(Lua) Pi-threads

Concurrent Programming made Simple

(C) 2007-2008, Frédéric Peschanski
UPMC Paris Universitas – LIP6
first.last@lip6.fr

Introduction

The Pi-threads is a concise and efficient framework for concurrent programming. The version described in this tutorial is implemented as a library for the Lua programming language. The metaphor employed is that of communicating systems related to the pi-calculus. Pi-threads communicate through send/receive or broadcast primitives using communication channels. A higher-level and expressive choice construct allows to implement guarded commands, select-like behaviors and much more. The language also support an extended form of join patterns.

From a practical point of view, the Pi-threads greatly simplifies the thinking and programming of concurrent behaviors. For Lua programmers, the Pi-threads gives an abstraction layer above the coroutine mechanism so that programmers do not have to deal with the low-level aspects.

From a scientific point of view, the Pi-threads is part of a long-term research effort to design and develop integrated modeling, verification and programming techniques for day-to-day programmers of highly concurrent and distributed systems (see the Pi-graphs framework).

This document is a tutorial describing step-by-step examples involving most of the features of the Pi-threads framework.

Intended audience

This document assumes basic familiarity with the Lua programming language.

Outline

Installation

There is no installation procedure for the (Lua) Pi-threads (or LuaPi) framework per-se. The only "difficult" part is in expanding the archive, which like the following one any Linux/Unix/Mingw/Cygwin/MacOSX environment:

```
> tar xvzf LuaPithreads-<VERSION>.tar.gz
...
```

To check the library, the `LUA_PATH` environment variable should be set to the `lib/` directory of LuaPithreads and that's about all. You can try the (few) provided examples:

```
> source ./prepare.sh
> lua examples/tutorial/cs.lua
...
> (etc.)
```

1. Warming up: Ping-Pong

The classical ping-pong example is often presented as the hello world variant of concurrent and distributed systems. It simply consists in two Pi-threads sending and receiving messages to each other. Let's write the code of this example in a file `pingpong1.lua` (already present in the `examples/tutorial/` directory of the distribution).

The first step when writing a Pi-threads program is to **`require`** the `"pithreads"` module:

```lua
-- file "pingpong1.lua"

-- importing the LuaPi module
require "pithreads"
```

The pi-threads run code chunks called *behaviors*, which in Lua are represented by Lua functions. We describe below the common *behavior* of the Ping and Pong Pi-threads as a Lua function, named `PingPong`, and implemented as follows:

```lua
-- the behavior of Ping and Pong
function PingPong(thread,inp,out,message)
  print(thread.name,"started")
  while true do
    local msg = thread:receive(inp)
    print(thread.name .. " receives '" .. msg .. "'")
    thread:send(out,message)
  end
end
```

The first parameter of a behavior function, generally noted `thread`, references the Pi-thread actually running the behavior. The other parameters may vary depending on the behaviors.

After displaying the `"... started"` message, a Pi-thread running the `PingPong` behavior will enter an infinite loop which consists firstly in receiving a message from an input channel named `inp`. The syntax:

```
val1, val2, ... = thread:receive(inp)
```

means: “the `thread` listens to channel `inp` and waits/blocks until the values `val1`, `val2`, ... can be received.”

This denotes a synchronization with a corresponding sender Pi-thread on the same channel. The received value becomes the value of the local variable `msg`. In the most general case, multiple values can be received at once.

The following syntaxes are equivalent:

```
-- (1) the communication metaphor
local val1,val2, ... = thread:receive(chan)

-- (2) the pure synchronization metaphor (no return value expected)
thread:wait(chan)

-- (3) the conversation metaphor (for broadcast)
local val1,val2, ... = thread:listen(chan)
```

These are all synonyms but in general we use the send/receive communication metaphor. However, when no value is to be sent nor received, a case named "pure synchronization", the signal/wait metaphor is preferred. Finally, the talk/listen conversation metaphor is used for broadcast interactions (see section 5). By default interactions are unicast (or point-to-point, 1-to-1 etc.), they involve only one sender and one receiver.

Let's go back to the `PingPong` behavior. After the reception, the content of the received message `msg` is displayed and another `message` is sent on an output channel named `out`.

The syntax:

```
thread:send(out,val1, val2, ...)
```

means: “the Pi-thread `thread` is willing emit the values `val1`, `val2`, ... on channel `out`.”

The emission of values is blocking until (at least) one receiver on the same channel synchronizes with the sender Pi-thread so that it can proceed further on.

The variant for pure synchronization (no emission of value) is as follows:

```
thread:signal(out)
```

As seen from the code example we use some specific color conventions in code excerpts:

- Pi-thread references are as in `thread`
- (Lua) Pi-threads specific functions and methods are denoted `receive`, `send`, etc.
- channel names are `inp` and `out`

- Lua keywords are e.g. `function`, `while`, etc.
- Functions and module names are e.g. `PingPong`
- Comments are written `-- like this`
- Litteral strings are `"like that"`

A second behavior we need for our example is a one-shot initialization behavior that will initiate the ping-pong interactions. This is needed since the PingPong behavior starts by waiting for something to be received on channel `inp`. This `Init` behavior is written as follows:

```
-- the initialization behavior
function Init(thread,chan)
  print(thread.name,"started")
  thread:send(chan,"<<INIT>>")
  print(thread.name,"sent is value")
end
```

This mostly consists in sending an initial message on some `chan` channel.

Now we must set up an architecture of three Pi-threads running in parallel, which in the LuaPi terminology corresponds to creating a LuaPi **agent**. The basic rule is that there is at most one agent for each LuaPi script. The distributed LuaSpaces extension describes the possible interactions among agents, but this is another story. So let's create the agent for the ping-pong script using the `init` function of the `pithreads` module:

```
-- the pingpong agent
agent = pithreads.init()
```

An `agent` object provides a certain number of methods, the most important ones being:

- the method `new` or `newchan` to create a channel
- the method `spawn` to create a Pi-thread
- the method `run` to start the agent

The agent we want to create in the ping-pong example is composed out of three Pi-threads interconnected by two channels, which may be pictured as in Figure 1.

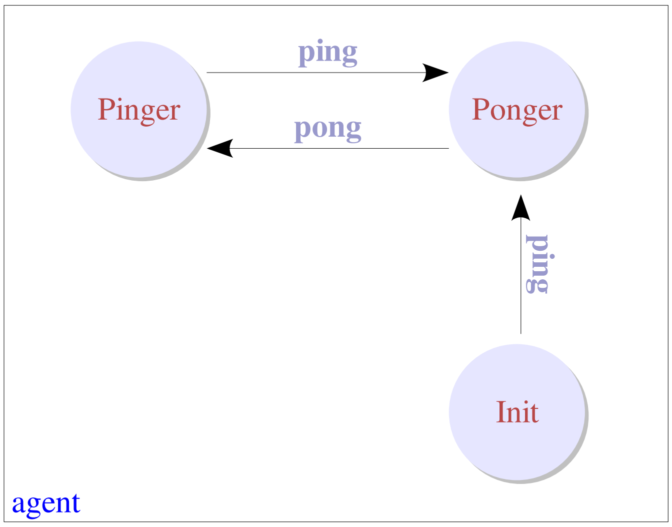

Figure 1 : the Ping-Pong architecture (agent)

The architecture of **Figure 1** can be easily composed using the agent methods:

```
-- first create the two channels
ping = agent:new("ping")
pong = agent:new("pong")

-- then create the ping and pong Pi-threads
pinger = agent:spawn("Pinger",PingPong,ping,pong,"<<PING>>")
ponger = agent:spawn("Ponger",PingPong,pong,ping,"<<PONG>>")

-- the initialization Pi-thread
agent:spawn("Init",Init,ping)
```

The syntax:

`chan = agent:new("name")`

means: “tell the `agent` to create a channel named `"name"` and store the channel reference into variable `chan`.”

One may also write equivalently:

`chan = agent:newchan("name")`

The syntax:

`thread = agent:spawn("name",Behavior, ... args ...)`

means: “tell the `agent` to spawn a new Pi-thread named `"name"` running behavior `Behavior`, passing `args` as arguments.”

Note that no Pi-thread runs just after spawning. The main agent loop must be entered so that the Pi-threads actually start, which simply consists in invoking the **run** method of the agent without argument:

```
-- and let's start everything
print("Starting the agent")
agent:run()
```

To run the program we simply use the Lua interpreter, the resulting output should look like the following:

```
> lua examples/tutorial/pingpong1.lua
Starting the agent
Start scheduling
Pinger  started
Ponger  started
init    started
Pinger receives '<<INIT>>'
Ponger receives '<<PING>>'
Pinger receives '<<PONG>>'
Ponger receives '<<PING>>'
Pinger receives '<<PONG>>'
Ponger receives '<<PING>>'
Pinger receives '<<PONG>>'
Ponger receives '<<PING>>'
Pinger receives '<<PONG>>'
Ponger receives '<<PING>>'
Pinger receives '<<PONG>>'
Ponger receives '<<PING>>'
... infinitely ...
```

You may type `Ctrl-C` to stop the program. Keep in mind that many concurrent programs keep the control like this (e.g. a web server).

Exercise :

Write a variant `pingpong2.lua` of the ping-pong example so that the execution terminates after some time. A suggested idea is to add a parameter `fuel` (an integer) to the `PingPong` behavior, and loop until all fuel has been consumed (i.e. the `fuel` value is `0`).

Interlude: Concurrent Programming Metaphors

There exist many ways to design and write concurrent, parallel and distributed programs. The two most common metaphors are:

1. the **shared memory metaphor** based on concurrent read/write on shared variables and locking mechanisms. This is the most common approach to concurrent programming, e.g. using Posix threads, Win32 or Java threads.

