

Multiagentové systémy

Dr. Andrej Lúčný

KAI FMFI UK

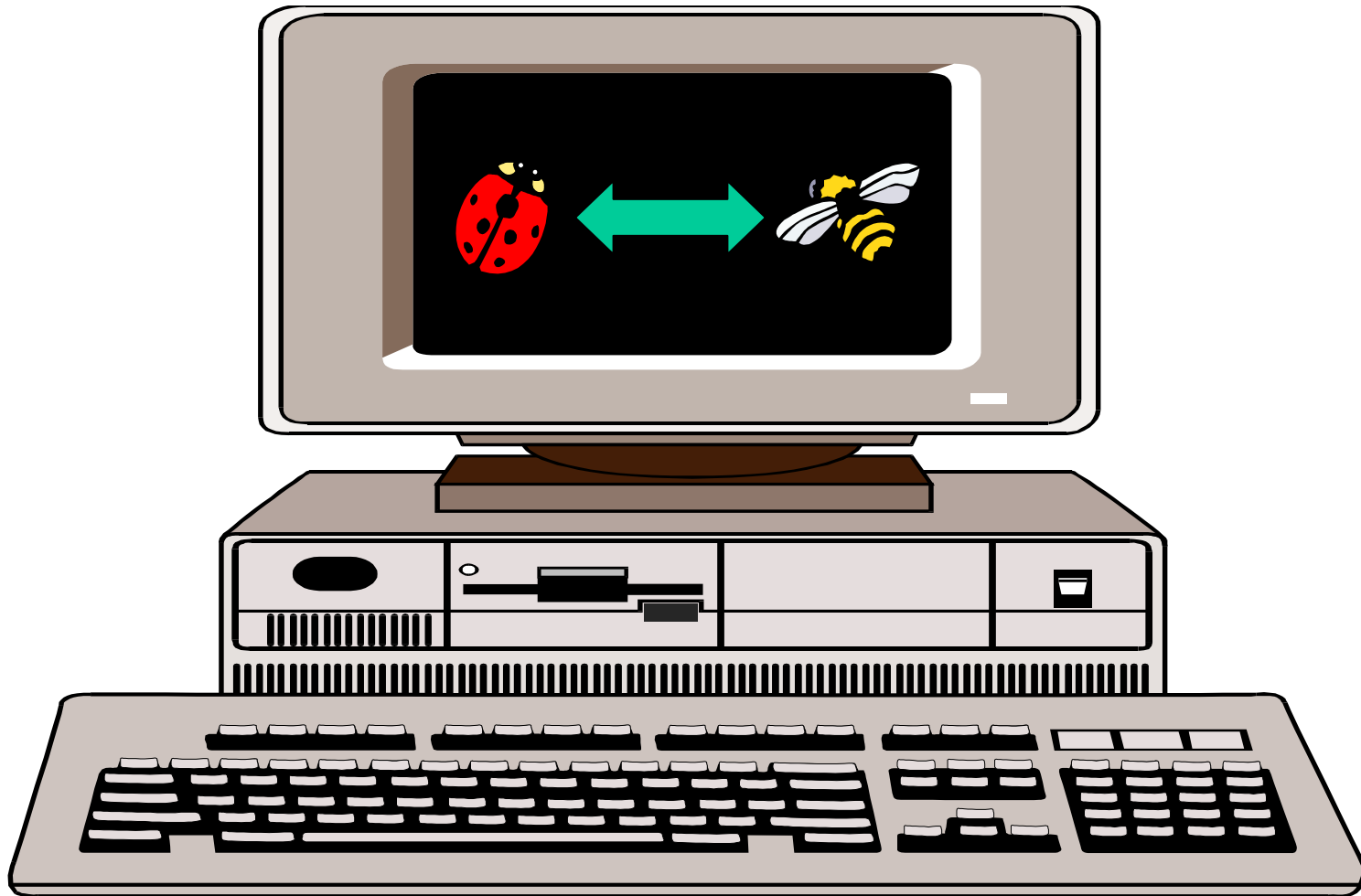
andy@microstep-mis.com

<http://www.microstep-mis.sk/~andy>

Opakovanie

- čo umožňuje výmenu dát medzi vláknami JVM ?
 - aké synchronizačné mechanizmy JVM má ?
 - čo je messageQueue ?
 - akého vzoru je objekt Space ?
 - aké postavenie majú MAS na VM ?
-
- akými softwarovými prostriedkami implementovať komunikáciu medzi agentami v rámci IPC ?

MAS ako IPC



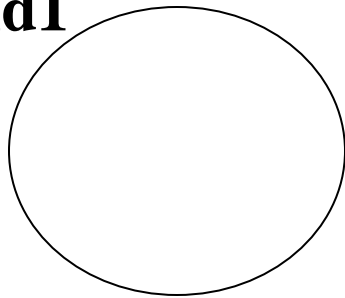
Inter Process Communication

História komunikácie medzi procesmi:

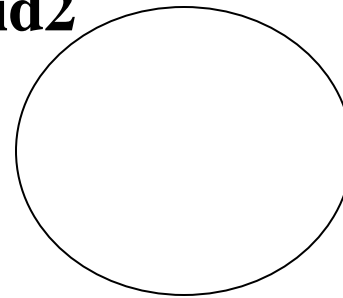
- signály
- zdieľaná pamäť
- pipe, socket
- posielanie správ – message passing

