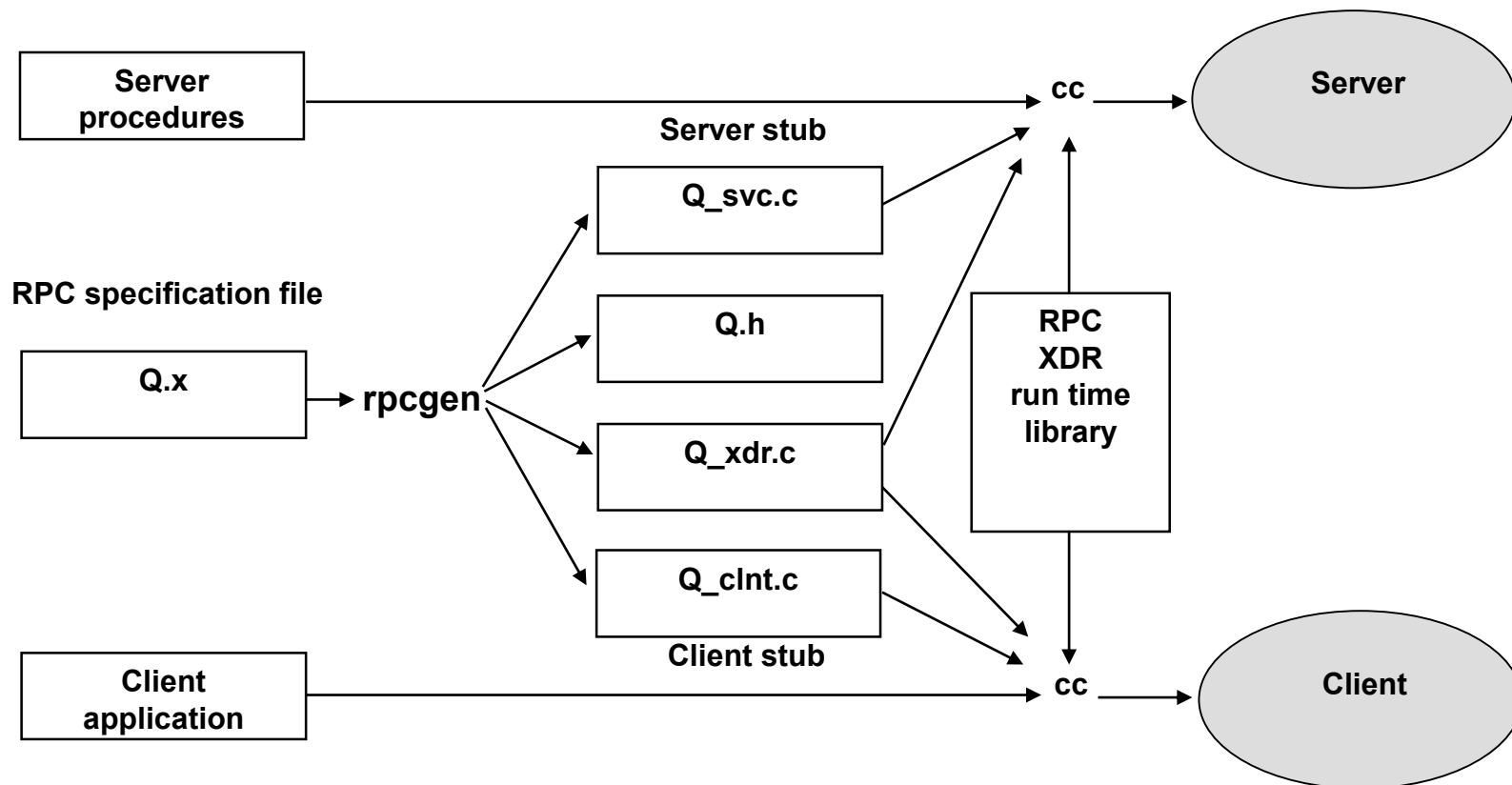
- Изпращане само на стойности, не на указатели;
- Изпращане на указатели към сървърния стъб и генериране на специален код за обработка на тези указатели.

Локализиране на RPC сървър

- Сървърът трябва да се регистрира в **binder**.
- Клиентът се свързва към binder за локализиране на сървъра (динамично свързване).