2. the **message-passing metaphor** based on communication primitives as exemplified by more or less famous languages, e.g. Erlang, CML, Occam, etc.

It is clear that LuaPi, despite the predominance of the shared memory model, is clearly a member of the message-passing family of concurrent frameworks. Even if programming style is before all a matter of taste (and in my own opinion message-passing tastes better), there are more objective reasons for this choice. First, the LuaPi project is part of a research effort consisting in putting state of the art theoretical results into practice. Our works related to LuaPi include the interaction spaces, the pi-graphs and the cube-vm.

There are also practical reasons why the Pi-calculus metaphor was chosen. The first of these is the simplicity and minimalism of the proposed constructs and principles. This tutorial emphasizes on illustrating the expressiveness of the few Pi-calculus constructs and the extensions we propose. We show in particular that they are quite powerful to design higher-level concurrency patterns through composition. This makes a big difference if compared to shared-memory constructs that are a lot more difficult to compose. Another focus of this tutorial is the illustration of the smooth integration with the standard Lua programming model.

If compared to actor-based communication models, for example used in the Erlang programming language, the Pi-calculus frameworks employ a model of **explicit synchronous channels**. The advantage of using channels as intermediate is that Pi-threads do not need to know about each other, which provides a lot of flexibility, in particular for systems with dynamic behaviors. The synchronous semantics are in our opinion easier to control (and debug) than their asynchronous variants.

2. The classics (1) : Mutual exclusion

The second example we will design is the classic of all classics: the **mutual exclusion problem**. The first of part of the script (in `examples/tutorial/cs.lua`) is as follows:

```lua
-- import the LuaPi module
require "pithreads"

-- the number of Pi-threads with Critical Sections
N = tonumber(arg and arg[1]) or 1000
-- a counter for running critical sections
CS_COUNT = 0
```

The parameter `N` gives the number of mutually exclusive Pi-threads that will we spawned by the main agent. The `CS_COUNT` global variable counts the number of Pi-threads that actually entered their critical section at a given time. This counter should be at most 1 to ensure mutual exclusion, i.e. the fact that no more than 1 Pi-thread is in the state of executing a protected block of code called its **critical section**.

Each of the mutually exclusive Pi-threads runs the same `CS` behavior defined as follows:

```lua
-- the common behavior for all Critical Section Pi-threads
function CS(thread,n,lock,csfun)
  local enter,leave = thread:receive(lock)
  -- START of Critical Section
  CS_COUNT = CS_COUNT + 1
  thread:send(enter,n)
  -- the critical section content
  csfun(thread)
  thread:send(leave,n)
  -- END of Critical Section
  CS_COUNT = CS_COUNT - 1
  thread:send(lock,enter,leave)
end
```

The lock channel, as its name suggests, is used to ensure the mutual exclusion condition of the code located between the --START and --END comments in the CS behavior. The first thing any CS Pi-thread does is thus to block (i.e. acquire the lock) until someone synchronizes (i.e. releases the lock). We demonstrate here that the lock channel is not used only as a pure synchronization means (we would use a signal/wait pair in this case), but also to transmit some values. Here, the two received values are respectively stored in the variables enter and leave, that are indeed channel references. The channel enter is used to tell an observer Pi-thread (see below) that we entered the critical section. Symmetrically, the leave channel is used to tell the observer that the critical section is terminated. The generic csfun function represents the body of the critical section, which is passed as argument. This shows that our protocol is generic and can be specialized by passing different csfun functions. Finally the CS Pi-thread releases the lock by transmitting again the enter and leave references along channel lock. This ability to communicate channel references, the most distinctive trait of the underlying theory of the pi-calculus, is sometimes called *name passing* or *channel passing*, or even sometimes *mobility* though the later is rather misleading. In the case of LuaPi, this feature is probably best designated as **channel reference passing**.

Next is the behavior for an Observer Pi-thread that is used to test the behaviors of the CS Pi-threads, and in particular ensure that the mutual exclusion property holds:

```
-- A behavior to observe the Critical sections
function Observer(thread,enter,leave)
  while true do
    local n = thread:receive(enter)
    print("Lock taken by "..tostring(n))
    -- check the mutual exclusion property
    assert(CS_COUNT==1)
    local m = thread:receive(leave)
    print("Lock released by "..tostring(m))
    -- check the take/release pair
    assert(n==m)
  end
end
```

This behavior follows a **reactive** pattern. A reactive behavior blocks until the occurrence of a given event. Here the event corresponds to the reception of an integer value on a channel enter. When triggered, the even leads to the execution of some short-term computations called a *reaction*. The reaction here consists in displaying some informations on the console, and most importantly to check our mutual exclusion property. Since at this point at most one CS Pi-thread should have entered its critical section, the value of CS_COUNT should exactly be 1. A purely reactive Pi-thread would then go back to its blocking mode waiting for the next occurrence of an event. Here we wait for a reception on the leave channel, which allows to check the further property that the Pi-thread who entered its critical section is indeed the one leaving the same critical section now.

Such a behavior that does not participate directly to the protocol, but that can show informations and test properties, denotes the notion of an **observer**. This is the principal means of testing concurrent systems.

We need to be able to spawn an arbitrary number of CS Pi-threads. We can use a standard Lua loop to perform so but for illustrative purpose we will rely here on a Pi-thread spawning other Pi-threads, as follows:

```
-- A Pi-thread behavior that spawns other Pi-threads
function Launch(thread,n,lock,start,csfun)
  while n>0 do
    print("Start Pi-thread : "..tostring(n))
    thread:spawn("CS"..tostring(n),CS,n,lock,csfun)
    n = n - 1
  end
  print("Go !")
  thread:signal(start)
end
```

The spawn method, originally an agent method, is thus also available at the Pi-thread level. In fact we could equivalently write `thread.agent:spawn(...)` for the same purpose. Of course, the same holds for all agent methods. After having spawned all the `CS` Pi-threads, the `Launch` Pi-thread *signals* this fact on channel `start`. Remember that `signal` is a synonym for `send` but generally used only for pure synchronization purposes. The pending synonym for *waiting* a signal is of course `wait`, a synonym for `receive`. This is used in the following behavior:

```
 -- The synchronization barrier
function Barrier(thread,start,lock,enter,leave)
  thread:wait(start)
  thread:send(lock,enter,leave)
end
```

A **synchronization barrier**, such as this, makes an arbitrary number of Pi-threads wait altogether until some event occur, and then allow the them to proceed. Here, the event that triggers the barrier is a signal on channel `start`. This occurs when all the `CS` Pi-threads have been spawned, so that we are sure all the Pi-threads are running before we start observing our protocol.

Finally, we set up the initial configuration of the system, as follows:

```
-- create the agent
agent = pithreads.init()

-- create the channels
lock = agent:new("lock")
enter = agent:new("enter")
leave = agent:new("leave")
start = agent:new("start")
```

```
-- spawn the Pi-threads
agent:spawn("Launch",Launch,N,lock,start,
  -- the critical section implementation
  function (thread)
    print("Pi-thread "..thread.name..": critical section")
  end)

agent:spawn("Observer",Observer,enter,leave)
agent:spawn("Barrier",Barrier,start,lock,enter,leave)

print("Starting the agent")
agent:run()
```

This initial configuration can be pictured as in **Figure 2**.

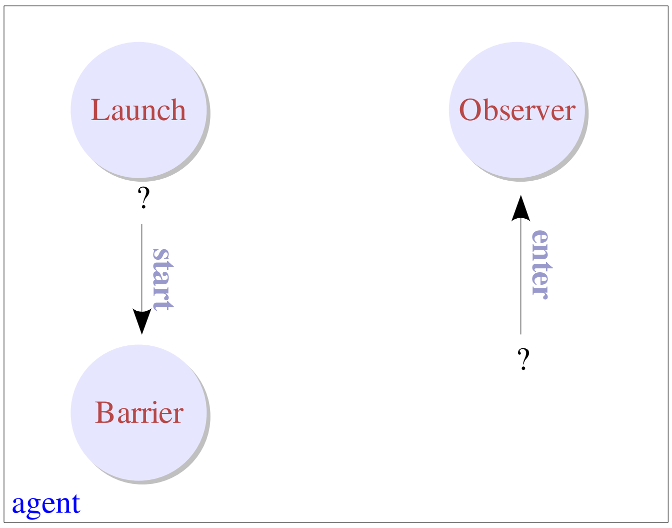

Figure 2 : initial configuration for mutual exclusion

We use question marks to show the end(s) of a channel that are not (yet) enabled. For example, the `Observer` does not know who will enter the critical section, and the `Launch` Pi-thread is not yet ready to signal on `start`. Let us start the program with only 5 `CS` Pi-threads so that we can analyze the traces and show the evolving architecture:

```
> lua examples/tutorial/cs.lua 5
Starting the agent
Start scheduling
Start Pi-thread : 5
Start Pi-thread : 4
Start Pi-thread : 3
Start Pi-thread : 2
Start Pi-thread : 1
Go !
```

At this point the five `CS` Pi-threads have been started (note that the order of launch is arbitrary, but generally quite deterministic) and the `Launch` Pi-thread is ready to signal this to the synchronization barrier.