Procesy: pid

pid1



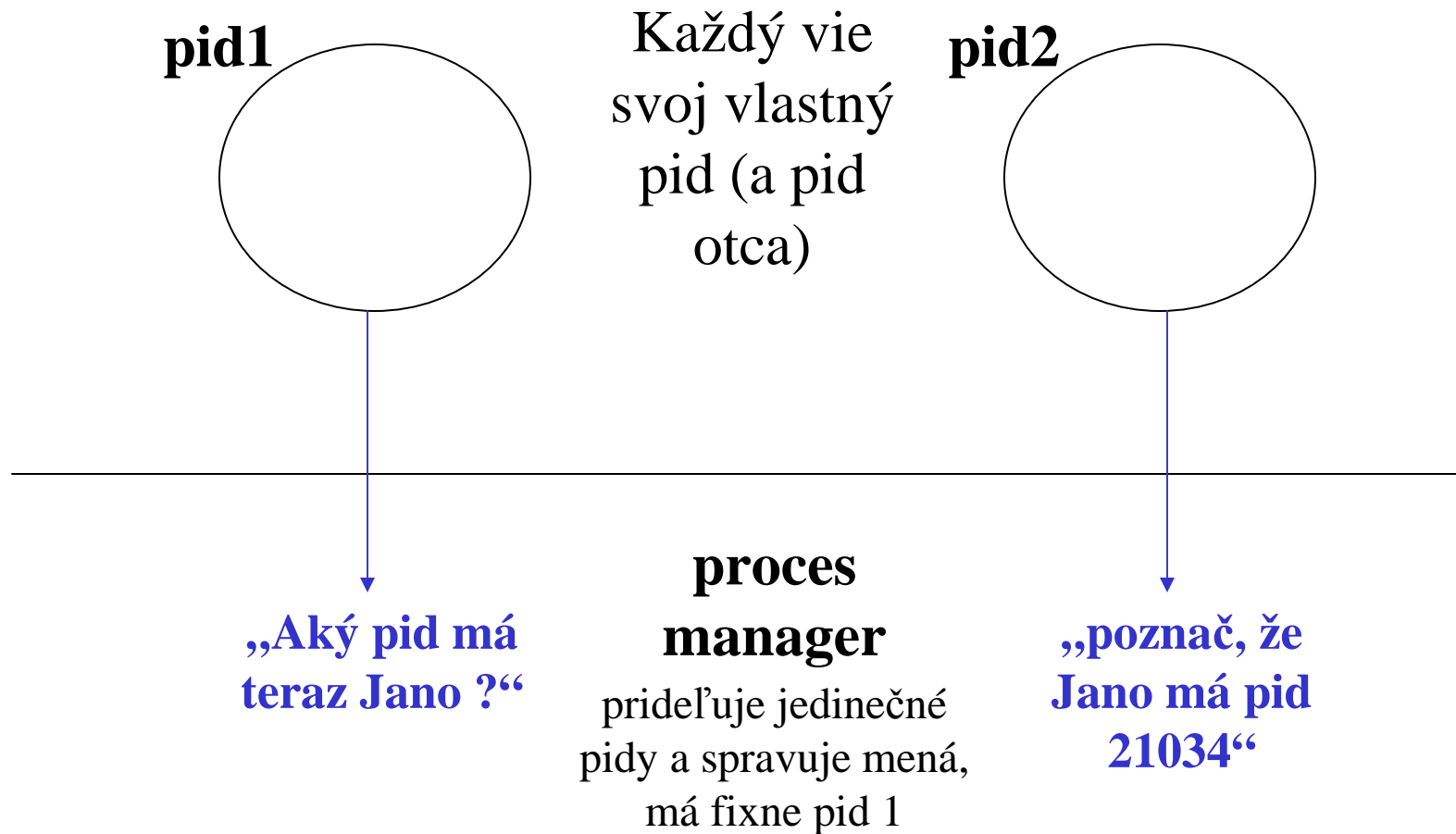
pid2



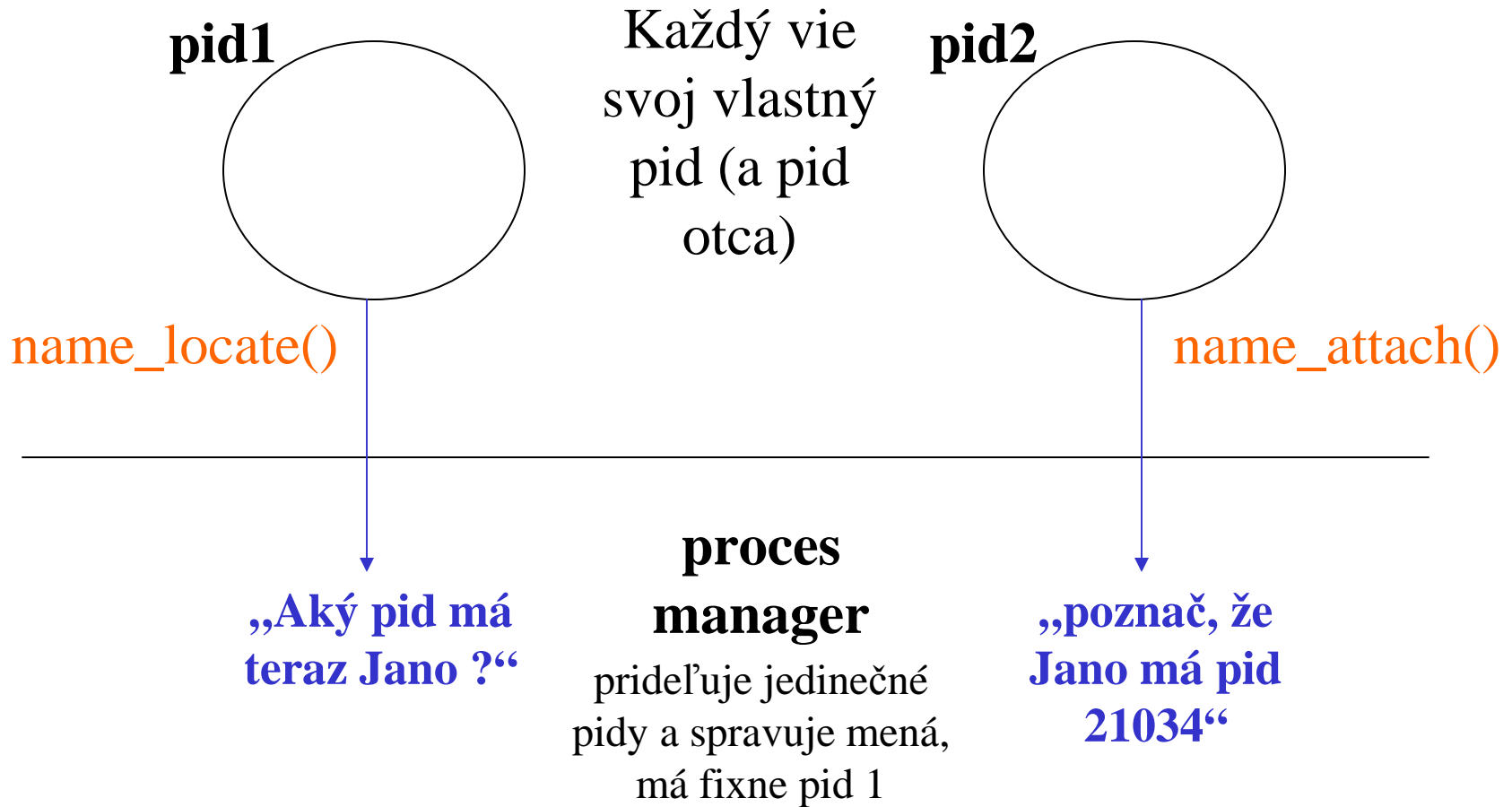
OS

jeden Proces môže komunikovať s
druhým pokiaľ vie jeho pid

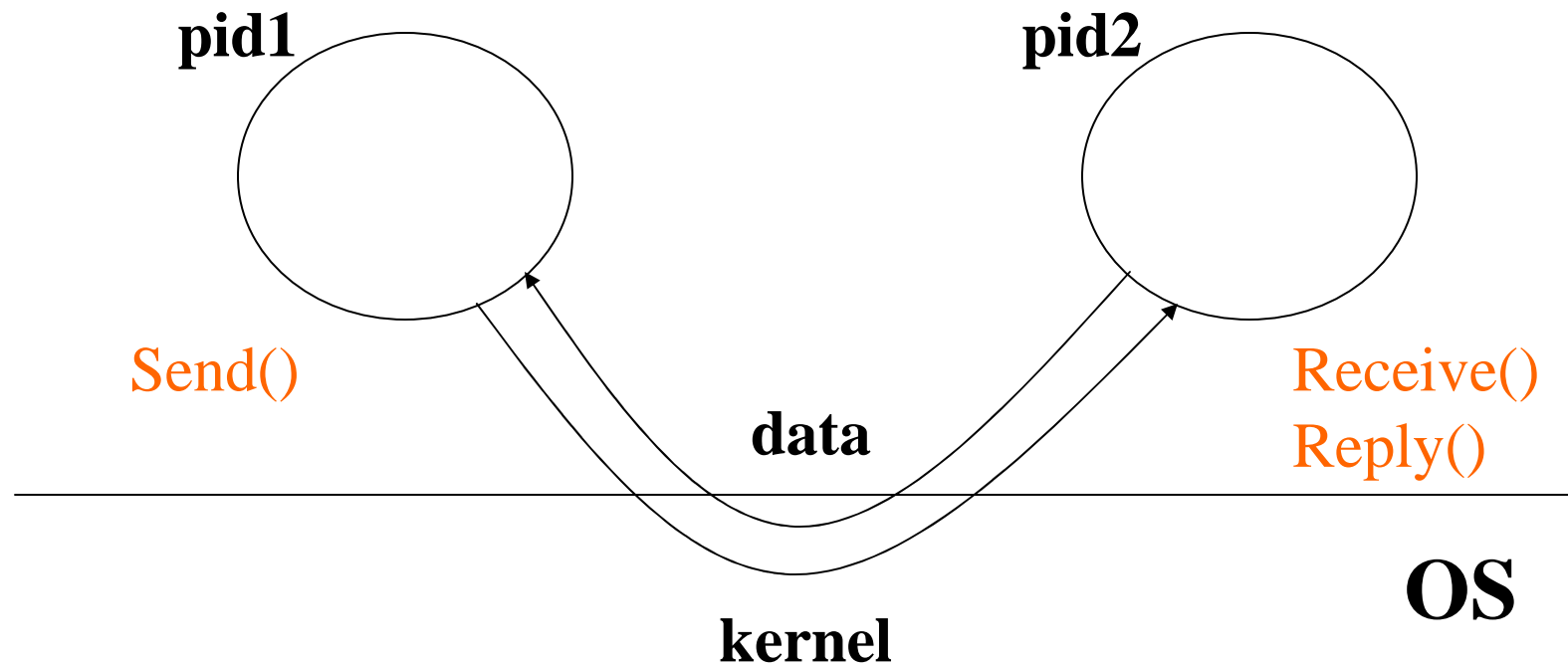
Procesy: names



Procesy: names

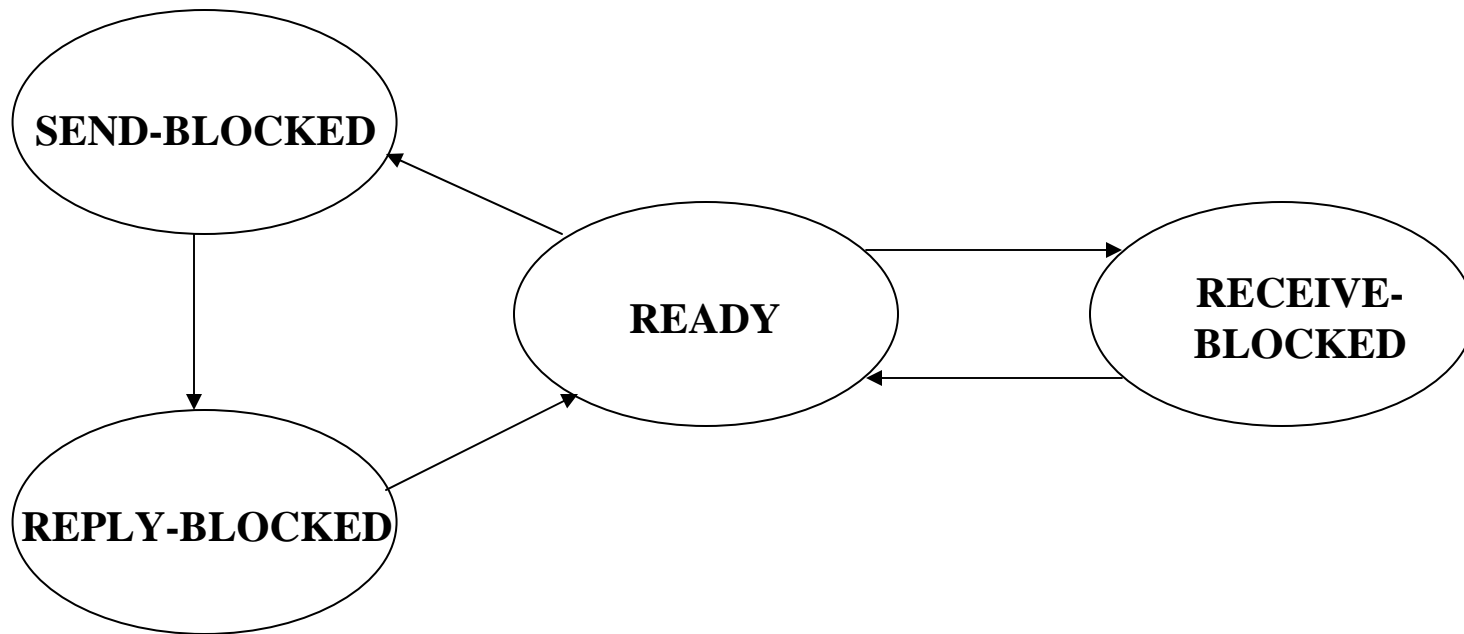