Генериране на стъбове (sunRPC)



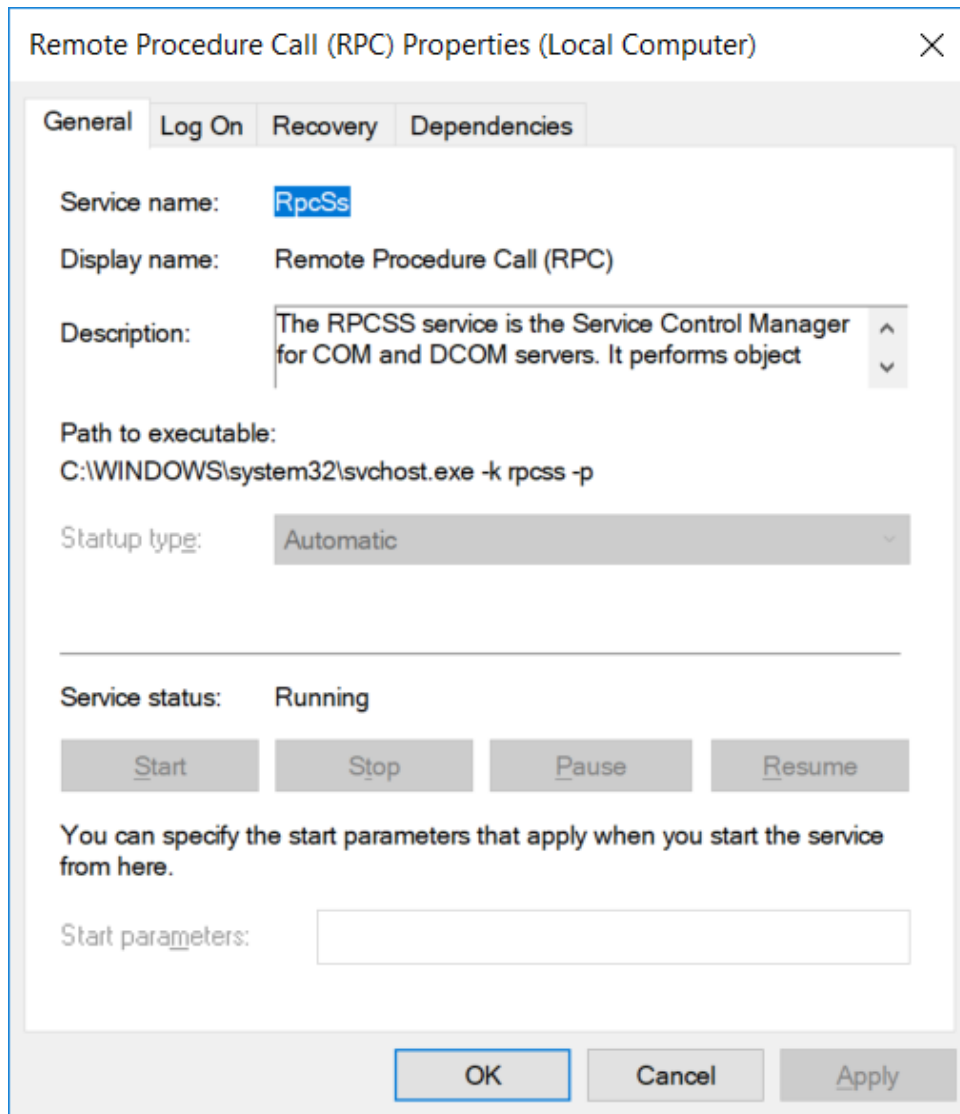
Спецификация на протокола

Използва се RPCL (RPC Language)

```
/*  time.x  */
/* Defines two procedures
    gettime_1 - returns the binary date and time
    getstrtime_1 - takes a binary date and returns a string
*/

program TIMEPROG {
    version TIMEVERS {
        long GETTIME(void) = 1;    /* procedure number = 1 */
        string GETSTRTIME(long) = 2; /* procedure number = 2 */
    } = 1;                        /* version number = 1 */
} = 0x20000001;                  /* program number = 0x20000001 */
```

Пример: Windows RPC



Пример: Windows RPC

```
Administrator: Command Prompt
C:\WINDOWS\system32>sc enumdepend rpcss 100000
[SC] EnumDependentServices: entriesread = 183

SERVICE_NAME: GoogleChromeElevationService
DISPLAY_NAME: Google Chrome Elevation Service
        TYPE               : 10  WIN32_OWN_PROCESS
        STATE                : 1  STOPPED
        WIN32_EXIT_CODE       : 1077  (0x435)
        SERVICE_EXIT_CODE   : 0  (0x0)
        CHECKPOINT           : 0x0
        WAIT_HINT            : 0x0

SERVICE_NAME: PrintWorkflowUserSvc_1d07169e
DISPLAY_NAME: PrintWorkflow_1d07169e
        TYPE               : f0  ERROR
        STATE                : 4  RUNNING
                        (STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE       : 0  (0x0)
        SERVICE_EXIT_CODE   : 0  (0x0)
        CHECKPOINT           : 0x0
        WAIT_HINT            : 0x0

SERVICE_NAME: DevicesFlowUserSvc_1d07169e
DISPLAY_NAME: DevicesFlow_1d07169e
        TYPE               : e0  USER_SHARE_PROCESS_INSTANCE
        STATE                : 1  STOPPED
        WIN32_EXIT_CODE       : 1077  (0x435)
        SERVICE_EXIT_CODE   : 0  (0x0)
        CHECKPOINT           : 0x0
        WAIT_HINT            : 0x0
```


Въпроси?