The configuration at this point is the following:

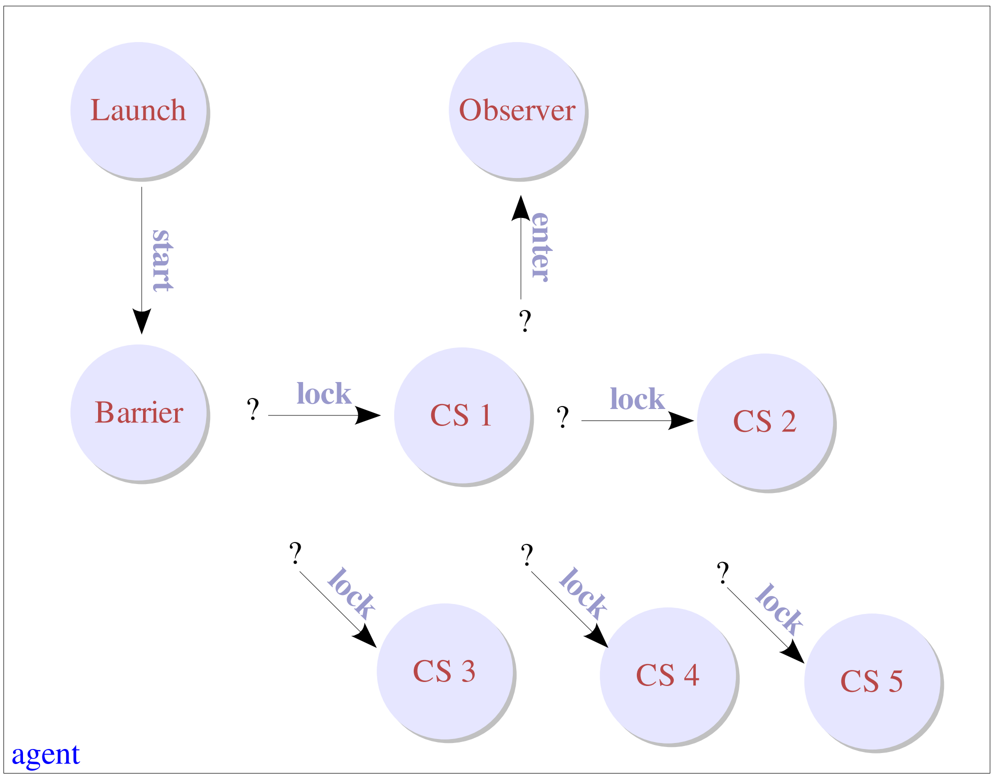

Figure 3 : synchronization barrier

In the next step the synchronization barrier is ready to activate one of the `CS` Pi-thread by synchronizing on the shared channel `lock`. The execution thus evolves to the following configuration:

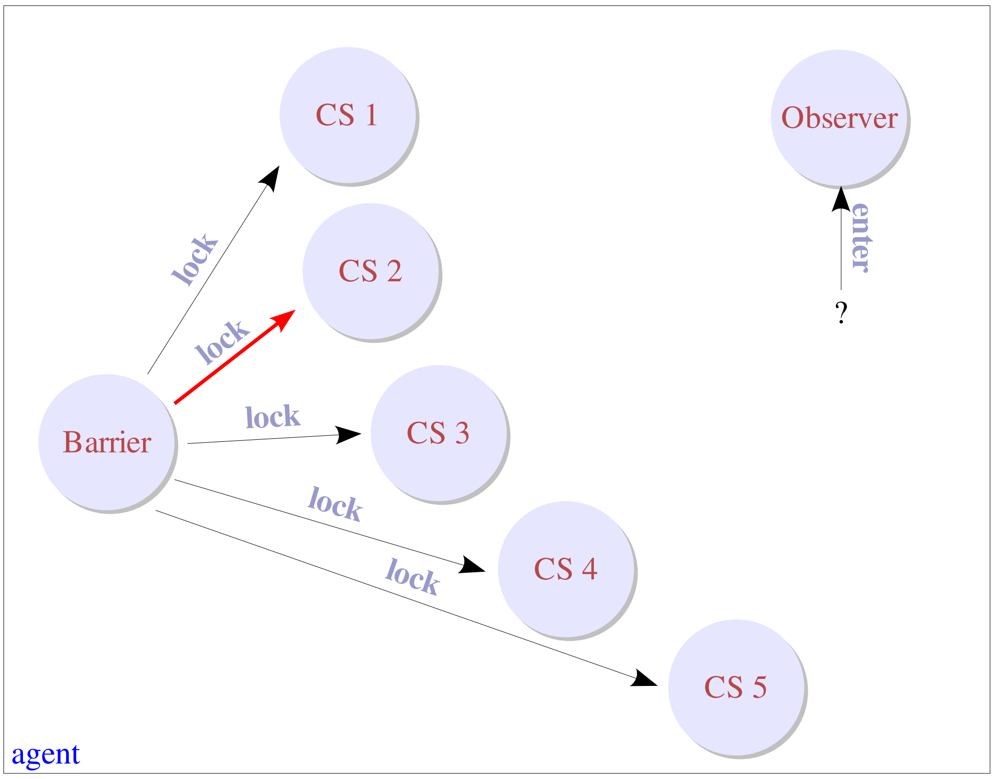

Figure 4 : before entering a critical section

At this point LuaPi will choose an arbitrary `CS` Pi-thread to synchronize with the `Barrier`. We suppose LuaPi selects the synchronization with the `CS 2` Pi-thread, as the red arrow on **Figure 4** shows.

In the next step, the selected Pi-thread will thus interact with the Observer, in the following configuration:

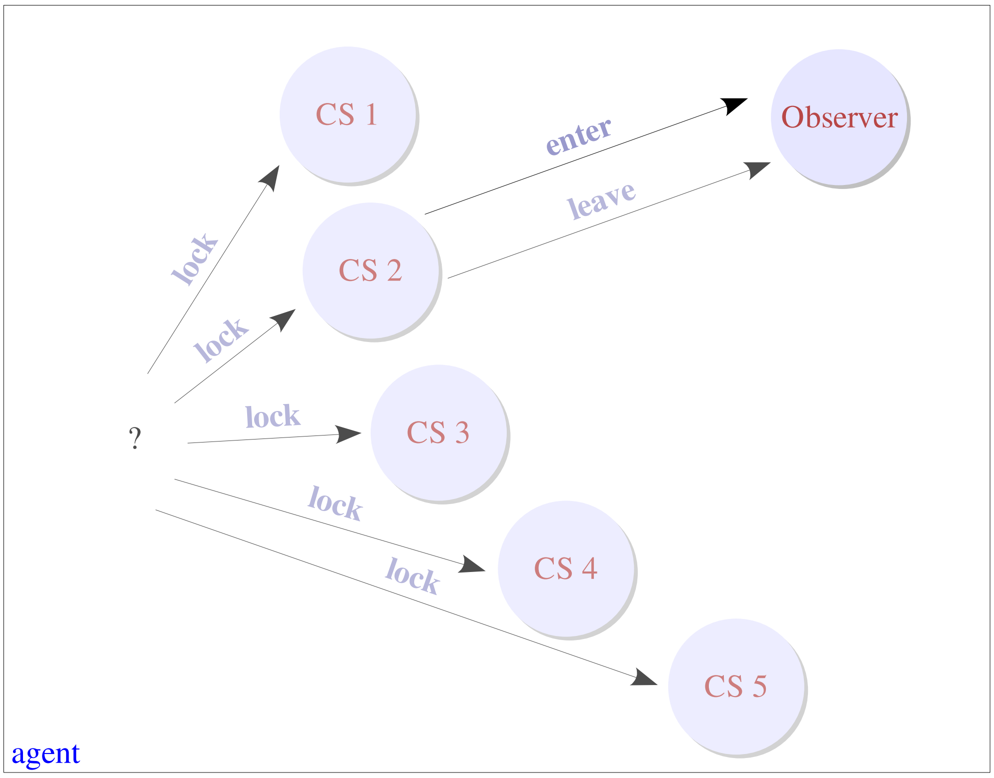

Figure 5 : interactions with the Observer

The Observer will display the following lines on the console:

```
Lock taken by 2
Pi-thread CS2: critical section
Lock released by 2
```

In the next step, the `CS 2` Pi-thread will release the `lock` and synchronize with an arbitrary Pi-thread, for example `CS 4`, which will lead to the following configuration:

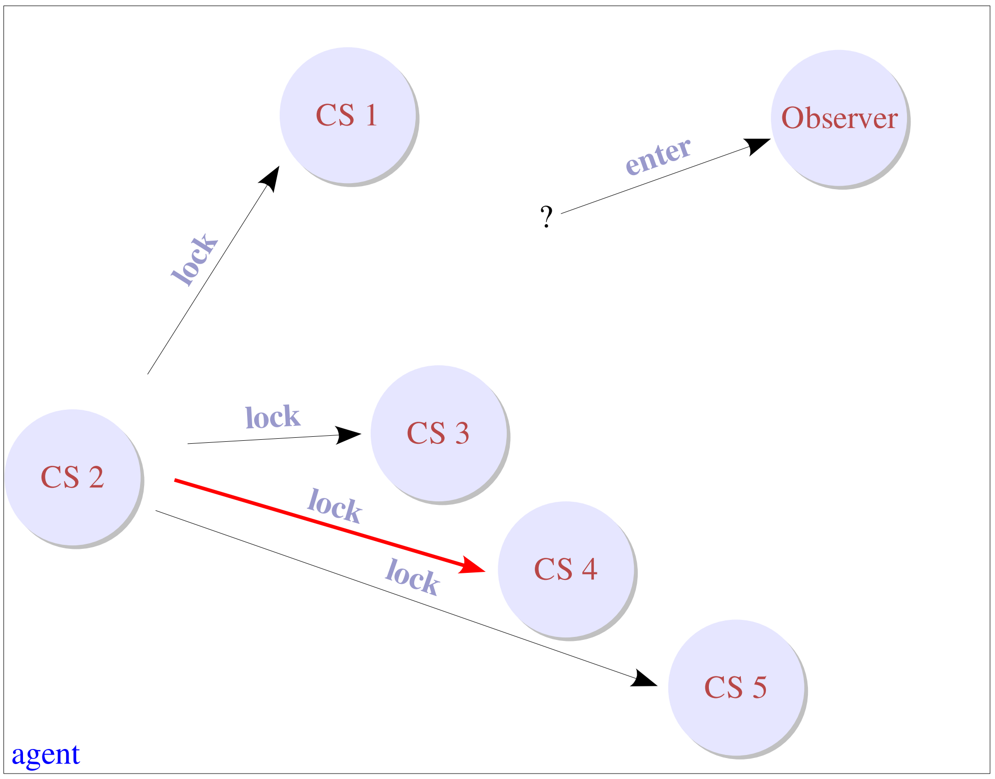

Figure 6 : synchronization between critical sections

And the procedure goes on until all `CS` Pi-thread executes their critical section... with for example the following observations:

```
Lock taken by 4
Pi-thread CS4: critical section
Lock released by 4
Lock taken by 1
Pi-thread CS1: critical section
Lock released by 1
Lock taken by 5
Pi-thread CS5: critical section
Lock released by 5
Lock taken by 3
Pi-thread CS3: critical section
Lock released by 3
All Pi-threads terminated
End of scheduling
```

