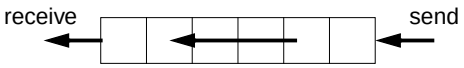
# 7. Asynchronous Message Passing

**Processes send and receive messages via channels**

**Message**: value of a composed data type or object of a class

**Channel**: **queue** of arbitrary length, containing messages




**operations** on a channel:

- **send (b)**: adds the message b to the end of the queue of the channel; does **not block** the executing process (in contrast to synchronous send)
- **receive()**: yields the oldest message and deletes it from the channel; **block**s the executing process as long as the channel is empty.
- **empty()**: yields true, if the channel is empty; the result is **not necessarily up-to-date.**

**send** and **receive** are executed under mutual exclusion.





# Channels implemented in Java

```
public class Channel
{                                  // implementation of a channel using a queue of messages
   private Queue msgQueue;

   public Channel ()
      { msgQueue = new Queue (); }

   public synchronized void send (Object msg)
      { msgQueue.enqueue (msg); notify(); }     // wake a receiving process

   public synchronized Object receive ()
      {  while (msgQueue.empty())
            try { wait(); } catch (InterruptedException e) {}
         Object result = msgQueue.front();        // the queue is not empty
         msgQueue.dequeue();
         return result;
      }

   public boolean empty ()
      { return msgQueue.empty (); }
}
```

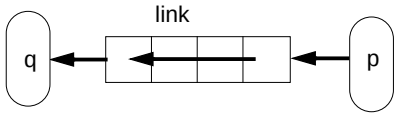
All waiting processes wait for the same condition => notify() is sufficient.
After a notify-call a new receive-call may have stolen the only message => wait loop is needed



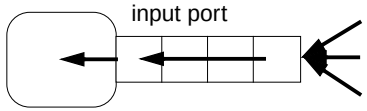


# Processes and channels

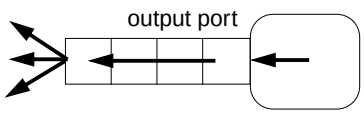
**link**:
**one sender** is connected to **one receiver**;
e. g. processes form chains of transformation steps (pipeline)



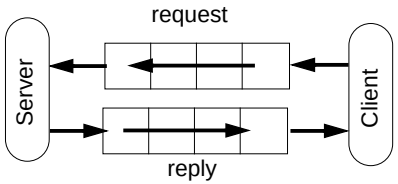

**input port** of a process:
**many senders - one receiver;**
channel belongs to the receiving process;
e. g. a server process receives tasks from several client processes




**output port** of a process:
**one sender - many receivers**;
channel belongs to the sending process;
e. g. a process distributes tasks to many servers (unusual structure)




pair of **request and reply channels;**
one process requests - the others replies;
tight synchronization,
e. g. between client and server








# Termination conditions

When system of processes terminates the following **conditions** must hold:

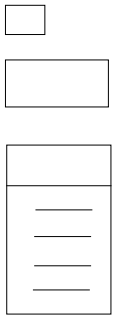
1. **All channels are empty.**
2. **No** processes are **blocked on a receive** operation.
3. **All** processes are **terminated**.

Otherwise the **system state is not well-defined**, e.g. task is not completed, some operations are pending.

**Problem:**
In general, the processes **do not know the global system state**.

# Message structures

A **message object** may have arbitrary structure suitable for the **particular purpose**:



empty — **synchronization** only

kind — different **kinds** of messages, without data
e. g. signal different kinds of events

kind
argument vector — different kinds of messages with **data**
e. g. number and or identities of resources

special case:
a channel on which the sender expects a reply

**Operations** on messages:
constructors

setKind (k), getKind ()