Procesy: komunikácia

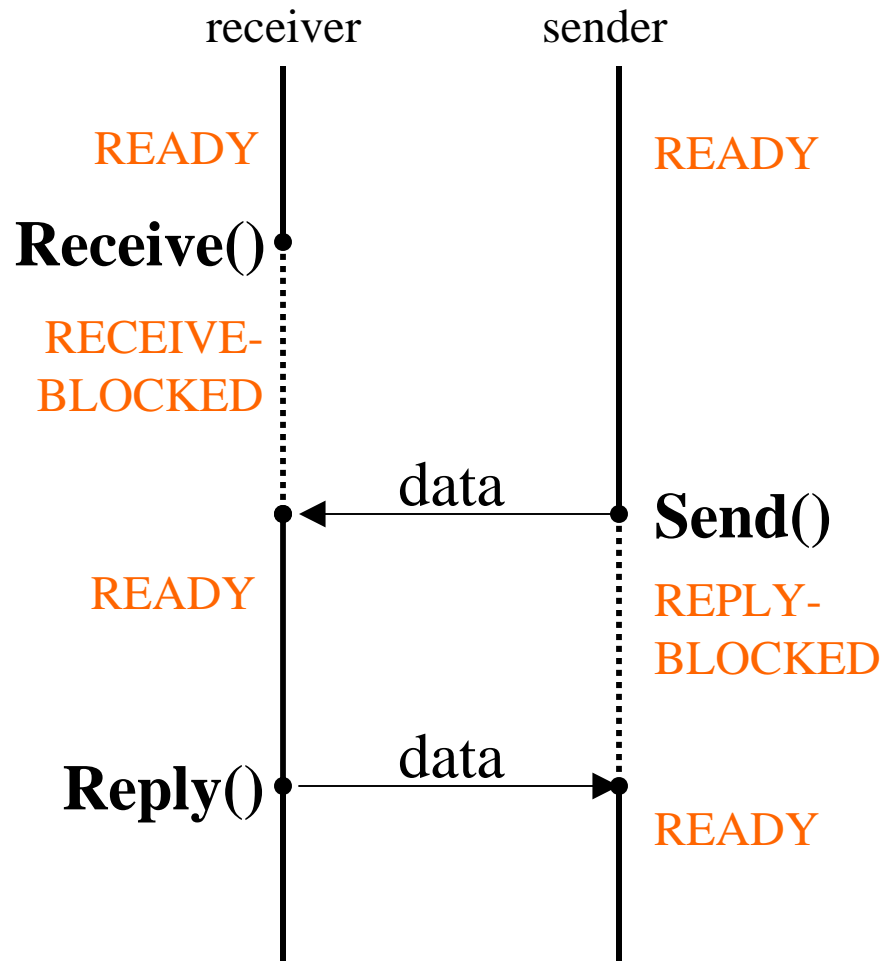


SRR model

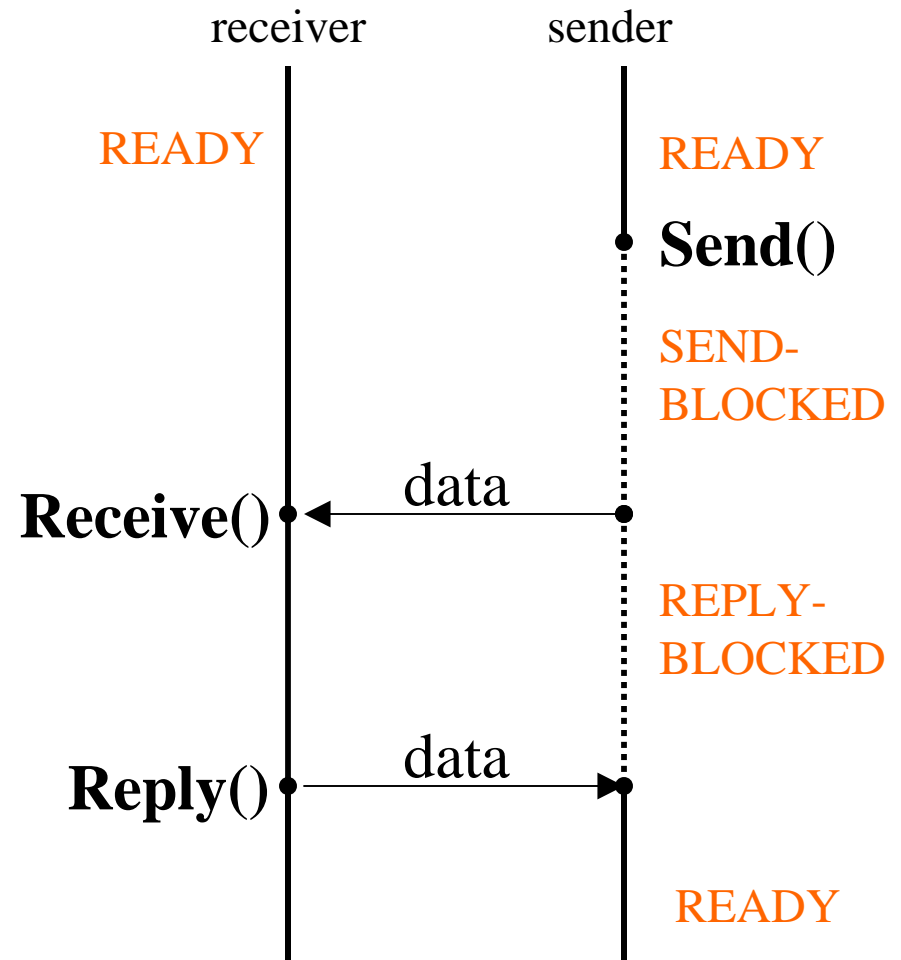
Procesy: stavy



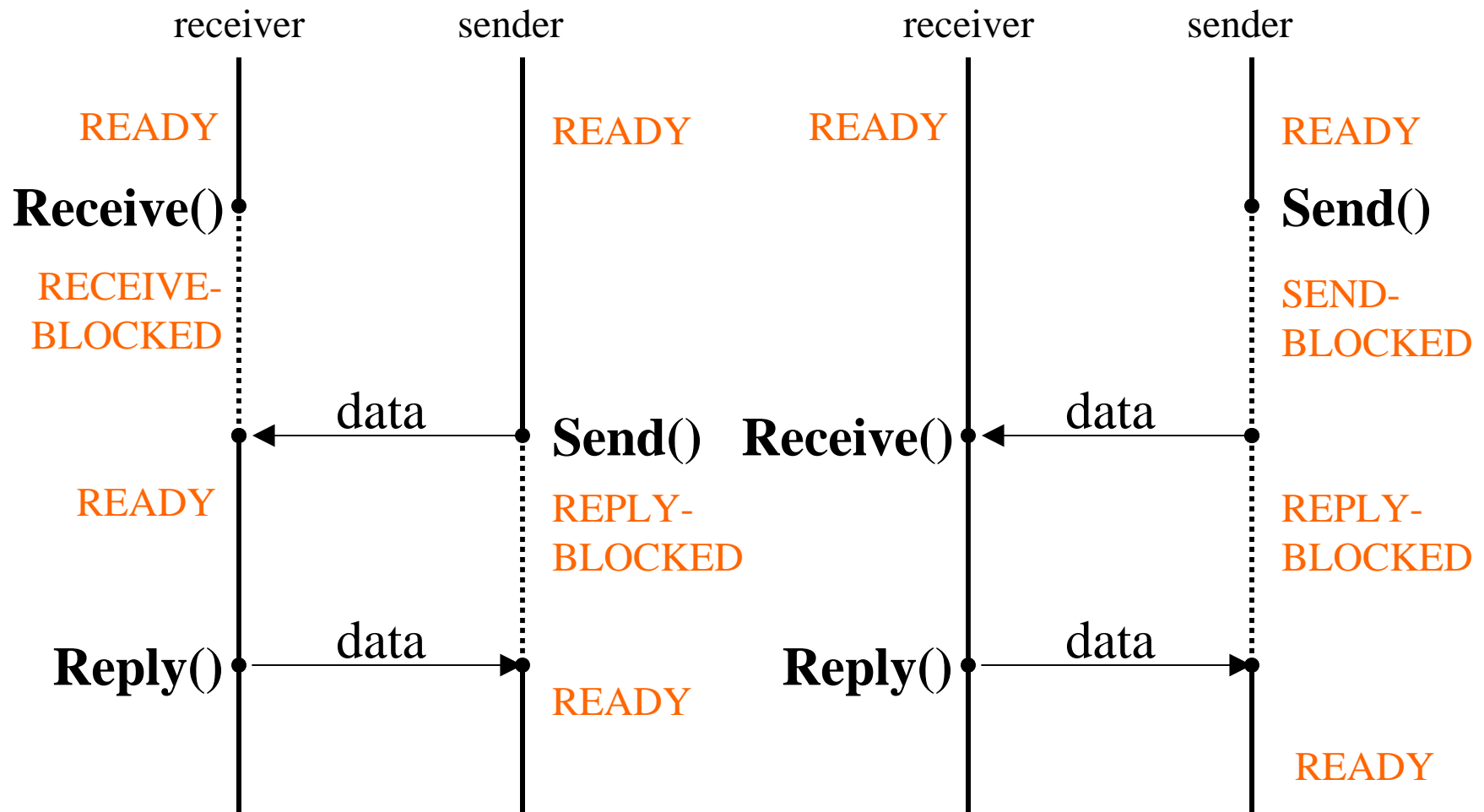
SRR model



SRR model



SRR model

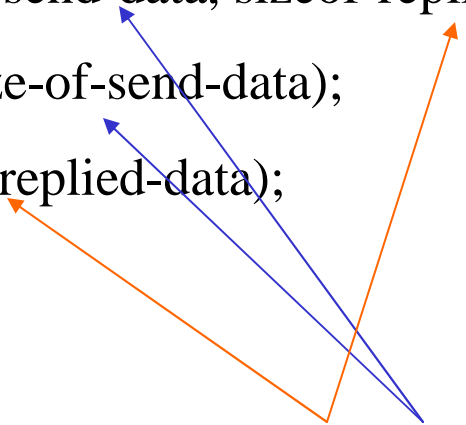


SRR model

Ktorú možnosť si programátor želá ?