In conclusion, what do we exactly ensure with this protocol ? The answer if that if the `lock` channel is only known to the participating `CS` Pi-threads, then the critical sections are guaranteed to execute in a mutually exclusive way.

3. Asynchrony: an example of choice

LuaPi Pi-threads uses *synchronous* channels to communicate with each other. This is the most effective way of communicating as long as the Pi-threads share their memory. In a distributed systems, communication channels are most often *asynchronous*: an emitter does not have to (or simply cannot) synchronize with a receiver to proceed. To simulate such an asynchronous behavior we have to use an intermediate queue to separate the emitters and receivers. This is useful, for example, to implement a concurrent queue of GUI events.

We describe below a classical example: concurrent writers and readers on a shared queue buffer. The program is named `async.lua` and first defines a few constants as follows:

```
-- import the LuaPi module
require "pithreads"

-- the number of writer Pi-threads
NWRITERS = tonumber(arg and arg[1]) or 100
-- the number of writes per writer
NWRITES = tonumber(arg and arg[2]) or 10
-- the number of writer Pi-threads
NREADERS = tonumber(arg and arg[3]) or 50
-- the capacity of the queue
CAPACITY = tonumber(arg and arg[4]) or 25
```

We next describe a `MsgQueue` behavior that implements our asynchronous queue:

```
-- a behavior for a message Queue behavior
function MsgQueue(thread,put,take,capacity)
  local queue = {}
  while true do
    print("Queue size: ",#queue)
    -- empty queue (can only put)
    if next(queue)==nil then
      print("Queue is empty")
      local msg = thread:receive(put)
      table.insert(queue,1,msg)
    -- full queue (can only take)
    elseif #queue==capacity then
      print("Queue is full")
      local msg = table.remove(queue)
      thread:send(take,msg)
    else -- other cases (can put or take)
      local msg = queue[#queue]
      thread:choice(
        { put:receive(),  function(msg)
                           table.insert(queue,1,msg)
                          end },
        { take:send(msg), function()
                           table.remove(queue)
                          end })()
    end
  end
end
```

This is a slightly more involved behavior. A message queue is a Pi-thread governed by two channels whose purpose is respectively to put or to take a message in the queue. A writer Pi-thread must send on the put channel to emit an asynchronous message. A receiver, on the other hand, must receive on the take channel. We implement a First-In First-Out (FIFO) behavior but other ordering properties might be proposed as well (e.g. using **math.random** to simulate unordered unreliable communication channels). The queue itself is implemented as a Lua table (used as an array) stored in the local variable queue. We use a CAPACITY constant to limit the number of messages that can be stored in the queue. Technically-speaking, this makes the system more *pseudo-asynchronous* (asynchronous when the queue is not full, but temporarily synchronous when the capacity is reached) than truly asynchronous. A truly asynchronous system is somewhat a vision of the mind since an infinite amount of memory would be needed to implement it in practice.

In the main (infinite) loop of the behavior we have three cases to consider:

(1) the queue is empty: the only available operation is to put a message in the queue,
(2) the queue is full: the only available operation is to take a message from the queue,
(3) other cases: we have the **choice** between the put and take operations.

The first two cases are easy to implement from what we already know (and a basic knowledge of the table/array manipulation functions of plain Lua). The real novelty is in the implementation of the notion of **choice**. In the underlying theory of the Pi-calculus, the expression P + Q with P and Q Pi-thread expressions is not an arithmetic operation, it is the choice between either executing P or executing Q. As always with theoretical work, a notion with trivial semantics at the most abstract level can become quite complex at the implementation level. The LuaPi realization of this choice operator takes the form of a functional construct whose general syntax is as follows:

The syntax:
```
choice = thread:choice({guard1, react1},
                       {guard2, react2}
                       -- ... etc ...
                       {guardN, reactN})
```
means: "prepares a choice for the Pi-thread thread, with the reaction react1 guarded by guard1, the reaction react2 guarded by guard2, ..., and finally the reaction reactN guarded by guardN"

This may seem a little bit obscure so let us detail this construct. The reaction react1 explains the code to execute only if the guard1 is enabled. Similarly, the reaction react2 explains the code to execute only if the guard2 is enabled, and so on. There are three basic kinds of guards:

(1) chan:**receive**() : an input guard enabled if a sender on chan is available,
(2) chan:**send**(val) : an output guard enabled if a receiver on chan is available,
(3) fun() : a functional guard, an argument-less function returning a **boolean**, and enabled only if **true** is returned.

The reactions are functions with either no argument (for output and functional guards) or the received messages as arguments.

A choice, as returned by the choice method, is just a passive functional structure. To activate the choice, the only thing to do is to call it as a function without any argument, i.e. type choice() in Lua. The behavior of the choice is then as follows:

- evaluating guard1, and if it is enabled, executing the reaction react1, or
- evaluating guard2, and if it is enabled, executing the reaction react2, or
- ... etc ...
- evaluating guardN, and if it is enabled, executing the reaction reactN, or
- if no evaluated guard is enabled, then block until one of them becomes enabled.

Remark: When a reaction reactK has been executed, the choice returns with the integer K. In general, we do not have to use this return value.

We can now go back to our MsgQueue behavior and finally understand the case when the queue is neither empty nor full, as reproduced below :

```
    else -- other cases (can put or take)
      local msg = queue[#queue]
      thread:choice(
        { put:receive(),  function(msg)
                           table.insert(queue,1,msg)
                          end },

        { take:send(msg), function()
                           table.remove(queue)
                          end })()
```

The first step is to fetch (without removing it) the oldest element that has been put in the queue (we insert at the front of the queue, and remove at the end so the oldest element is the last element of the table). Then, we prepare (and immediately enact, remark the parenthesis at the end of the extract) a choice between our two possibilities: either receive a new message in the queue (using an input guard on channel put) or send the oldest message (using an output guard on channel take). Note that using the default strategy, the receptions are somewhat privileged over emissions since we first try to enable the input guard, and only try the output guard if no reception is possible.

The behaviors of the writer and reader Pi-threads are as follows:

```
-- A behavior for writer Pi-threads
function Writer(thread,id,put,msg,count)
  local first = count
  while count>0 do
    print("Writer #",id," sends: ",msg..tostring(first+1-count))
    thread:send(put,msg..tostring(first+1-count))
    count = count - 1
  end
end
```

```lua
-- A behavior for reader Pi-threads
function Reader(thread,id,take)
  while true do
    local msg = thread:receive(take)
    print("Reader #",id," receives: ",msg)
  end
end
```

Finally we set up the architecture of the concurrent system:

```lua
-- create the agent
agent = pithreads.init()

-- create the channels
put = agent:new("put")
take = agent:new("take")

-- spawn the Pi-threads
agent:spawn("MsgQueue",MsgQueue,put,take,CAPACITY)

agent:replicate("Writer",NWRITERS,Writer,put,"<BEEP>",NWRITES)
agent:replicate("Reader",NREADERS,Reader,take)

print("Starting the agent")
agent:run()
```

We notice that unlike the previous examples, we do not introduce a dedicated Pi-thread behavior to launch the readers and writers. Here we rely on the `replicate` agent method:

The syntax:

`agent:replicate("Prefix",NB,Behavior,arg, ...)`

means: “start `NB` Pi-threads of respective name `"Prefix1"`, `"Prefix2"`..., `"PrefixNB"`. Each Pi-thread runs the same `Behavior`. The first two arguments of this `Behavior` function are the created Pi-thread `thread` followed by the index of the Pi-thread in the replication, respectively `1`, `2`, ..., `NB`. The other arguments are `arg`, ...”

In our example we use this `replicate` method to start all the writers and readers.

Note that there is no magic behind the replicate method, its actual code is quite simple (largely thanks to the host Lua language):

```lua
function replicate(agent,threadname,n,threadfun,...)
  for i=1,n do
    agent:spawn(threadname..tostring(i),threadfun,i,...)
  end
end
```

4. The classics (2): join the philosophers

Perhaps the most famous testbed for concurrent languages is the dining philosophers invented by Edsger W. Dijsktra. This is not a directly applicable example but it is a small problem that exposes clearly the tension between *safety* and *liveness* issues in concurrent systems. A system is safe if it performs its task correctly, which is often explained as "nothing bad happens". A system is live if complementarity the "good things eventually happen". To understand the tension between the two notions in concurrency, we have to understand what is needed to ensure safety on the one side, and liveness on the other side. To ensure safety, one has to restrict the possible executions, for example by making some Pi-thread wait or similar things. If we restrict too much the possibility of execution, we might end up with a system that does nothing bad, but nothing good either, and thus fails to ensure liveness.

