Multi-agent systems

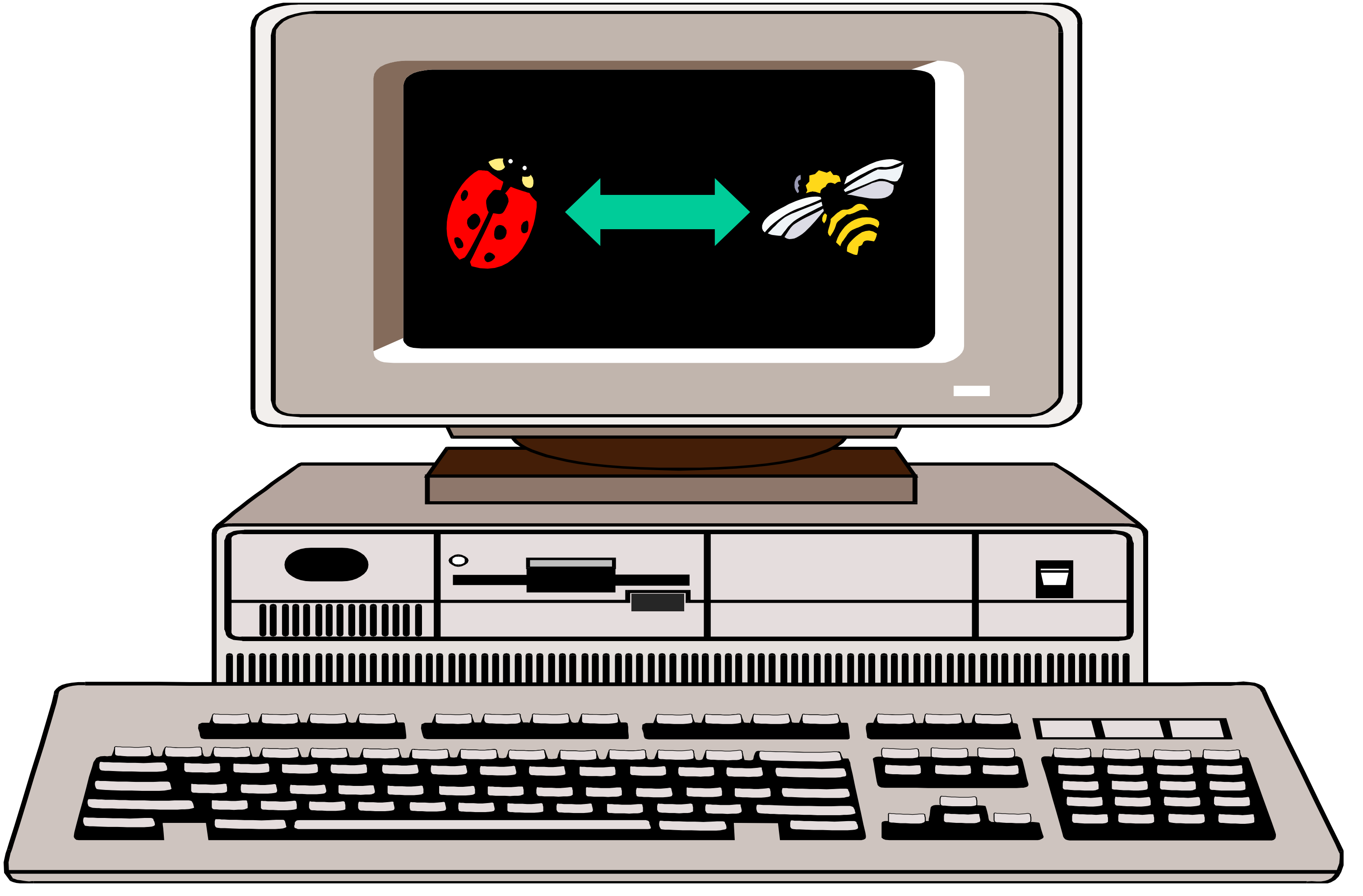
Andrej Lúčny

KAI FMFI UK

lucny@fmph.uniba.sk

http://www.agentspace.org/mas

MAS via IPC



Inter Process Communication

History of IPC:

- signals
- shared memory
- pipe, socket
- message passing

Signals

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

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

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

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

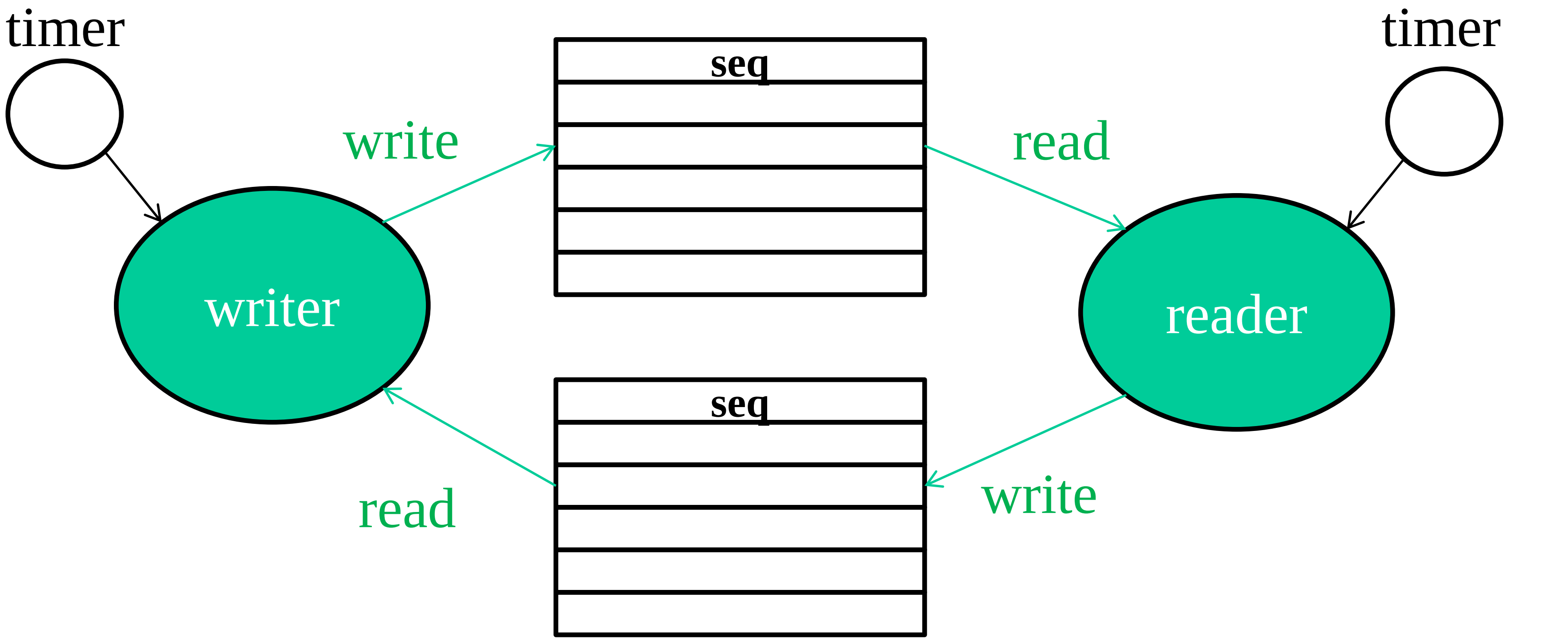
```
event = CreateEvent( NULL, TRUE,
         FALSE, TEXT("Event"));
…
waitForSingleObject (event,
                     timeout);
```

Shared memory

```
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();
...
```

Shared Memory - Synchronization



Synchronization among processes

- Dependent on platform (Windows), from Java we call it via JNI

- Process waiting on event:

```
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);
```

