G52CON:
Concepts of Concurrency

Lecture 14 Remote Invocation

Brian Logan

School of Computer Science

bsl@cs.nott.ac.uk

Outline of this lecture

- distributed processing
- message passing
- remote invocation
  - Remote Procedure Call (RPC)
  - Extended Rendezvous
- examples:
  - time server using RPC
  - Ada rendezvous

Implementations of concurrency

We can distinguish three types of implementations of concurrency:

- **multiprogramming**: execution of concurrent processes by timesharing them on a single processor;
- **multiprocessing**: the execution of concurrent processes by running them on separate processors which all access a shared memory; and
- **distributed processing**: the execution of concurrent processes by running them on separate processors which communicate by message passing.

Distributed processing

- processors share only a communication network, e.g., networks of workstations or multicomputers with distributed memory
- the processes don't share a common address space, so they can't communicate via shared variables
- instead they communicate by *sending and receiving messages*

Message passing

Processes communicate by sending and receiving messages using special message passing primitives which include synchronisation:

- **`send`**(*`destination`*) *`message:`* sends *`message`* to another process *`destination`*

- **`receive`**(*`source`*) *`message:`* indicates that a process is ready to receive a message *`message`* from another process *`source`*

Synchronising communication

If a process tries to **`receive`** a message before one has been sent, it will block until there is a message for it to read.

The differences are mainly in the behaviour of the **`send`**ing process:

- asynchronous communication: the sending process continues without waiting for the message to be received, e.g., Unix sockets, `java.net`
- synchronous communication: the sending process is delayed until the corresponding receive is executed, e.g., CSP, occam
- remote invocation: the sending process is delayed until a reply is received, e.g., RPC (`java.rmi`), Extended Rendezvous

Asynchronous Message Passing

If a process sends a message and continues executing without waiting for the message to be received, then the communication is termed *asynchronous*

- **`send`** operations are non-blocking:
- a sending process can get arbitrarily far ahead of a receiving process;
- message delivery is not guaranteed if failures can occur; and
- since channels can contain an unbounded number of messages messages have to be buffered.

Synchronous message passing

If the sending process is delayed until the corresponding receive is executed, the the message passing is *synchronous*

- both the **`send`** and **`receive`** operations are blocking
- a process **`send`**ing to a channel delays until another process is ready to receive from that channel;
- messages don't need to be buffered.

Problems with message passing

Both asynchronous and synchronous message passing assume *one-way* communication:

- messages are transmitted in one direction only, from sender to receiver
- message passing is well suited to problems in which the flow of information is essentially one-way, e.g., producer-consumer problems
- two way information flow between, e.g., between clients and servers, has to be programmed with two explicit message exchanges

Remote invocation

With *remote invocation* a process executes a synchronous send and waits until the reply is received:

- combines aspects of *monitors* and *synchronous message passing*:
  - as with monitors interaction is via public procedures
  - as with synchronous **`send`**, calling a procedure delays the caller
- provides *two way* communication from the caller to the process servicing the call and back
- implemented using message passing

RPC & Extended rendezvous

There are two main forms of remote invocation:

- *Remote Procedure Call* creates a *new* process to handle each call
- *Extended Rendezvous* services a request using an *existing* process.

Modules

A *module* is an abstraction which can be used to describe both RPC and Extended Rendezvous

A module contains both *processes* and *local* and *exported procedures*:

- the *header* contains the signatures of the exported procedures
- the *body* contains local procedures and processes, local variables, and initialisation code
- at any point in time, a module contains zero or more *processes*
- different modules may reside in different addresses spaces

Module syntax

```
module <moduleName>
    // header (signatures of exported procedures)
    export <procID1>(args);
    export <procID2>(args);
    ...
body
    // local variables
    // initialisation code

    // implementations of exported module procedures
    // local procedures
    // local (background) processes
```

Modules and message passing

Communication *between* modules is by calls to exported procedures:

- arguments and return values are passed as *messages*
- the sending and receiving of messages is *implicit* rather than explicitly programmed

Communication *within* modules is similar to monitors:

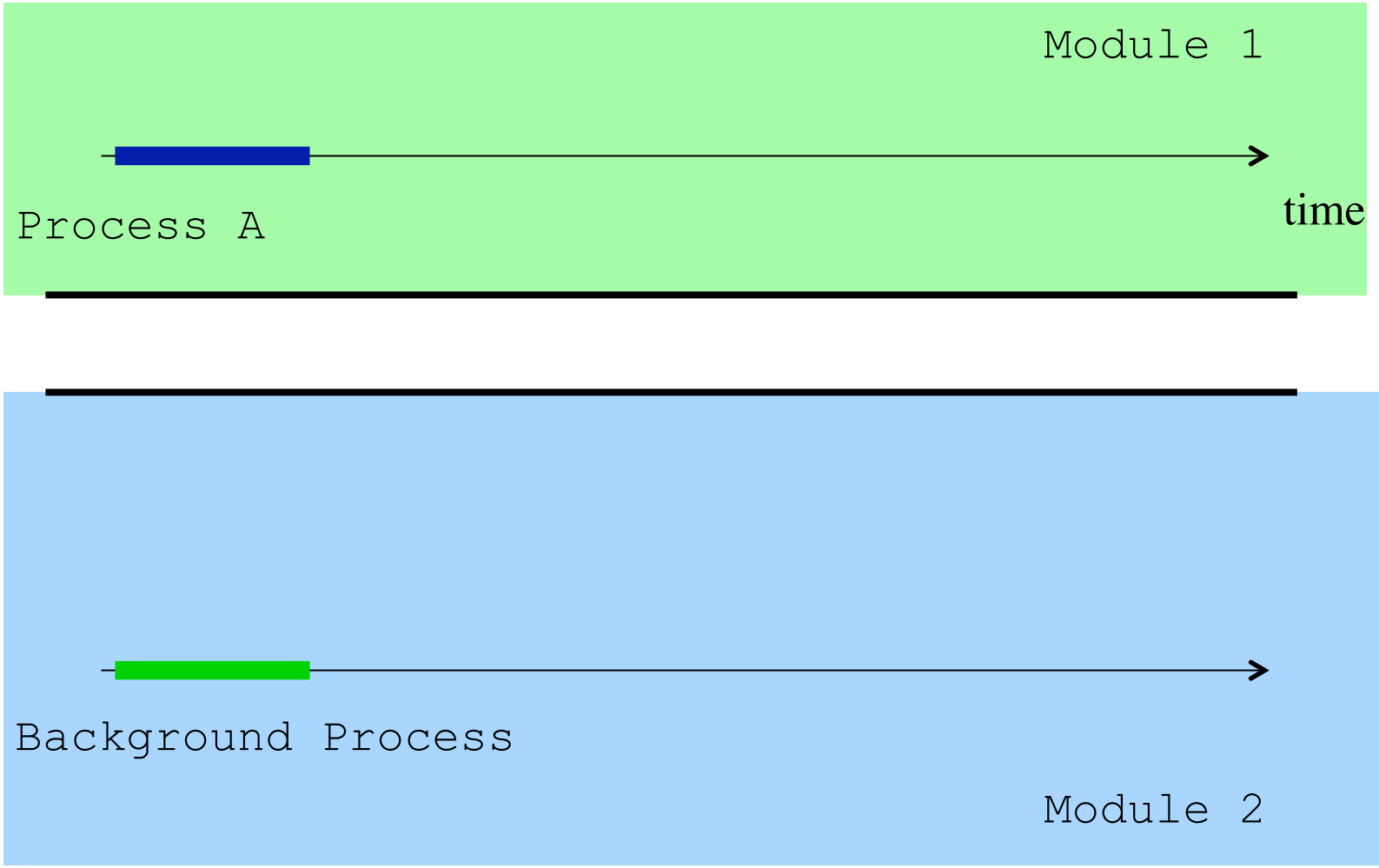
- processes within a module can share variables and call procedures declared in that module.