The notion of *deadlock* and *livelock* correspond to the worst case in term of safety and liveness. A deadlock occurs when a set of Pi-threads cannot advance and perform their computations. It can be seen as a liveness issue (a deadlocked system is not live for sure), but it is simpler to explain that a deadlock is something bad happening and, indeed, safety measures can be taken to detect and solve deadlock problems. On the contrary, a livelock occurs when the system does not perform its intended meaning, but it is not blocked in any way. This is in general more difficult to detect and solve. The notion of fairness, also, is related to liveness: a fair system ensure that a Pi-thread capable of doing some work will, indeed, perform this work at some point.

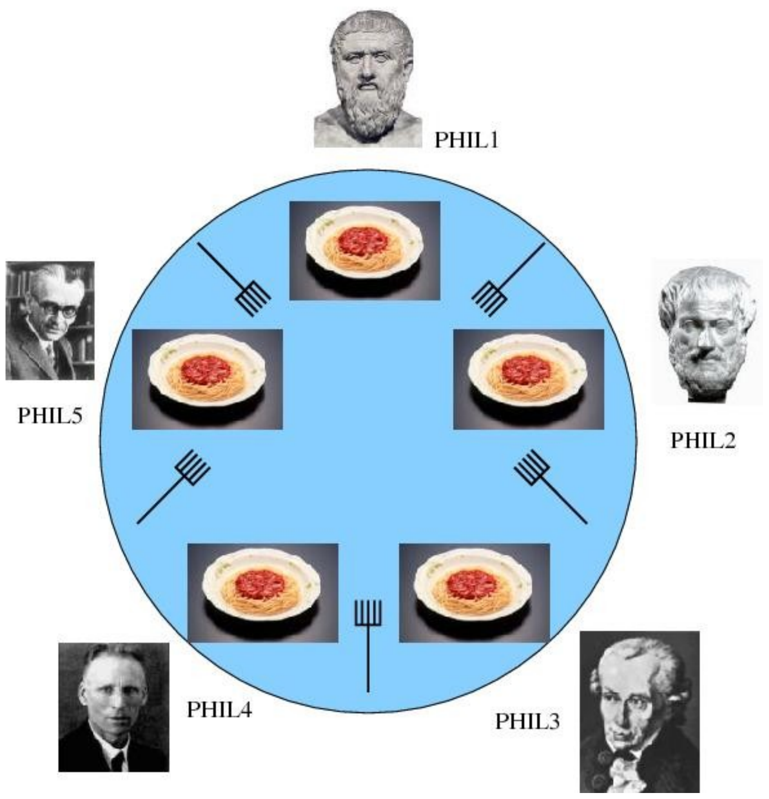

Figure 6 : The dining philosphers

The dining philosopher example, despite its simplicity, is useful to discuss the issues of safety, liveness and fairness. The problem is pictured on **Figure 6**. The activity of a philosopher can be decomposed in two distinguished phases: (1) eating and (2) thinking. Suppose there are NBPHILOS philosophers (NBPHILOS=5 on the figure). The dining table is set with NBRES forks (or chopsticks in some variants), NBRES plates (with noodles) and NBRES chairs (NBRES=5 on the figure). A constraint is that a philosopher needs two forks in order to eat (well, that probably motivates the infamous chopstick variant).

We provide three implementations of the examples in the files: dining1.lua, dining2.lua and dining3.lua. We first describe the code shared by all the versions, starting with the header:

```lua
require "pithreads"
-- the number of philosophers
NBPHILOS = tonumber(arg and arg[1]) or 5
-- the number of resources (forks,plates,chairs)
NBRES = tonumber(arg and arg[2]) or 5
-- then quantity of noodle in each plate (nil cancels termination)
QUANTITY = tonumber(arg and arg[3]) or 10

-- global statistics
STATS = {}
```

The constants allow to configure the number of philosophers NBPHILOS, the number of available resources NBRES (plates, forks and chairs) as well as the QUANTITY of noodle in each plate. The interesting instances are when the number of philosophers is at least as big as the number of resources. The default values demonstrates the original example. Note that our variant of the dining philosophers also deal with the correct termination of the protocol. If QUANTITY is initialized with the **nil** value, then the program does not terminate. Explicit termination makes the protocol a little more complex to implement, but it will allow us to make some statistics that will prove quite useful when dealing with fairness issues. The statistics are collected in the global variable STATS.

The behavior for the fork Pi-threads is as follows:

```lua
-- the behavior of forks
function Fork(thread,take)
  while true do
    thread:signal(take)
    thread:wait(take)
  end
end
```

Each fork Pi-thread possesses a dedicated take channel. A philosopher takes a fork simply by waiting on take, and it is released by sending a signal on the same channel. Each plate is also a running Pi-thread written as follows :

```lua
-- the behavior of plates
function Plate(thread,eat,quantity)
  while quantity==nil or quantity>0 do
    thread:send(eat,1) -- one noodle
    if quantity~=nil then
      quantity = quantity - 1
    end
  end
  thread:send(eat,0)
end
```

The eat channel drives the behavior of the plate Pi-threads, and one of the main safety issue is to ensure that there is no race condition on the eat channel of each plate, which relates to mutual exclusion. To trigger termination, an amount of 0 is sent to the eating philosopher if the quantity of the plate is exhausted. The plate Pi-thread will then terminates but there is more to do to ensure a *safe global termination* when the total quantity has been consumed.

It is not required though useful to introduce dedicated Pi-threads to manage the combination of forks and plates (each plate is surrounded by two designated forks). And indeed, a philosopher should first seat before taking its forks and eat (a philosopher would undoubtedly *not* stand up while eating). The behavior of chairs is as follows:

```
-- the behavior of chairs
function Chair(thread,seat,fork1,fork2,plate)
  local leave = thread:new("leave:"..thread.name)
  local ok = true
  while ok do
    thread:send(seat,fork1,fork2,plate,leave)
    ok = thread:receive(leave)
  end
end
```

A chair manages a seat channel (that a philosopher use to seat on the chair), two fork channels fork1 and fork2 as well as a plate channel. Upon seating, a philosopher will receive through seat all the channels required to enjoy his dinner (note the heavy use of channel passing in this example). A special secret channel leave is also created and sent to the philosopher so that it can leave the table in a mutually exclusive way. When leaving, a philosopher will send a boolean ok to propagates the termination event, if any. If ok is **false** then the plate in front of the chair is empty, which means there is no point in using the chair anymore.

Now comes the heart of the protocol, embodied by the common behavior of philosophers. Of course this behavior is somewhat more involved. The beginning is the following:

```
-- the bevahior of philosophers
function Philo(thread,seat,activate)
  while true do
    print(thread.name .. " thinks")
    local fork1,fork2,eat,leave = thread:receive(seat)
    print(thread.name .. " seats")
    -- next step: take the forks
```

A philosopher first, and before all, does is main routine: *think* ! Buf after some thinking, of course, a philosopher has to eat. To do so, a chair must be acquired which is done by receiving the "dinner package" through the seat channel.

In the next step, the philosopher must in turn:

1. acquire two forks by waiting on fork1 and fork2
2. eat by receiving on eat (and record some statistics)
3. release the forks
4. leave the table

Our first implementation of this protocol (in file dining1.lua) is as follows:

```
    -- philosopher behavior (cont'd) : first try
    thread:wait(fork1)
    thread:wait(fork2)
    local nb = thread:receive(eat)
    STATS[thread.name] = STATS[thread.name] + nb
    print(thread.name .. " eats " .. tostring(nb) .. " noodle(s)")
    thread:signal(fork2)
    thread:signal(fork1)
    print(thread.name .. " releases the forks")
    if nb==0 then
      thread:send(leave,false)
      print(thread.name .. " .. Nothing to eat ?
                          leaves the table (unhappily)")
    else
      thread:send(leave,true)
      print(thread.name .. " leaves the table")
    end
  end
end
```

A subtle treat for safety (i.e. a bug) is hidden in the code above. To discover this bug let us anticipate a little bit and show a run of the system using this behavior for philosophers:

```
> lua examples/tutorial/dining1.lua
Starting the agent
Philo5 thinks
Philo4 thinks
Philo3 thinks
Philo2 thinks
Philo1 thinks
Philo5 seats
Philo5 left fork taken
Philo4 seats
Philo4 left fork taken
Philo2 seats
Philo2 left fork taken
Philo3 seats
Philo3 left fork taken
Philo1 seats
Philo1 left fork taken
====
All Pi-theads terminated
Ending the agent
Consumption statistics:
Total consumption = 0
```

So the program ends up with a total consumption of 0 ! The problem is quite common in concurrent systems: a cyclic deadlock. In the run above we can see that all five forks are taken, but each of the five philosophers took the left fork, and thus there is no right fork available anymore and no one can neither eat nor release their fork and leave: we are deadlocked !

Remark: in contrast with many threading libraries, LuaPi is able to detect deadlock and finish the execution of the program with an informative message.

