A Simple Model for Parallel and Distributed Algorithms

Jordi Bataller Mascarell

November 11, 2024

Abstract

In this article, we introduce a notation for describing algorithms accurately. Precise notations are necessary even if no formal proofs are required. Many algorithms, mainly parallel and distributed, are complex, and their textual descriptions are often unclear, if not misleading, making it difficult to translate them into programs faithfully. Moreover, potential bugs are challenging to spot, especially race conditions, even when an algorithm is transcripted into a programming language.

A key point to help solve all those issues is to identify the *atomic steps* of the algorithm. This is, what changes in the state are modeled by the algorithm to take place instantaneously.

Therefore, we suggest expressing algorithms as transition systems. Each transition, labeled with a postcondition and an effect, represents an atomic step the algorithm can take.

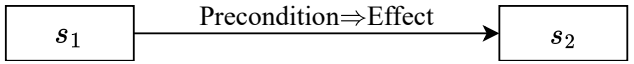


Figure 1: Step

1 How to Define Algorithms using Transition Systems

A transition system [Wik23] is a set of states along with a set of transitions defined on those states. Each transition represents a step that a thread [Wik24c] executing the algorithm can take. defines an atomic, i.e. instantaneus, transition from one state to another one. A step represents

- a change in the thread's own state –i.e., its program counter, and
- a change in the values of the variables available to the thread.

It is very convenient to depict a step as a directed edge of a graph, see figure

1. The edge is annotated with a precondition and an effect, both defined on the variables of the algorithm.

If the precondition holds for the state where a thread is on –the step is enabled, the transition may occur. If the step is taken, then the postcondition and change of state are immediate by definition. Note that the original thread's state and the precondition, combined, are the overall conditions for the step, while the effects on the variables and the new thread's state are the overall effects of the step.

A step is not guaranteed to happen as soon as it is enabled: other threads could also be allowed to take its own step at that moment.

An execution of an algorithm is a sequence of steps, implying that not two steps happen simultaneously. This is a realistic assumption even for parallel and distributed algorithms, for two main reasons:

- there is no global clock and, therefore, no way to tell which of two parallel steps taken by different threads happened before.
- if two steps are truly parallel, there will be executions where one appears before the other.

What step is executed when several ones are enabled is not a deterministic choice. In either case, an enabled step can't be delayed infinitely, and will eventually occur.

An execution of a parallel or distributed algorithm is difficult to understand if written as a sequence. It is much more advisable to depict an execution using several timelines, where changes in threads' states and variables can be shown more clearly. The caveat is not to put two steps (thread's state change) on the same vertical. Consider figure 2 for an example.

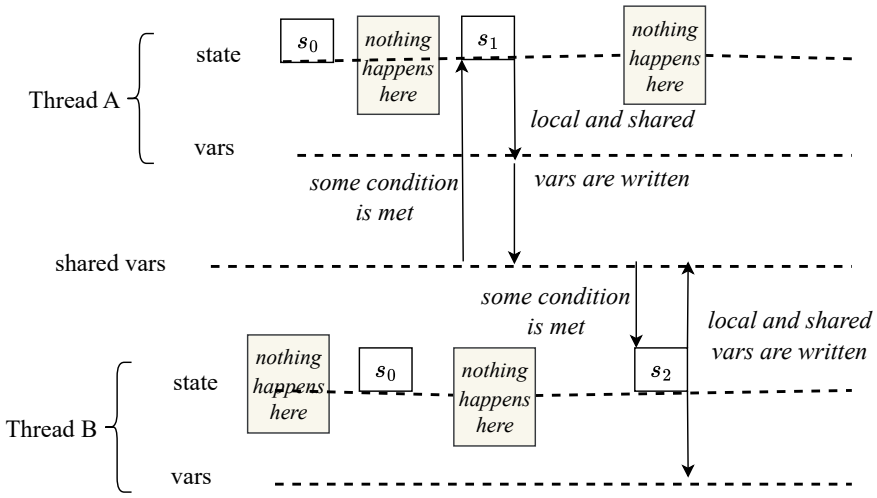


Figure 2: A parallel execution: $a.s_0, b.s_0, a.s_1, b.s_2$

In some algorithms, a thread can start up a new one. For such a case, the step is defined as in figure 3. There, $s_1 \to s_2$ is the step by the thread launching a second one, which starts in its s_0 state.

Reciprocally, two threads may join into one. Figure 4 shows how this is to be modeled.

Finally, we would like to stress that the key to modelling an algorithm is

- to define when and how the states and variables change.
- to specifiy what changes happen atomically.