Synchronization among processes

- Dependent on platform (Windows), from Java we call it via JNI

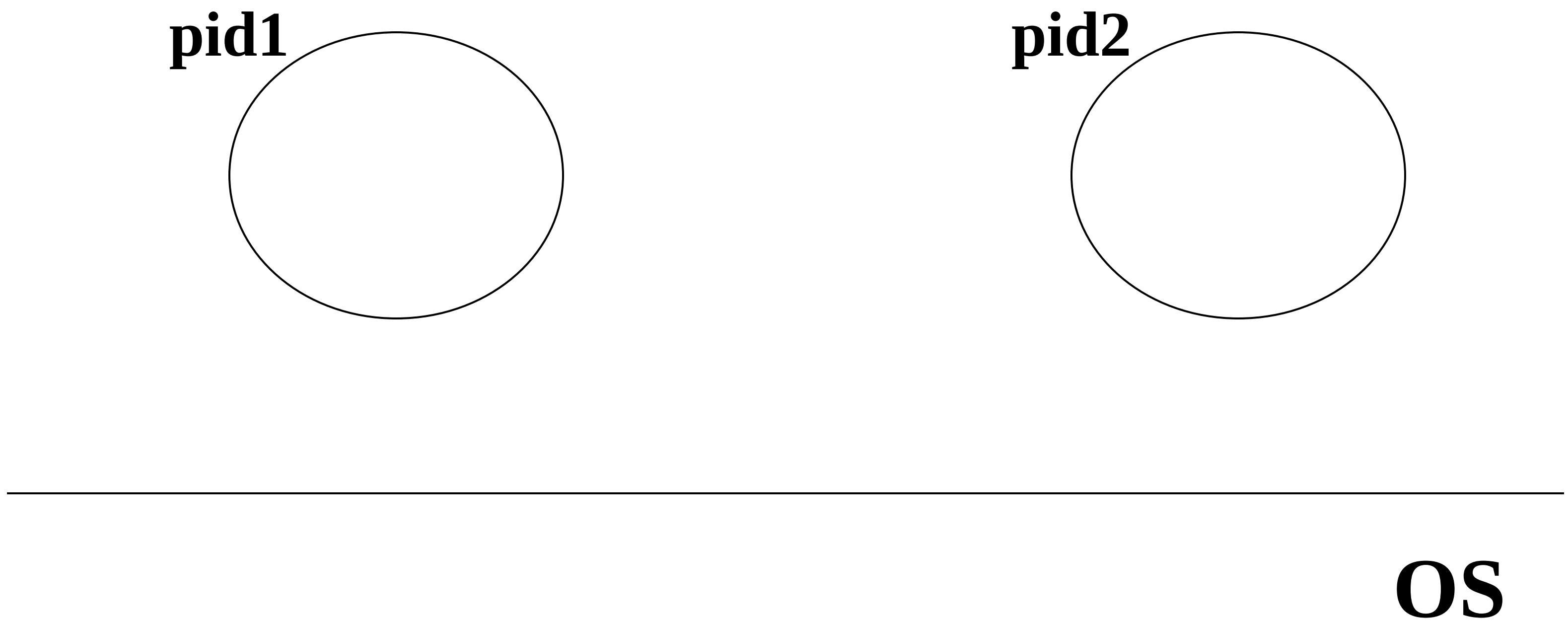
- Process triggering the event:

```
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

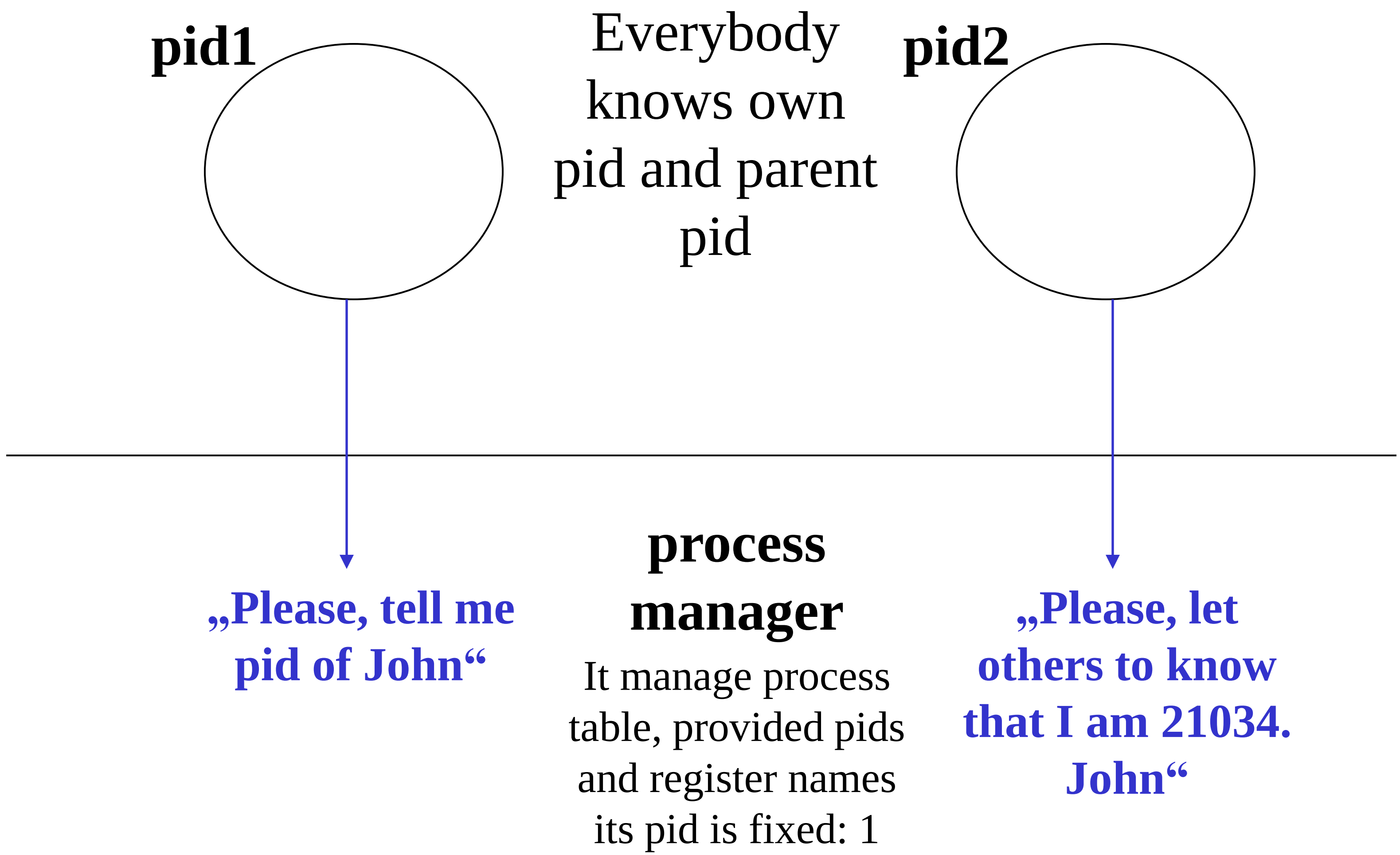
SRR

Processes: pid

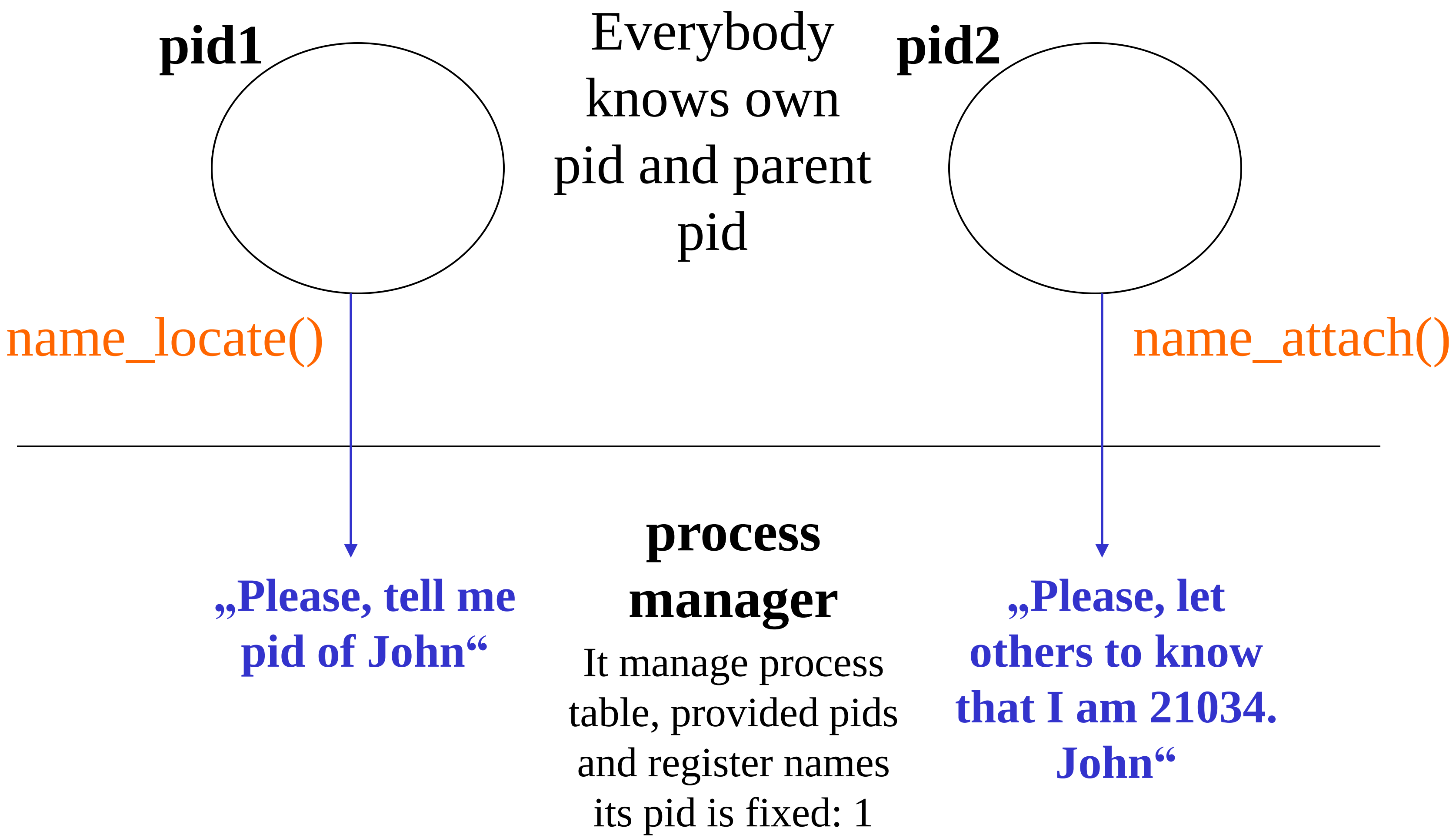


One process can communicate with
other when it knows its process ID

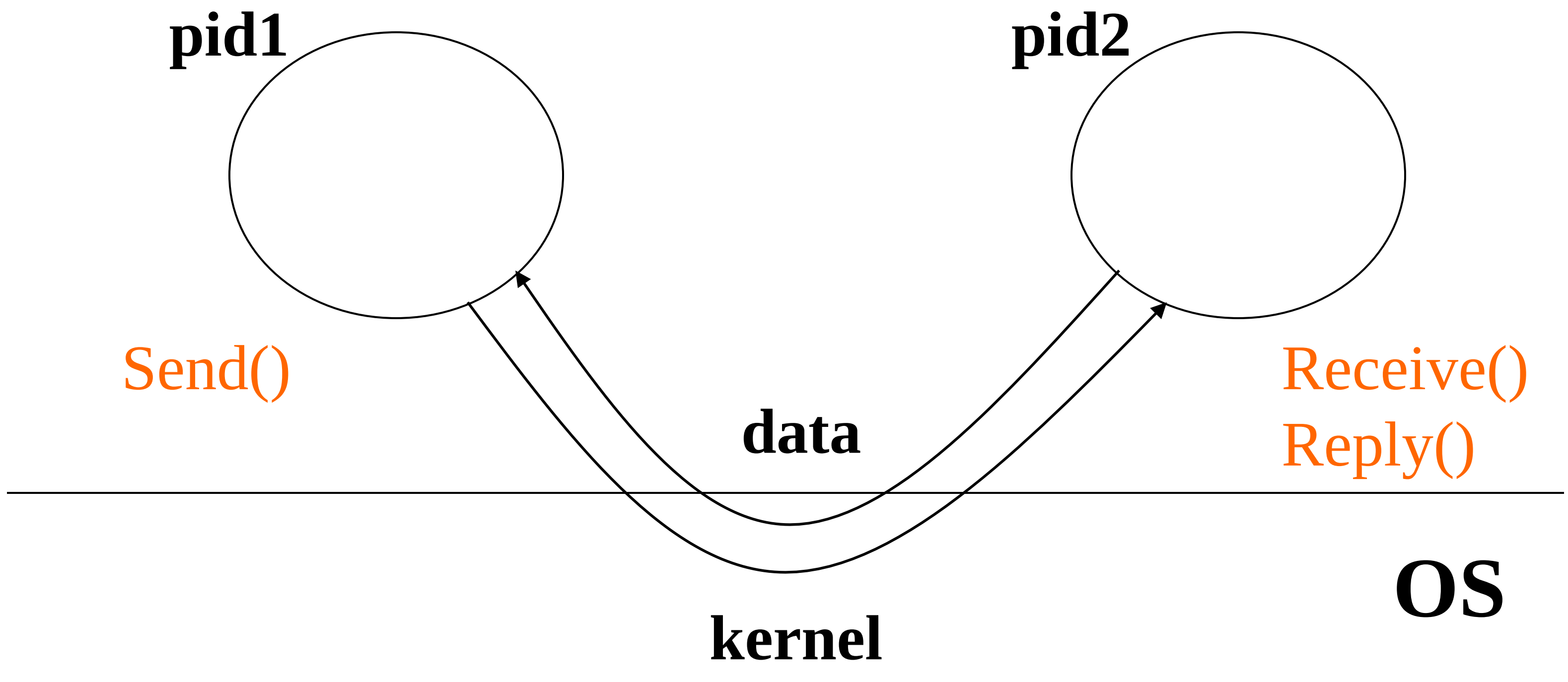
Processes: names



Processes: names

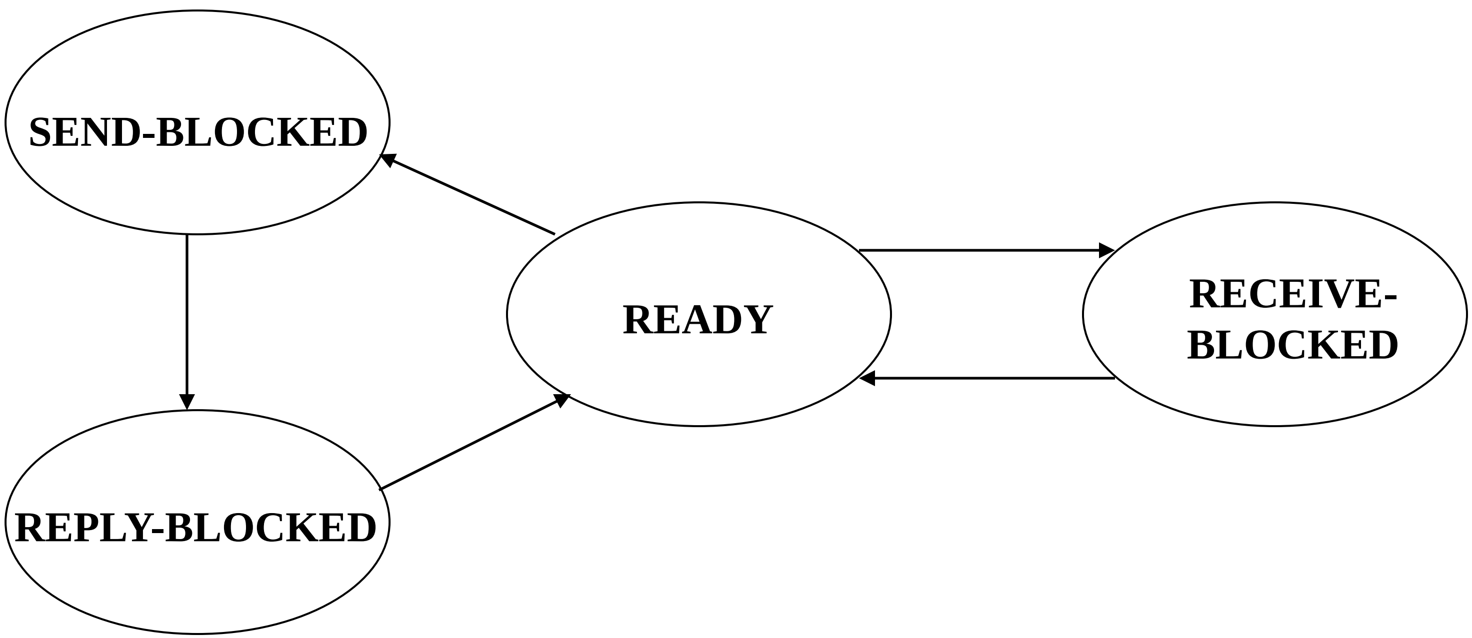


Processes: communication

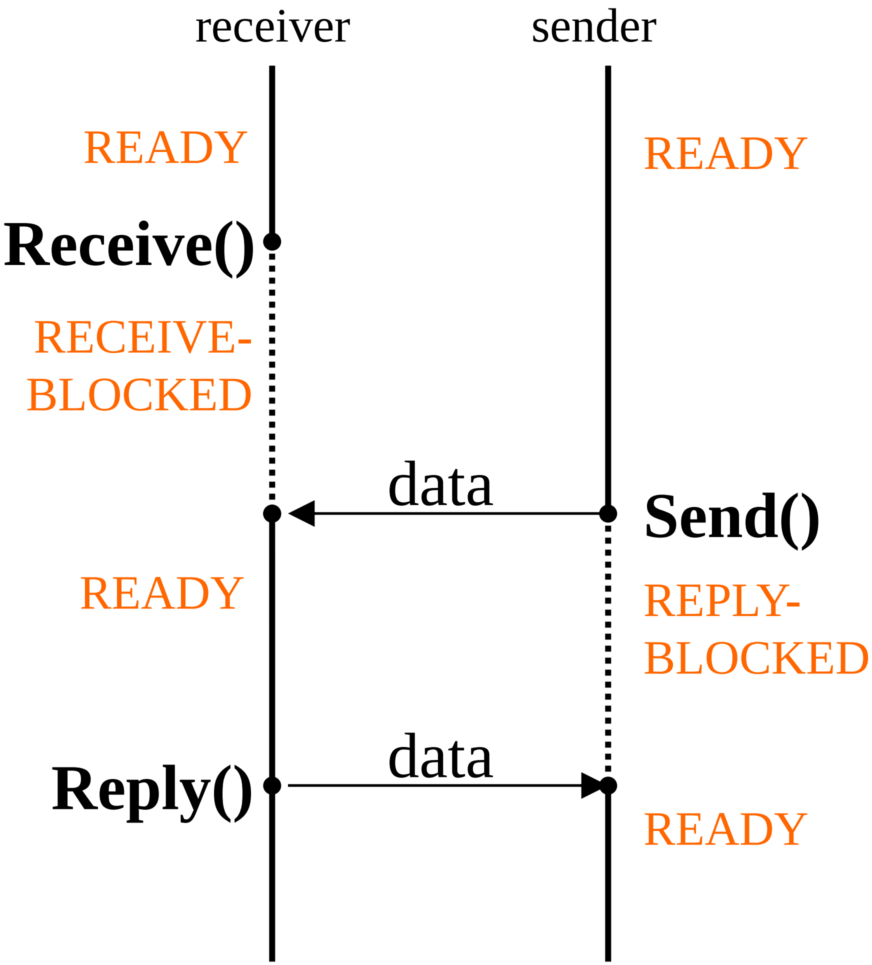


SRR model

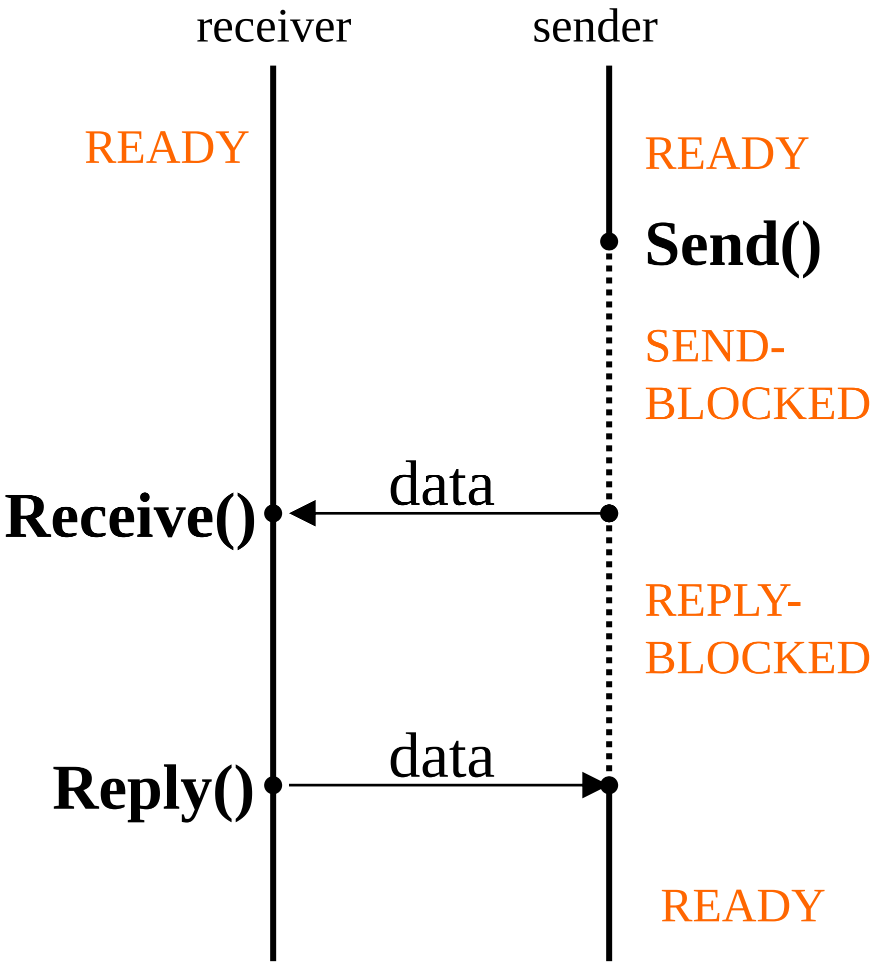
Processes: states



