Multiagentové systémy

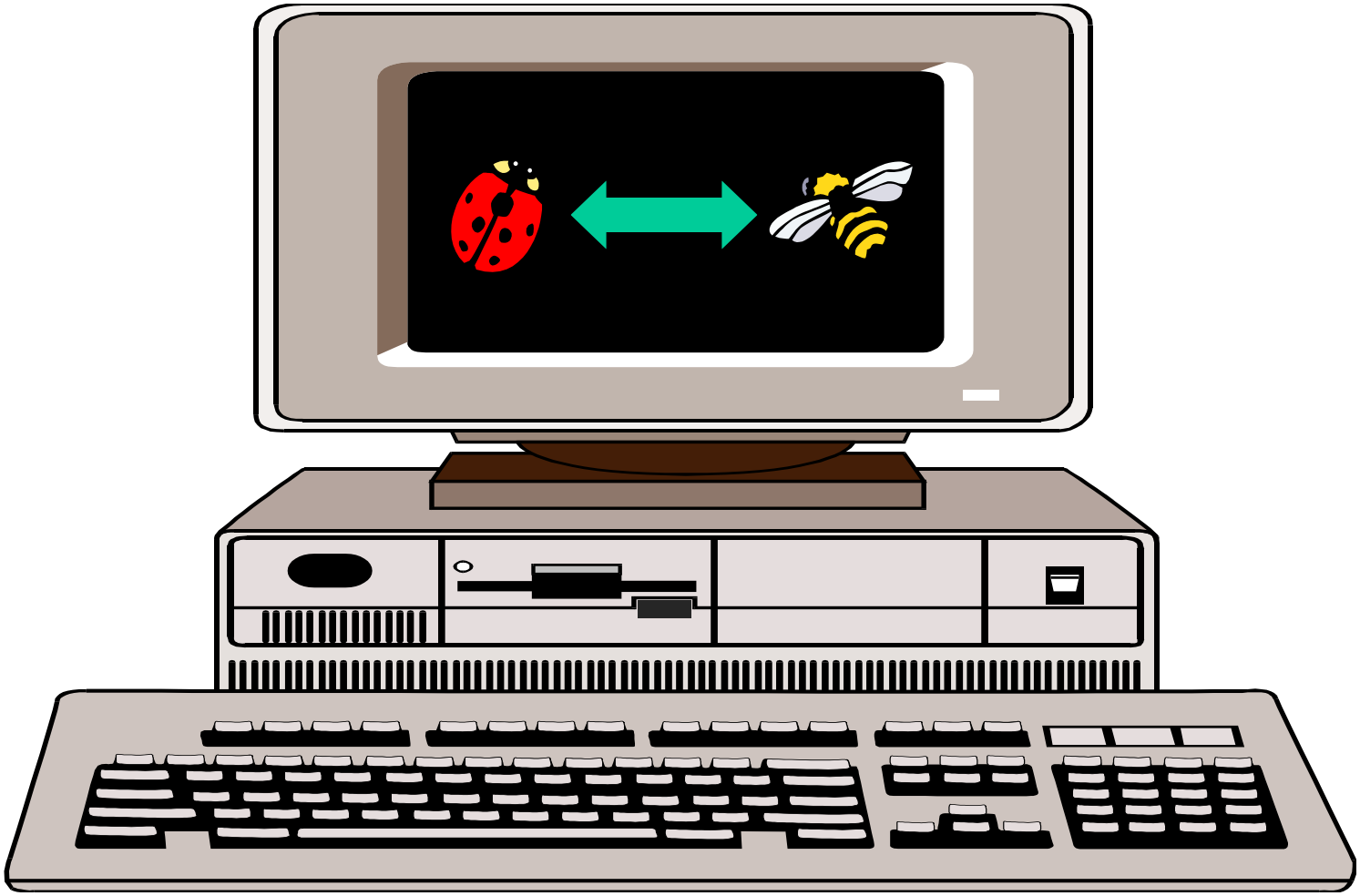
Andrej Lúčny

KAI FMFI UK

lucny@fmph.uniba.sk

http://www.agentspace.org/mas

MAS ako IPC



Inter Process Communication

História komunikácie medzi procesmi:

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

Signály

```
void handler (int sig) {
   // signal received
}

void main ()
{
    signal(SIGUSR1,handler);
    …
}
```

```
    …
    pid = …
    kill(pid,SIGUSR1);
    …
```

```
event = CreateEvent( NULL, TRUE,
         FALSE, TEXT("Event"));
…
SetEvent(event)
```

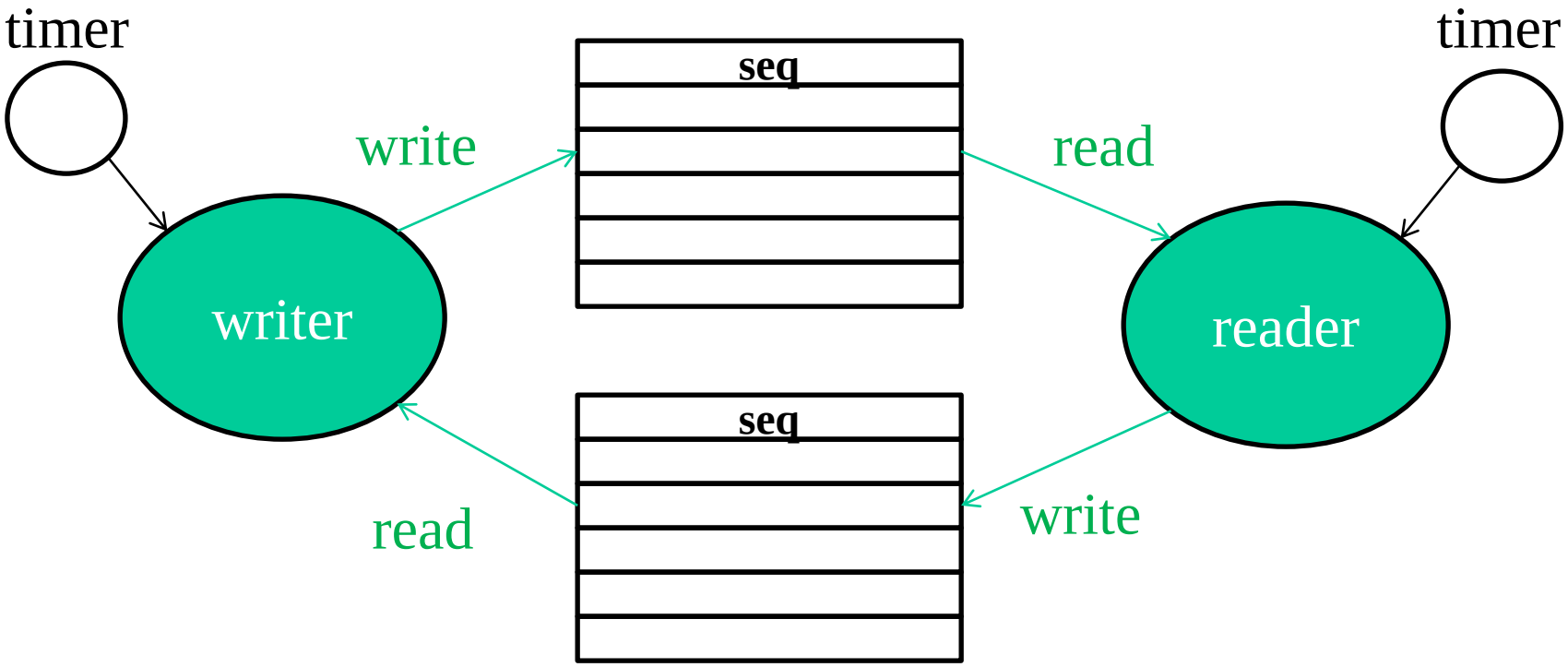
```
event = CreateEvent( NULL, TRUE,
         FALSE, TEXT("Event"));
…
waitForSingleObject (event,
                     timeout);
```

Zdieľaná pamäť

```
ch = new RandomAccessFile(name, "rw");
buf = ch.getChannel().map(FileChannel.MapMode.READ_WRITE, 0, size);
for (...) {
    byte b = buf.get();
    ...
}
buf.rewind();
...
```

```
ch = new RandomAccessFile(name, "rw");
buf = ch.getChannel().map(FileChannel.MapMode.READ_WRITE, 0, size);
for (...) {
    byte b = ...;
    buf.put(b);
    ...
}
buf.rewind();
...
```

Zdieľaná pamäť - Synchronizácia



Synchronizácia medzi procesmi

- Platformovo závislá, z Java voláme cez JNI

- Proces čakajúci na udalosť:

```
HANDLE ghWriteEvent;

ghWriteEvent = CreateEvent(
    NULL,               // default security attributes
    TRUE,               // manual-reset event
    FALSE,              // initial state is nonsignaled
    TEXT(_name)         // object name
);

DWORD dwWaitResult;
dwWaitResult = WaitForSingleObject(
    ghWriteEvent,       // event handle
    INFINITE
);                      // indefinite wait

ResetEvent(ghWriteEvent);
```

Synchronizácia medzi procesmi

- Platformovo závislá, z Java voláme cez JNI

- Proces posielajúci na udalosť:

```
HANDLE ghWriteEvent;

ghWriteEvent = CreateEvent(
    NULL,               // default security attributes
    TRUE,               // manual-reset event
    FALSE,              // initial state is nonsignaled
    TEXT(_name)         // object name
);

SetEvent(ghWriteEvent);
```