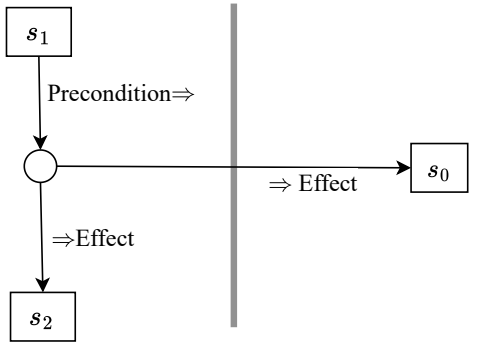
1.1 Parallel Algorithms

Parallel algorithms are algorithms designed to be executed by several threads that communicate and coordinate using shared variables. As an example, we show how the well-known Dijkstra's symmetrical algorithm for mutual exclusion, [Dij65], can be expressed using our notation. First off, let us present the algorithm in pseudocode.

Dijkstra's Mutual Exclusion Algorithm

Shared variables:
turn: $0, ..., N.\ turn \leftarrow 0$
flag: $[\{0, 1, 2\}]_{0..N}.\ flag[i] \leftarrow 0\ \forall i \in 0..N$

Algorithm por process $i \in \{1, ..., N\}$:



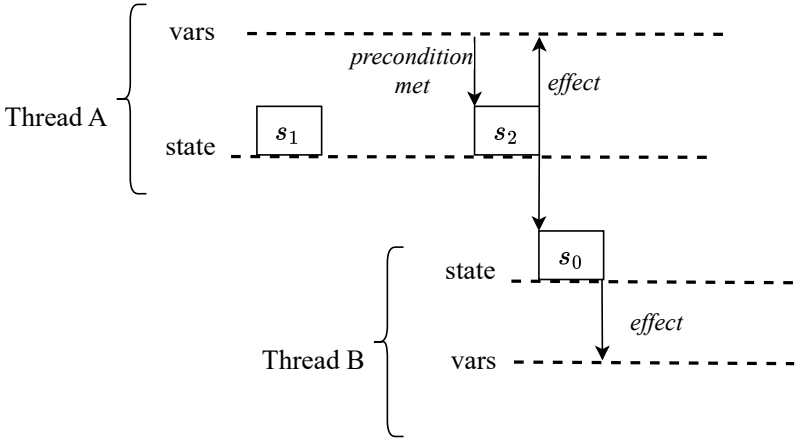


Figure 3: Starting up a thread.