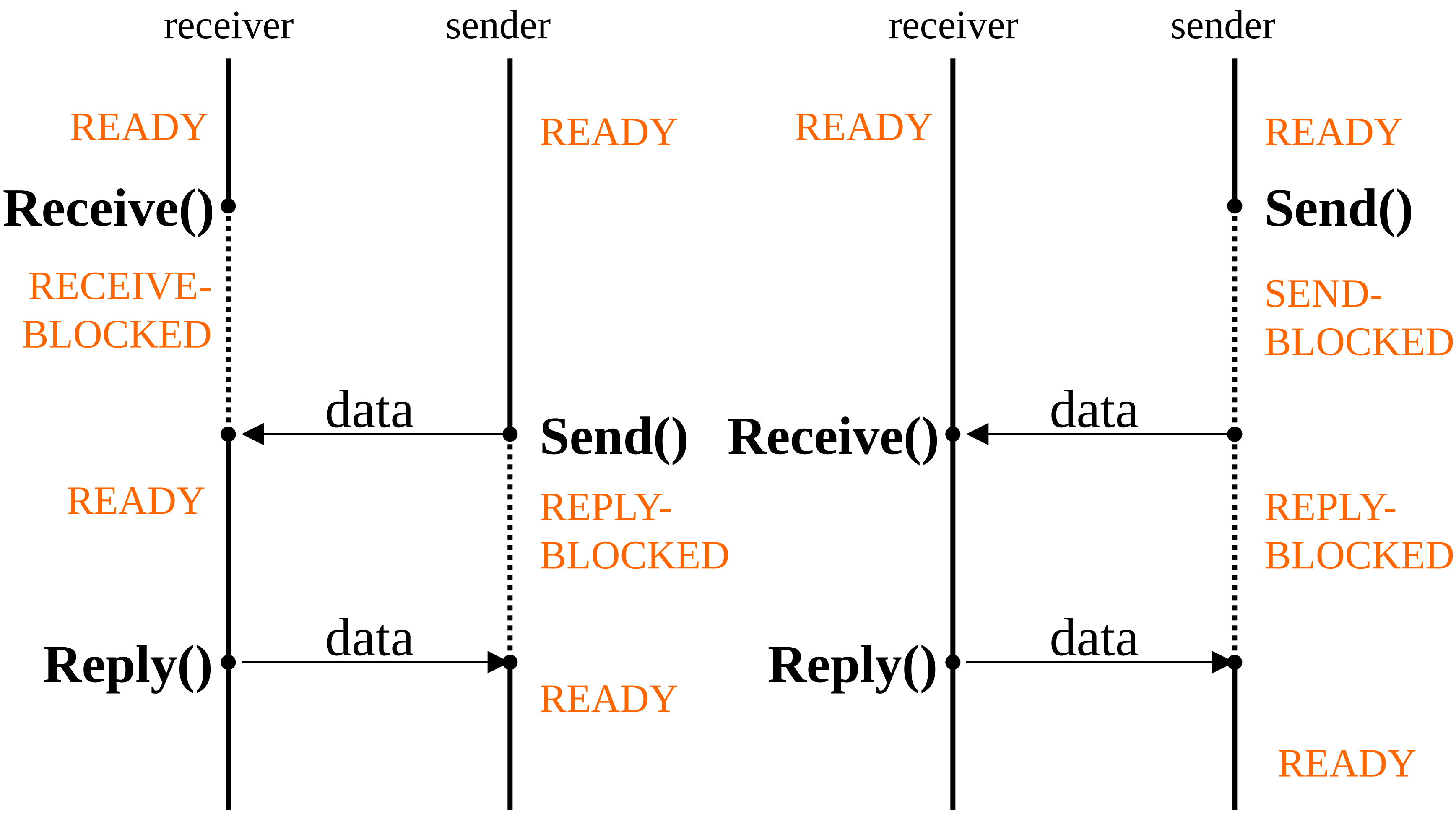
SRR model


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

Which case is preferred by programmer ?

Primitives

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);

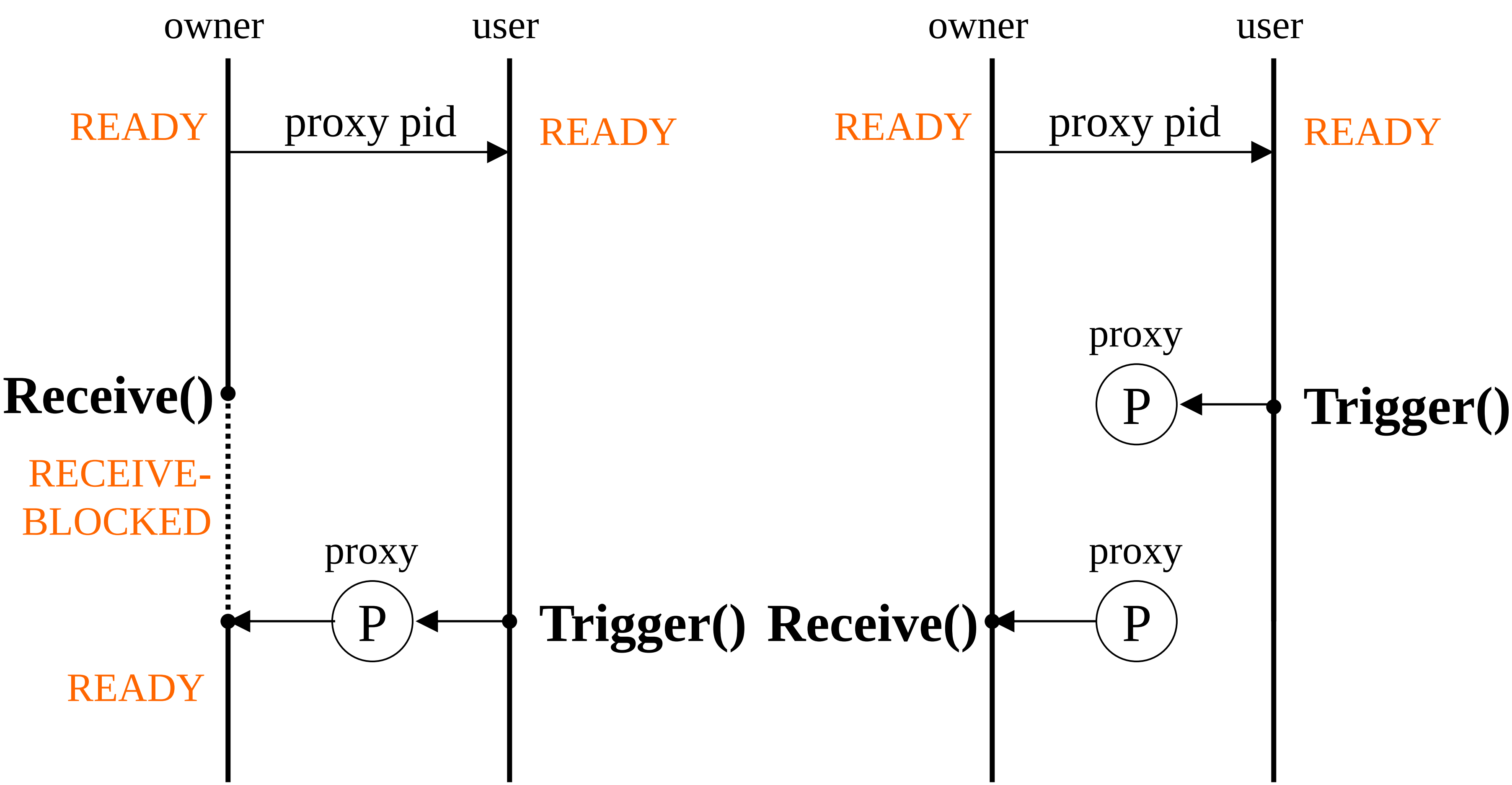
Who does grant that these sizes are corresponding ?

SRR model

Non blocking message passing

- virtual process proxy
- virtual proces timer

SRR model

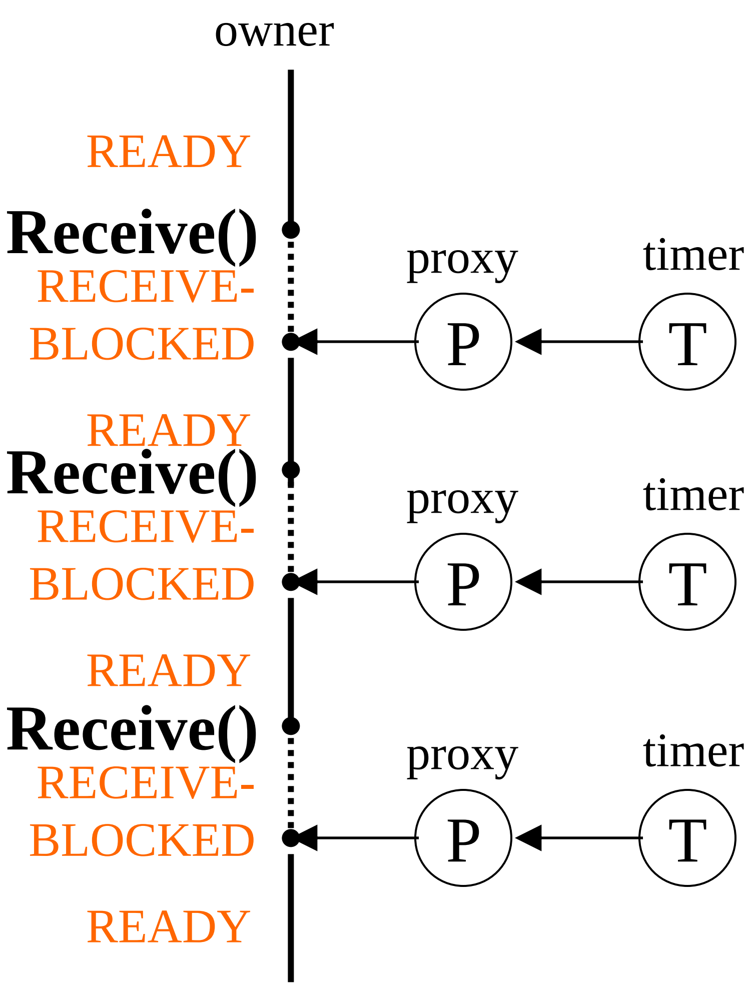


pidp = proxy_attach(0,0,0,-1)

SRR model

Trigger(pidp)

TIMER



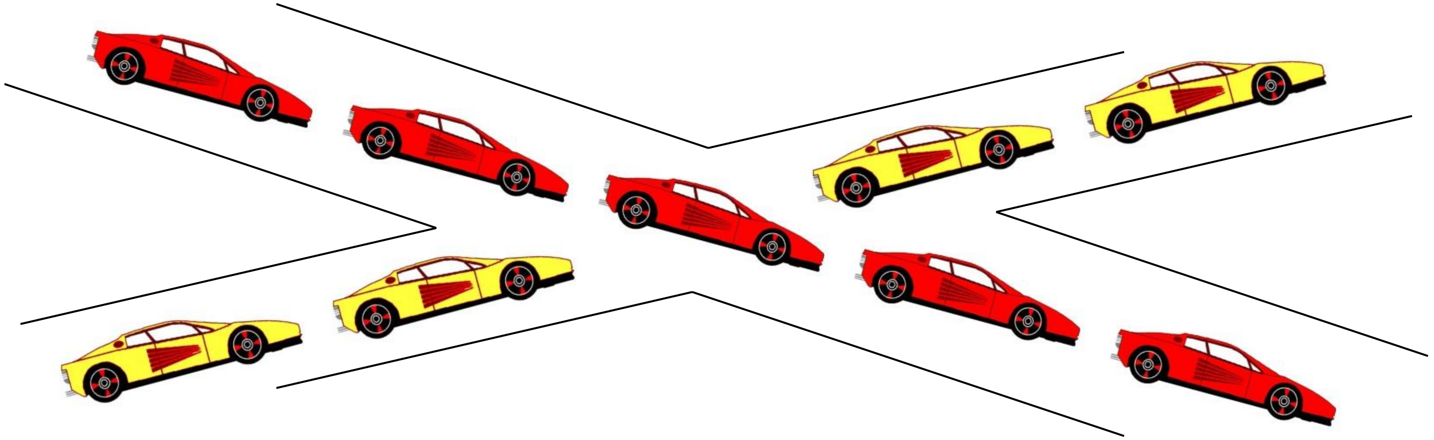
pidt = timer_create(-pidp)

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

SRR model

