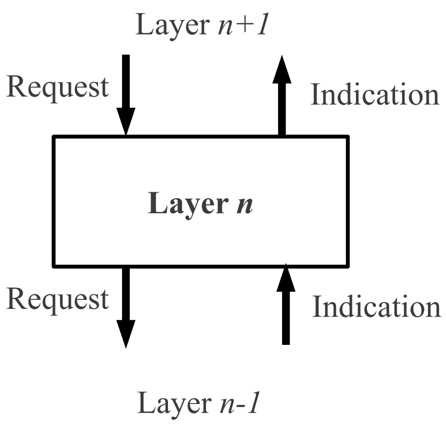
Module specification and implementation



Drawing 1: The Figure shows a layer in protocol stack. Layer n is below Layer n+1 and above Layer n-1. Layer n+1 invokes services at Layer n, by sending request events. Layer n replies to Layer n by sending back indication events.

<u>**Module 1:** Interface and properties of protocol module "layer n"</u>

Module:

  Name: LayerN, **instance** *ln.*

Events:

  Request: < *ln*, *EventName1* | *parameter1*, *parameter2* > : Sends request event 1.

  Indication: < *ln*, *EventName2* | *parameter1*, *parameter2* > : Sends indication event 2.

Properties:

  LN1: *Always correct:* Everything this module does is correct.

  LN2: *Always replies*: Every request will eventually lead to an indication being delivered.

<u>**Algorithm 1:** Implementation of module 1</u>

```
Implements:
        LayerN, instance ln.
Uses:
        LayerNMinus1, instance ln1.                   //we need to know the layer below

upon event < ln, Init > do                            //initialize internal state
        state := IDLE;

upon event < ln, EventName1 | param1, param2 > do     //event received from upper level
        state := WORKING;
        trigger < ln1, AnotherEventName | param1 >;   //send a request to lower level

upon event < ln1, AnIndicationEvent | param > do      //event received from lower level
        state := IDLE;
        trigger < ln, EventName2 | param, param + 7 >; //send indication to upper level
```

Types of events

```
**upon event** < *co1* , *Event1* | *param1*, *param2*, … > **do**
        do something;
```

Each event is processed through a dedicated handler by the process (i.e., by the corresponding component). A handler is formulated in terms of a sequence of instructions introduced by upon event, which describes the event, followed by pseudo code with instructions to be executed. The processing of an event may result in new events being created and triggering the same or different components. Every event triggered by a component of the same process is eventually processed, if the process is correct (unless the destination module explicitly filters the event; see the such that clause ahead). Events from the same component are processed in the order in which they were triggered. This first-in-first-out (FIFO) order is only enforced on events exchanged among local components in a given stack. The messages among different processes may also need to be ordered according to some criteria, using mechanisms orthogonal to this one.

We assume that every process executes the code triggered by events in a mutually exclusive way. This means that the same process does not handle two events concurrently. Once the handling of an event is terminated, the process keeps on checking if any other event is triggered. This periodic checking is assumed to be fair, and is achieved in an implicit way: it is not visible in the pseudo code we describe.

```
upon condition do                                    //an internal event
        do something;
```

For writing complex algorithms, we sometimes use handlers that are triggered when some condition in the implementation **becomes** true, but do not respond to an external event originating from another module. The condition for an internal event is usually defined on local variables maintained by the algorithm.

```
upon event < co1 , Event1 | param1, param2, … > such that condition do
        do something;
```

An upon event statement triggered by an event from another module can also be qualified with a condition on local variables. This handler executes its instructions only when the external event has been triggered and the condition holds.

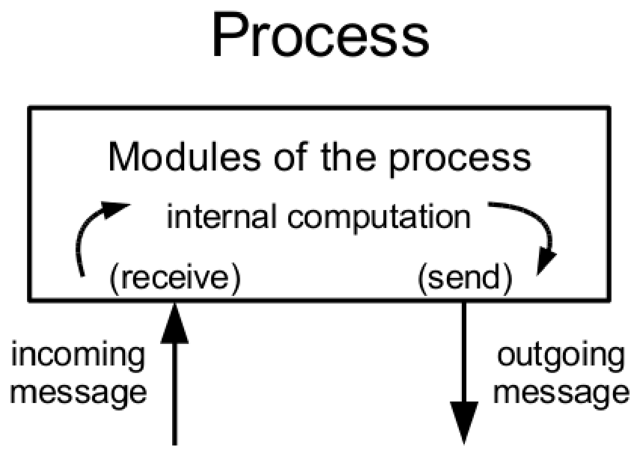
Processes and Messages

We abstract the units that are able to perform computations in a distributed system through the notion of a *process*. We consider that the system is composed of *N* different processes, named *p, q, r, s*, and so on. The set of processes in the system is denoted by Π. Unless stated otherwise, this set is static and does not change, and every process knows the identities of all processes. Sometimes a function rank : Π → {1, . . . , N } is used to associate every process with a unique index between 1 and N . In the description of an algorithm, the special process name *self* denotes the name of the process that executes the code. Typically, we will assume that all processes of the system run the same local algorithm. The sum of these copies constitutes the actual distributed algorithm.

We do not assume any particular mapping of our abstract notion of process to the actual processors or threads of a specific computer machine or operating system. **The processes communicate by exchanging messages** and the messages are uniquely identified, say, by their original sender process using a sequence number or a local clock, together with the process identifier. In other words, we assume that all messages that are ever exchanged by some distributed algorithm are unique. Messages are exchanged by the processes through communication links. We will capture the properties of the links that connect the processes through specific link abstractions, which we will discuss in the class.

The message passing assumption entails that **there is no notion of shared memory among processes**. Consequently, the modules you will use for implementing a part of the algorithm may perform one of the following actions:

1. *local computation*: where they can use local variables, and parameters received through events or messages.
2. *events*: a module can send and receive events to/from another module within the same process.
3. *messages*: a process can send a message to another process. This is typically abstracted by the *PerfectPointToPointLinks (pl)* module. This module resides in the lowest layer of a process. A process sends a message to another process by issuing an *event* to the *PerfectPointToPointLinks* module (e.g., *<pl, Send | TargetProcess, message>*).



Drawing 2: Step of a process. The process receives an incoming message, does local processing, and sends a response message.