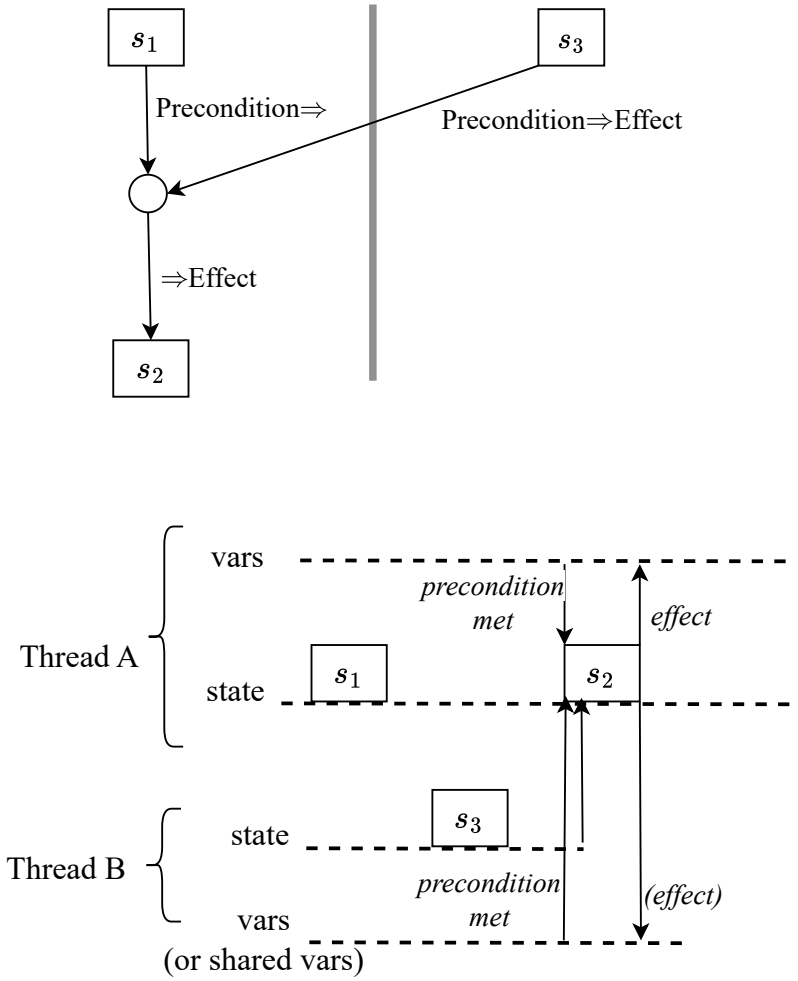


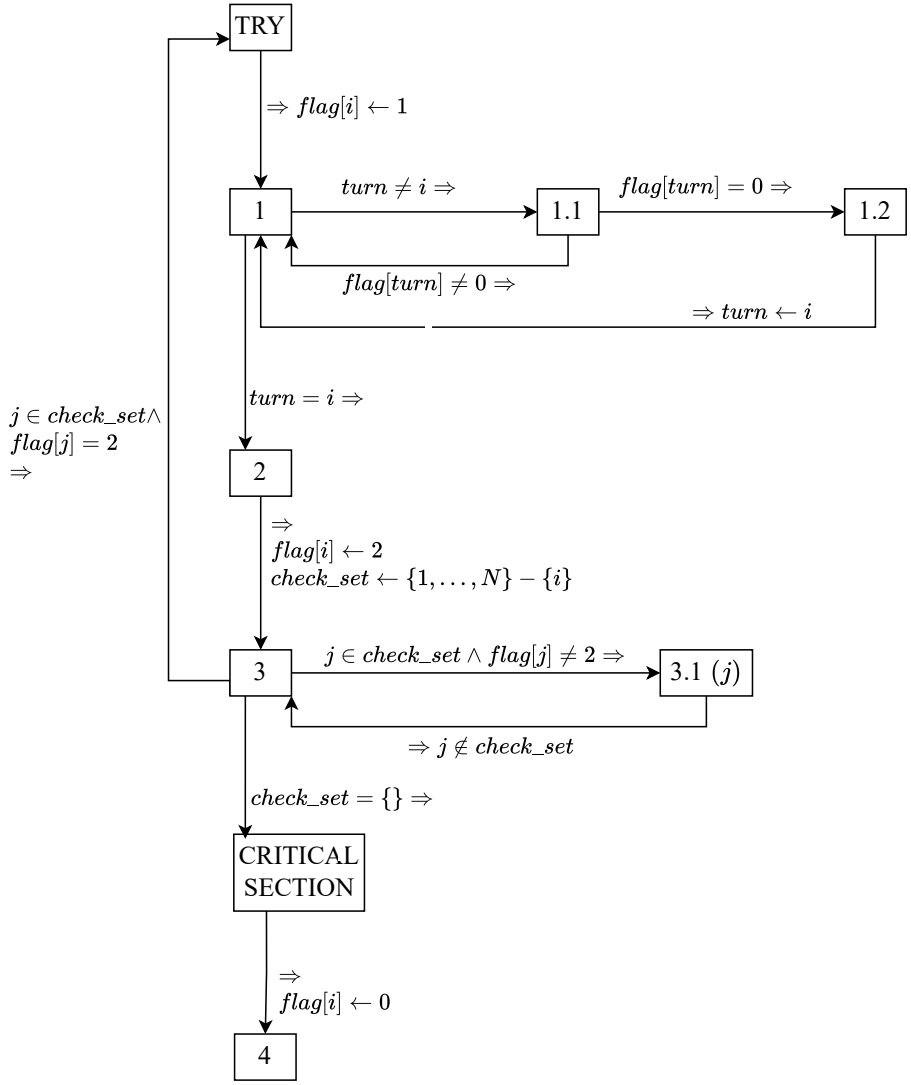
Figure 4: Two threads join.

TRY: $flag[i] \leftarrow 1$
while $turn \neq i$
    if $flag[turn] = 0 \Rightarrow$
        $turn \leftarrow i$
$flag[i] \leftarrow 2$
for $j \in \{1, ..., N\} - \{i\}$
    if $flag[j] = 2 \Rightarrow$
        go to TRY
CRITICAL SECTION
$flag[i] \leftarrow 0$

Despite being quite clear, the above algorithm fails to point out which are its atomic steps. For example, "if $flag[turn] = 0 \Rightarrow turn \leftarrow i$" is not meant to be a single step; otherwise, the second loop would be unnecessary. Two processes can simultaneously see that $flag[turn] = 0$ if one of them "rests" an instant on the condition. Later, both will do $turn \leftarrow i$. Even more, they could reach the second loop, which decides the thread that proceeds to the critical section.