Modules and RPC

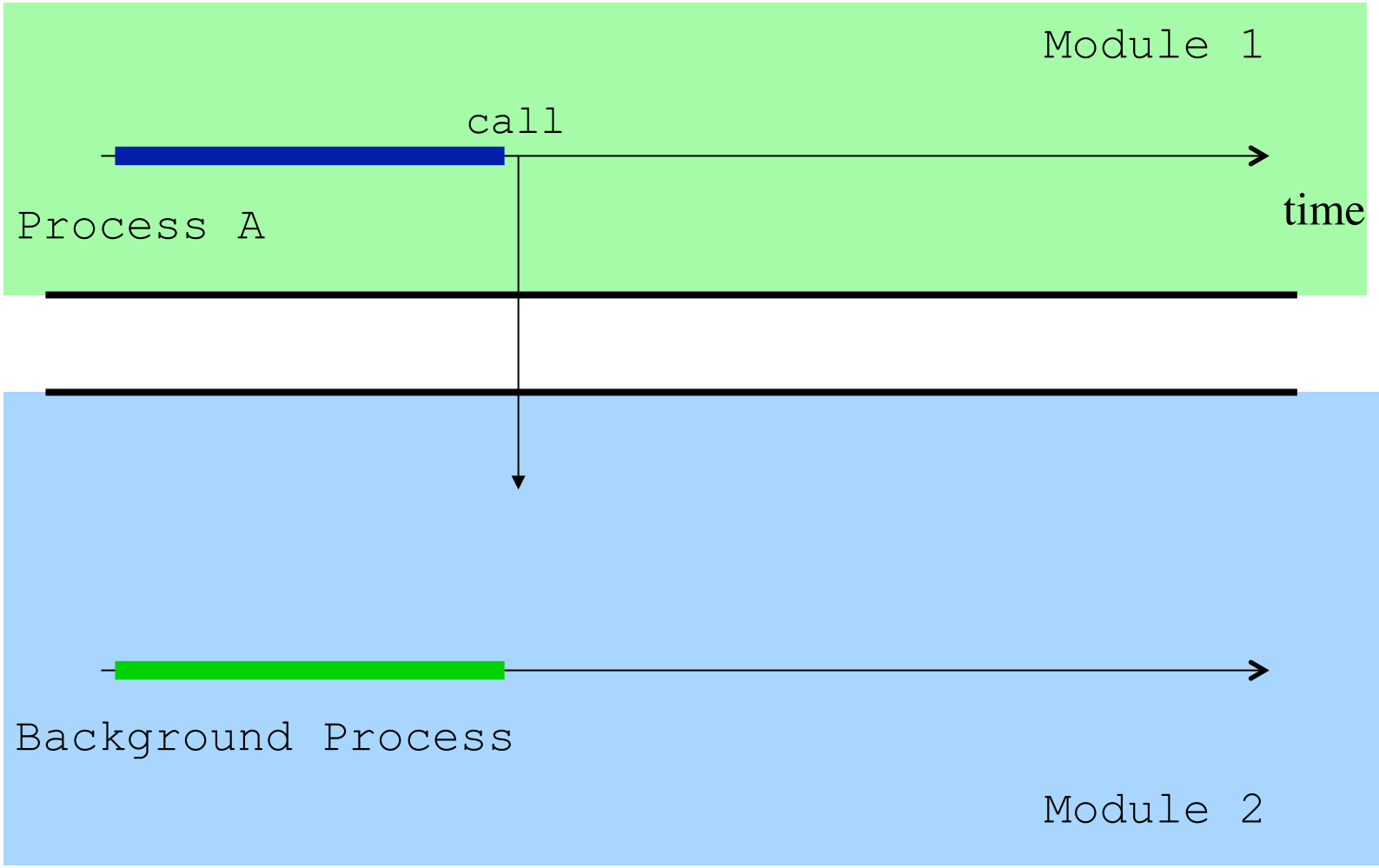
In RPC, a module contains *zero or more* processes and some exported procedures:

- local processes are called *background processes*
- processes that result from remote calls to exported procedures which are called *server processes*