Primitívy

```
Send (pid, send-data, replied-data, sizeof-send-data, sizeof-replied-data);  
pid-of-sender = Receive (0, send-data, size-of-send-data);  
Reply(pid-of-sender, replied-data, sizeof-replied-data);
```



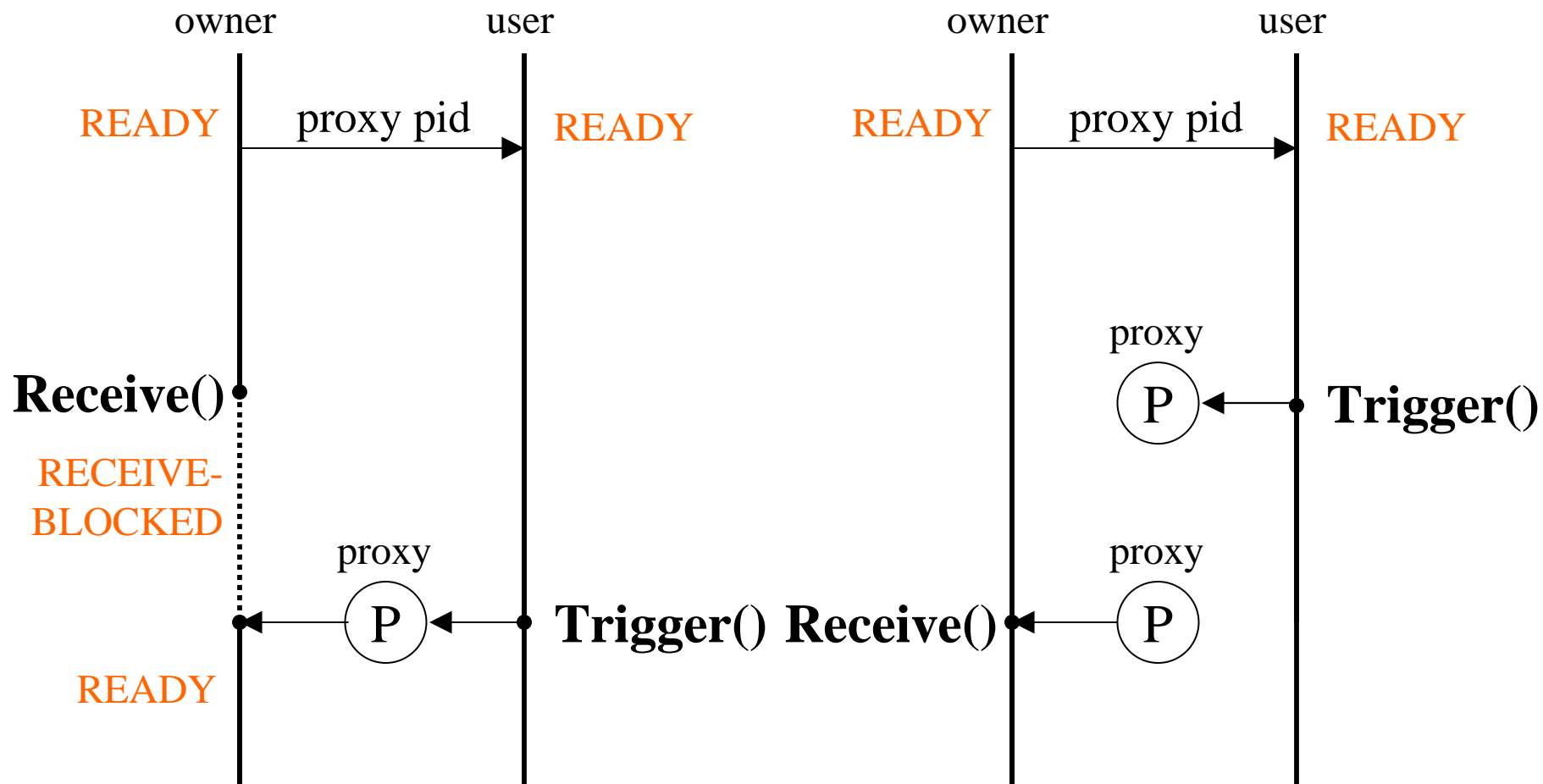
Kto zaručí, že tieto veľkosti si budú zodpovedať ?

SRR model

neblokujúce posielanie správ

- virtuálny proces proxy
- virtuálny proces timer

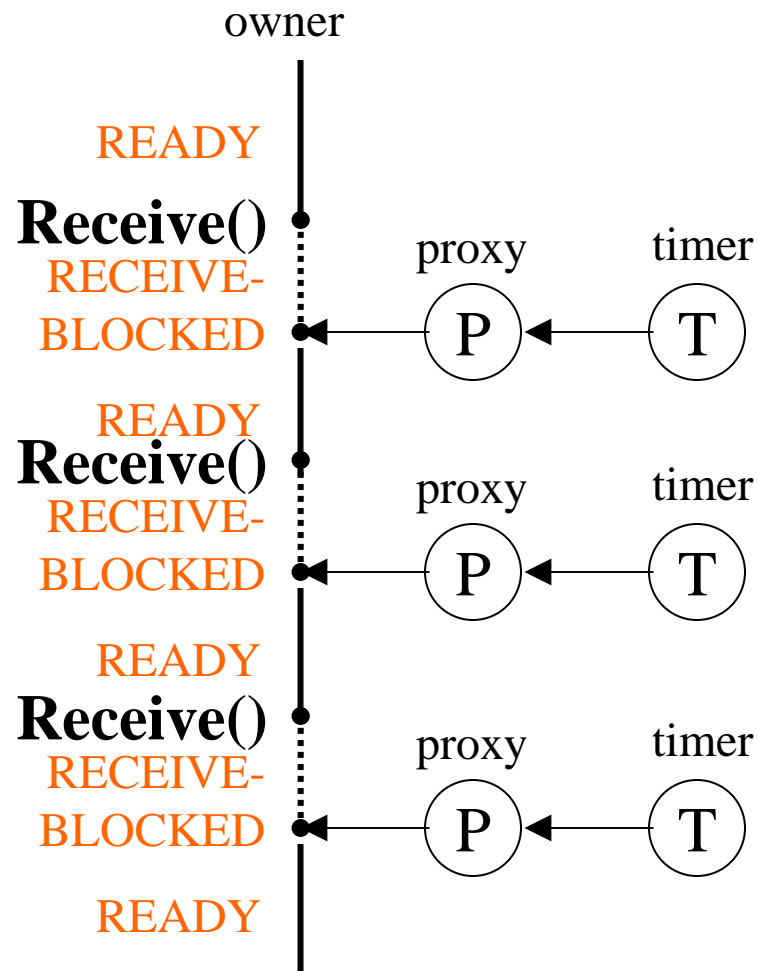
SRR model



pidp = proxy_attach(0,0,0,-1)

Trigger(pidp)

SRR model



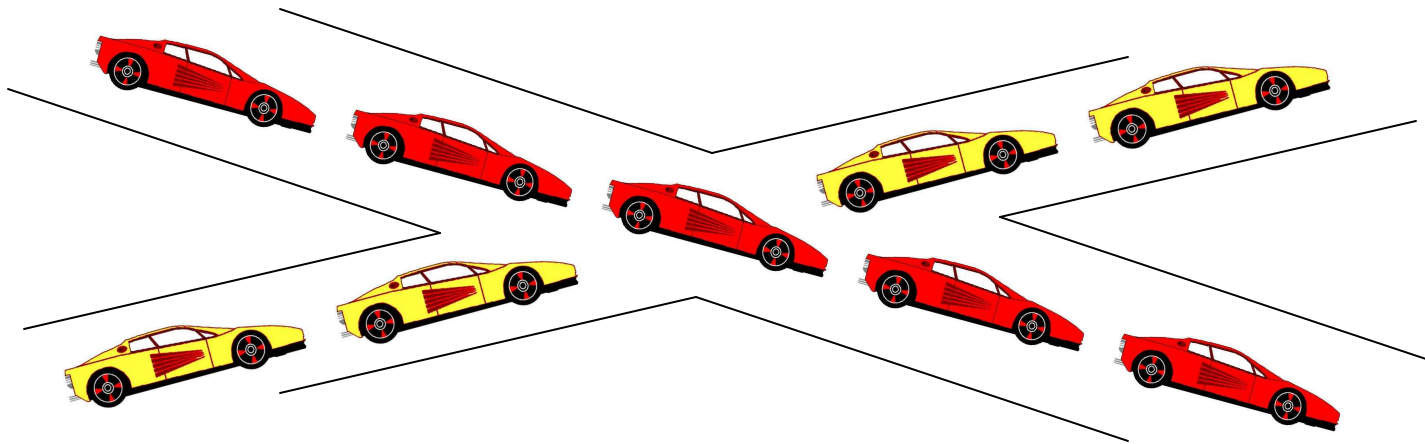
TIMER

`pidt = timer_create(-pidp)`

SRR model `timer_set(pidt, typ, sec0, nsec0, sec, nsec)`

Problémy komunikácie medzi procesmi

- deadlock
- livelock
- negarantovaná odozva



Riešenie

- zaviesť pravidlá, ktoré musí programátor pri návrhu systému dodržiavať, t.j. **architektúru**