Message passing

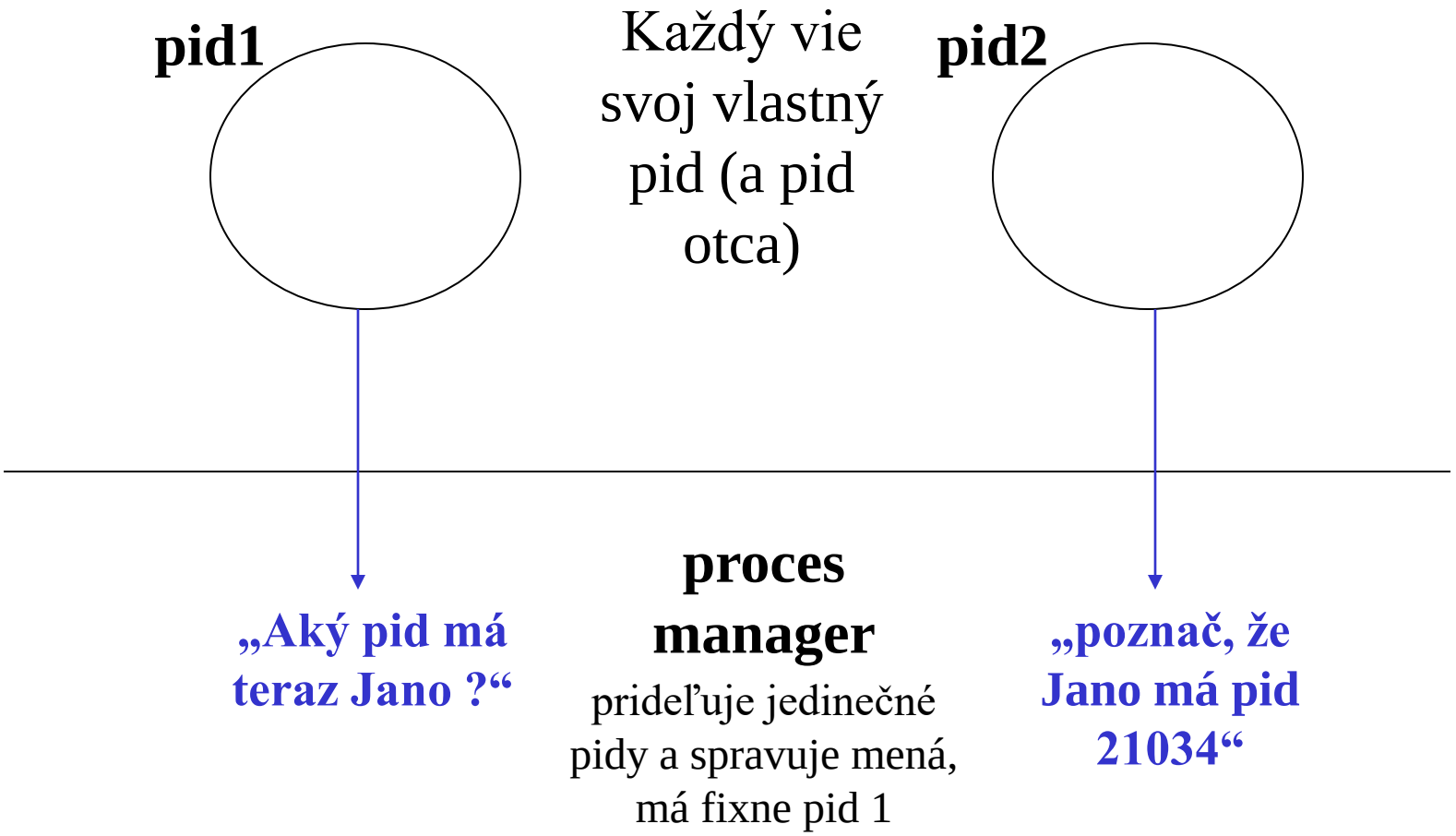
SRR

Procesy: pid

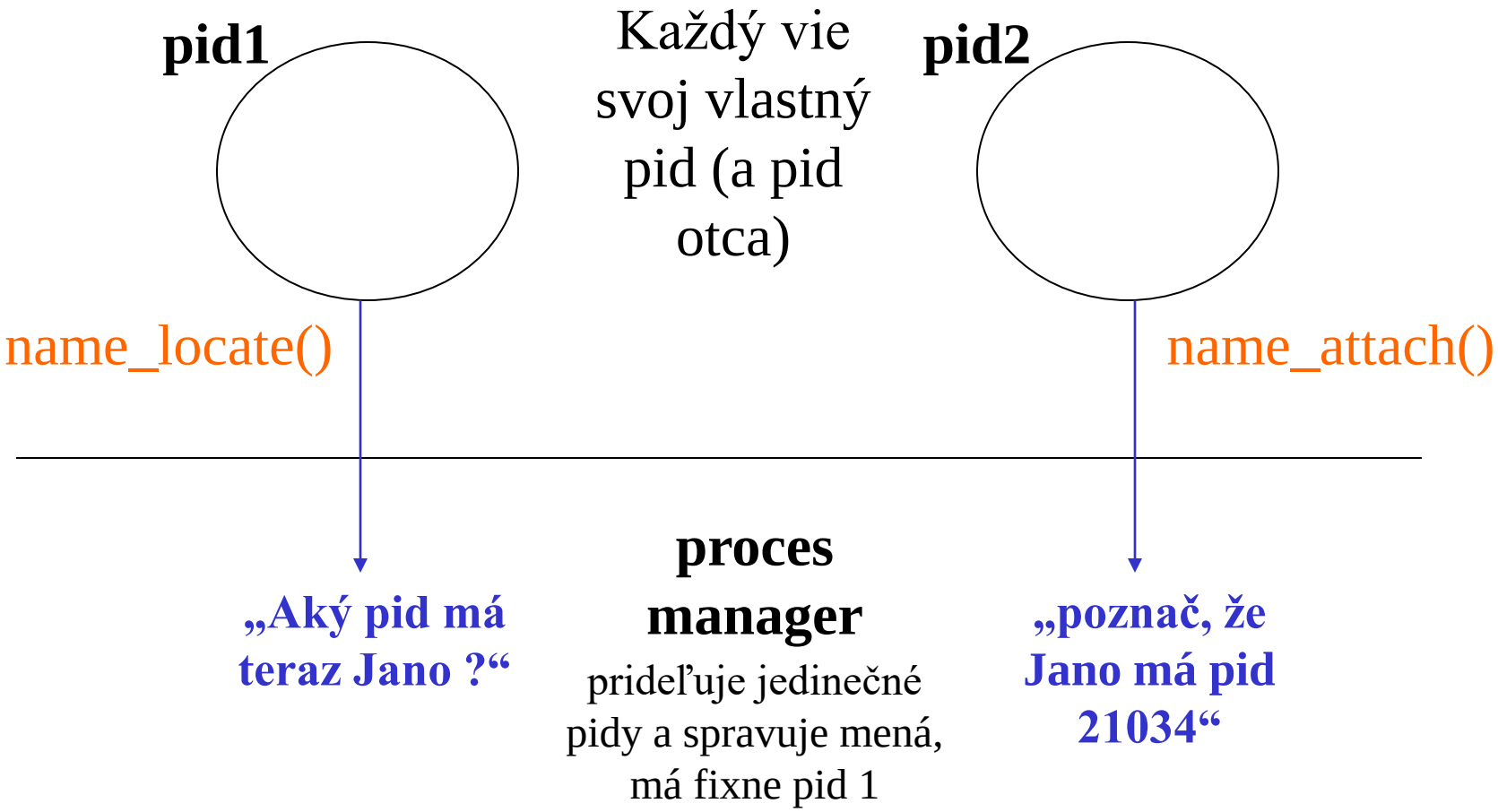
OS

jeden Proces môže komunikovať s druhým pokiaľ vie jeho pid

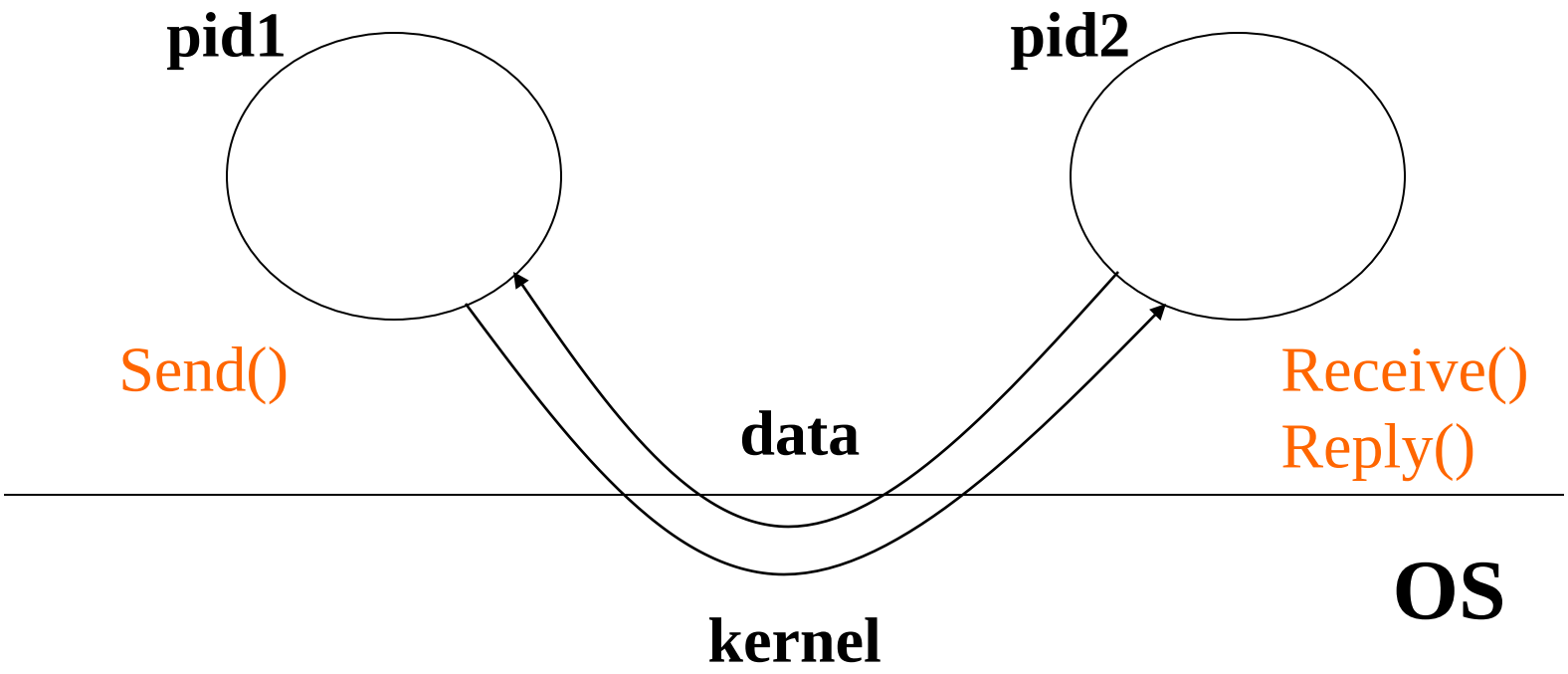
Procesy: names



Procesy: names

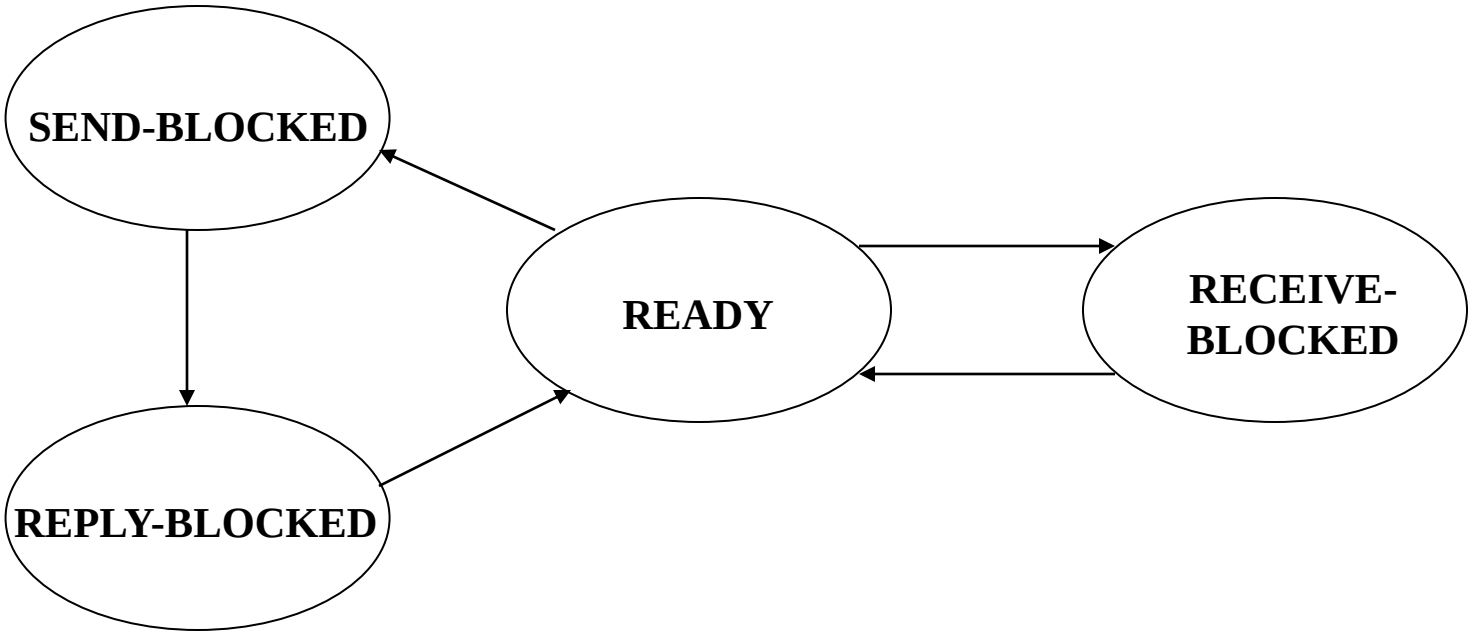


Procesy: komunikácia