Problems of communication between processes

- deadlock
- livelock
- lagged response



Solution

- **architecture -** rules for developer

One possible solution:

pyramidal client-server architecture

Client-Server

In SRR model

- Server is receiver
- Client is sender

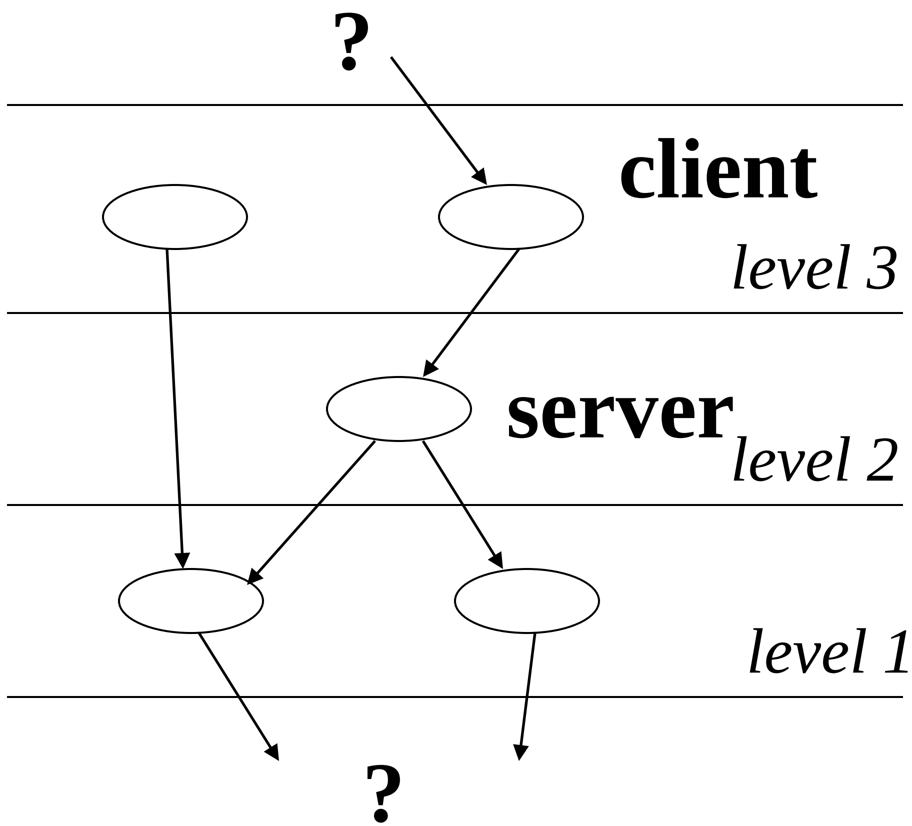
Of course a process and be sever and client in the same time concerning more relations to other different processes. Thus we can find both Receive and Send calls in its code

where ?

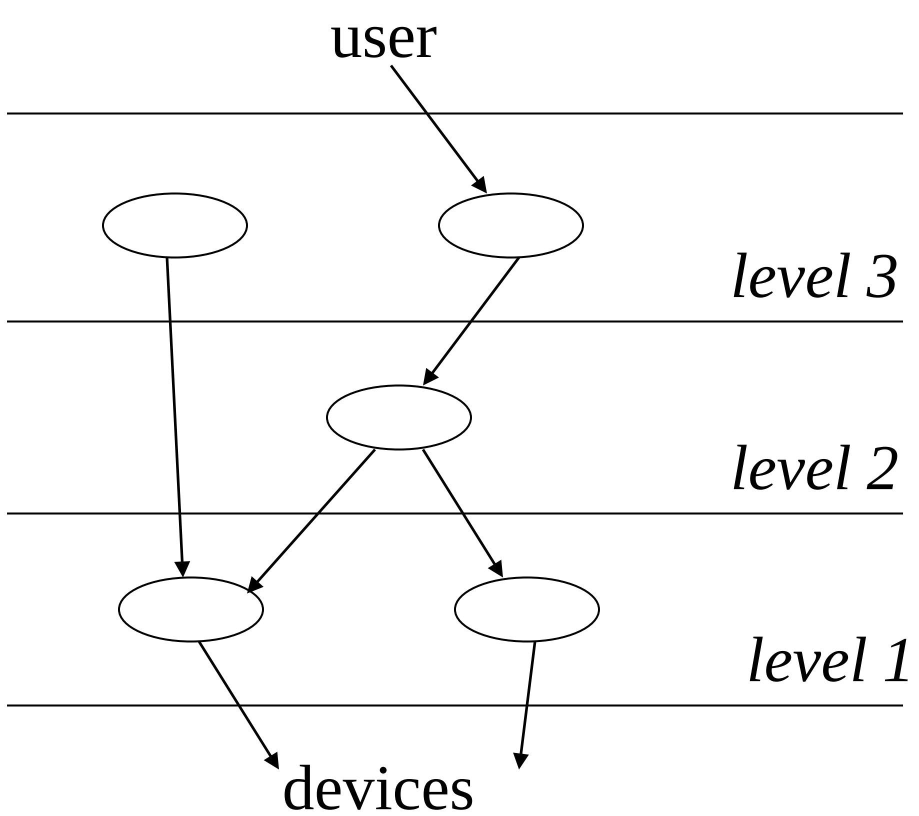
Pyramidal architecture
Client-Server

1. System is divided to levels
2. Each server is put to certain level