Jedným z možných riešení je tzv. pyramidálna client-server architektúra

Client-Server

z hľadiska SRR je

- Server receiverom
- Client senderom

Samozrejme server z hľadiska jedného vzťahu môže byť klientom z hľadiska vzťahu druhého a teda v ňom nájsť ako Receive, tak Send

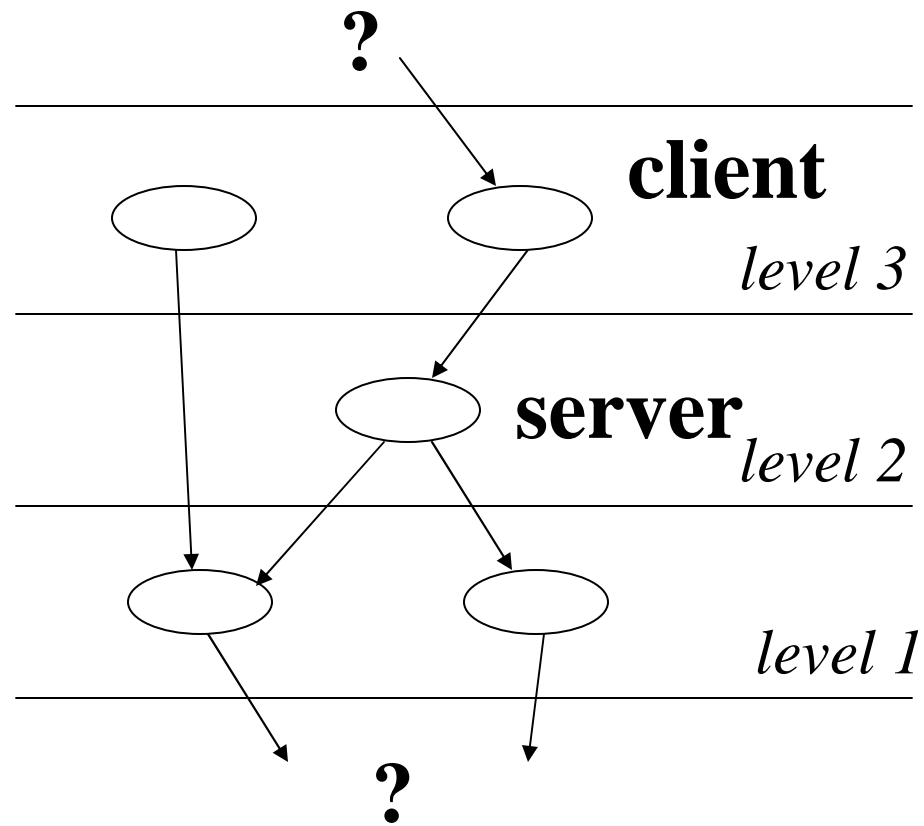
kde ?



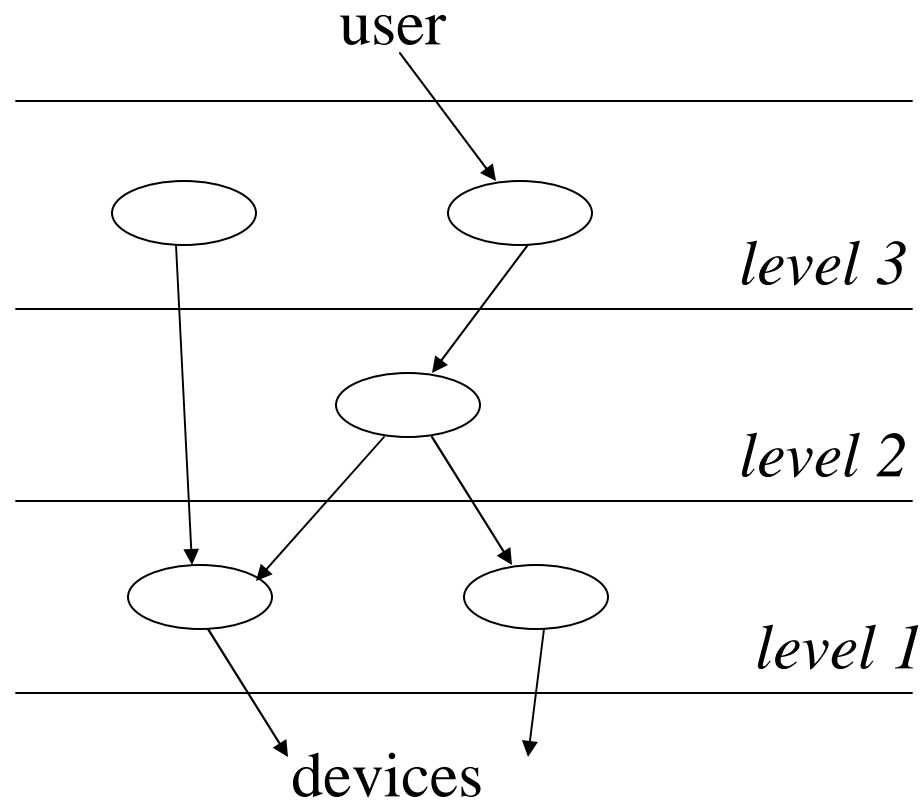
Pyramidálna architektúra Client-Server

1. Systém rozdelíme na úrovne
2. Každý server dáme na určitú úroveň
3. Každý klient musí byť na vyššej úrovni než jeho server

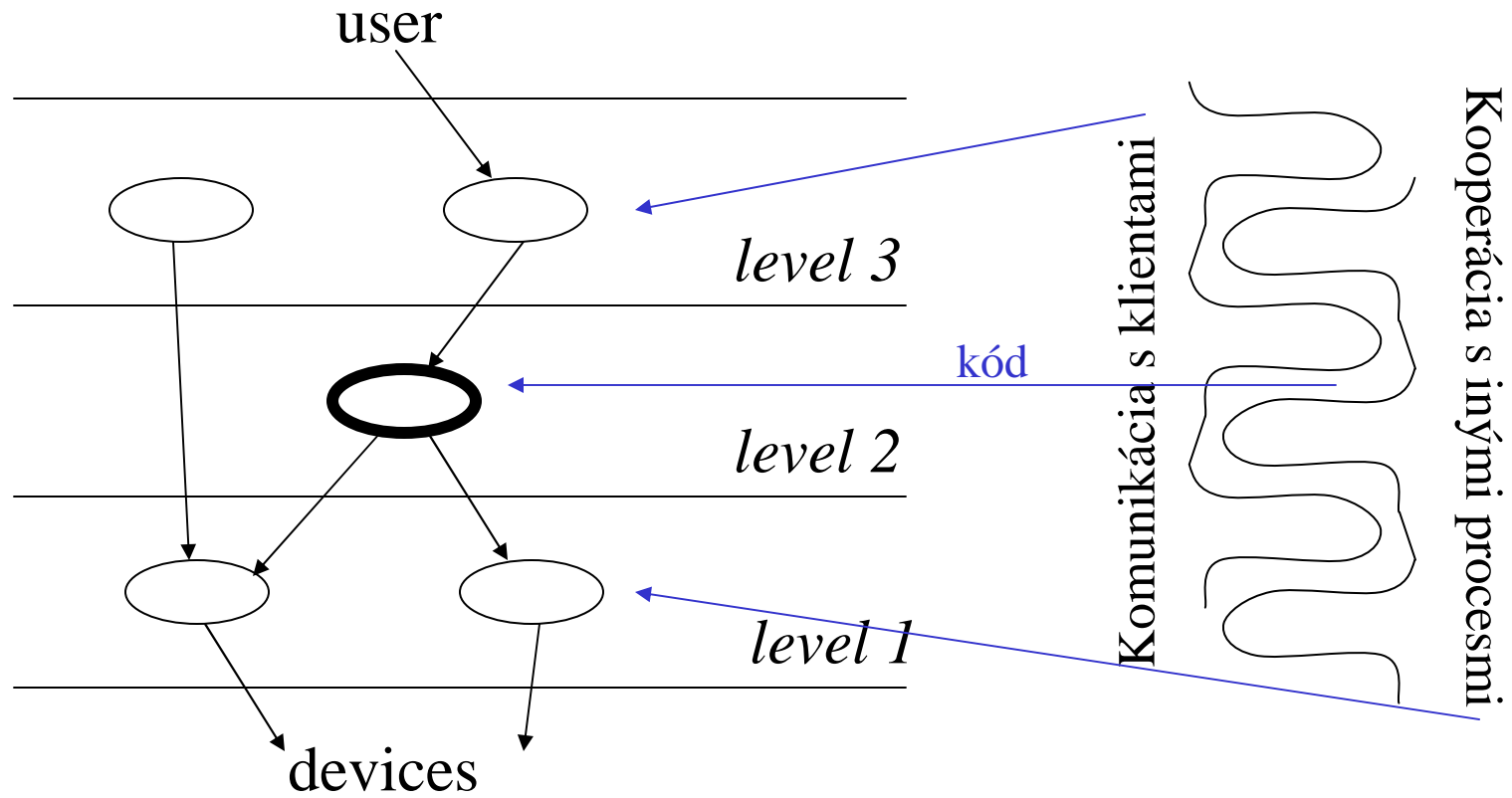
Pyramidálna architektúra Client-Server



Pyramidálna architektúra Client-Server



Štruktúra serveru



Server

```
typedef struct server_msg {
    short header;
    short action;
    union {
        ...
    } data;
};
#define SERVER_HEADER 'SH'
#define SERVER_ACTION1 'A1'
...
#define SERVER_ACTIONx 'Ax'
main ()
{
    struct server_msg msg;
    // inicializacia
    for (;;) {
        pid = Receive(0,&msg,sizeof(msg));
        if (msg.header != SERVER_HEADER)
            continue;
        switch (msg.action) {
            case SERVER_ACTION1:
                // spracuj msg
                break;
            ...
            case SERVER_ACTIONx:
                ...
                break;
        }
        Reply(pid,&msg,sizeof(msg));
    }
}
```