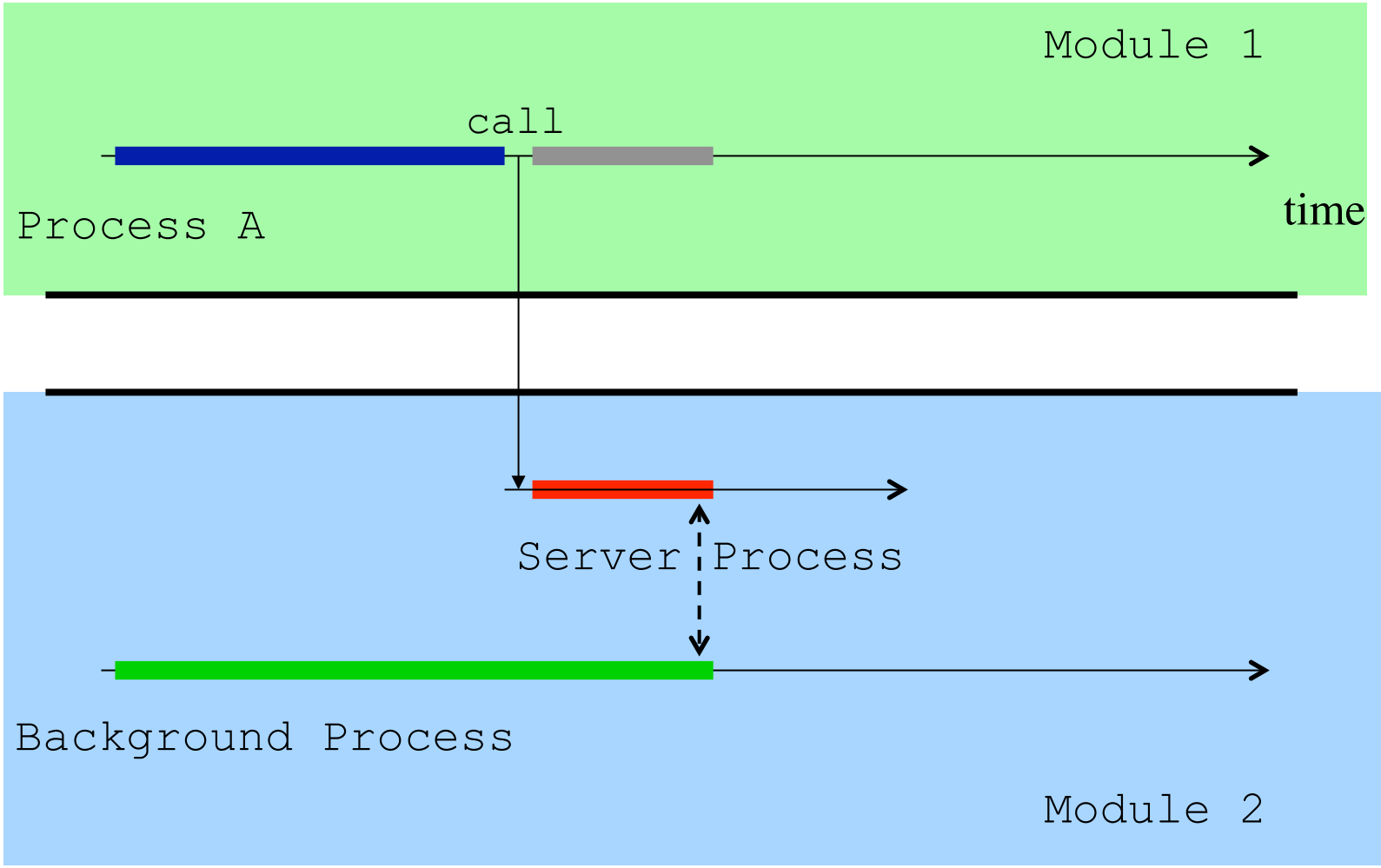
Servicing an RPC call 1



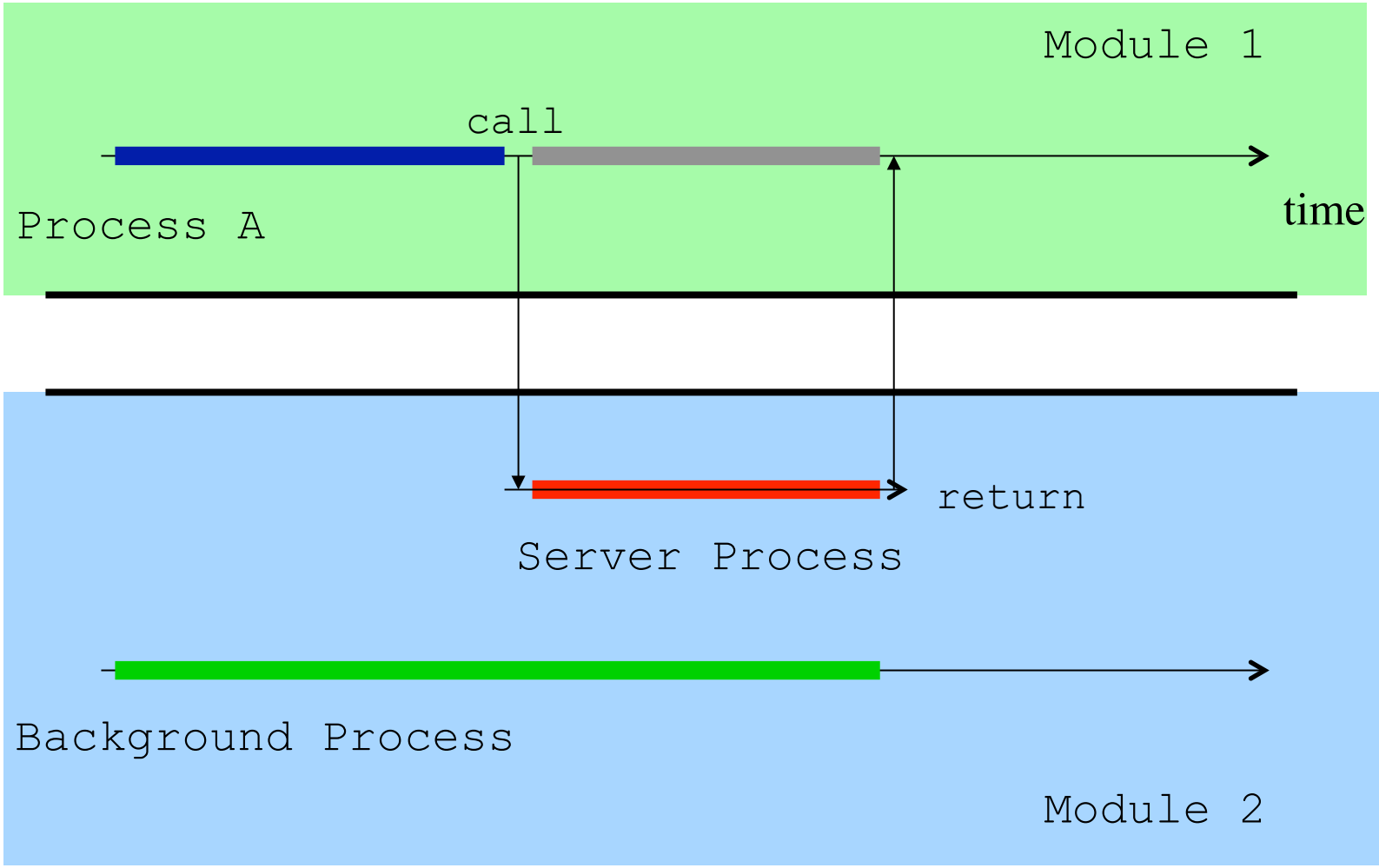
Servicing an RPC call 2



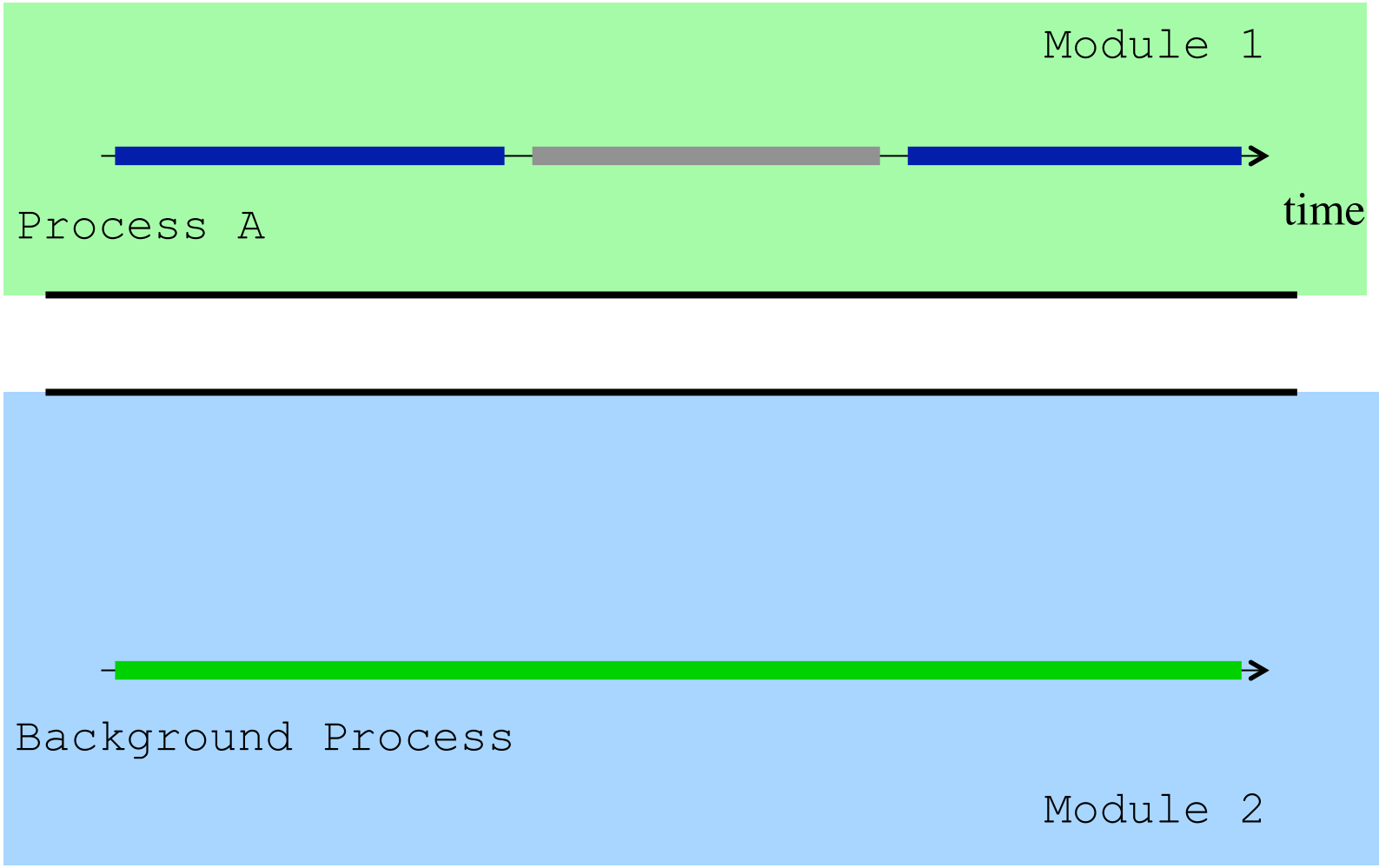
Servicing an RPC call 3



Servicing an RPC call 4



Servicing an RPC call 5



Synchronisation in modules

There are two ways for server and background processes in an RPC module to have mutually exclusive access to shared variables and to synchronise with each other

- all the processes in the same module execute with mutual exclusion (as in monitors)–condition synchronisation is programmed explicitly using *semaphores* and/or *condition variables*

- processes execute concurrently within a module and both mutual exclusion and condition synchronisation are programmed explicitly using *semaphores* and/or *condition variables*

Example: time server

A *time server* provides timing services to client processes :

- the time server defines two procedures: `get_time` and `delay`
- a client process gets the time of day by calling `get_time()`
- a client process calls `delay(interval)` to block for `interval` time units

Time server RPC implementation 1

```
module TimeServer
    export integer get_time();
    export void delay(integer interval);

body
    integer time = 0;
    binary semaphore m = 1;
    binary semaphore[] d = new binary semaphore[n] {0};
    queue napQ;

    // exported module procedures ...
```