The solution is in breaking the cycle while acquiring forks, and there are many ways to do this. It is in general a matter of replacing the following two lines:

```
    thread:wait(fork1)
    thread:wait(fork2)
```

A first solution would be to acquire the first one, and then try to acquire the second one. In case we fail to acquire the second fork then we simply release the first one and start again, until we finally succeed in acquiring the two forks:

```
    local acquired = false
    while not acquired do
      thread:wait(fork1)
      if thread:tryWait(fork2) then
        acquired = true
      else
        thread:signal(fork1)
      end
    end
```

This "solution" if far from being satisfying, and indeed should be rejected because we might still end up in no Pi-thread being able to acquire the forks. The reason is because we always acquire the fork in the same order for every philosophers: first `fork1` and then `fork2`. The difference is that we have an even more subtle bug: we transformed our deadlock in a livelock. The Pi-threads are all "working" but all they do is to try to acquire the forks ... infinitely. It is a lot more difficult to observe a livelock than a deadlock. LuaPi, for instance, is able to detect deadlock when they occur, and simply terminates the whole agent at once in such case. For a livelock, the symptom is different: the program never stops. This is the main reason why we always try to implement an explicit termination when developping a concurrent system. A livelock is detected when the system does not terminate, as it is the case if you try the "solution" above.

The choice construct allow us to implement a satisfying solution, written as follows:

```
   local acquired = false
   while not acquired do
     local first, second
     thread:choice({ fork1:receive() , function() first = fork1
                                                    second = fork2 end },
                   { fork2:receive() , function() first = fork2
                                                  second = fork1 end })()
     thread:choice({ second:receive() , function() acquired = true end },
                   { first:send() })()

    end
```

Now the situation is better and the above can be claimed as a solution to the problem. The use of the choice construct makes the selection of the first fork non-deterministic, which breaks the cycle that made the first two propositions fail. Now we can make a second, dependent choice of either acquiring the second fork or releasing the first one (because someone else needs it). Though we found a solution to the problem there is a more *beautiful* option.

What we would like to express is the simultaneous acquisition of the forks an *atomic* principle. For this we can use a **join pattern**, a concurrent construct developed in the framework of the join-calculus and the Jocaml language.

In fact, LuaPi implements a generalization of the join patterns (in LuaPi the join actions are not restricted to input guards). A solution equivalent to the previous one can be written as follows (see `dining2.lua`):

```
thread:join({ fork1:receive(), fork2:receive() },
          function(f1,f2) print(thread.name .. " takes the forks")
          end)()
```

This code simply means that the Pi-thread thread will wait until it can receive simultaneously on the `fork1` and `fork2` channels. Before returning a corresponding reaction (a Lua function) is executed.

A join pattern is syntactically speaking quite similar to a choice but the semantics are largely different and more complex.

The syntax:

```
join = thread:choice({guard1_1, guard1_2, ... }, react1},
                     {guard2_1, guard2_2, ... }, react2},
                     -- ... etc ...
                     {guardN_1, guardN_2, ... }, reactN})
```

means: “prepares a `join pattern` for the Pi-thread `thread`, with the reaction `react1` guarded by the conjunction of the guards `guard1_1, guard1_2, ...`, the reaction `react2` guarded by `join2_1, ...`, and finally the reaction `reactN` guarded by `joinN_1, ...`”

As for the choice, the guards can be either boolean values or functions, and input or output guards (note that only input guards are available in Jocaml). The default strategy for the join pattern activation consists in:

- evaluating atomically all guards `guard1_1`, `guard1_2`, etc. and if there are all enabled, executing the reaction `react1`, or
- evaluating atomically all guards `guard2_1`, `guard2_2`, etc. and if there are all enabled, executing the reaction `react2`, or

- ... etc ...
- evaluating atomically all guards `guardN_1`, `guardN_2`, etc. and if there are all enabled, executing the reaction `reactN`, or
- if no join pattern is enabled, then block until one of them becomes enabled.

The rest of the dining philosopher example is routine, but the setting of the architecture (composed of many Pi-threads) is a little bit verbose, so we do not reproduce it here (the examples are available in the LuaPi distribution). But we can still try our join pattern solution.

```
> lua examples/tutorial/dining2.lua 5 5 10
Starting the agent
Philo5 thinks
Philo4 thinks
Philo3 thinks
Philo2 thinks
Philo1 thinks
Philo5 seats
Philo5 takes the forks
Philo5 eats 1 noodle(s)
Philo5 releases the forks
Philo4 seats
Philo4 takes the forks
Philo4 eats 1 noodle(s)
Philo4 releases the forks
Philo3 seats
Philo3 takes the forks
Philo3 eats 1 noodle(s)
Philo3 releases the forks
Philo1 seats
Philo1 takes the forks
Philo1 eats 1 noodle(s)

... etc (many lines omitted) ...

====
All Pi-threads terminated
Ending the agent
Consumption statistics:
Philo4: 12
Philo3: 10
Philo1: 8
Philo5: 14
Philo2: 6
Total consumption = 50
```

Now the example works ! To be more precise, the example is safe, which can be seen by the fact that no strange output nor deadlock occur. The system is also live, which is seen by the fact that all resources (noodles) have been consumed by the philosophers. However, the example is not fair, some philosopher ate more than others, and it may be even the case that some philosopher starves, unable to consume any resource.

Exercise :

Write a variant `dining3.lua` of the Dining philosopher example so that the execution shows a fair distribution of resources among the philosopher. The output should look like the following:

```
> lua examples/tutorial/dining2.lua 5 5 10
Starting the agent
... etc (many lines omitted) ...
====
All Pi-threads terminated
Ending the agent
Consumption statistics:
Philo4: 10
Philo3: 10
Philo1: 10
Philo5: 10
Philo2: 10
Total consumption = 50
```

A suggested idea is to add a Dispatcher Pi-thread whose purpose is to distribute "secret codes" (a channel of course) to the philosophers. A philosopher can not eat before having a secret code. After eating, a given philosopher releases its secret code but the dispatcher must ensure that no other secret code will be sent to this philosopher until all the other philosophers also had a secret code (and thus ate) in the meantime. The dispatcher thus acts as a scheduler among philosophers.

Remark: this exercise is not a simple one, do not hesitate to check the provided solution.

5. Global clock: a case for broadcast communication

Sometimes, strangely as it is, the computer world does not resembles the real world. For instance, most computation models are based on 1-to-1 round-trip interaction protocols: the function call to the function (and back), the method call to the method (and back), etc. In the distributed world, most of the time an emitter sends a message to one receiver, and a client sends a request to one server, which in turn replies to the very same client. But think about the real world. Imagine a room where you have a conversation with other people. Generally most people in the room will receive your "messages" (to really *get* the messages is another story). Then some of them might want to say something in return, but maybe they won't (because they are too shy) or maybe things will get confused because multiple "messages" will flow at the same time. So the real world most often follows a broadcast kind of interaction. There is indeed a computational model which favors, if not enforces, broadcast interaction semantics: the **synchronous languages** (see e.g. ReactiveML). These are not really mainstream but occupy a niche in the embedded world because they match the synchronous hardware of many embedded devices. Remember that LuaPi Pi-threads use synchronous channels to interact with each others, but this does not make the language a synchronous language *per se*. To become synchronous, we must be able to implement a synchronous clock, emitting a sequence of "ticks", each "tick" being a synchronization barrier where all Pi-threads must meet before leaving the previous "tick" and entering the next one. We will not describe all the features of synchronous languages, but we will see how we might implement a synchronous clock in LuaPi.

We begin with the implementation of a synchronous clock using the primitives we described so far. A global clock might be useful to implement a simulator in which all Pi-threads must perform a simulation step, perhaps concurrently, but in this case they have to wait for the next simulation step. The example is in the file `clock.lua`, it has the following header:

```
require "pithreads"

-- the number of worker Pi-threads
NB_WORKERS = tonumber(arg and arg[1]) or 100
-- the clock lifetime
TIME_TO_LIVE = tonumber(arg and arg[2]) or 10
```

The system involves two kinds of behavior: the `Clock` itself and a set or `Worker` Pi-threads. The protocol has three distinct phases:

1) the worker Pi-threads register themselves to the global clock,

2) the clock emits a tick to all the registered Pi-threads,

3) each Pi-thread perform its computation step (work) and then wait for the next tick, meaning it enters the synchronization barrier,

4) when all the Pi-threads are ready for the next tick, we go back to phase 2).

The `Worker` behavior is quite simple, it is defined as follows:

```
function Worker(thread,id,register,tick,barrier,work)
  thread:send(register,id)
  while true do
    -- wait for the next tick
    local count = thread:receive(tick)
    print(thread.name,"Tick #",count," received")
    -- perform some work
    work(thread,id,count)
    -- enter the synchronization barrier
    thread:send(barrier,id)
  end
end
```

In the first phase we register the worker Pi-thread to the global clock, using the `register` channel. Next the Pi-thread enters a cyclic behavior beginning with waiting the next clock `tick` by receiving on the latter channel. The value received is the tick count (starting from `1` until `TIME_TO_LIVE`). Then a message is displayed and the `work` function is called, meaning the worker is now doing its step work. Finally, the Pi-thread enters the synchronization barrier by sending its `id` on the `barrier` channel. And then the behavior goes on “forever” (as long as the clock lives).