Client - utilita

```
main ()
{
    struct server_msg msg;
    // inicializacia
    pid = name_locate("...");
    msg.header = SERVER_HEADER;
    msg.action = SERVER_ACTIONy;
    // naplni msg.data;
    Send(pid,&msg,&msg,
        sizeof(msg),sizeof(msg));
    // spracuje msg.data
}
```

Client - data collector

```
main ()
{
    // inicializacia
    pids = name_locate("...");
    pidp = proxy_attach();
    pidt = timer_create(-pidp)
    timer_set(pidt,RELATIVE,0,0,1,0);
    for (;;) {
        pid = Receive(0,NULL,0);
        if (pid == pidp) {
            // vytvor msg
            Send(pids,&msg,&msg,
                sizeof(msg),sizeof(msg));
            // spracuj msg
        }
    }
}
```

Dekompozícia servera

Problém negarantovanej odozvy

(Pamäťovo) nestabilné riešenie (ktoré už poznáme):

- zavoláme `fork()` a pustíme separátny proces, ktorý službu vybaví

Stabilné riešenie (nevyhnutné tam, kde máme procesy ale nemáme vlákna):

- Master - slave

Master - slave

Riešenie: master – server, slave – pomocná úloha ktorú si púšťa master

Tejto dokáže zveriť spracovanie služby a seba tým uvoľní pre obsluhu ďalších klientov

Pozor, slave (otrok) nie je klient

Slave

```
void main ()
{
    pid_m = getppid();
    Send(pid_m,...); //co si prajes pane?
    for (;;) {
        task ();
        Send(pid_m,...); //urobene, co si prajes pane?
    }
}
```

v akom stave prežije slave väčšinu času ?

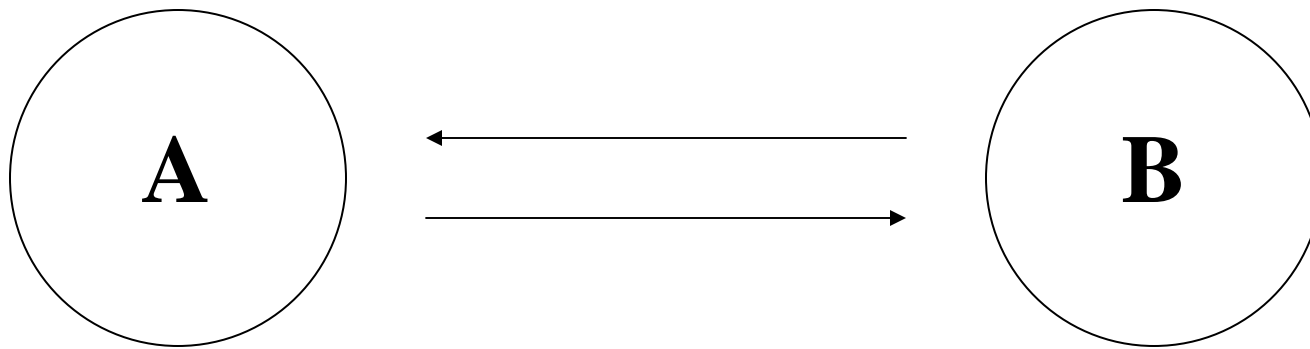
```

main ()
{
    struct server_msg msg;
    struct server_port *port;
    // inicializacia
    ports_init();
    mam = 0; komu = 0; spid = start_slave(); // spawn
    for (;;) {
        pid = Receive(0,&msg,sizeof(msg));
        if (pid == spid) {
            mam = 1;
            if (komu > 0) Reply(komu,...);
        }
        if (msg.header != SERVER_HEADER)
            continue;
        ports_reinit();
        if ((port = port_get(pid)) == -1) {
            port = port_new();
            port_setdefaults(port);
        }
        switch (msg.action) {
            case SERVER_ACTION1:
                // spracuj *port a msg
                Reply(pid,&msg,sizeof(msg));
                break;
            ...
            case SERVER_ACTIONx:
                Reply(spid,...); komu = pid;
                break;
        }
    }
}

```

Master

Servers na rovnakej úrovni

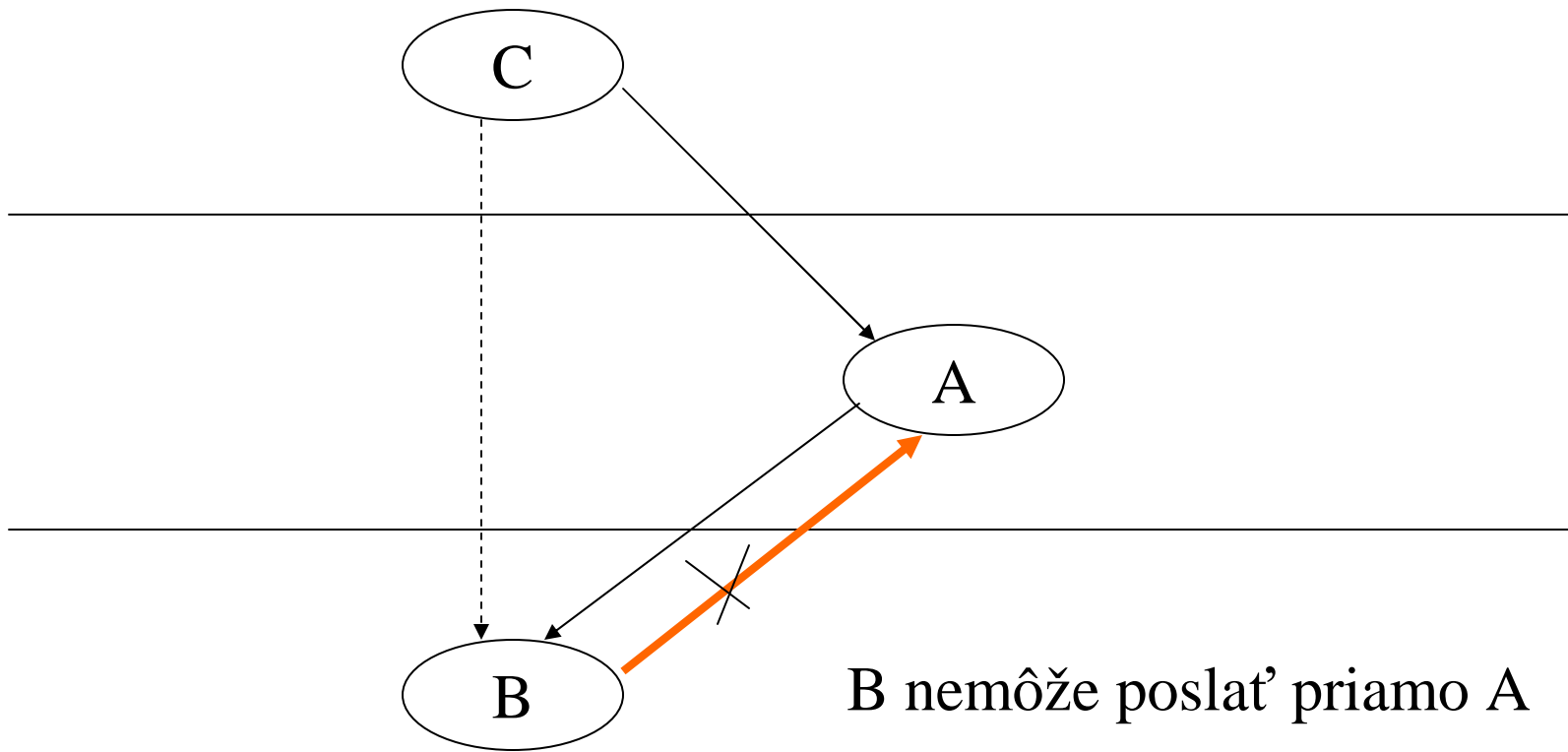


- A je clientom servera B
- B je clientom servera A

Čo s tým ?