Time server RPC implementation 2

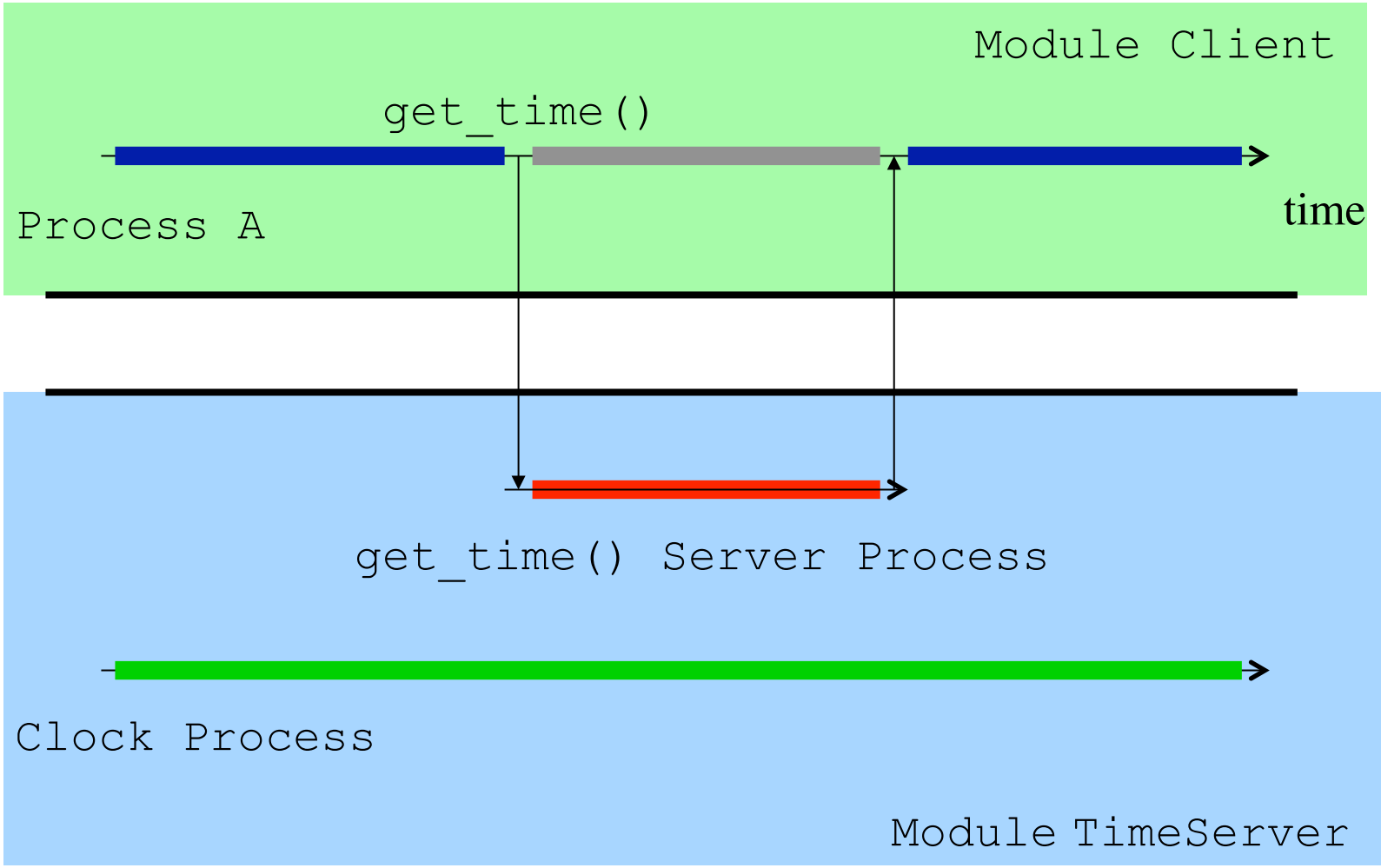
```
// exported module procedures
integer get_time() {
    return time;
}

void delay(integer interval) {
    integer waketime = time + interval;
    P(m);
    insert(napQ, <waketime, serverProcID>);
    V(m);
    P(d[serverProcID]);
}
// background Clock process ...
```

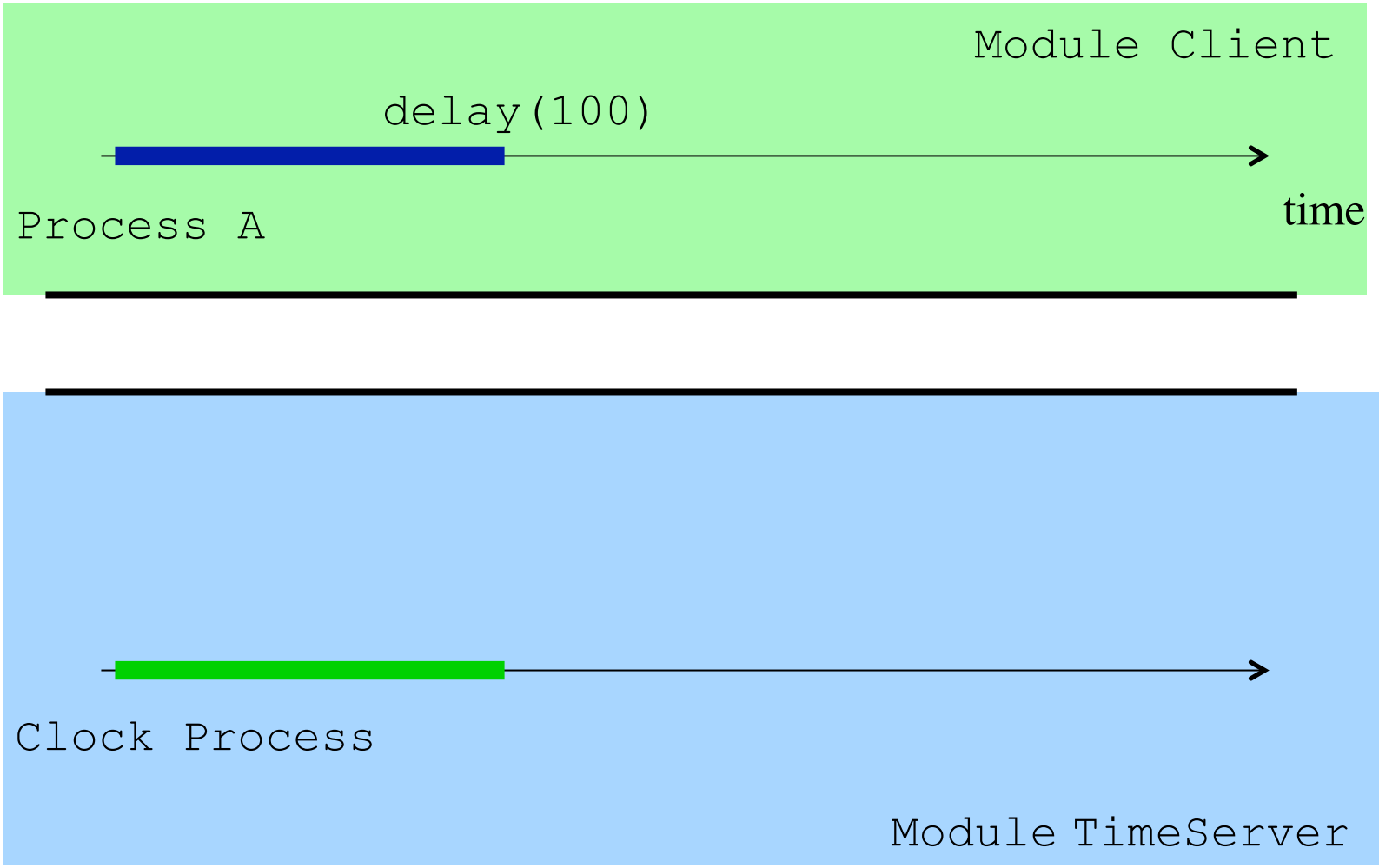
Time server RPC implementation 3

```
// background Clock process
process Clock {
    // start hardware timer (omitted) ...
    while(true) {
        // wait for interrupt then restart timer...
        time++;
        P(m);
        while(time >= smallest waketime on napQ) {
            remove(napQ, <waketime, serverProcID>);
            V(d[serverProcID]);
        }
        V(m);
}
```