Dijkstra's algorithm can be written with our notation in this way.

Dijkstra's Mutual Exclusion Algorithm



Our notation makes it much easier to show, for instance, that two processes might arrive at the second loop, having set *turn* to themselves. Consider the execution in figure 5.

1.2 Distributed Algorithms

Distributed algorithms are algorithms for several threads that communicate by exchanging messages; they don't share any variable. This type of encapsulated thread is better known as a process, [Wik24a]. However, modeling a process having several threads that share some variables could also be necessary. In this case, the algorithm must be precise about how many threads a process may contain and when a new thread is started.

It is worth noting that implementing a remote procedure call [Wik24b] library on top of a message passing system is simple and widely used for building distributed systems. Thus, a distributed algorithm may need, or be easier, to be expressed in terms of remote calls instead of messages. If this is the case, replicating the message exchange underlying the calls may be unnecessary. Remote calls are easily modeled in our notation by means of starting up a new thread.

In summary, we assume either of these options for inter-process communication in a distributed algorithm:

- exchanging messages
- calling remote funcions

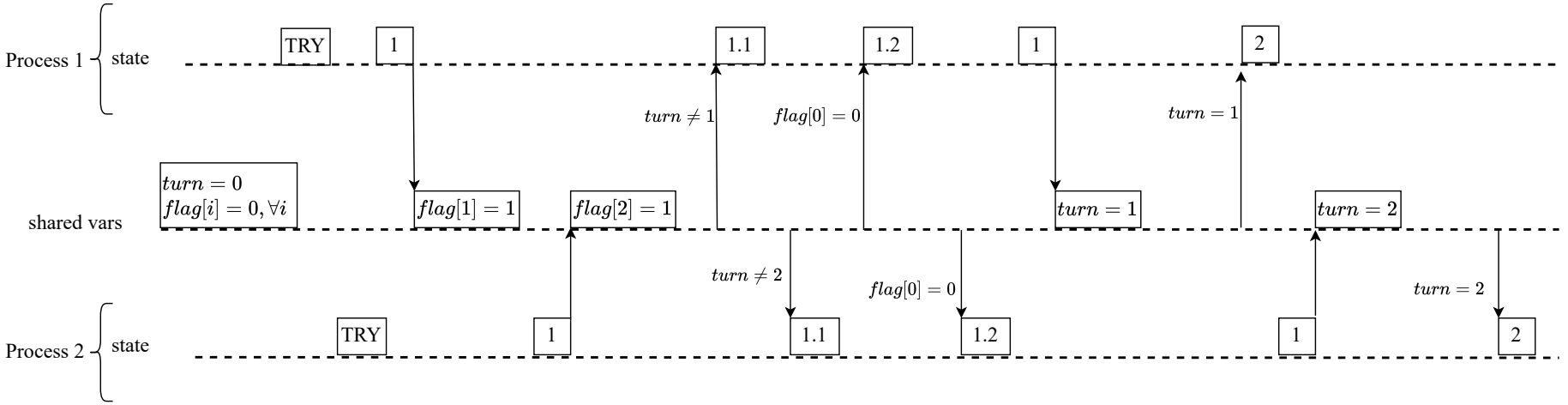


Figure 5: Execution of Dijkstra's Algorithm. Two processes make it to the second loop.

1.2.1 Distributed Algorithms Based on Messages

In order to model the dispatch of a message, we assume that every process

- has at least one mailbox.
- can add a message to every other process' mailbox

This way, the delivery of a message is modeled as the effect of a sender's step, which instantly adds the message to the receiver's inbox. This is not as unrealistic as it might seem. On the contrary, there is always a delay between the reception and the point when a thread starts processing a message. This delay captures, without loss of generality, the time of transit of a message between two processes. This is shown in figure 6.

This is an algorithm for mutual exclusion [RA81].

Ricart and Agrawala Mutual Exclusion Algorithm

All events are sorted using Lamport's clocks [Lam78].

- A process that wants to enter the critical section sends a TRY message to every other process.
- Upon receiving a TRY message:
  - If the receiver isn't trying to access the critical section, it replies OK.