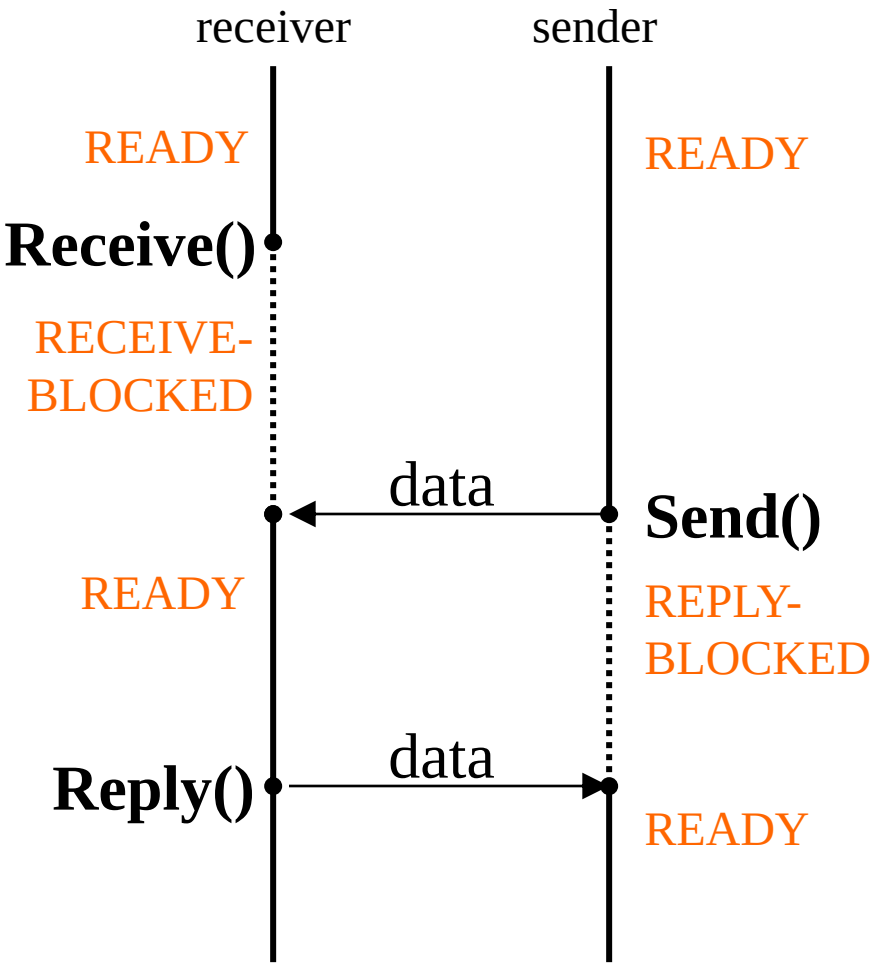


SRR model

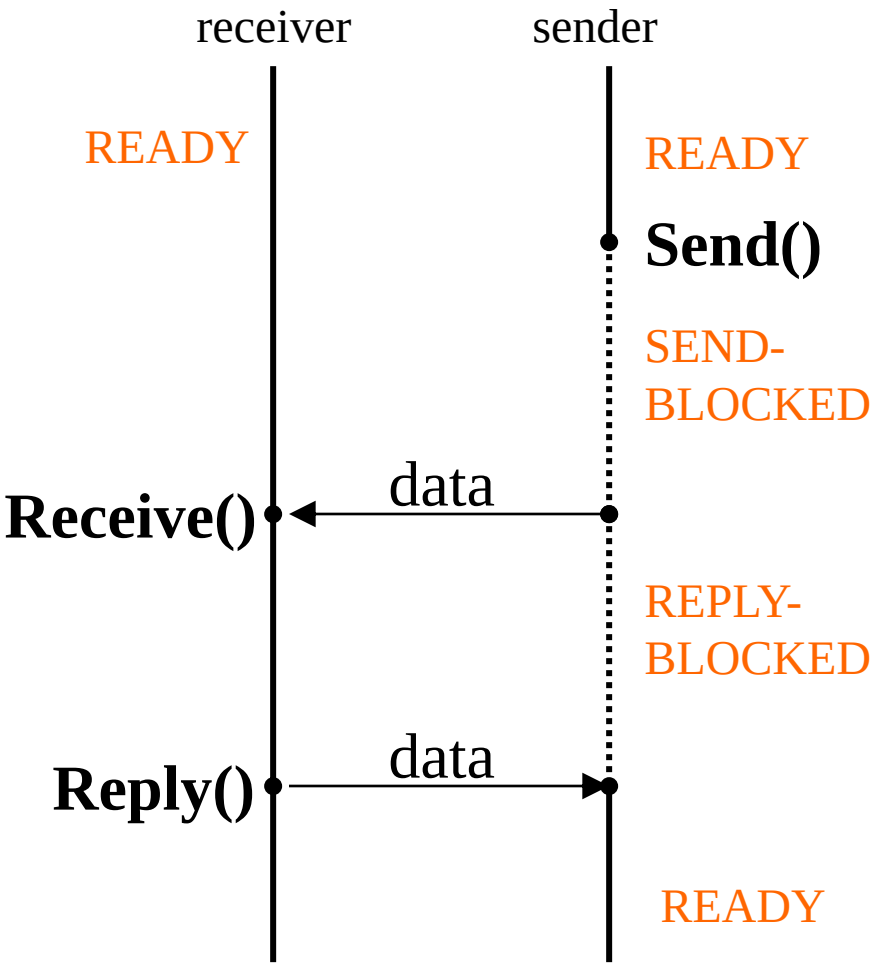
Procesy: stavy



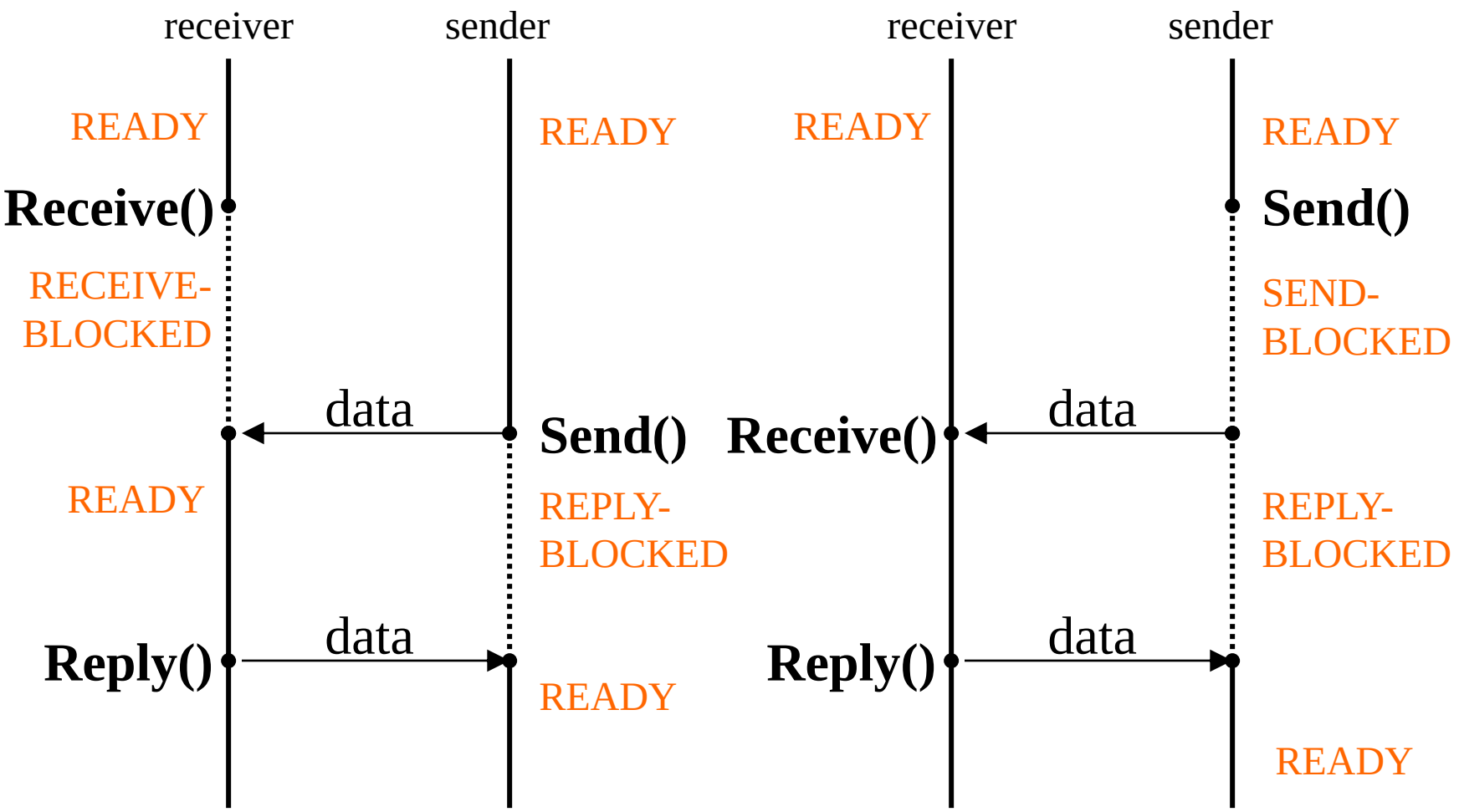
SRR model


receiver
sender
READY
READY
Receive()
RECEIVE-
BLOCKED
data
Send()
READY
REPLY-
BLOCKED
data
Reply()
READY

SRR model


receiver
sender
READY
READY
Send()