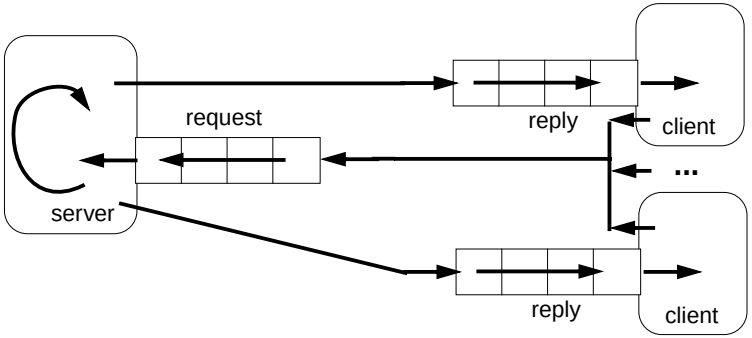
setArg (ind, val), getArg (ind), getArgList ()





# Client / server: basic channel structure

One **server process responds to requests of several client processes**




**request channel:**
input port of the server

**reply channel:**
one for each client (input port),
may be sent to the server included in the request message

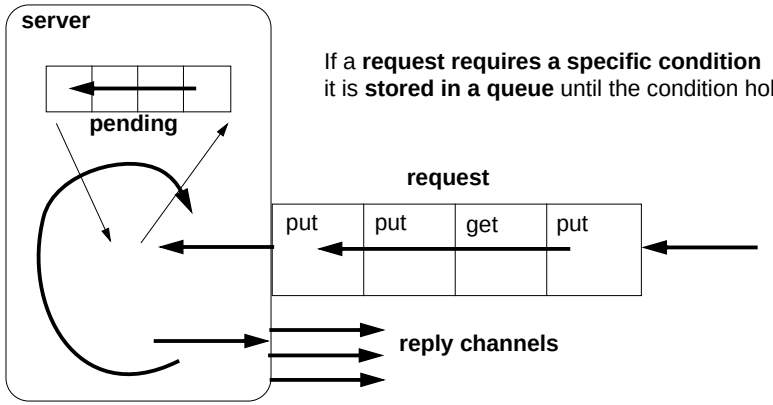
**Application**: server distributes data or work packages on requests





# Server processes: different kinds of operations

**Different requests** (operations) are represented by different **kinds of messages**.




If a **request requires a specific condition**
it is **stored in a queue** until the condition holds.

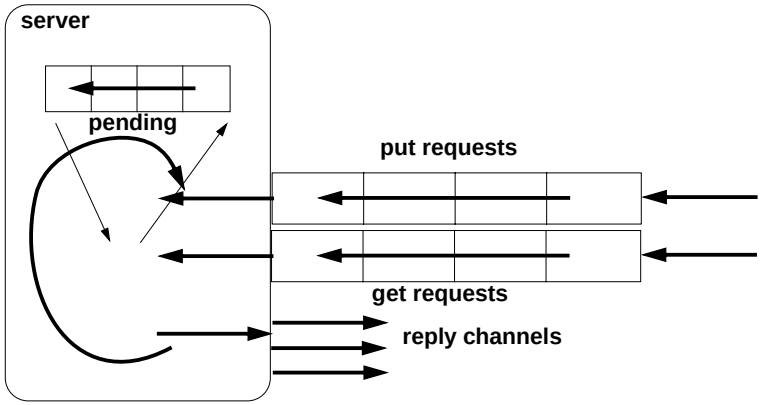
The server processes the requests **strictly sequentially**;
thus, it is guaranteed that critical sections are **not executed interleaved**.