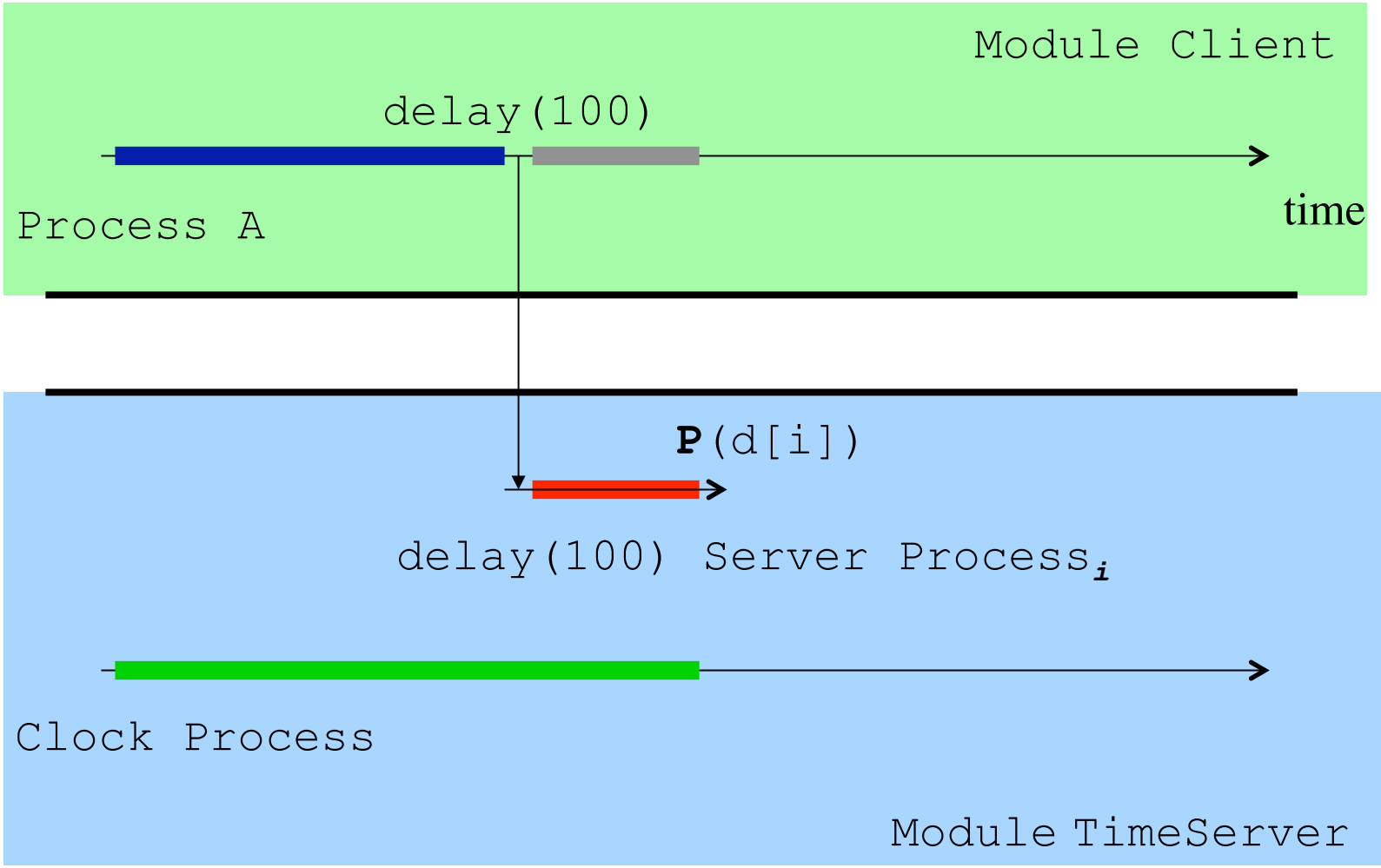
Servicing a `get_time()` call



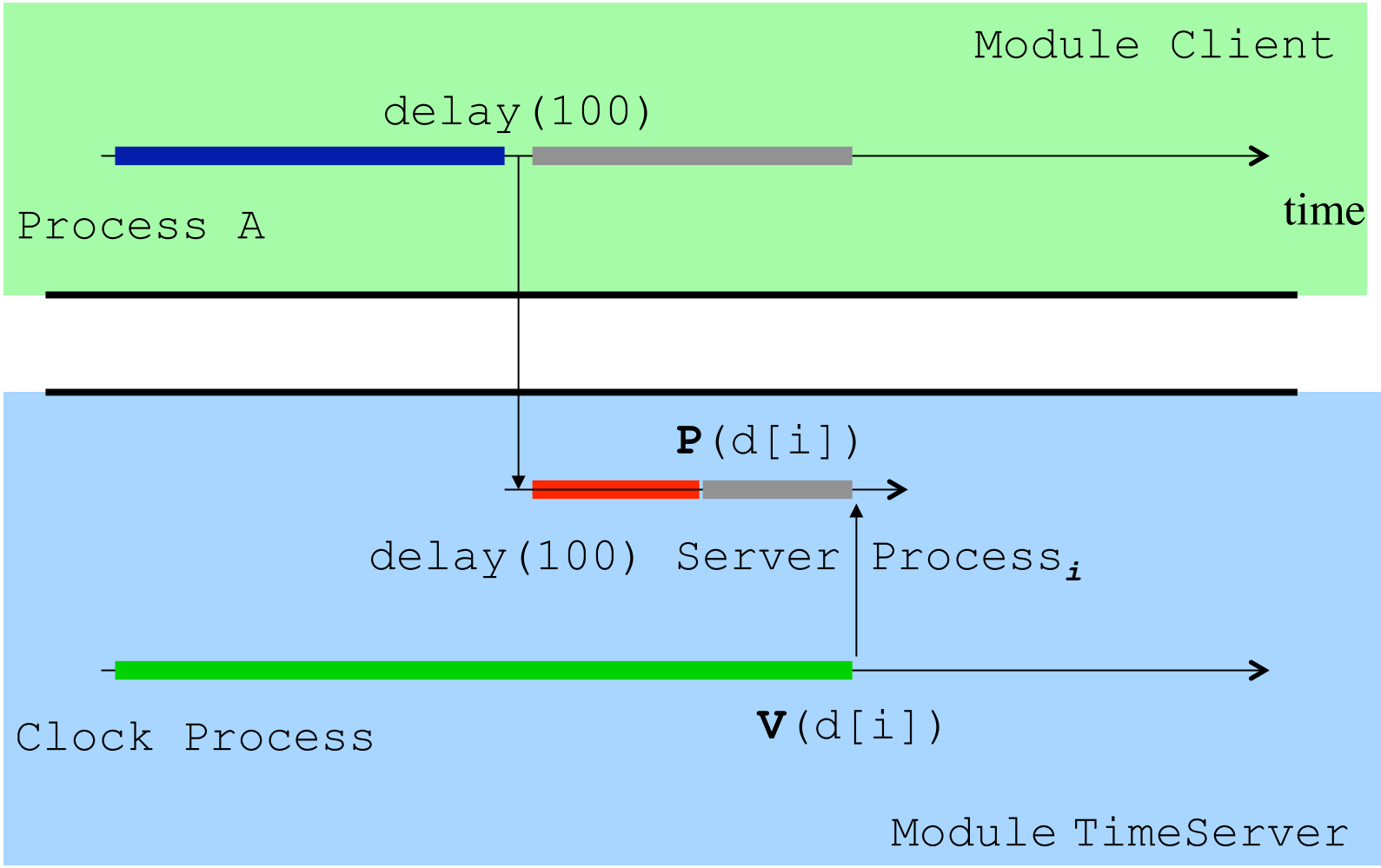
Servicing a `delay()` call 1



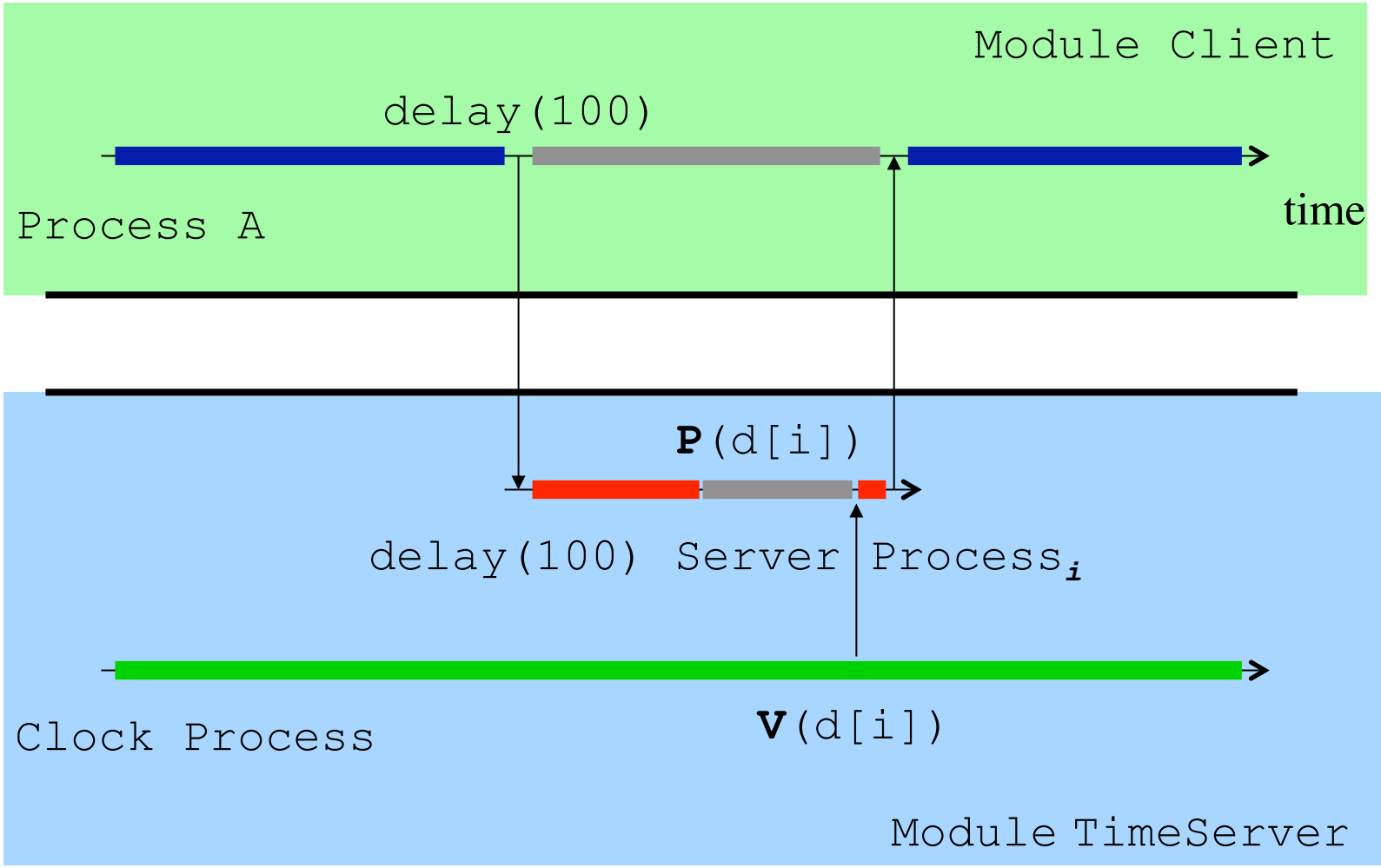
Servicing a `delay()` call 2



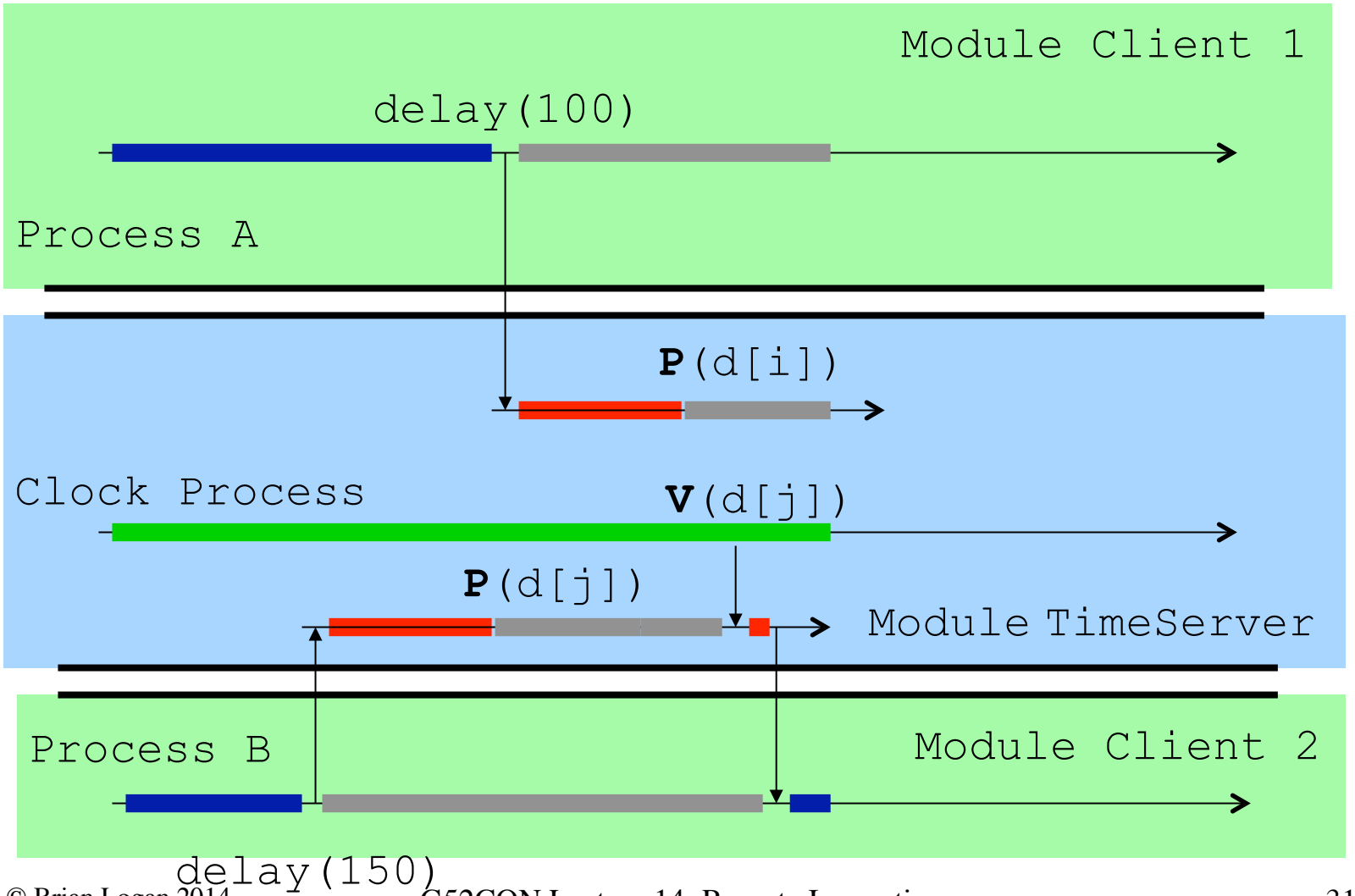
Servicing a `delay()` call 3



Servicing a `delay()` call 4



Servicing multiple `delay()` calls



Extended rendezvous

Extended rendezvous combines *communication* and *synchronisation*