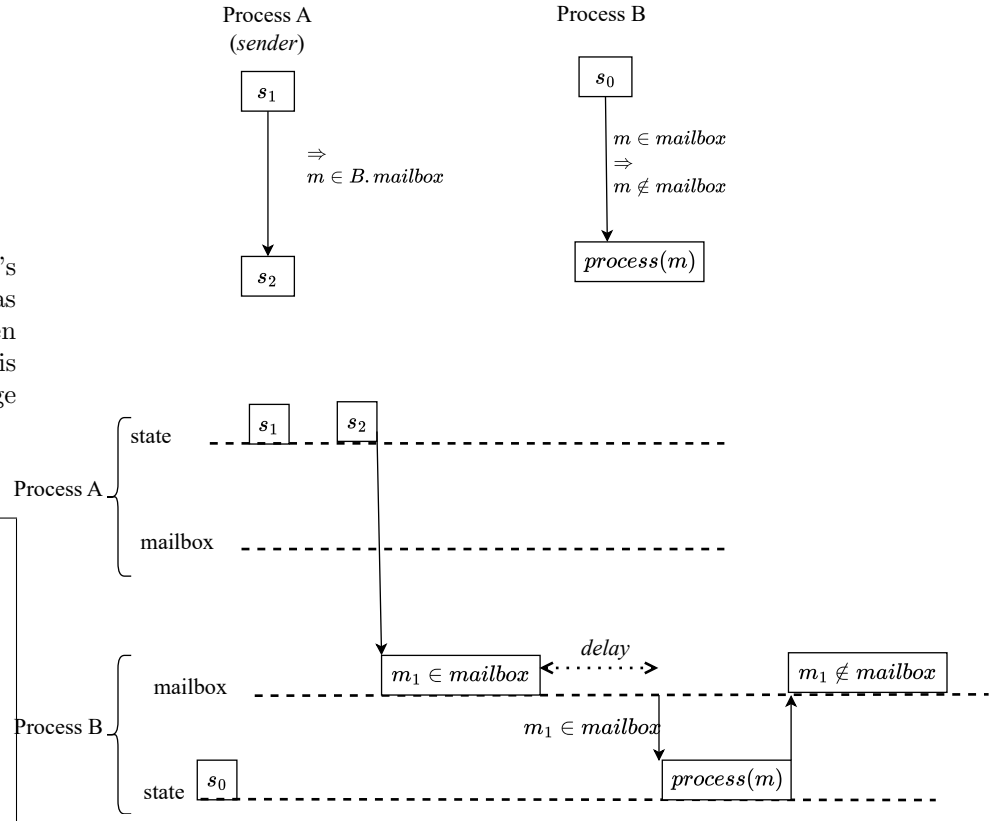


Figure 6: Distributed Algorithm. Sending and receiving a message.

  - If the receiver is in the critical section, it adds the request to the queue and doesn't reply.
  - If it is trying to access the critical section but not there yet, it compares the clock of its own TRY message with this one's. Earliest timestamp wins
    * If it loses, it replies OK.
    * If it wins, it adds the request to the queue and doesn't reply.
- A process enters the critical section when it gets everyone's OK.
- When a process exits the critical section, it sends an OK for each TRY message it queued.

This is how we model the algorithm.