SEND-
BLOCKED
data
Receive()
REPLY-
BLOCKED
data
Reply()
READY

SRR model



SRR model

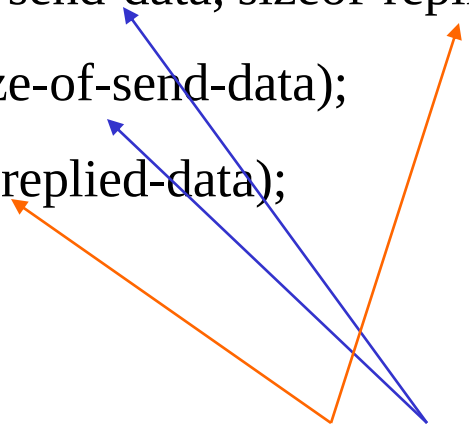
Ktorú možnosť si programátor želá ?

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);



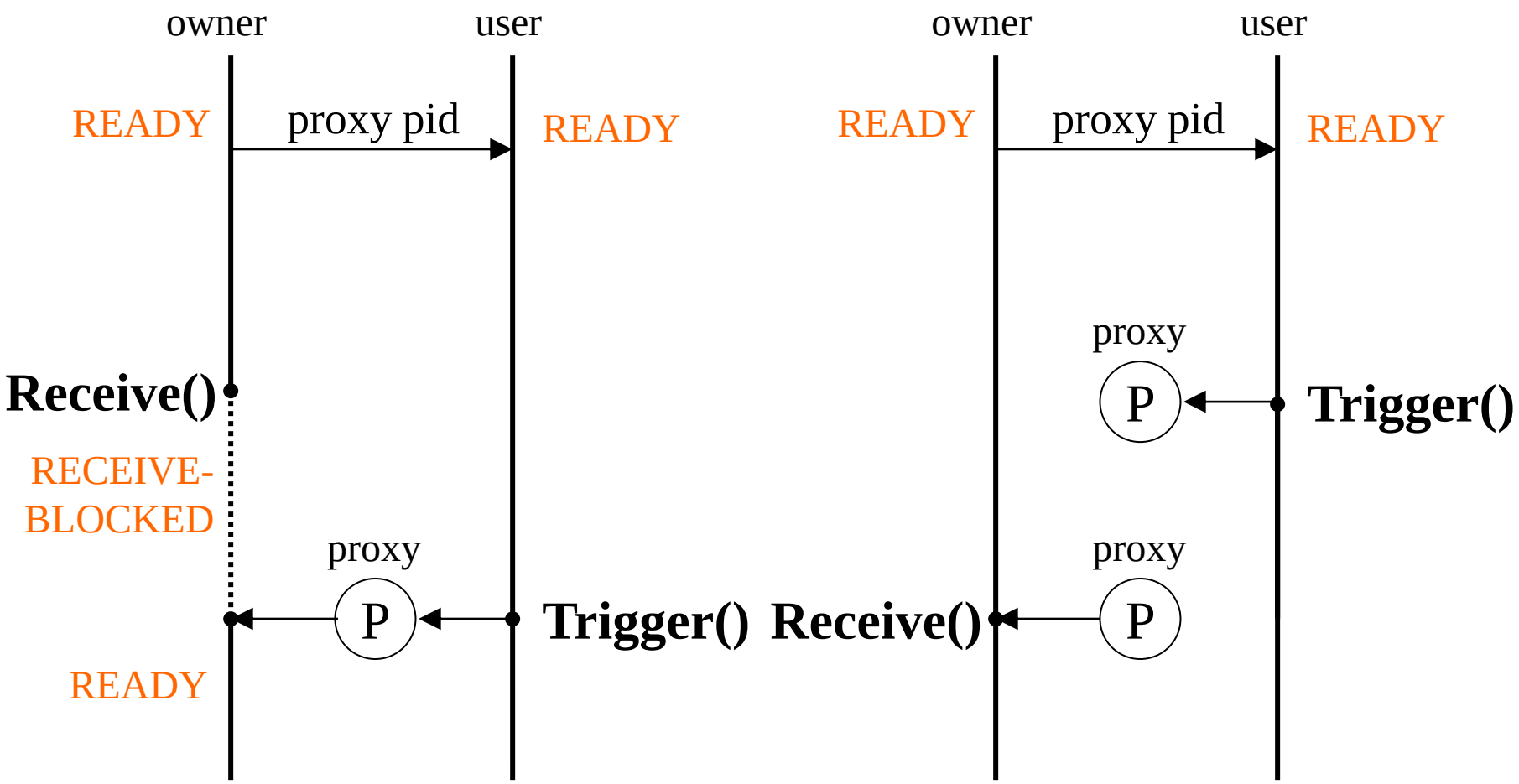
Kto zaručí, že tieto veľkosti si budú zodpovedať ?