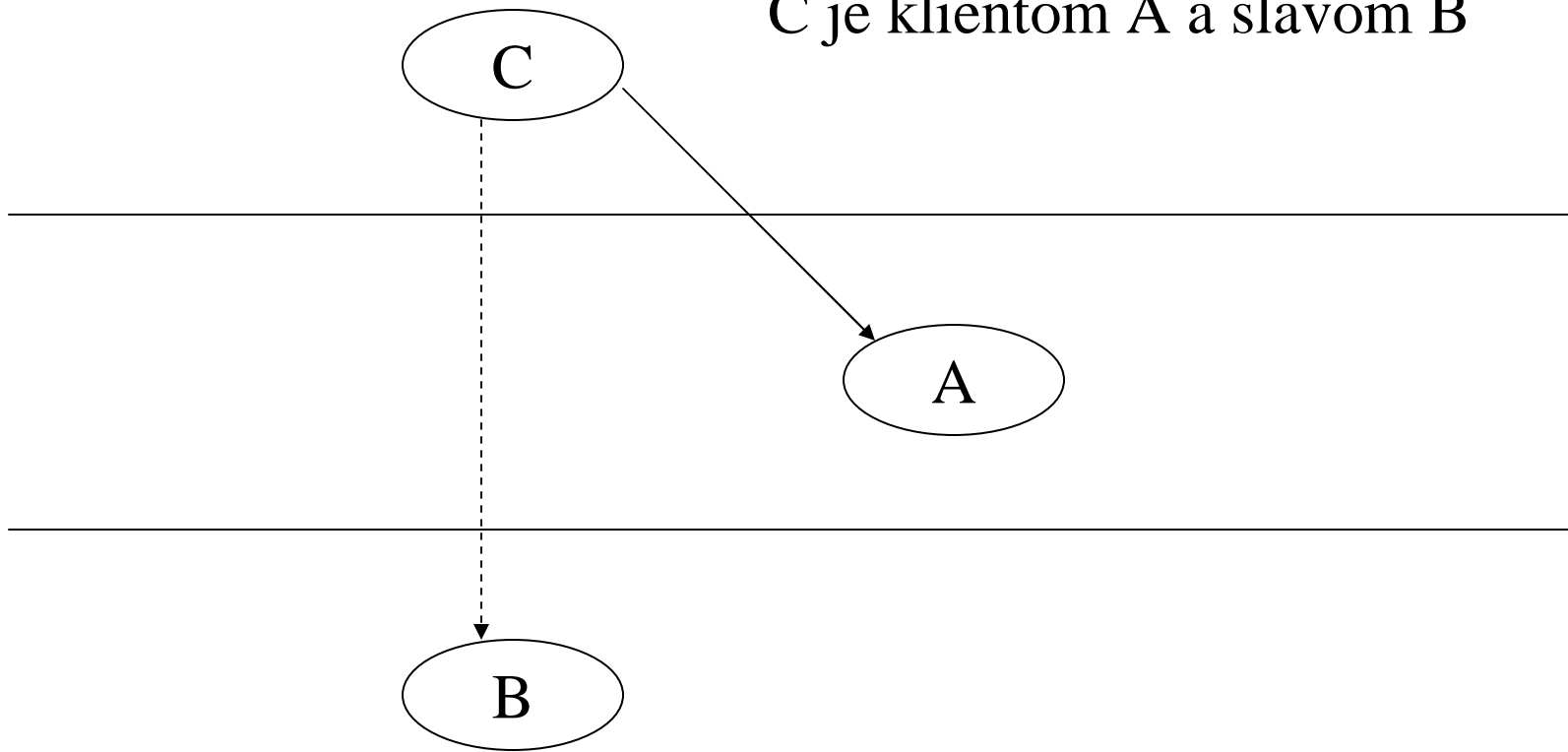
Tento problém už nie je
všeobecný ale týka sa len
pyramídálnej architektúry
client-server

Prievozník



Prievozník

C je klientom A a slavom B



Servery na rovnakej úrovni

Riešenie: prievozník + buffrovanie

```
void main ()
{
    pid_low = getppid();
    pid_high = name_locate("...");
    for (;;) {
        Send(pid_low,...); //co chces poslat?
        Send(pid_high,...); //odkazuju ti toto
    }
}
```

prievozník je slave jedného servera a klient druhého

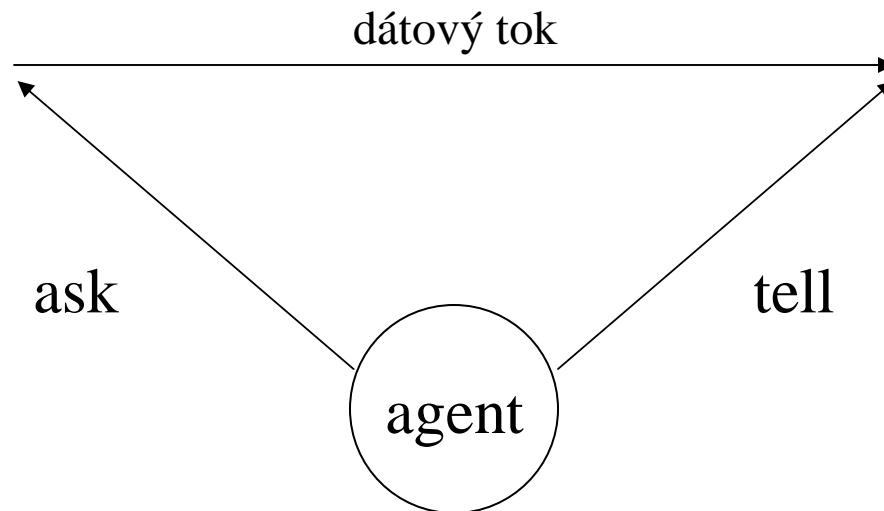
iné riešenia: Proxy + buffrovanie, pipe

Servery na rovnakej úrovni

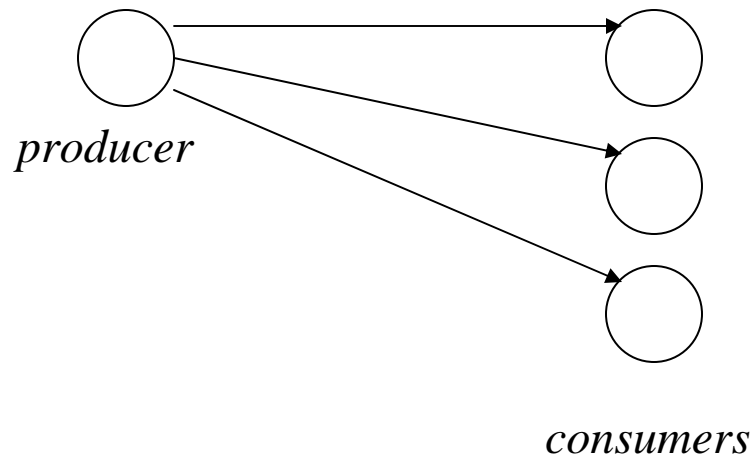
Prievozník (ferryman) je už kódovo veľmi podobný agentom

Nabáda na nastolenie inej architektúry, ktorá by neriešila problém komunikácie medzi servermi na rovnakej úrovni ako špecifický prípad, ale riešenie tohto problému brala ako základný princíp komunikácie medzi dvomi procesmi.

Dátový tok pomocou agentov

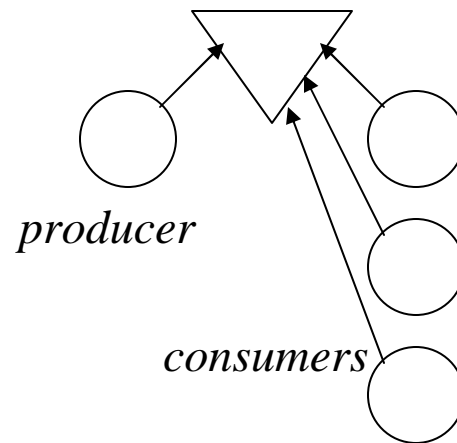


Dátový tok pomocou priamej komunikácie medzi agentami



- nerieši deadlock

Dátový tok pomocou nepriamej komunikácie medzi agentami



- deadlock nemožný

Agent-Space

Komunikačná architektúra ktorá vie riešiť problémy komunikácie medzi procesmi, založená na nepriamej komunikácii medzi agentami

Každý proces je buď

- agent

alebo

- space

Agent

```
void main ()
{
    // initialization
    ...
    pidp = proxy_attach();
    pidt = timer_create(-pidp);
    timer_set(pidt,RELATIVE,0,0,...);
    for (;;) {
        pid = Receive(0,NULL,0);
        if (pid == pidp) {
            // sense
            Send(...);
            Send(...);
            Send(...); ...
            // select
            ...
            // act
            Send(...);
            Send(...);
            Send(...); ...
        }
    }
}
```

budený timerom

Agent

```
void main ()
{
    // initialization
    ...
    pidp = proxy_attach();
    Send(...); // send pidp to space

    for (;;) {
        pid = Receive(0, NULL, 0);
        if (pid == pidp) {
            // sense
            Send(...);
            Send(...);
            Send(...); ...
            // select
            ...
            // act
            Send(...);
            Send(...);
            Send(...); ...
        }
    }
}
```