The `Clock` behavior is more involved, because it has two different responsibilities:

1) emit the current tick to all the worker Pi-threads

2) implement the synchronization barrier

We also add the possibility to register new workers between the synchronization phase and the tick emission phase. The definition is as follows:

```
-- the global clock behavior
function Clock(thread,register,tick,barrier,nbReg,ttl)
  for count=1,ttl do
    if nbReg==0 then
      return
    end
    -- emit the tick
    for i=1,nbReg do
      thread:send(tick,count)
    end
    -- synchronization barrier
    for i=1,nbReg do
      local pid = thread:receive(barrier)
      print("worker #",pid," synchronized")
    end
    -- allow some new registrations
    while thread:tryReceive(register) do
      nbReg = nbReg + 1
    end
  end
end
```

The parameters of the `Clock` behavior, beyond the clock Pi-thread `thread`, are the three management channels `register`, `tick` and `barrier`, as well as the number of registered workers `nbReg` and the time-to-live value `ttl`. The behavior is a for-loop, and the loop body is executed once for each tick `count` (from `1` to `ttl`). First, we take care of aborting the clock if there is no registered Pi-thread. Then, we emit the `tick` count to each one of the worker Pi-thread. To perform so, we only need to know their number as recorded in `nbReg`. Here we simulate a broadcast emission by performing multiple point-to-point interactions. The clock then waits for all the `nbReg` workers to enter the synchronization barrier. To account for the fact that maybe new worker would like to join the clock protocol dynamically, we allow new registrations but only between the two main phases of the behavior. Note the use of a non-blocking variant of `receive`, simply named `tryReceive`. The first value received is either **`true`** (followed by the received values) or **`false`** (cannot receive). There of course also exists a non-block variant of `send` named `trySend` (also returning **`true`** or **`false`** depending on the success or failure of the emission). And finally we either proceed to the next tick or terminate the clock if the `ttl` value is reached.

We also need a dedicated behavior to register all the workers active before the clock activation:

```
-- Initial registrations of workers
function Init(thread,register,tick,barrier)
  local nbReg = 0
  while thread:tryReceive(register) do
    nbReg = nbReg + 1
  end
  thread:spawn("Clock",Clock,register,tick,barrier,nbReg
                       ,TIME_TO_LIVE)
end
```

As long as there are receivers on register, we record their count and then launch the global clock Pi-thread with the correct registration number. The final part of the script is as follows:

```
agent = pithreads.init()

register = agent:new("register")
tick = agent:new("tick")
barrier = agent:new("barrier")

agent:replicate("Worker",NB_WORKERS,Worker,register,tick,barrier,
                function(thread,id,tick)
                  print("Worker #",id," works at tick=",tick)
                end)

agent:spawn("Init",Init,register,tick,barrier)

agent:run()
```

The work job consists here simply in displaying some feedback message, but an arbitrarily complex step-behavior could be implemented in a real-world application. This emphasizes once that the example described in this tutorial are not toy examples, the all implement useful and reuseable *concurrency patterns*. Let us try our example with 5 work Pi-threads and 3 ticks. The output is as follows:

```
> lua examples/tutorial/clock.lua 5 3

Start scheduling
Worker1, Tick #, 1,  received
Worker #, 1,  works at tick=, 1
Worker3, Tick #, 1,  received
Worker #, 3,  works at tick=, 1
Worker2, Tick #, 1,  received
Worker #, 2,  works at tick=, 1
Worker #, 1,  synchronized
Worker #, 3,  synchronized
Worker #, 2,  synchronized
Worker2, Tick #, 2,  received
Worker #, 2,  works at tick=, 2
Worker3, Tick #, 2,  received
Worker #, 3,  works at tick=, 2
Worker1, Tick #, 2,  received
Worker #, 1,  works at tick=, 2
Worker #, 1,  synchronized
Worker #, 3,  synchronized
Worker #, 2,  synchronized
Worker2, Tick #, 3,  received
Worker #, 2,  works at tick=, 3
Worker3, Tick #, 3,  received
Worker #, 3,  works at tick=, 3
Worker1, Tick #, 3,  received
Worker #, 1,  works at tick=, 3
Worker #, 1,  synchronized
Worker #, 3,  synchronized
Worker #, 2,  synchronized
All Pi-threads terminated
End of scheduling
```

This output might seem surprising since we started 5 worker Pi-threads and only 3 of them are doing their job. The reason is that there is some non-determinism involved in our example. In fact, dealing with concurrency is before all about understanding the non-deterministic part of the programs we write. Here, the aspect we do not control is in the `Init` behavior. What happened in the execution just above is that only 3 workers got the chance to activate their registration action (receiving on the `register` channel) before the start of the initialization Pi-thread. Actually we can enforce this by yielding at the start of the initialization behavior, and also each time we register a new worker. This is one of the few cases where explicit yielding would be required.

Another limit of the clock protocol is that we have to keep track of the number of registered workers at all time.

Exercise :

Modify the global synchronous clock protocol to account for workers willing to `unregister` from the clock (you might need to introduce another explicit channel as well as a choice between either registering or unregistering a Pi-thread).

LuaPi provides a much better way to solve the global synchronous clock problem, thanks to the broadcast primitives it supports. The underlying theory, the broadcast Pi-calculus (b-pi), defines two operations:

- the **`broadcast`** (**`bcast`**) or **`talk`** (or **`say`**) primitive for sending a broadcast message
- the **`listen`** primitive to receive a "spoken" message (in fact a synonym of **`receive`**)

The syntax:

`thread:broadcast(chan,msg, ...)`

or `thread:bcast(chan,msg, ...)`

or `thread:talk(chan,msg, ...)`

or `thread:say(chan,msg, ...)`

means: "`thread` is sending the values `msg`, ... *atomically* to all receivers on channel `chan`."

The precise semantics of the broadcast can decomposed in two phases:

1) an active receiver on `chan` is a Pi-thread that can advances with interactions until it blocks receiving on `chan` (e.g. by performing `thread:listen(chan)`),
2) the message `msg` is sent to all of the active receivers at once, *atomically*,
3) all the active receivers are put back in the ready queue for further execution.

This powerful construct can be used to simplify a lot the implementation of synchronous clocks (or any synchronous broadcast protocol indeed). The broadcast variant is in the file `bclock.lua`.

The Worker behavior can be rewritten as follows:

```
-- the worker behavior
function Worker(thread,id,tick,work)
  while true do
    -- wait for the next tick
    local count = thread:listen(tick)
    print(thread.name,"Tick #",count," received")
    -- perform some work
    work(thread,id,count)
  end
end
```

Now the workers do not have to take care of any registration or synchronization, they simply wait for the next clock tick by listening on the `tick` channel. As a matter of fact, we could have received on the channel, listening and receiving are synonymous. The `listen` method can serve as an indicator that the Pi-thread is willing to participate in a broadcast protocol.

The most striking characteristic of the broadcast version is in the Clock behavior that becomes a lot more simple, if not trivial:

```
-- the clock behavior
function Clock(thread,tick,ttl)
  for count=1,ttl do
    -- emit the tick
    thread:broadcast(tick,count)
  end
end
```

There is nothing to do except for broadcasting the current tick count. Given the semantics of the `broadcast` method, we know that all workers will receive the current tick count before they will all receive the next one, and so on. The remaining parts of the script is similar to the non-broadcast version, except that there is no registration phase and thus no need for any initialization behavior.

6. Final example: collecting votes

To wrap up our tutorial, we will describe the `collect` primitive – a more advanced concept - in a rather classical example of a vote protocol. The protocol simply consists in asking a group of Pi-threads to decide for a positive vote (`true`) or a negative vote (`false`) . There are many uses for such a protocol, a typical example being the commit protocol for a system of concurrent read/write database transactions.

The `send` primitive denotes a **1-to-1** kind of interaction, whereas the `broadcast` corresponds to **1-to-N** protocol: from 1 emitter to N receivers. There should exist some symmetrical argument for receivers. It is clear that `receive` (and also `listen`) corresponds to something we might call a **1-from-1** : 1 receiver receives from 1 emitter. We might wonder why a **1-from-N** kind of interaction protocol would not exist. In fact, people working on synchronous languages already studied similar concepts. Unlike LuaPi, all emissions performed on a given channel (called a signal in the synchronous world) are finally *collected* at the end of the tick count. This roughly corresponds to the synchronization barrier we implemented previously, except that each worker would register some value to be collected. But LuaPi is not a synchronous language (even if we argue it can be used to implement such a language) so what would be a useful equivalent in our framework. This equivalent is a new primitive, named `collect`, which is to `receive` (or `listen`) what `broadcast` is to `send`.

The syntax :

```
local values = thread:collect(chan)
```

means: "`thread` will collect all the values sent by all active senders on channel `chan`."

This works similarly to broadcast but here what is received is a table containing all of the values received from all the active sender. We remind that a Pi-thread is an active sender on `chan` if it can execute some internal steps (no interaction such as send or receive, in this case the Pi-thread could become a passive sender) so that it actually blocks to perform an emission.

We can now describe our final example : `voters.lua`, whose header is as follows:

```
require "pithreads"

NBVOTERS = tonumber(arg and arg[1]) or 10
NBTURNS = tonumber(arg and arg[2]) or 10
```

The parameters are the number of voters involved as well as the number of turns for the vote. For the demonstration purpose, all the voters run the same `Voter` behavior, defined as follows:

```
-- the common behavior of voters
function Voter(thread,id,vote)
  while true do
    local turn, secret = thread:receive(vote)
    if math.random() > 0.5 then
      print("Voter #" .. tostring(id) .. ": vote YES")
      thread:send(secret,true) -- vote YES
    else
      print("Voter #" .. tostring(id) .. ": vote FALSE")
      thread:send(secret,false) -- vote NO
    end
  end
end
```

All the voters share a common `vote` channel, and take there decision when the coordinator (see below) broadcasts a vote activation message. Each voter receives the `turn` number of the vote as well as a `secret` channel used to take the decision. The rationale is that each vote turn must not be performed in isolation for non-participating Pi-threads. In a real application (e.g. the voting phase of a Two-Phase Commit (2PC) protocol in a transaction system), the decision would involve analyzing some datastructure, but in our demonstration example we simply use a random number to decide wether we vote "yes" (`true`) or "no" (`false`) for the current turn.