SRR model

neblokujúce posielanie správ

- virtuálny proces proxy
- virtuálny proces timer

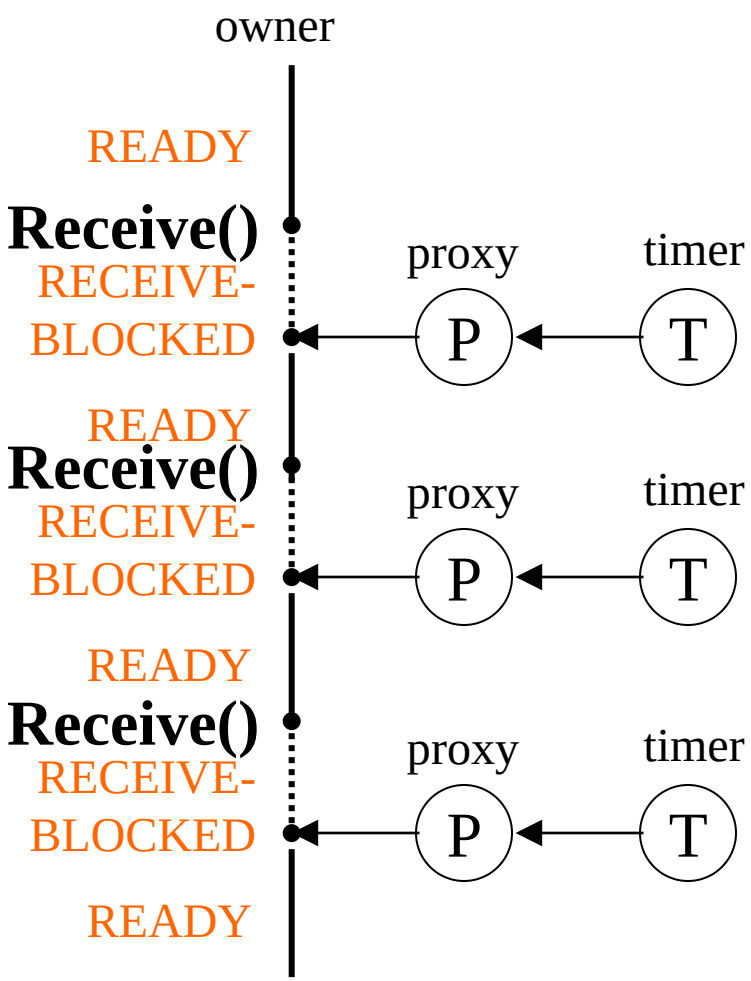
SRR model



SRR model

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

Trigger(pidp)



TIMER

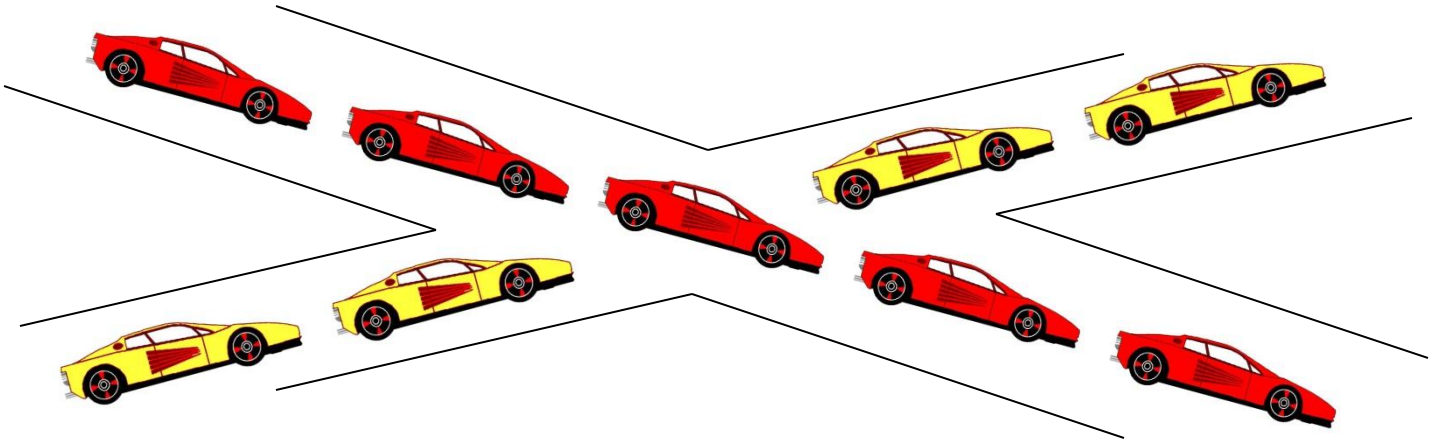
pidt = timer_create(-pidp)

timer_set(pidt, typ, sec0, nsec0, sec, nsec)

SRR model

Problémy komunikácie medzi procesmi

- deadlock
- livelock
- negarantovaná odozva



Riešenie

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

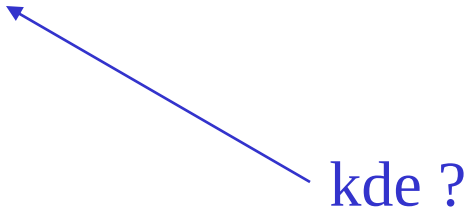
Jedným z možných riešení je tzv. pyramidálna client-server architektúra

Client-Server

z hľadiska SRR je

- Server receiverom
- Client senderom

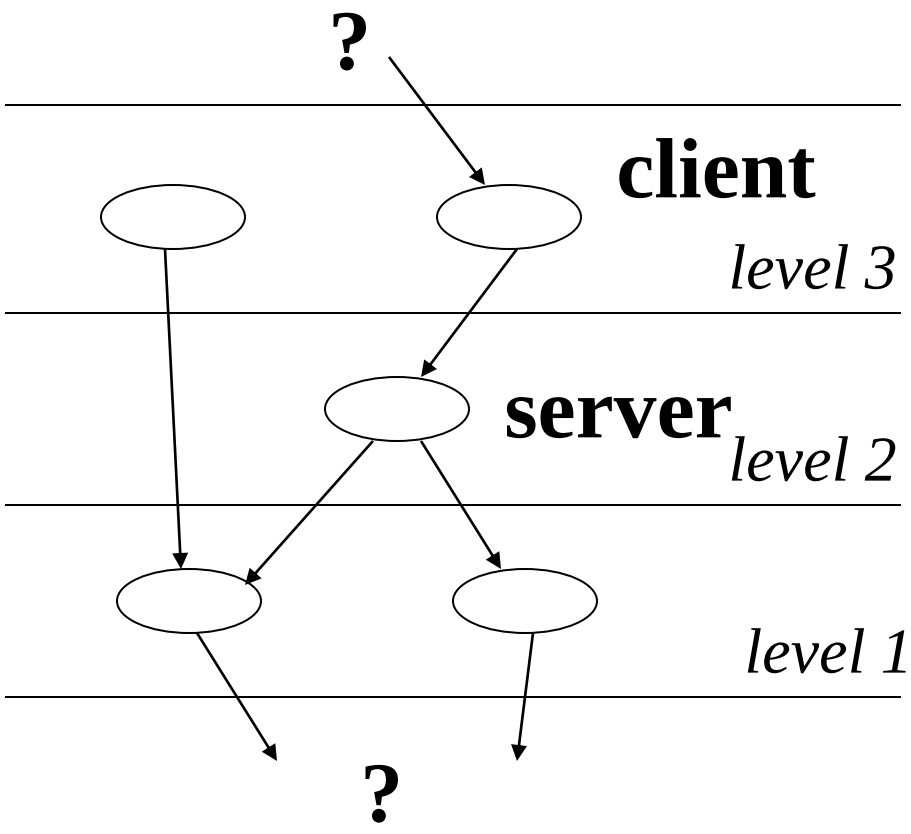
Samozrejme server z hľadiska jedného vzťahu može byť klientom z hľadiska vzťahu druhého a teda v ňom nájsť ako Receive, tak Send