- as with RPC a process invokes an operation by means of a remote call:
  - a server process waits for and then acts on a single call
  - calls are serviced one at a time rather than concurrently
- the caller and server processes synchronise (rendezvous) on the call

Modules and extended rendezvous

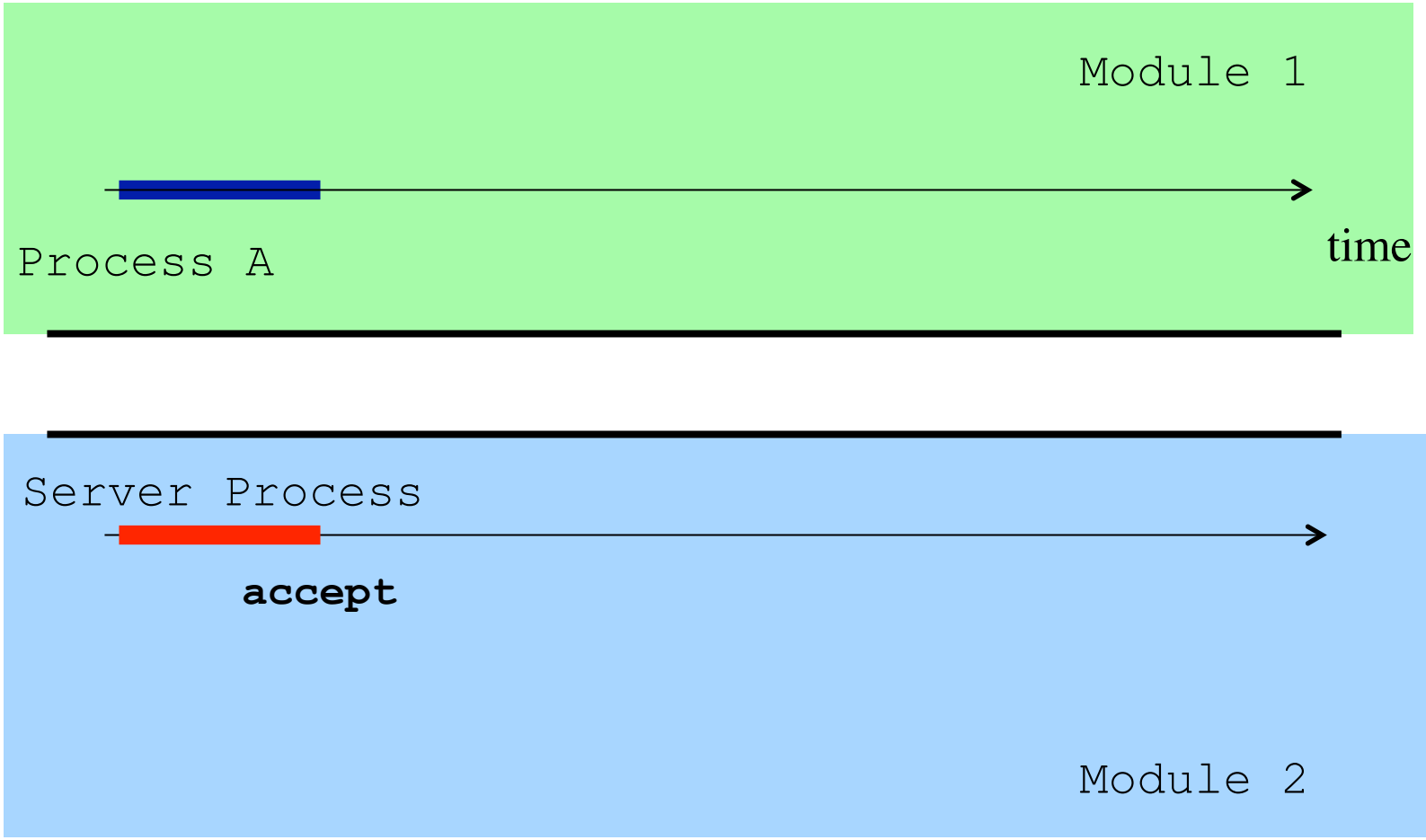
In extended rendezvous a module contains a *single* process and some exported operations:

- the header contains signatures of *operations* (or entry points) exported by the module
- the body of a module consists of a *single* process that services the call
- **`accept`** statements block the server process until there is at least one pending call of an exported operation

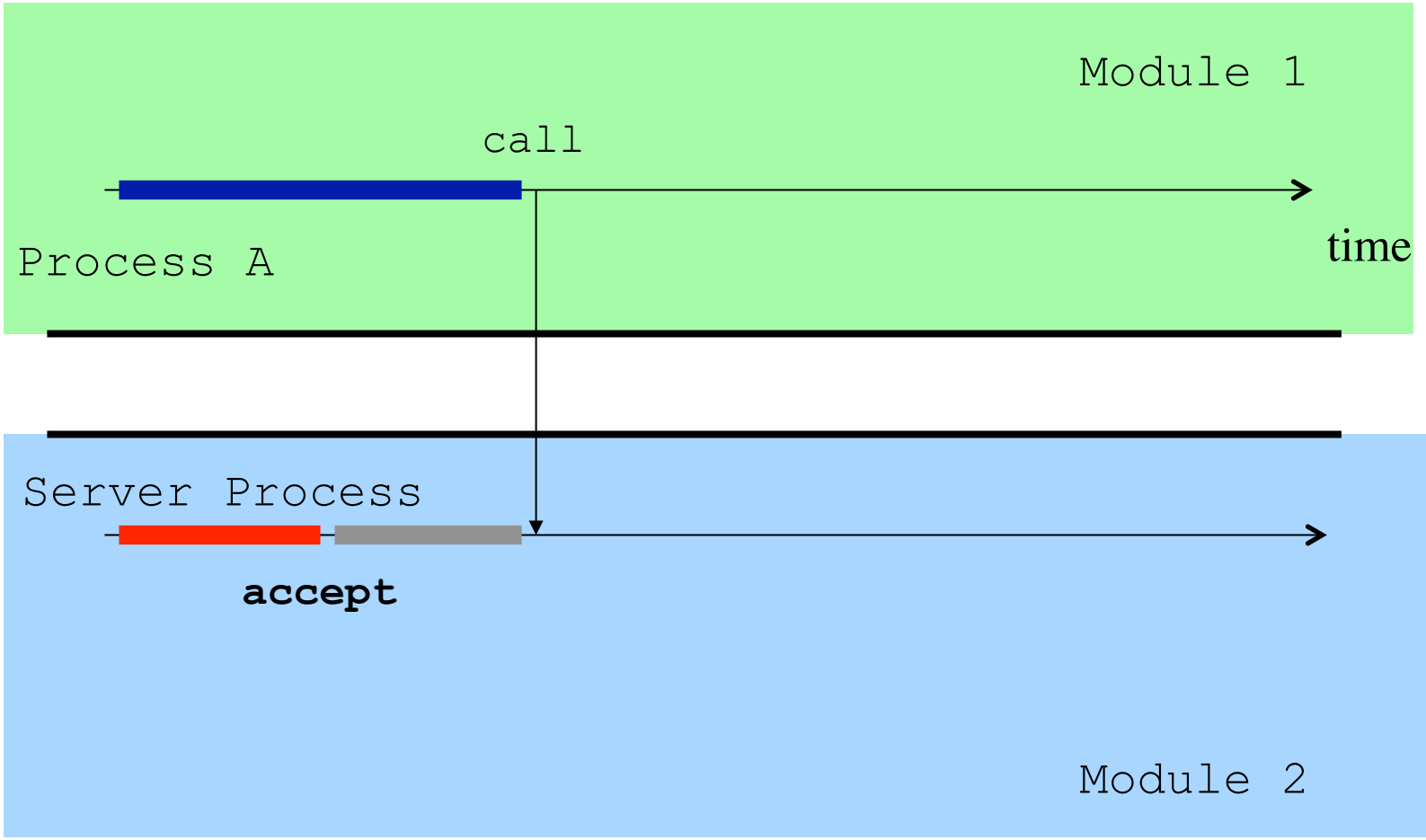
As with RPC

- arguments to the call and any return values are passed as messages between the caller and server processes