3. Each client must be put to higher level than its server

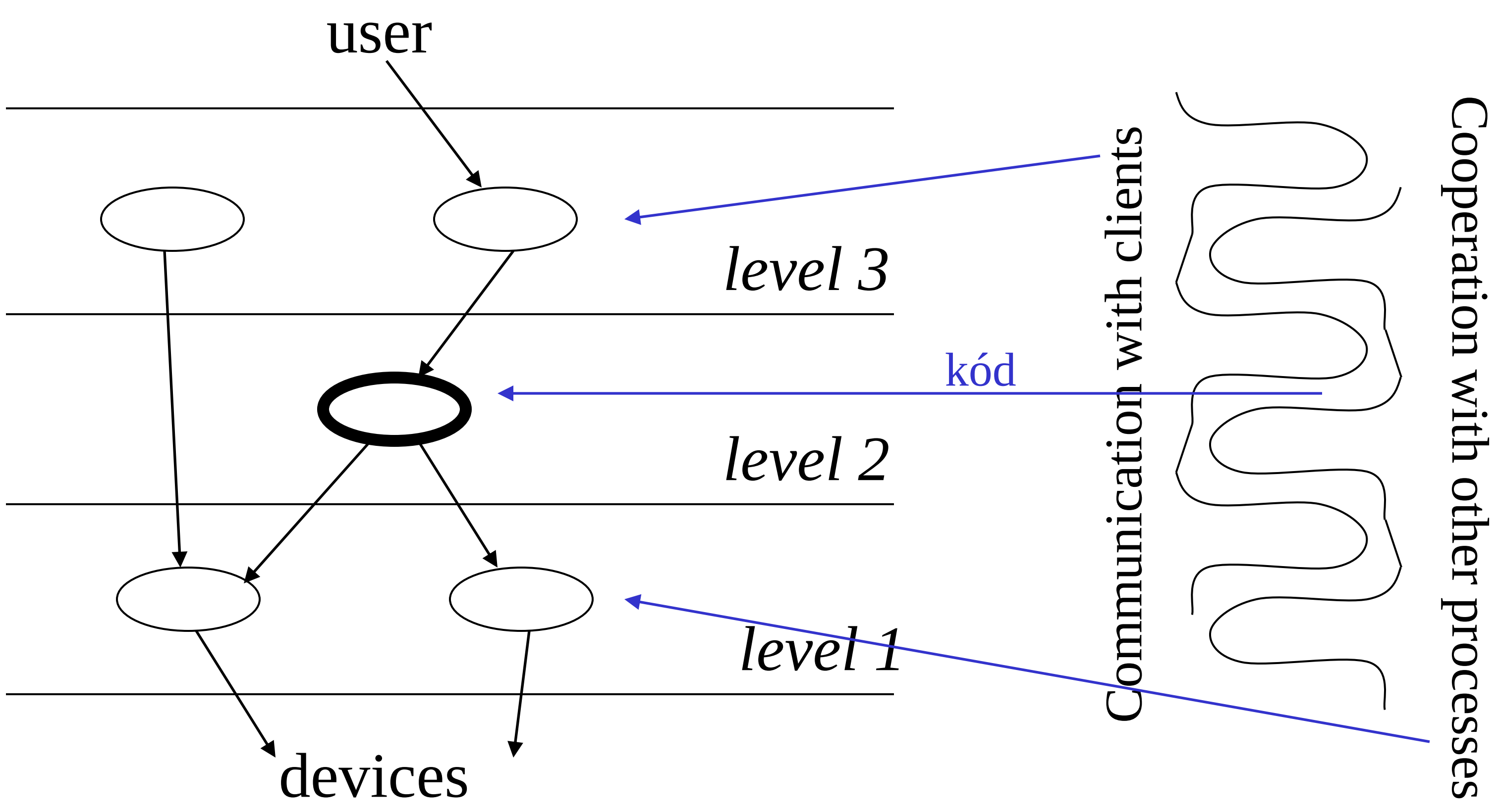
Pyramidal architecture
Client-Server



Pyramidálna architektúra Client-Server



Server structure



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;
   // inicialization
   if (name_attach("...") == -1) return;
   for (;;) {
      pid = Receive(0,&msg,sizeof(msg));
      if (msg.header != SERVER_HEADER)
         continue;
      switch (msg.action) {
         case SERVER_ACTION1:
            // process msg
            break;
         ...
         case SERVER_ACTIONx:
            ...
            break;
      }
      Reply(pid,&msg,sizeof(msg));
   }
}
```