Ricart and Agrawala Mutual Exclusion Algorithm

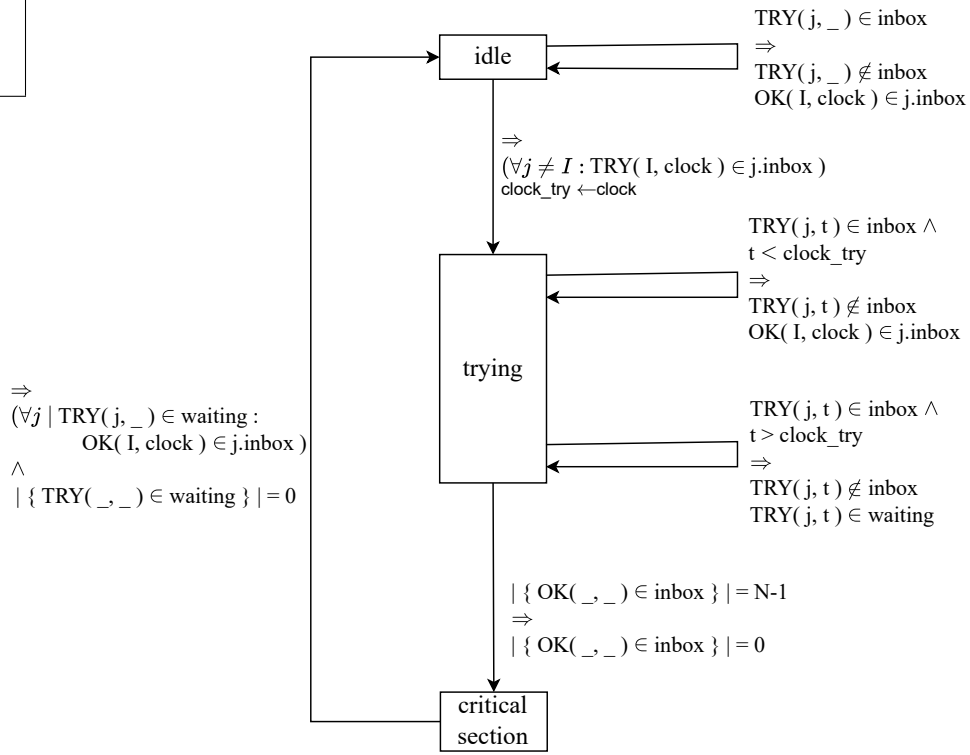
Types

- Process = {1, ..., N }
- Message = { OK(from: Process, clock: ℕ), TRY(from: Process, clock: ℕ) }
- Clock = (p: Process, t: ℕ) // *logical Lamport's clock, properly updated (not shown here)*

Variables for process I

inbox: { Message } ← ∅
clock: Clock ← $(I, 0)$
clock_try: Clock
waiting: { TRY(_,_) } ← ∅

Process I



1.2.2 Distributed Algorithms Based on Remote Function Calls

We have already shown how to represent the launch of a new thread. In this case, it would be started on a different process. Remarkably, our notation forces it to capture the asynchronous nature of remote calls. Figure 7 shows an example.

As an example, let's model the Berkeley algorithm [GZ89] for synchronizing the clocks of a group of processes.

Berkeley Algorithm

A leader process carries out the following steps when clocks need to be synchronized.

- Ask every follower what its *local* time is.
- For each answer, the one-way transmission time is estimated using the total round-trip time.
- The average of local times is calculated when all answers arrive.
- A δ_i value (positive or negative) for each follower is computed using the one-way lapse and the average time.
- Each process is sent its δ_i. Upon reception, it adds this value to its clock.

Figure 8 shows how we model the Berkeley algorithm.

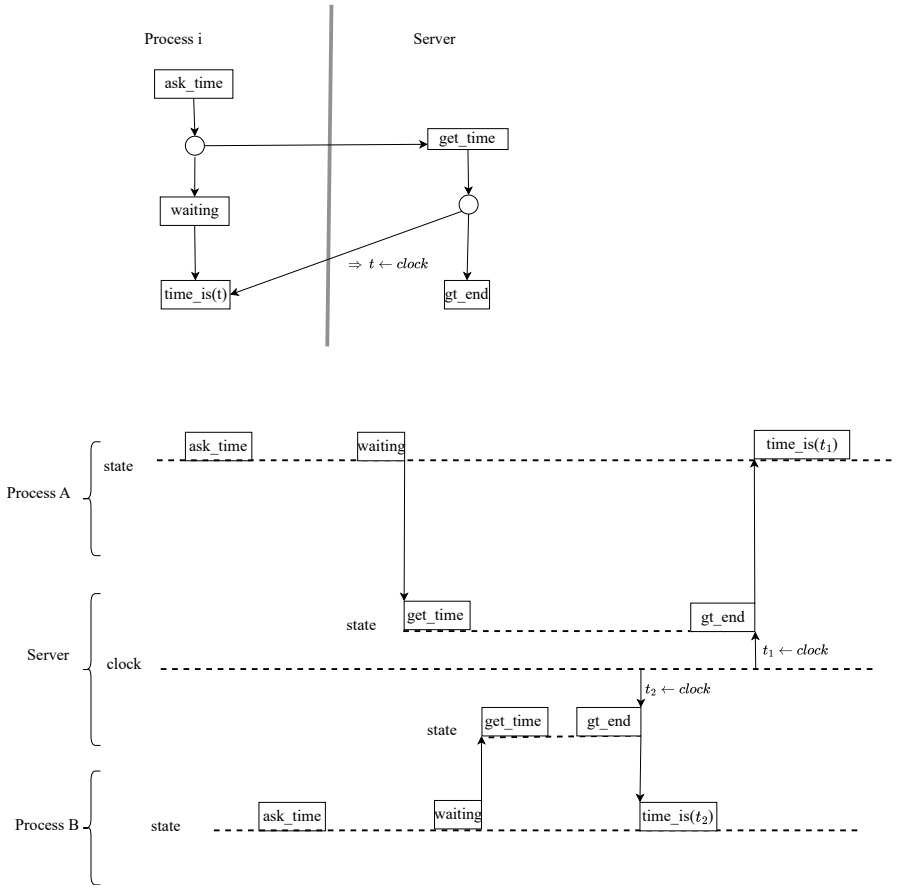


Figure 7: Distributed Algorithm. Remote call that launches a thread.