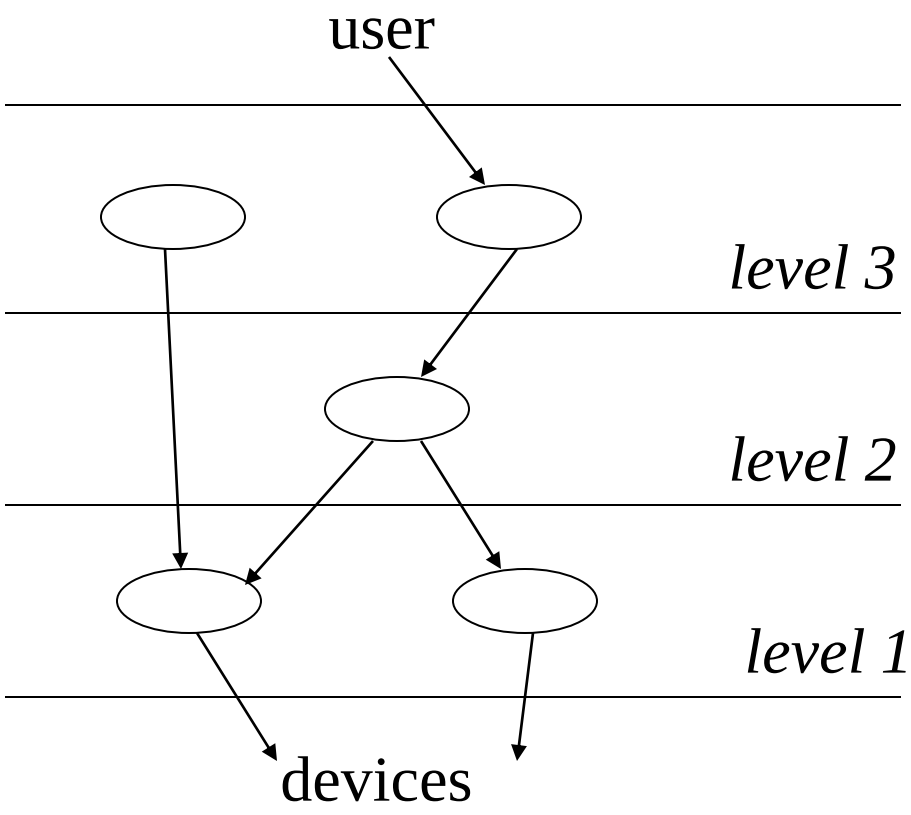
Pyramidálna architektúra Client-Server

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

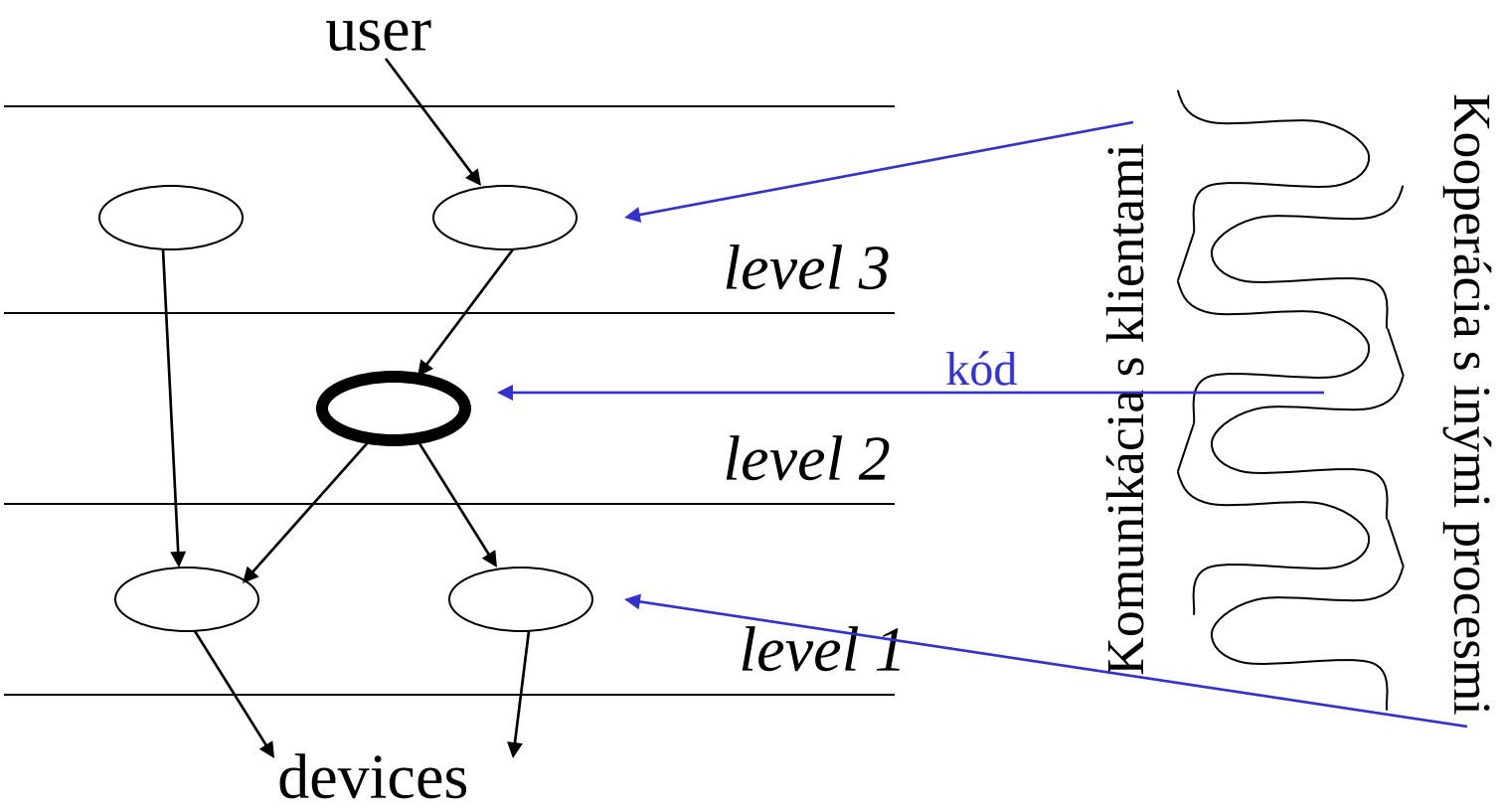
Pyramidálna architektúra Client-Server



Pyramidálna architektúra Client-Server



Štruktúra serveru



Server

```
typedef struct server_msg {
   short header;
   short action;
   union {
      ...
   } data;
};
#define SERVER_HAEDER 'SH'
#define SERVER_ACTION1 'A1'
...
#define SERVER_ACTIONx 'Ax'
void main ()
{
   struct server_msg msg;
   // inicializacia
   if (name_attach("...") == -1) return;
   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

```
void 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

```
void 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 púšta master

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

Pozor, slave (otrok) nie je klient

Slave

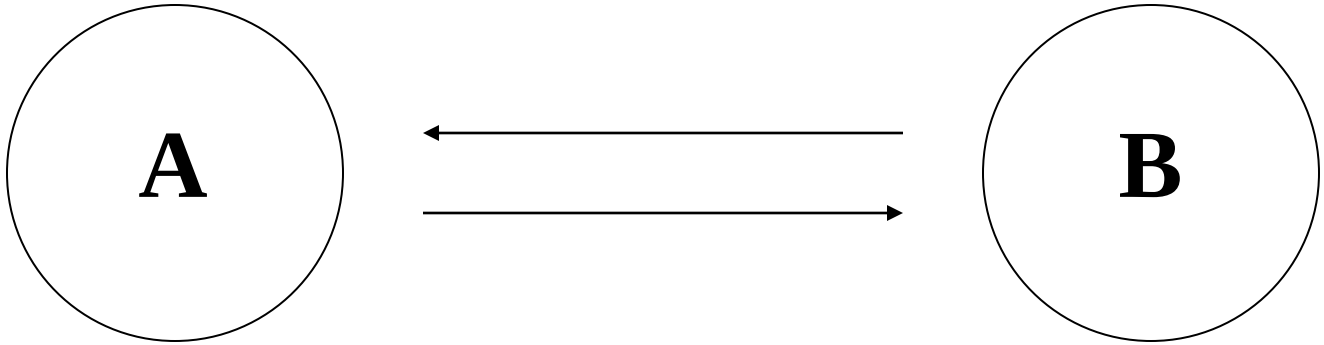
```
void main ()
{
    pid_m = getppid();
       Send(pid_m,...); //co si prajes pane?
    for (;;) {
       task ();
        Send(pid_m,...); //urobene, co si prajes pane?
    }
}
```

v akom stave prežije slave väčšinu času ?

Master