Client - utility

```
void main ()
{
   struct server_msg msg;
   // inicialization
   pid = name_locate("...");
   msg.header = SERVER_HEADER;
   msg.action = SERVER_ACTIONy;
   // prepare msg.data;
   Send(pid,&msg,&msg,
      sizeof(msg),sizeof(msg));
   // process msg.data
}
```

Client - data collector

```
void main ()
{
   // inicialization
   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) {
         // prepare msg
         Send(pids,&msg,&msg,
            sizeof(msg),sizeof(msg));
         // process msg
      }
   }
}
```

Server Decomposition

Problem of the lagged response:

memory-unstable solution:

- call fork() and run separated process which treats the requested service

memory-stable solution (the only solution if we have no threads):

- Master - slave

Master - slave

Solution: master – server, slave – auxiliary task which is launched by master

Master can pass treatment of a service to slave. Thus it is available for serving further clients

(slave is not client!)

Slave

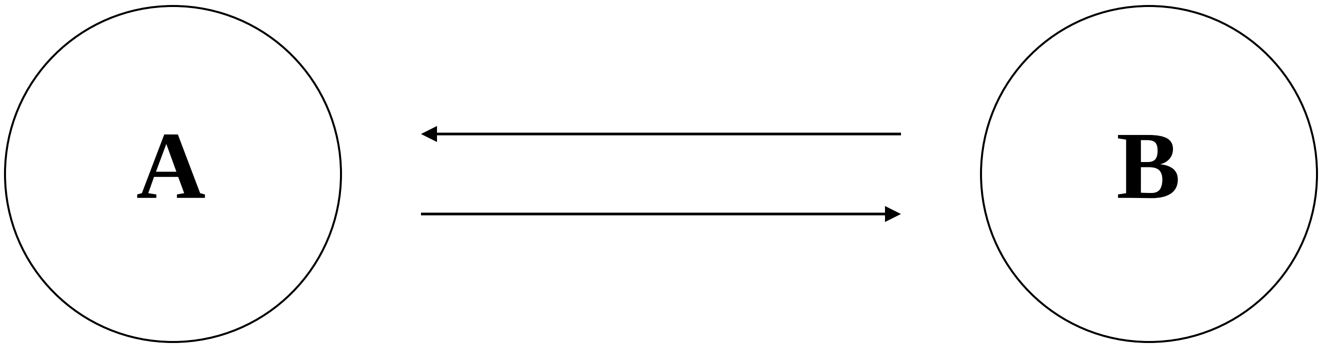
```
void main ()
{
    pid_m = getppid();
    Send(pid_m,...); //what do you wish, master ?
    for (;;) {
        task ();
          Send(pid_m,...); //your wish is completed,
                           //what do you which next master?
    }
}
```

In what SRR state the slave spends major part of its course ?

Master

```
main ()
{
    struct server_msg msg;
    struct server_port *port;
    // inicialization
    ports_init();
    ihave = 0; towhom = 0; spid = start_slave(); // spawn
    for (;;) {
        pid = Receive(0,&msg,sizeof(msg));
        if (pid == spid) {
            ihave = 1;
            if (towhom > 0) Reply(towhom,...);
        }
        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:
                // process *port and msg
                Reply(pid,&msg,sizeof(msg));
                break;
            ...
            case SERVER_ACTIONx:
                Reply(spid,...); towhom = pid;
                break;
        }
}}
```

Servers on the same level

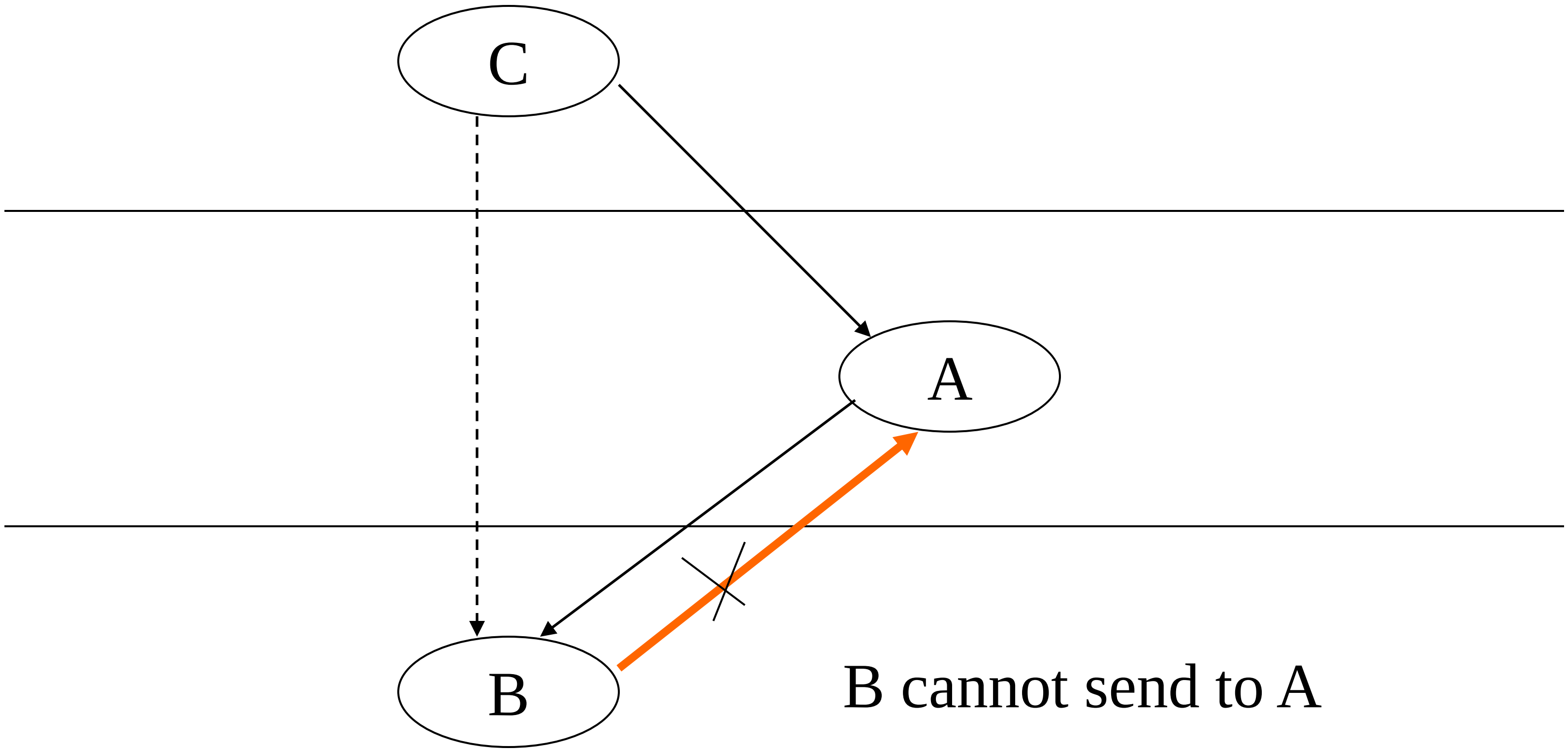


- A is client of server B
- B je client of server A

How to solve it ?