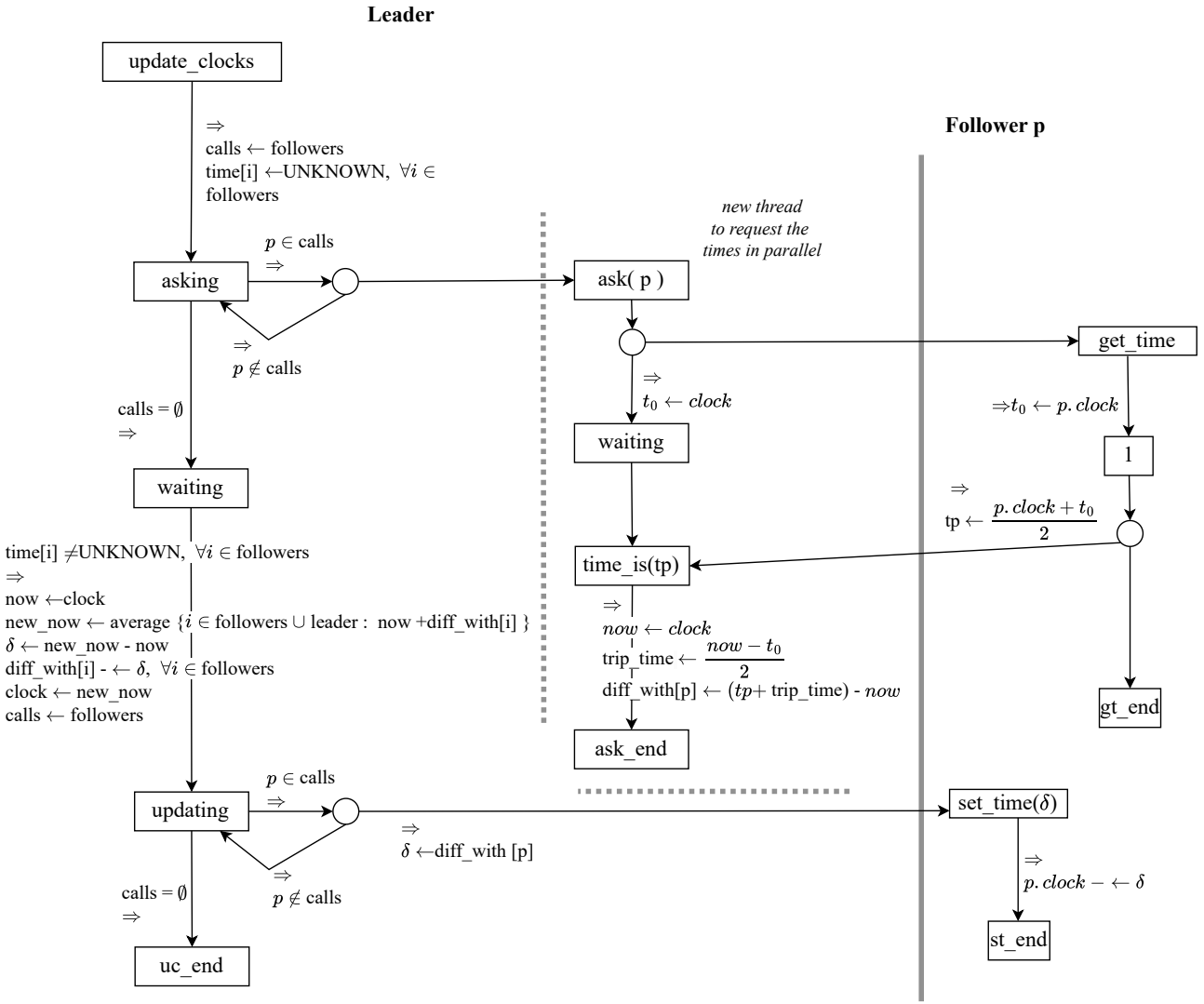


Figure 8: Berkeley Algorithm

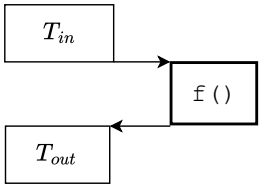


Figure 9: A function.

2 How to Design a System

A parallel or distributed system can comprise several components and combine different algorithms. In this case, using the algorithm's notation introduced before to express a system's architecture could lead to an over-complicated design with too many unnecessary details on this level. It is more convenient to have a way to express just the components that make up the system and how they interact.