```
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;
        }
    }
}
```

Servery na rovnakej úrovni

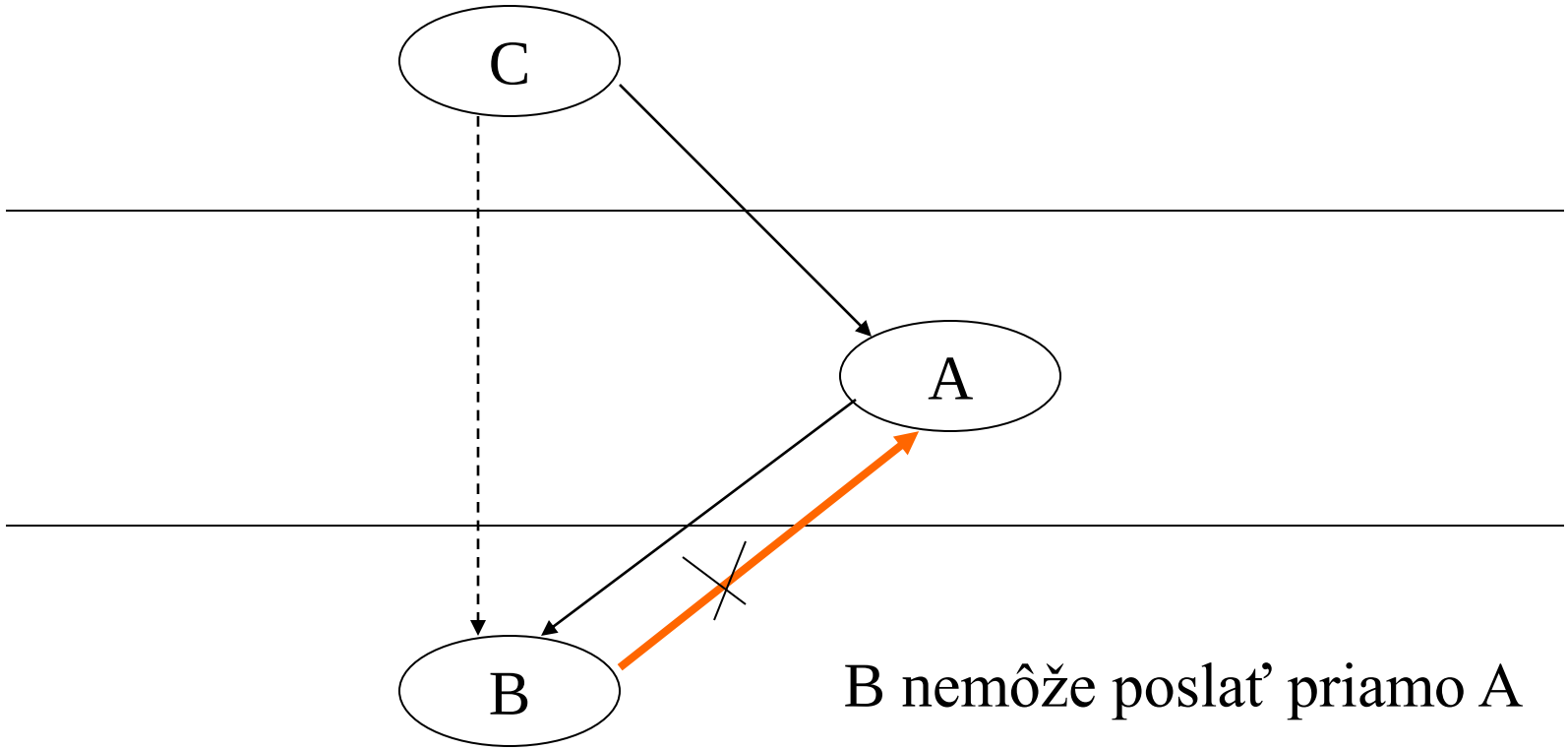


- A je clientom servera B
- B je clientom servera A

Čo s tým ?

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

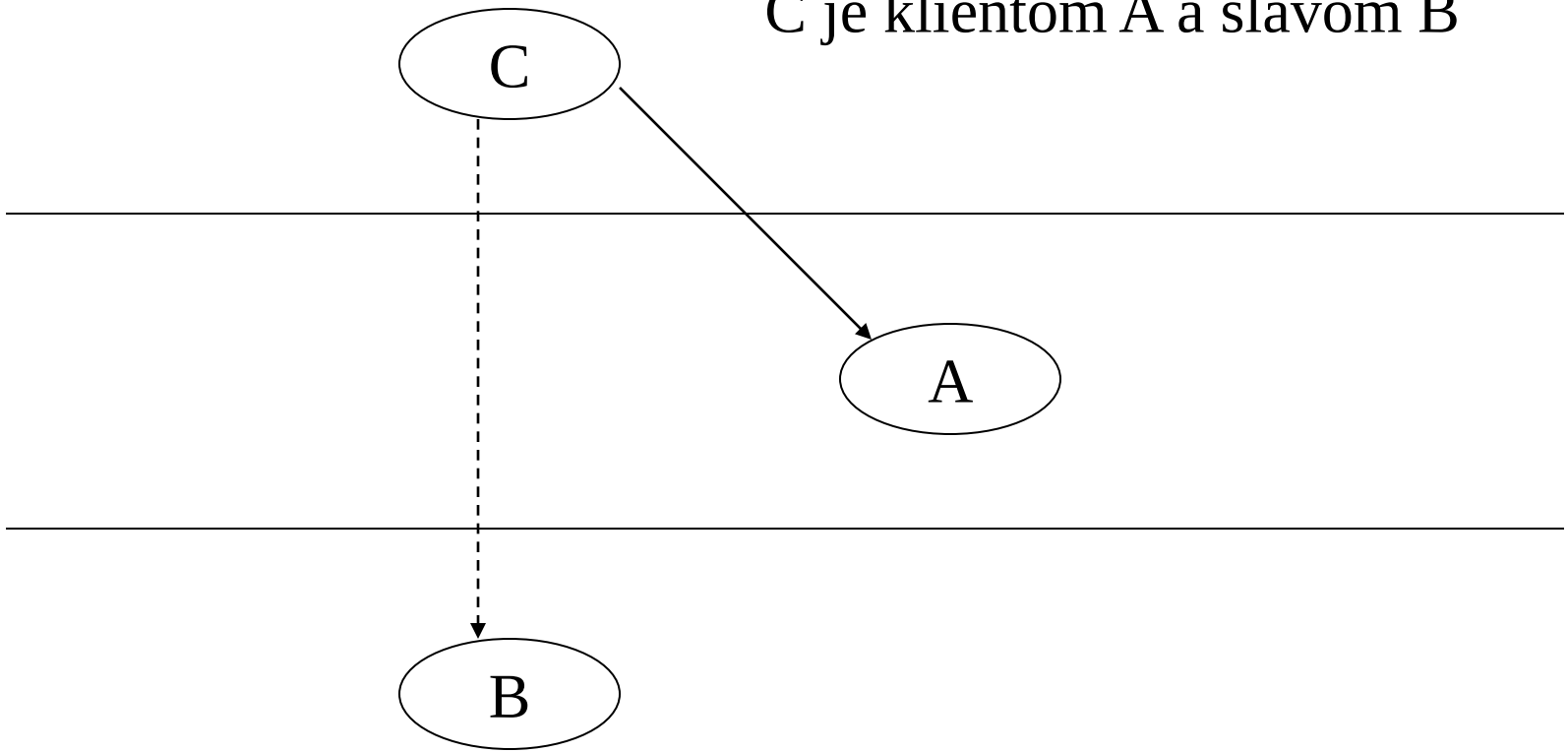
Prievozník



B nemôže poslať priamo A

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

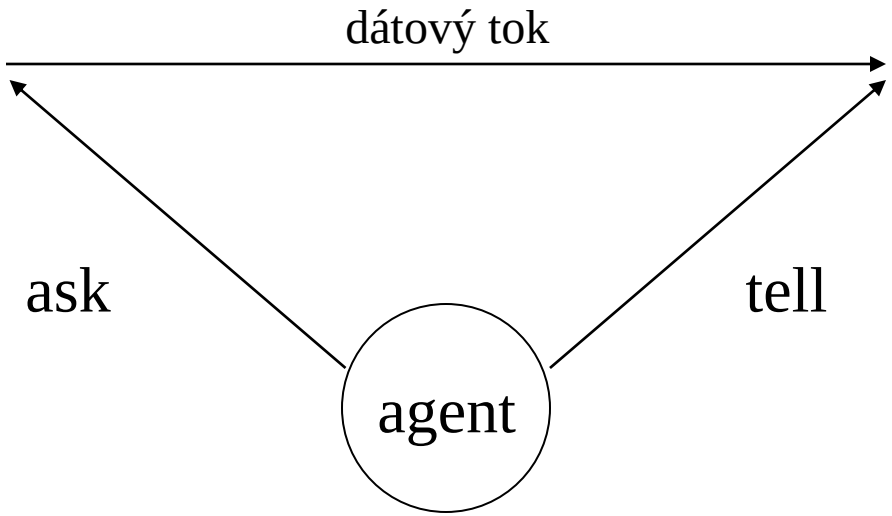
iné riešenia: Proxy + buffrovanie, pipe

Servery na rovnakej úrovni

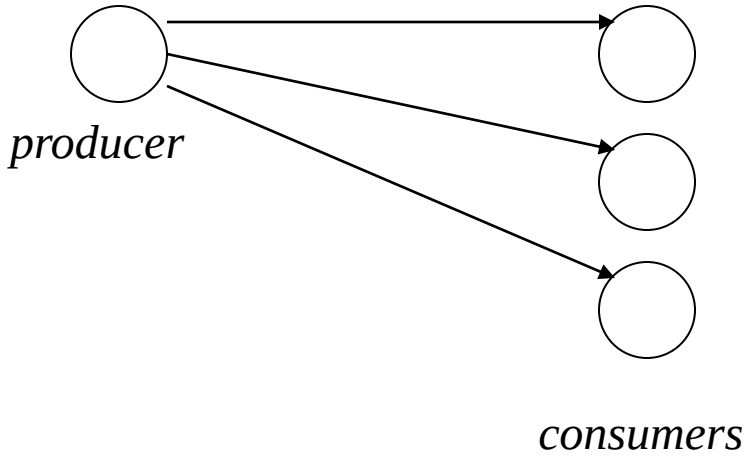
Prievozník (ferryman) je už kódovo veľmi podobný agentom

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

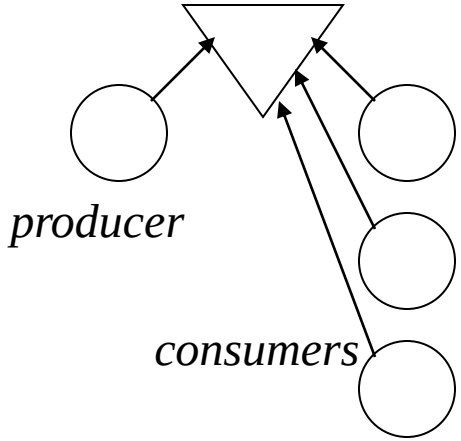


Dátový tok pomocou priamej komunikácie medzi agentami



- 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

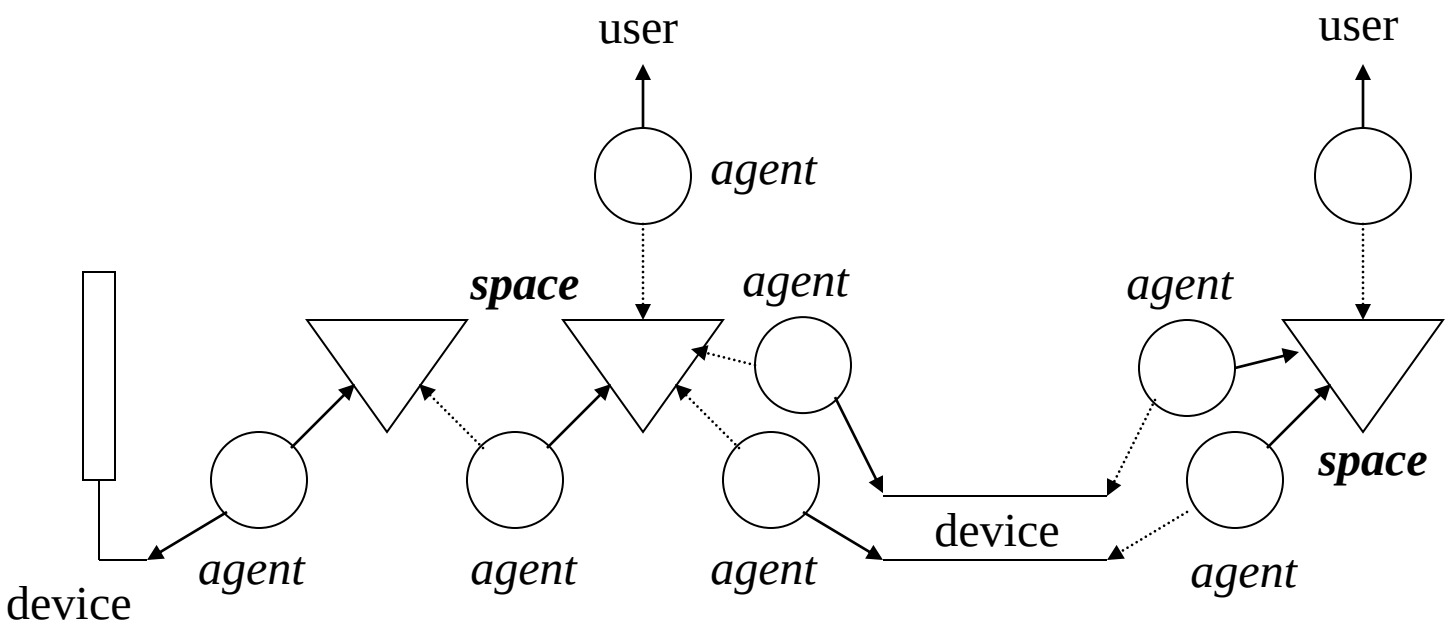
```
void main ()
{
    // initialization
    ...
    pidp = proxy_attach();
    Send(...); // send pidp to space