**Termination:** terminate clients, empty channel, empty queue.





# Different kinds of operations on different channels

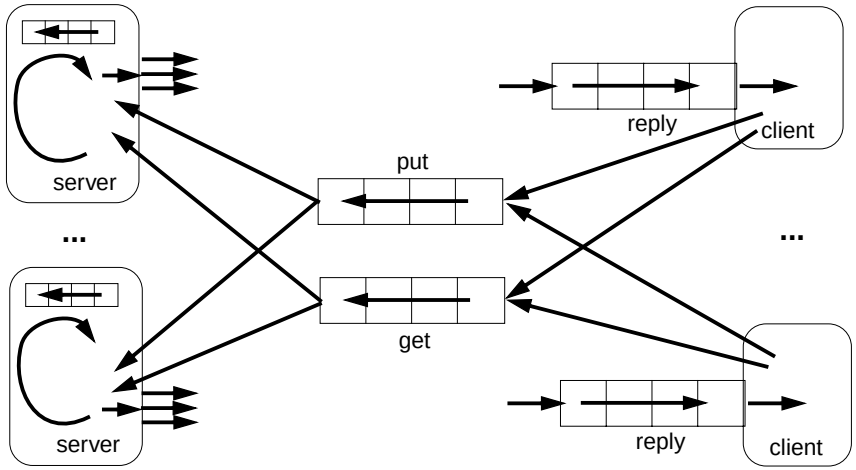



**Server must not block on an empty input port while another port may be non-empty:**

```
while (running) {
   if (!putPort.empty()) {  msg = putPort.receive(); ... }
   if (!getPort.empty()) {  msg = getPort.receive(); ... }
   if (!pending.empty()) {  msg = pending.dequeue(); ... }
}
```

# Several servers

**Several server processes,** several client processes**, several request channels**




**Termination:** empty request channels, empty queues, empty reply channels

**Caution:** a **receive** on a channel **may block** a server!





# Receive without blocking

If several processes receive from a channel **ch**, then the check

```
if (!ch.empty()) msg = ch.receive();
```

may block.
That is not acceptable when several channels have to be checked in turn.
Hence, a new non-blocking channel method is introduced:

```
public class Channel
{  ...
   public synchronized Object receiveMsgOrNull ()
   {  if (msgQueue.empty()) return null;
      Object result = msgQueue.front();
      msgQueue.dequeue();
      return result;
} }
```

Checking several channels:

```
while (msg == null)
{  if ((msg = ch1.receiveMsgOrNull()) == null)
   if ((msg = ch2.receiveMsgOrNull()) == null)
      Thread.sleep (500);
}
```

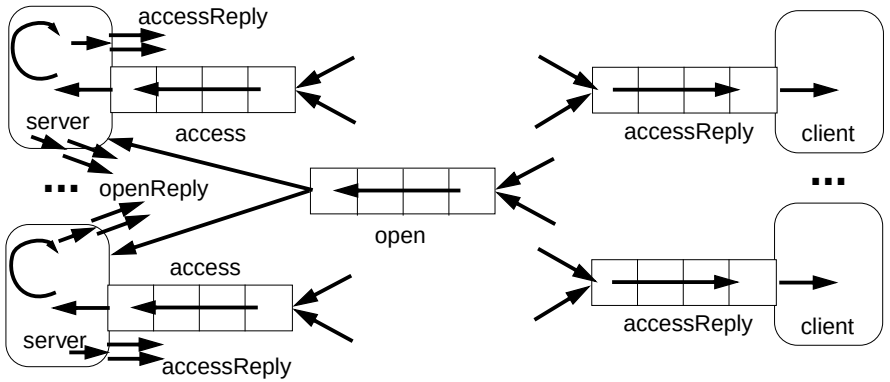




# Conversation sequences between client and server

Example for an **application pattern** is „file servers“:

- **several equivalent servers** respond to requests of **several clients**
- a client sends an **opening request** on a **channel common** for all servers (**open**)
- one server commits to the task; it then leads a conversation with the client according to a **specific protocol**, e. g.
  (**open openReply) ((read readReply) | (write writeReply))* (close closeReply)**
- **reply channels** are contained in the **open** and **openReply** messages.








# Active monitor (server) vs. passive monitor

| **active monitor** | | **passive monitor** |
|---|---|---|
| | **1. program structure** | |
| active process | | passive program module |
| | **2. client communication** | |
| request - reply via channels | | calls of entry procedures |
| | **3. server operations** | |
| kinds of messages and/or different channels | | entry procedures |
| | **4. mutual exclusion** | |
| requests are handled sequentially | | guaranteed for entry procedure calls |
| | **5. delayed service** | |
| queue of pending requests replies are delayed | | client processes are blocked condition variables, wait - signal |
| | **6. multiple servers** | |
| may cooperate on the same request channels | | multiple monitors are not related |