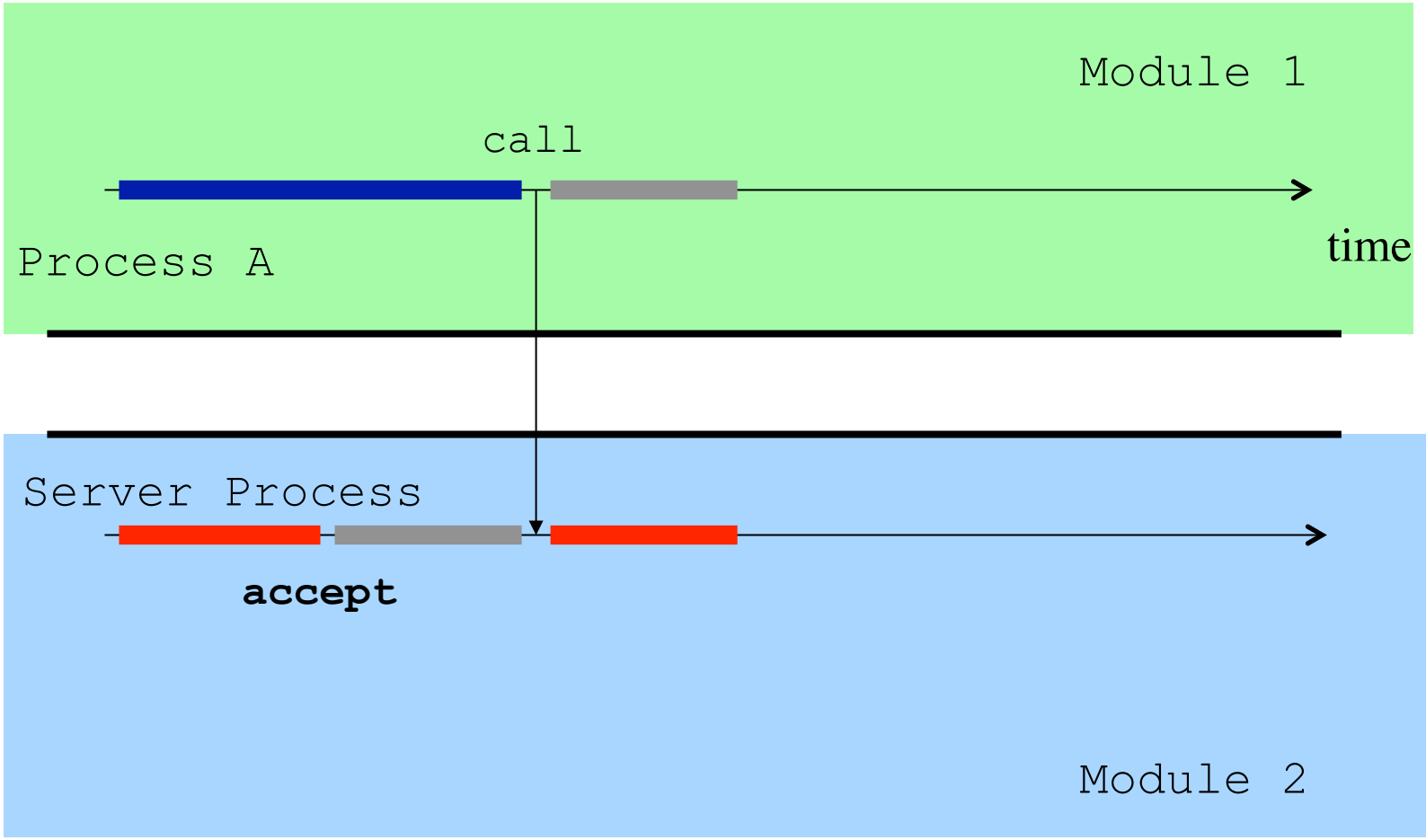
Servicing a rendezvous call 1



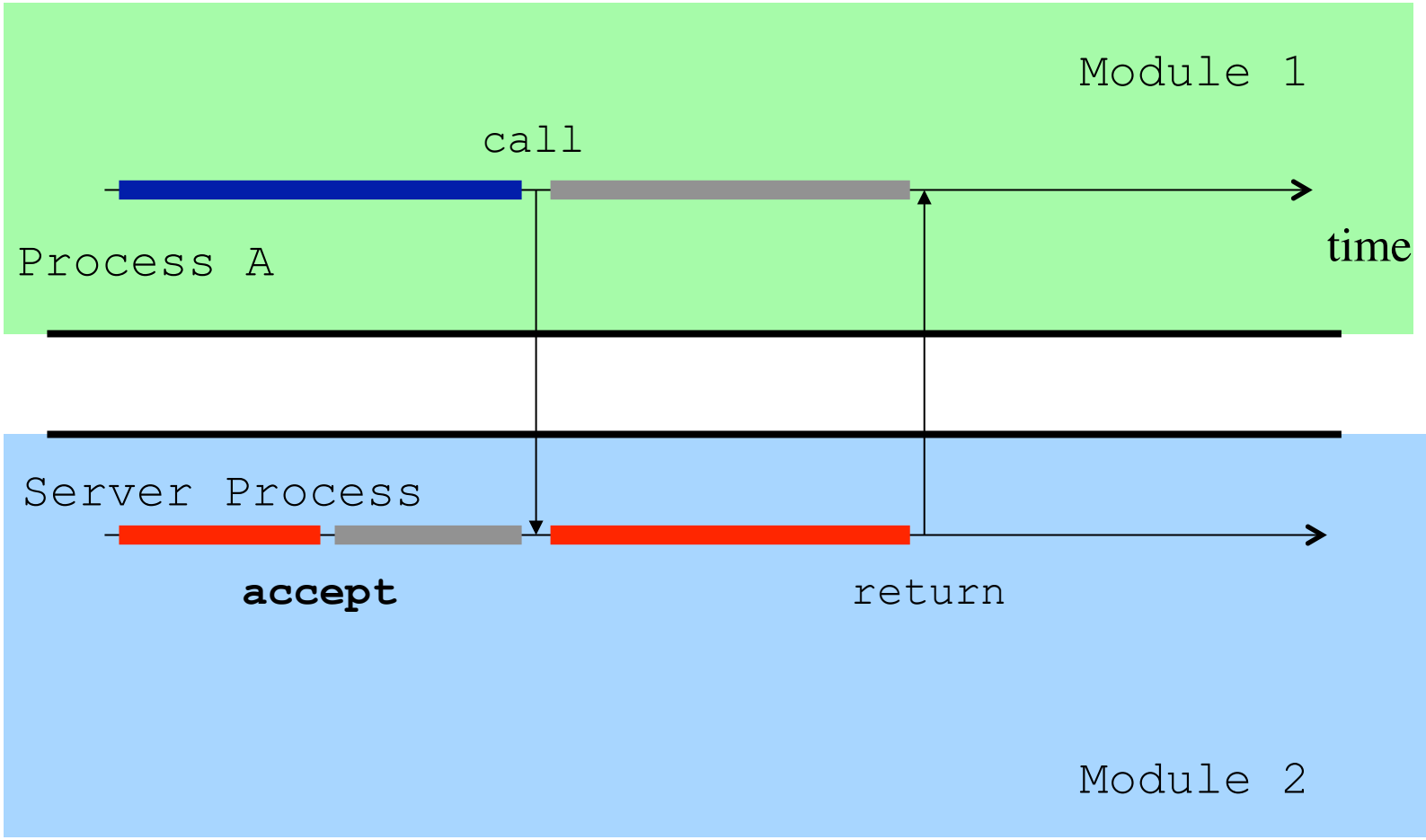
Servicing a rendezvous call 2



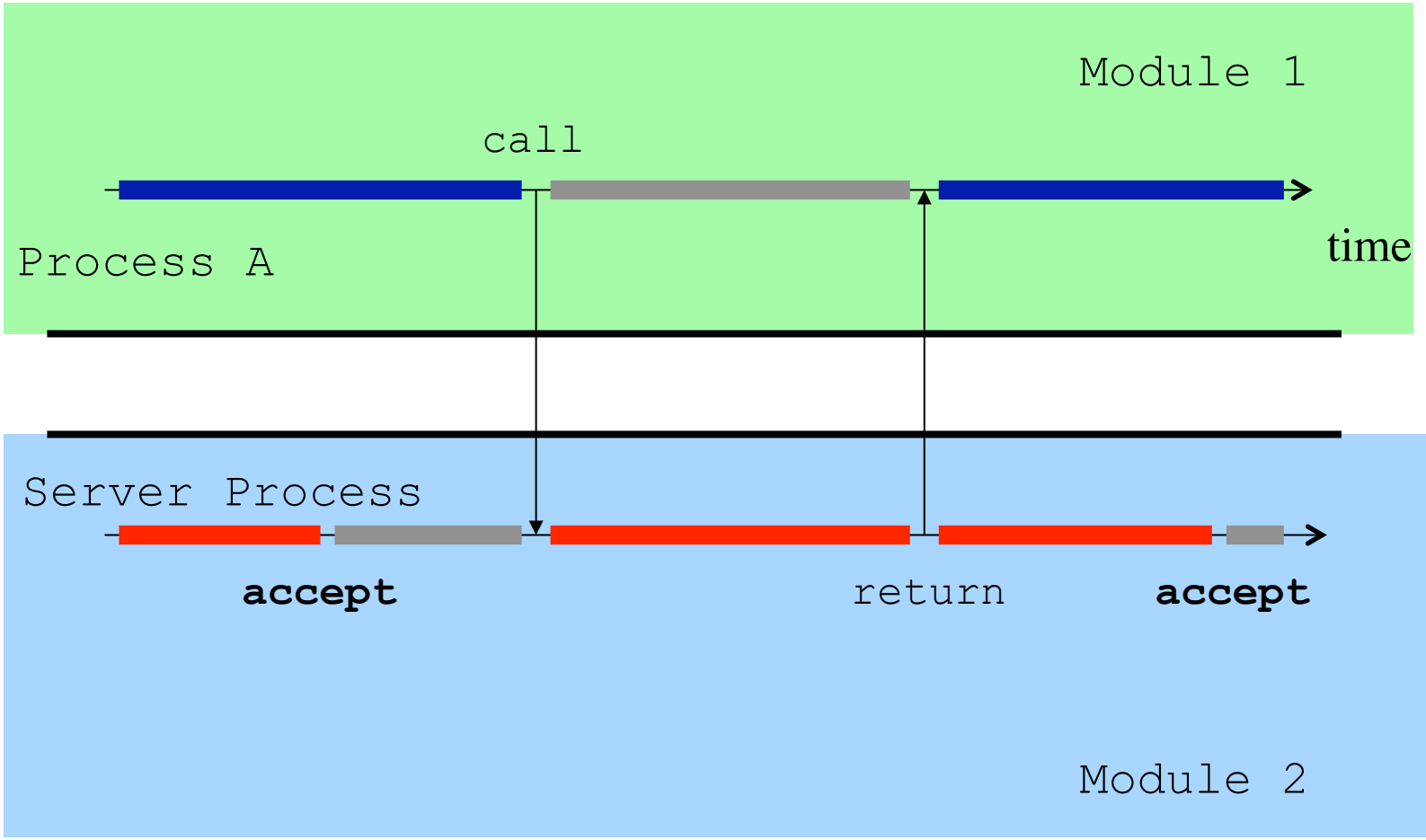
Servicing a rendezvous call 3



Servicing a rendezvous call 4



Servicing a rendezvous call 5



Example: Ada Rendezvous

In Ada, a module is called a *task* and exported operations are called *entries*

- the body of `task` contains variable and procedure declarations and the program statements executed by the `task`
- for each **`entry`** declared in the header, there is a corresponding **`accept`** statement
- execution of a `task` blocks at an **`accept`** statement, unless there is a call on the corresponding **`entry`**
- when there is a call on the **`entry`**, the statements that make up the body of the **`accept`** statements are executed, and any results returned

Ada **`entry`** and **`accept`** statements

```
task <name> is
    entry <entryID1>(args);
    entry <entryID2>(args);
end;
task body <name> is
    // local declarations
begin
    // statements
    loop
        select
            accept <entryID1>(args) do
                // statements
            end;
        or
            accept <entryID2>(args) do
                // statements
            end;
        end select;
    end loop;
    // more statements
end <name>;
```

Example: resource allocator in Ada

```
task ResourceAllocator is
    entry ACQUIRE(args);
    entry RELEASE(args);
end;
task body ResourceAllocator is
    // declaration list of free units, pending queue etc.
begin
    loop
        select
            accept ACQUIRE(args) do
                // process the ACQUIRE request
            end;
        or
            accept RELEASE(args) do
                // process the RELEASE request
            end;
        end select;
    end loop;
end ResourceAllocator;
```

Example: resource allocator in Ada

For example, a call to an **`entry`** called `ACQUIRE` in a `task` called `ResourceAllocator` has the form:

```
call ResourceAllocator.ACQUIRE(args);
```

and is serviced by an **`accept`** statement in the body of `ResourceAllocator` of the form:

```
accept ACQUIRE(args) do
    // statements to process the ACQUIRE request
end;
```

The next lecture

Distributed processing in Java

Suggested reading:

- Andrews (2000), chapter 8;
- Java Tutorial Sockets and RMI