This problem is not often but typical for pyramidal client-server architecture

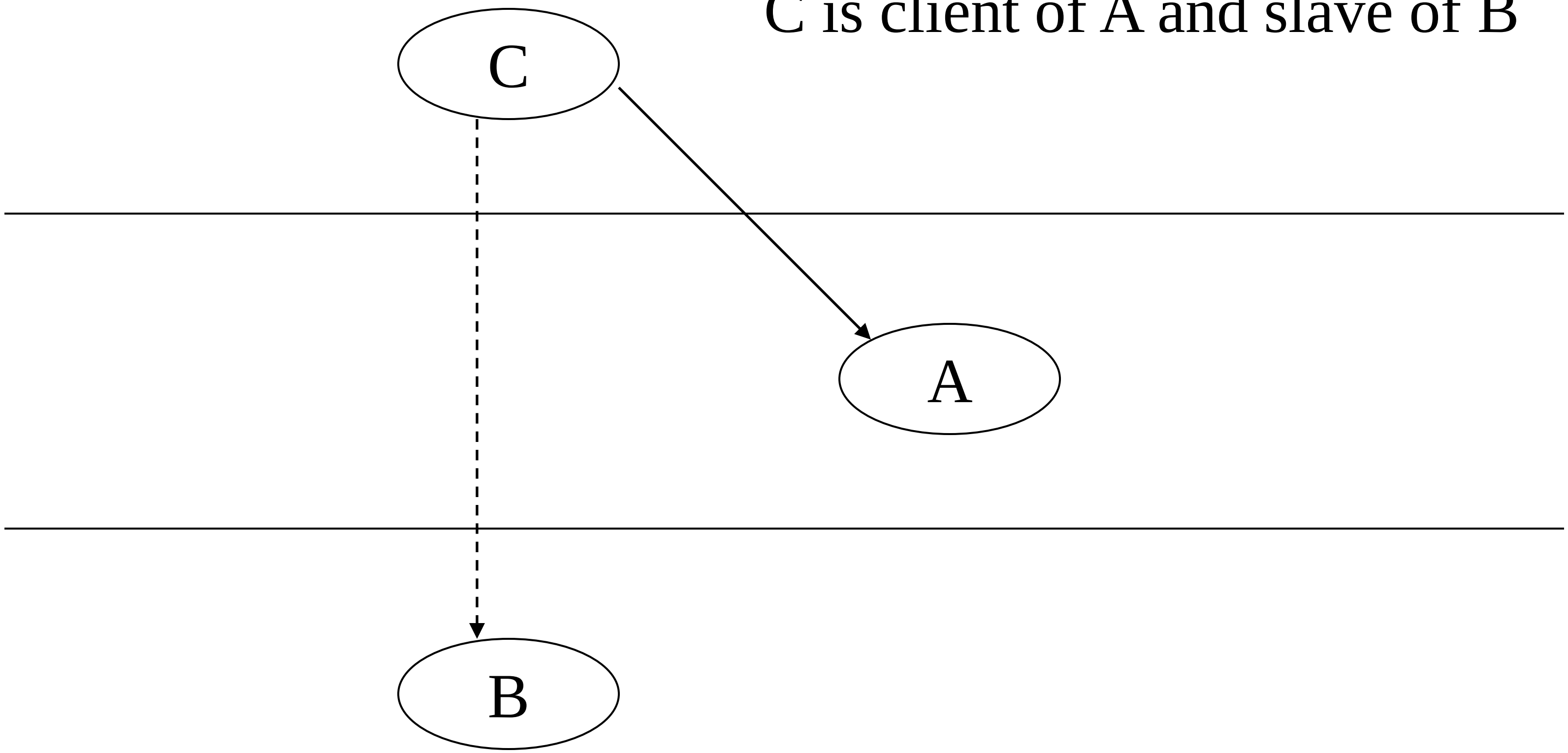
Ferryman



B cannot send to A

Ferryman

C is client of A and slave of B



Servers on the same level

Solution: ferryman + buffering

```
void main ()
{
    pid_low = getppid();
    pid_high = name_locate("...");
    for (;;) {
        Send(pid_low,...); //what do you want to send, master?
        Send(pid_high,...);//you client would like to send you
this, server
    }
}
```

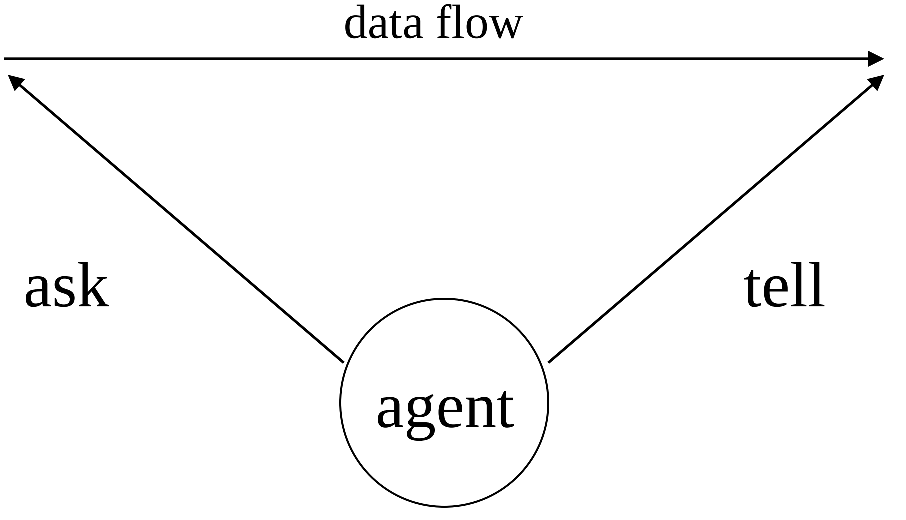
Ferrymen is slave of one server and client of the other

Servers on the same level

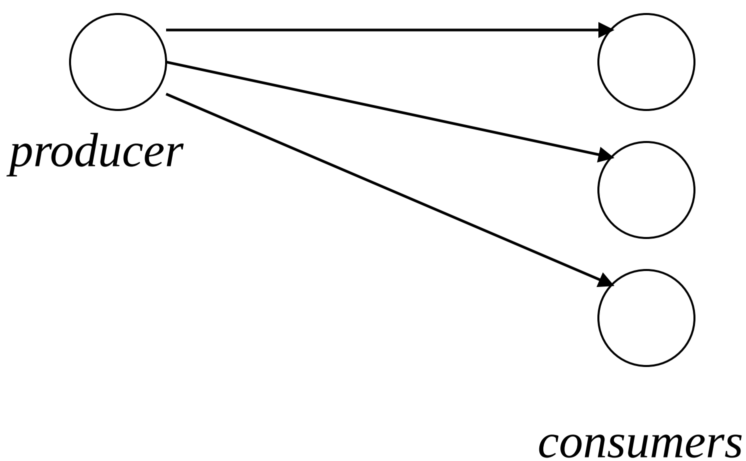
Ferryman’s code is very similar to code of agent

This inspire us for another architecture, which would not solve communication between servers on the same level as a special case, but which would employ solution of the case as the fundamental principle of communication among processes

Data flow via agent

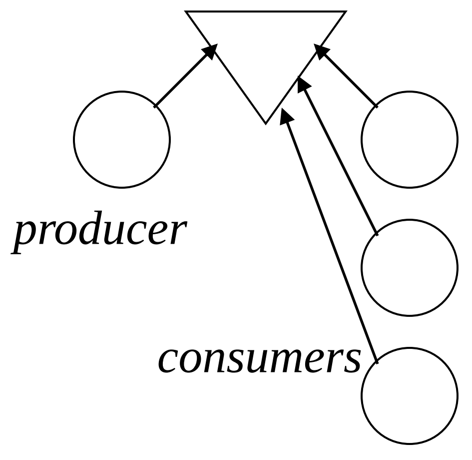


Data flow via direct communication among agents



- deadlock problem is not addressed

Data flow via indirect communication among agents



- deadlock is not possible

Agent-Space

- architecture which solves communication problems among processes
- based on indirect communication among agents