The Coordinator behavior controls the vote algorithm, it is specified as follows:

```lua
-- behavior for the vote coordinator
function Coordinator(thread,vote,nb,ifYes,ifNo)
  for turn=1,nb do
    -- (1) create a secret channel for this turn's vote
    local secret = thread:new("secret"..tostring(turn))
    -- (2) broadcast the vote activation message
    --      (with the secret channel)
    thread:broadcast(vote,turn,secret)
    -- (3) collect all votes
    local votes = thread:collect(secret)
    -- (4) analyze the votes and take the final decision
    --      Remark: votes is an array of returned values
    local nbYes = 0
    for i,yes in ipairs(votes) do
      if yes[1]==true then
        nbYes = nbYes + 1
      end
    end
    if nbYes >= #votes / 2 then
      ifYes(turn,#votes,nbYes)
    else
      ifNo(turn,#votes,nbYes)
    end
  end
end
```

The Coordinator cycles for the `nb` vote turns. The first step is to create the `secret` channel that will be sent to all the voters so that they can send back their local decision to the coordinator. We first `broadcast` the `secret` channel to all the voters.

The second phase is to wait for all the local decisions of the voters. In this case we use the `collect` primitive. Its behavior is to wait for all the potential senders and collect all the sent values in a Lua table named `votes`. If, for example, the protocol involves three voters, with the respective choices “yes”, “yes”, “no”, then the votes table will corresponds to: `{ true, true, false }`. Note that the ordering in the top-level table is arbitrary (i.e. `{ false, true, true }` would be equivalent).

The final part of the coordinator algorithm is to count the number of “yes” votes, which allows to take the global decision if there is a majority of “yes” votes. Once again, we delegate the handling of “yes” or “no” decisions to callbacks, making this coordinating pattern quite generic.

Now we can create the global architecture of the vote system:

```
agent = pithreads.init()

vote = agent:new("vote")

agent:spawn("Coordinator",Coordinator,vote,NBTURNS,
            function(turn,nbVotes,nbYes)
              print("Turn #" .. tostring(turn)
                    .. " global decision is YES")
              print("( " .. tostring(nbVotes)
                    .. " voters, " .. tostring(nbYes)
                                  .. " voted Yes)")
            end,
            function(turn,nbVotes,nbYes)
              print("Turn #" .. tostring(turn)
                    .. " global decision is NO")
              print("( " .. tostring(nbVotes) .. " voters, "
                    .. tostring(nbYes) .. " voted Yes)")
            end)

agent:replicate(agent,"Voter",NBVOTERS,Voter,vote)

print("Starting the agent")
agent:run()
```

The architecture when all voters Pi-thread are created and ready to take their decision is depicted on **Figure 7**.

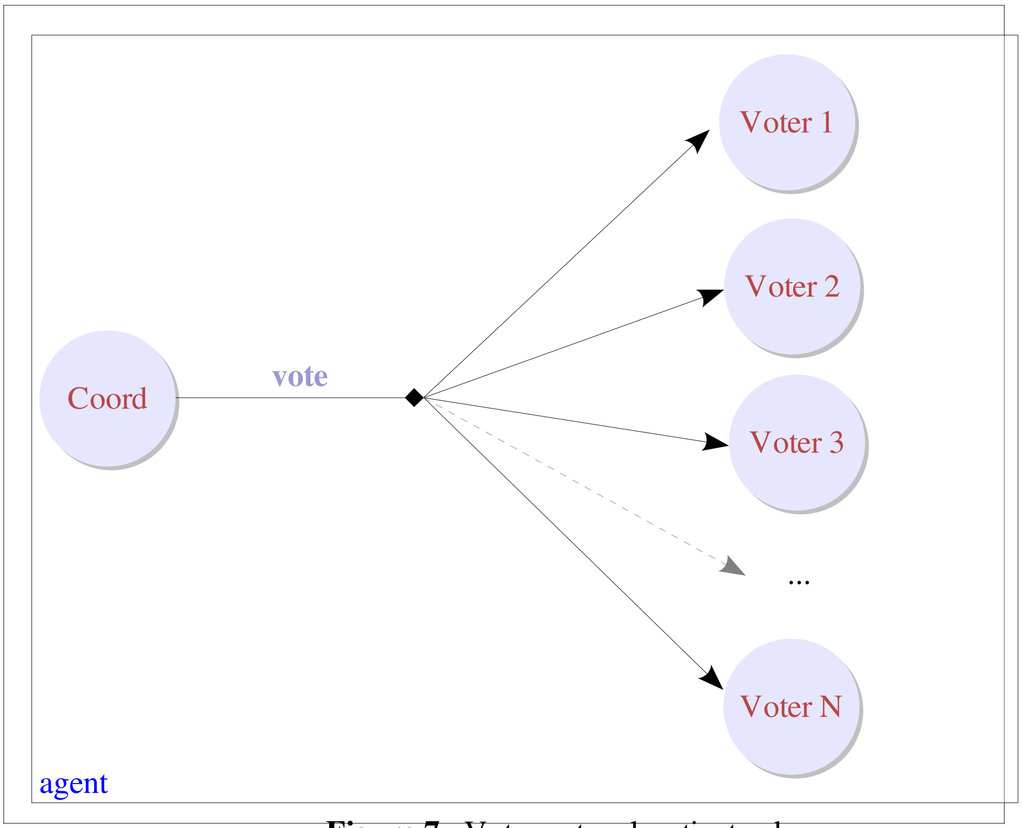

Figure 7 : Vote protocol, activate phase

After the broadcast phase, the collect phase can be depicted as in **Figure 8**.

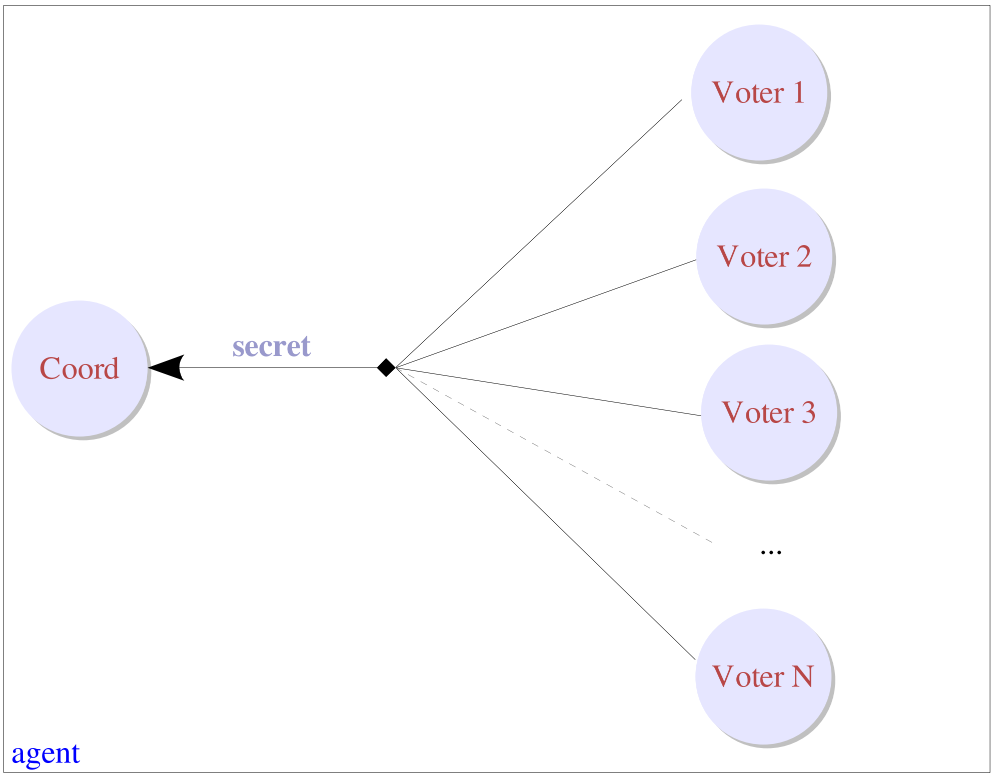

Figure 8 : Vote protocol, activate phase

We clearly see on the picture the 1-to-N (**broadcast**) and 1-from-N (**collect**) configurations. The most striking feature of LuaPi is to make these primitive fully atomic (and efficient in practice).

We give below a sample execution with 5 voters and 3 rounds:

```
> lua examples/tutorial/votes.lua 5 3
Starting the agent
Voter #1: vote YES
Voter #2: vote NO
Voter #3: vote YES
Turn #1 global decision is YES
( 3 voters, 2 voted Yes)
Voter #5: vote YES
Voter #2: vote YES
Voter #1: vote NO
Voter #4: vote NO
Voter #3: vote YES
Turn #2 global decision is YES
( 5 voters, 3 voted Yes)
Voter #5: vote NO
Voter #2: vote YES
Voter #1: vote NO
Voter #4: vote YES
Voter #3: vote NO
Turn #3 global decision is NO
( 5 voters, 2 voted Yes)
====
All Pi-theads terminated
```

The elections are over ... and the winner is ...

There are many more things to discover about these powerful **`broadcast`** and **`collect`** primitives, the limit is *your* imagination.

Conclusion

It is now time to conclude our journey through the enlightening world of concurrent programming with LuaPi.

There are many interesting applications that can be developed using LuaPi as the underlying framework: internet applications such as web servers, event-processing systems (e.g. for graphical user interfaces or games), simulators, etc.

More than being just a set of expressive primitives, the LuaPi constructs have well-defined, clear and precise semantics. Assistant tools are being developed so that fine-grained concurrent protocols implemented using this language (using LuaPi or one of the few others implementations we provide) can be checked for errors: semantic errors, deadlocks and livelocks and so on.