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

budený triggrom

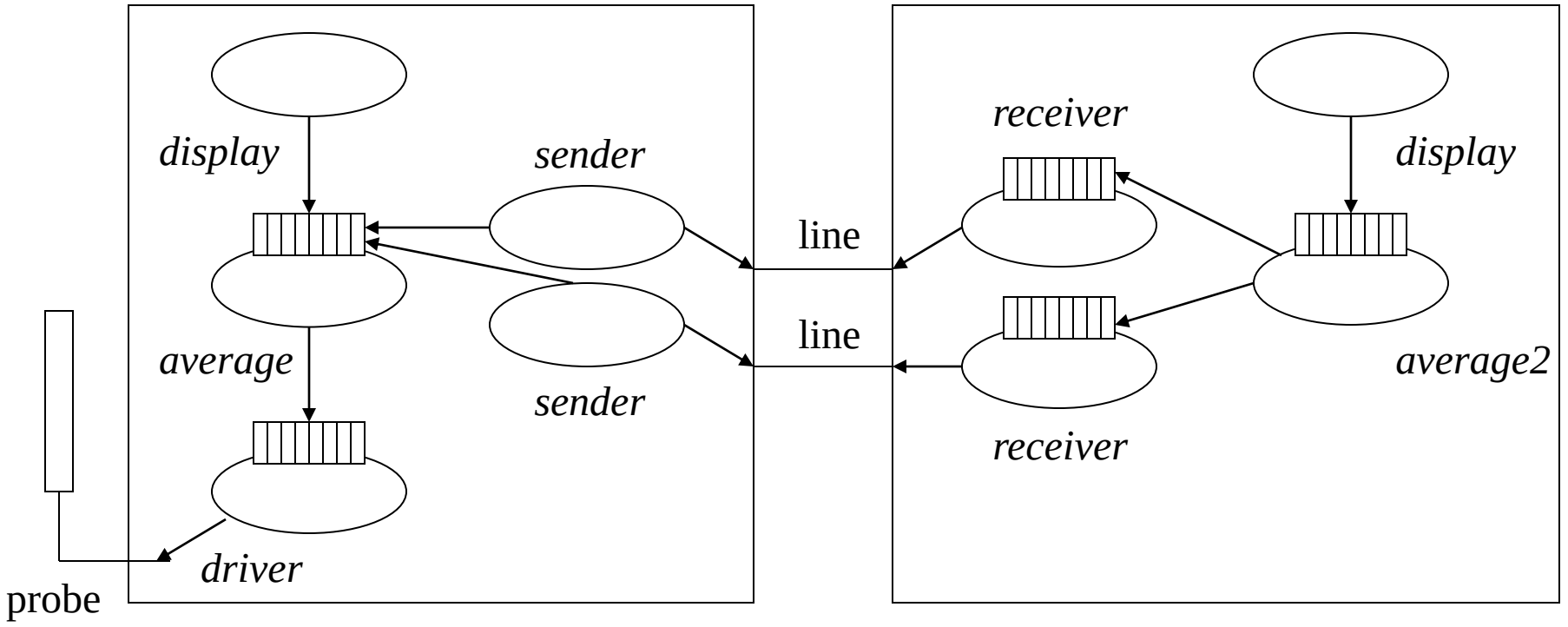
Space

```
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));
    }
}
```

Agent-Space

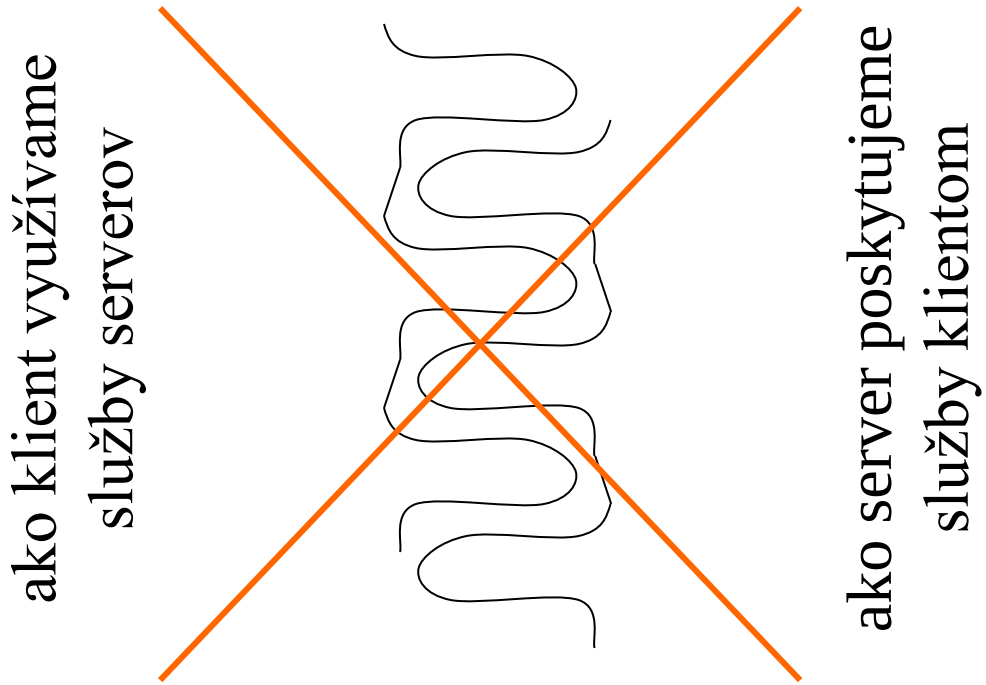

user
user
agent
space
agent
agent
space
device
agent
agent
agent
agent
device

Client-server (porovnanie)



Štruktúra

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



Deadlock

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

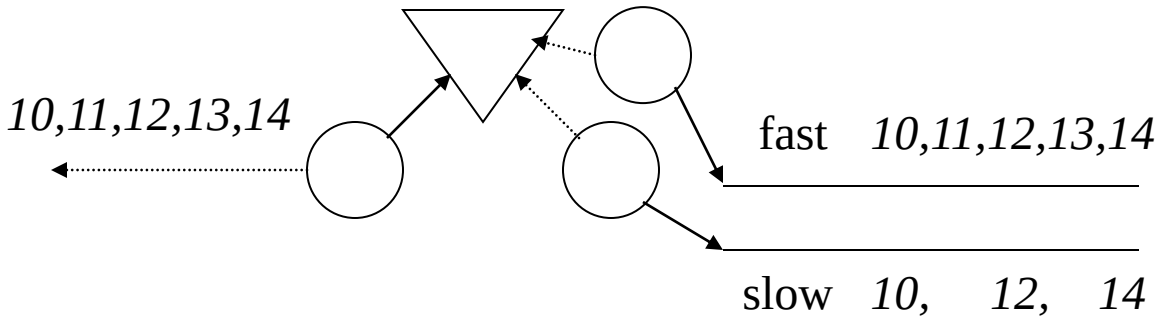
Live Lock

- live lock – každý proces dobrovoľne opúšťa 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

Ako si SRR vyskúšať:

- Doinštalovať do linuxu
  http://developers.cogentrts.com/srr

- Pod windowsami nainštalovať QNX6
  www.qnx.com

Postavenie MAS v IPC

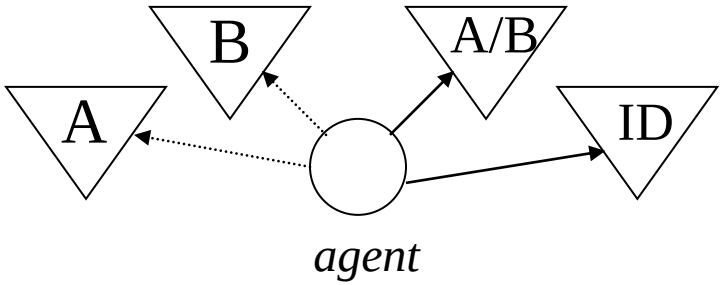
- MAS je jedným z riešení problémov IPC (deadlock, livelock a negarantovaná odozva)
- Je riešením veľmi dobrým, hoci netradičným

Soft crash landing

- V rámci architektúr nad IPC je obvyklé sledovať stav procesov, nechať si od nich hlásiť, že fungujú (watchdog) a inciovať opravnú akciu (recovery) ako napríklad reštartnúť proces, nahlásiť poruchu obsluhe, resetnúť počítač a podobne
- Táto snaha vychádza z predpokladu, že nie je možné urobiť programy bez chýb, ale je možné zariadiť, že tieto spôsobia len krátkodobú poruchu.

Soft crash landing

- Agentové riešenie je pre SCL prínosom v tom, že eliminuje väzby medzi procesmi
- najmä, ak sú agenty čisto reaktívne, takže keď pominie kritický stav, klon pokračuje tam, kde jeho predchdca prestal



Význam rýdzej reaktivity

```
Id = 0;

for (;;) {

        ask(A); ask(B);


        C = A / B;
        Id++;

        tell(C); tell(Id);


}
```

```
for (;;) {

        Id = 0; ask(Id);
        ask(A); ask(B);

        C = A / B;
        Id++;

        tell(C); tell(Id);

}
```