Every process is

- agent

or

- 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(...); ...
        }
    }
}
```

driven by timer

Agent

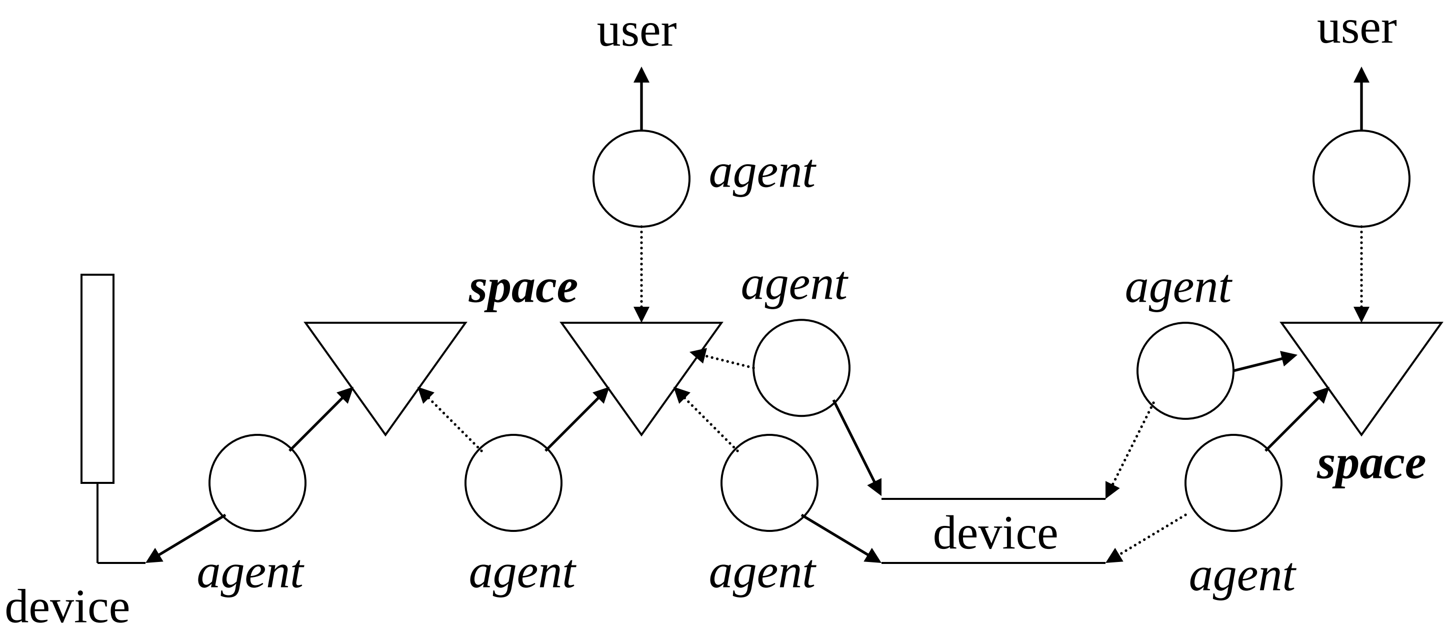
```
void main ()
{
    // initialization
    ...
    pidp = proxy_attach();
    Send(...); // send pidp to space

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

driven by trigger

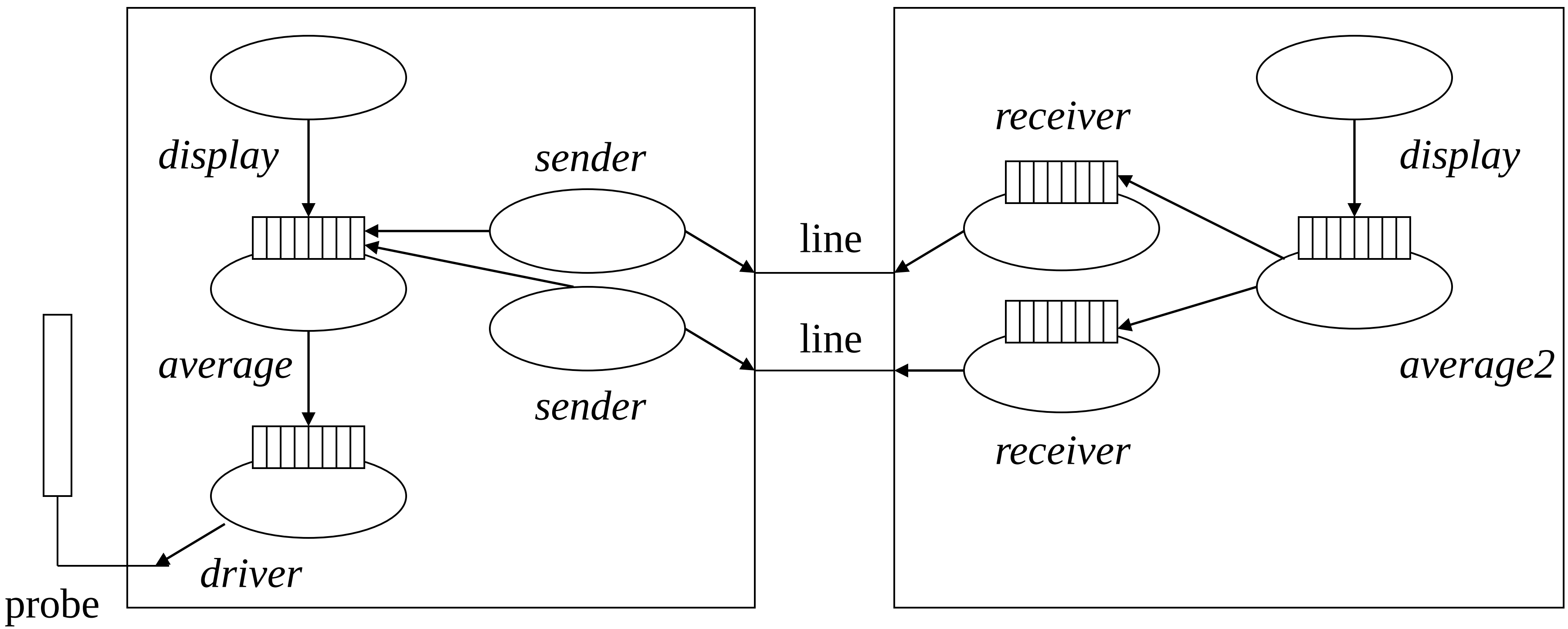
Space

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

Agent-Space

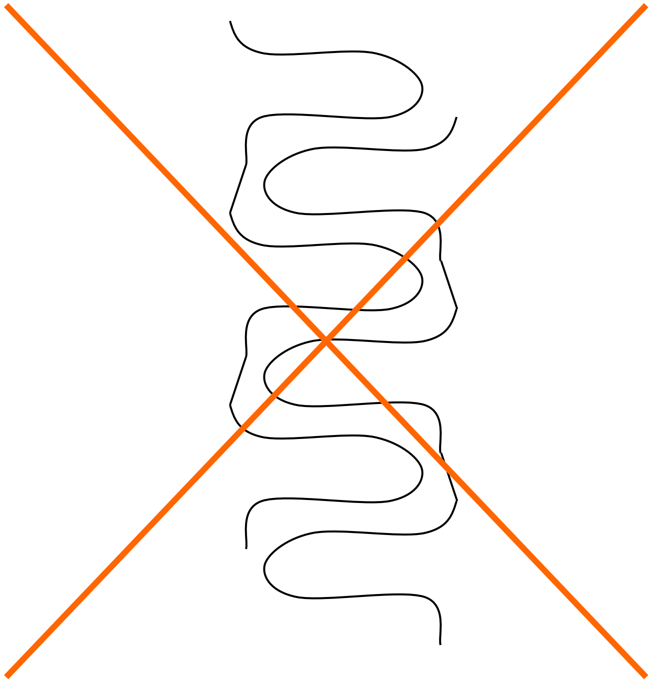

user
user
agent
space
agent
agent
space
device
agent
agent
agent
device
agent

Client-server (for comparison)



Code Structure

- Space contains only communication code
- Agents contains only application code



Deadlock

- Space calls only Receive and Reply
- Agent calls only Send and may be also Receive but just on proxy
- There is no other kind of process
- Thus deadlock is not possible

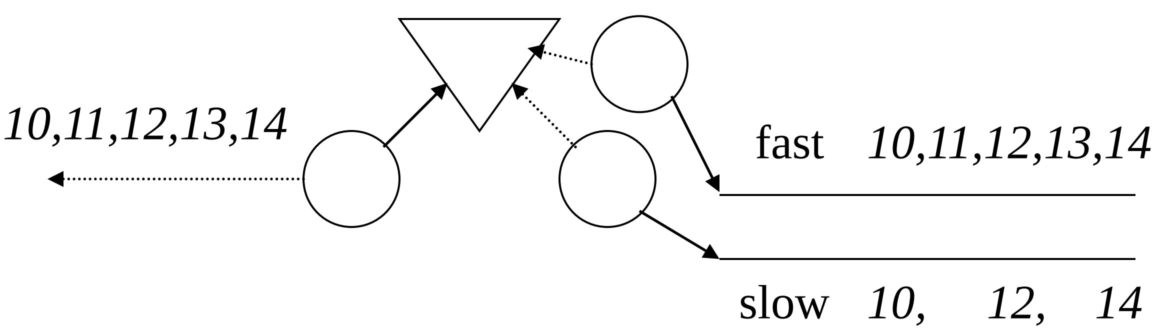
Live Lock

- live lock – every process calls Receive, thus it regularly yields processor

Lagged response

• swift response is supported by nature of information in Agent-Space architecture where the information is automatically sampled if there is lack time to process it

• (this is significant difference in comparison with actors)



Relation MAS and IPC

- MAS is one of possible solution of communication problems of IPC (deadlock, livelock and lagged response)
- It is not traditional but interesting and well-working solution

How to use SRR:

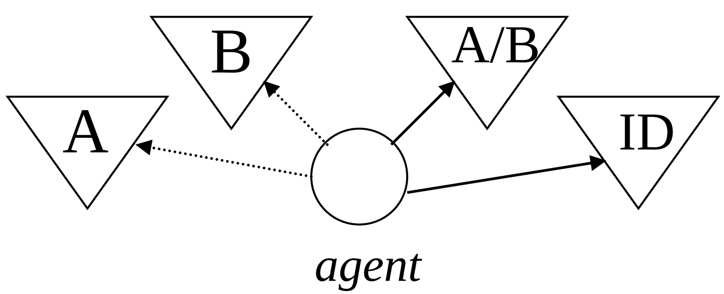
- Install proper module into Linux kernel http://developers.cogentrts.com/srr
- or install virtual machine called NC of QNX6 into MS Windows www.qnx.com and employ so called migration toolkit

Soft crash landing

- Architectures based on IPC often monitors state of processes, let them to report their operation (watchdog) and initialize a remedy operation (recovery) as restart of the process, restart of the computer or warning of user
- This is based on assumption that we are not able to develop system without error but can manage that the errors cause only tentative troubles.

Soft crash landing

- Agent based solution is a good choice for SCL, because of elimination of direct relations among processes
- Namely if agents are purely reactive, the system automatically return to normal operation after recovery from and error



Pure reactivity

```
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);

}
```