budený triggrom

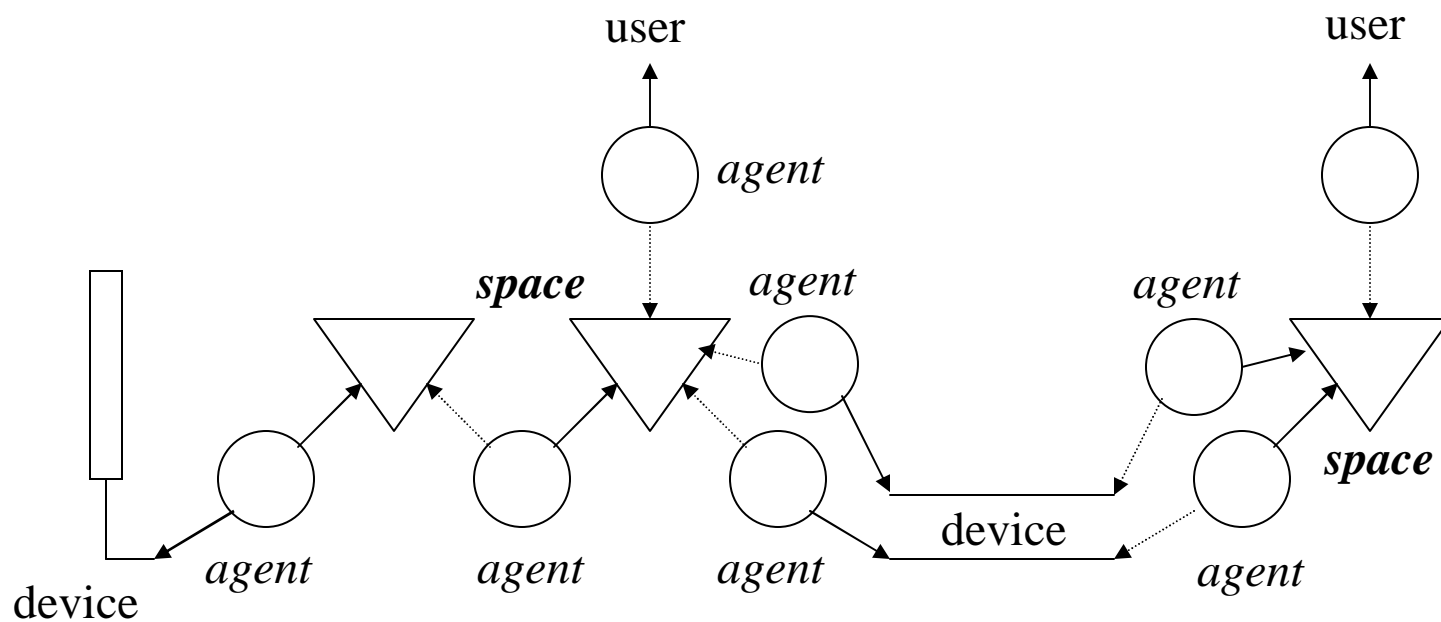
```

main ()
{
    struct server_msg msg;
    struct trigger *trg;
    struct block *data;
    // inicializacia
    for (;;) {
        pid = Receive(0,&msg,sizeof(msg));
        if (msg.header != SERVER_HEADER)
            continue;
        switch (msg.action) {
            case READ:
                // spracuj *port a msg
                break;
            case WRITE:
                ...
                break;
            ...
            case ATTACH_TRIGGER:
                ...
                break;
        }
        Reply(pid,&msg,sizeof(msg));
    }
}

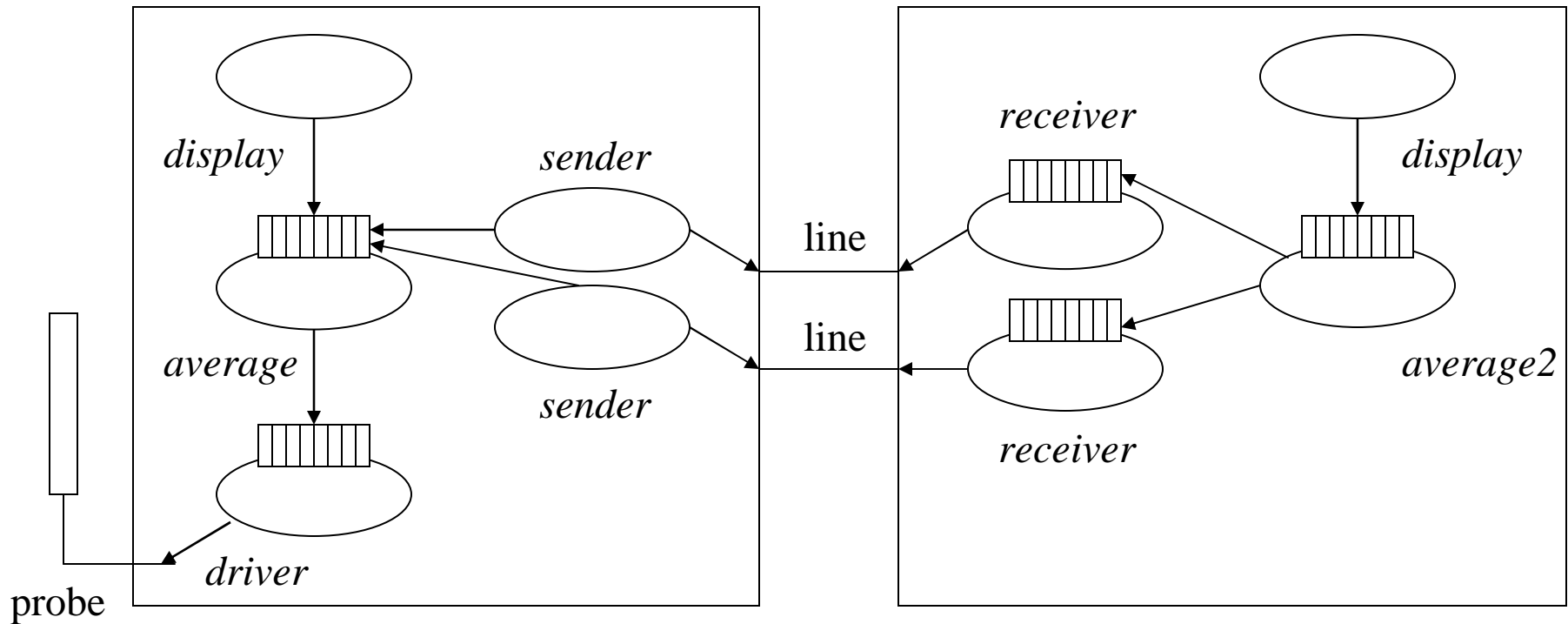
```

Space

Agent-Space



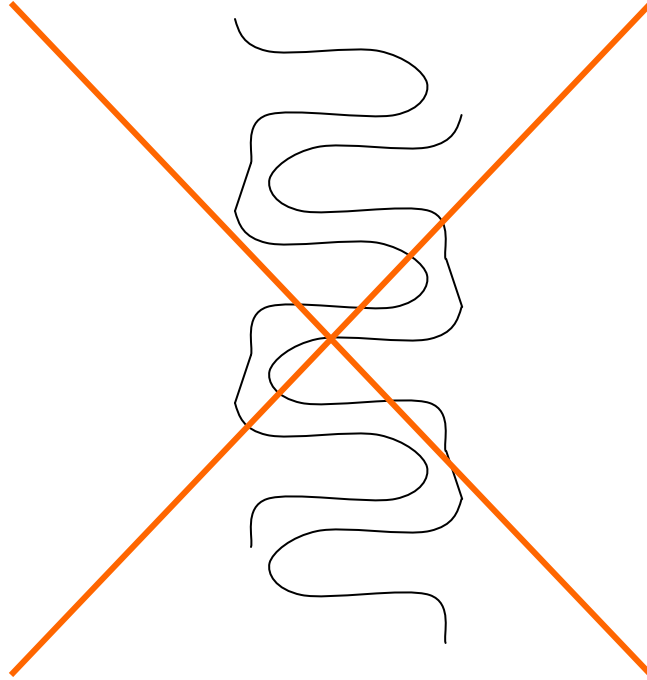
Client-server (porovnanie)



Štruktúra

- Space obsahuje len komunikačný kód
- Agenty obsahujú len aplikačný kód

ako klient využívame
služby serverov



ako server poskytujeme
služby klientom

Deadlock

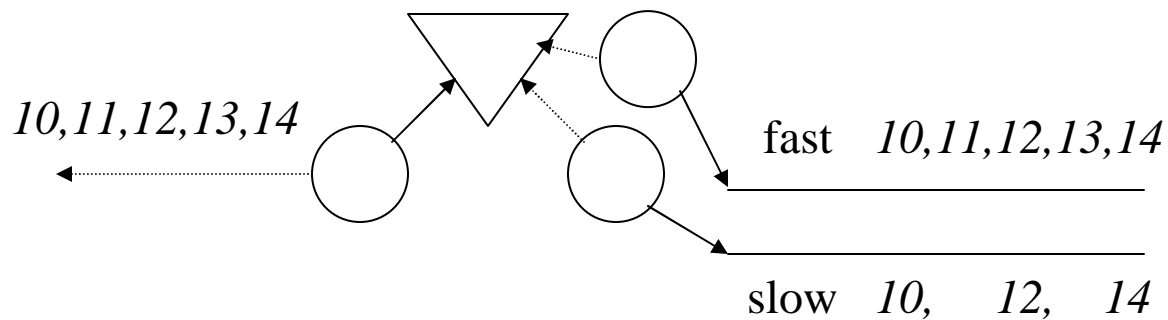
- Space volá Receive a Reply
- Agent volá Send a prípadne Receive na proxy
- iný proces tam nie je
- čiže deadlock nie je možný

Live Lock

- live lock – každý proces dobrovolně opouští procesor

Negarantovaná odozva

- garantovaná odozva – vyplýva z povahy informácie v Space – implicitné vzorkovanie
- rozdiel oproti aktorom



Postavenie MAS v IPC

- MAS je jedným z riešení problémov IPC (deadlock, livelock a negarantovaná odozva)
- Je riešením veľmi dobrým, hoci netradičným