In this section, we introduce a notation best suited to design the architecture of a system. It is based on the concept of function, which just transforms its input into an output, see figure 9. A function like that has no side effects nor does it make any assumptions about how data is passed in and out; it is just a mathematical definition of a function. For example, all the following source codes correspond, among many other possibilities in the same or different programming languages, with this function definition

$$\mathbb{R} \rightarrow multby2() \rightarrow \mathbb{R}$$

```
// R -> f() -> R in C/C++
double multby2( double a ) {
  return a * 2;
}

// R -> f() -> R in C/C++
void multby2( double * p ) {
  (*p) = (*p) * 2;
}

// R -> f() -> R in Java
void multby2( double a, double[] array ) {
  array[0] = a * 2;
}

// R -> f() -> R in Javascript
void multby2( a, callback ) {
   callback( a * 2 )
}
```

We regard a system as composed of "modules." A module aggregates functions that share the module's private variables. At least classes, objects, libraries, and processes fall under this definition of "module."

A module is simply depicted as a box, see figure 10. Its private components (i.e. variables and part of the functions) are to be shown completely inside. Public functions are to be shown half outside and half inside the box. A module's functions may access the private variables, then called methods, to read or change their values. In this case, that must be expressed:

- Read-only or const methods have an arrow pointing to the function from inside.

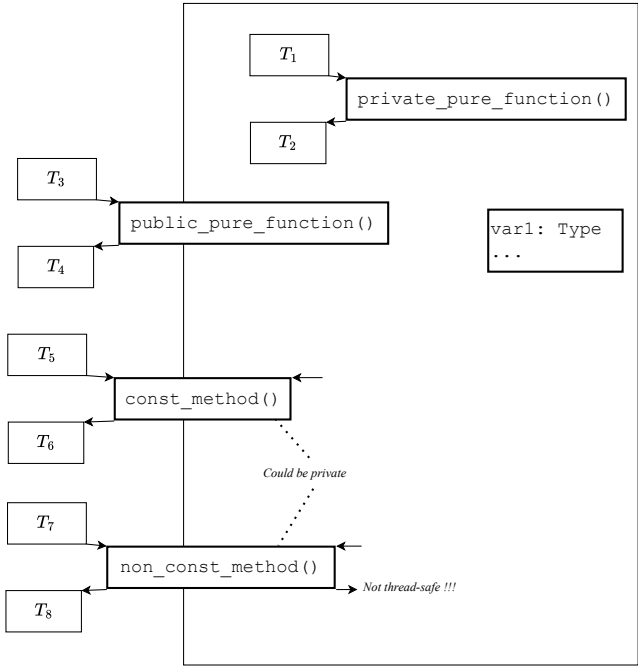


Figure 10: A module.

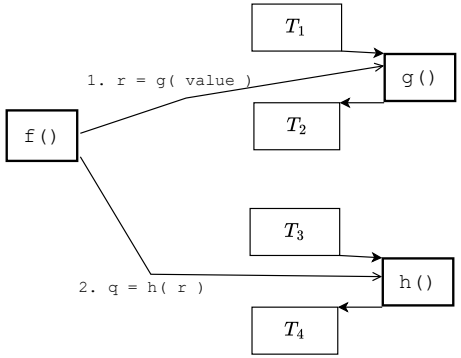


Figure 11: A algorithm with function calls.

- Non-const methods have an arrow pointing from the function to the inside.

Sometimes, an algorithm that involves calls between several methods or functions is required to be modeled. For such a case, we use an arrow to express each call. These arrows must be labeled with a sequence number and, optionally, the invocation; see figure 11.

Non-const methods must be carefully considered because, by definition, they have side effects on shared variables and, as a consequence, they are not "thread-safe". A significant advantage of our notation is how easy is to identify where race conditions may arise. Therefore, regarding threads, it is necessary to point out

- where a new thread starts, and

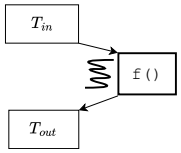


Figure 12: Function run by a new thread.

- how a non-const function, or several of them, must be made race condition-free.

A function executed by a new thread, different from the caller one, is marked as in figure 13. If needed in algorithms, we can explicitly write when a thread must be joined/waited; see figure 13.

2.1 Remote Functions

In the first stages of a distributed design, there is no problem in modeling a remote function, just like a regular synchronous function, i.e., figure 9.

But in a distributed system, a remote function call is always performed asynchronously from the point of view of the caller due to the time used to transmit underlying messages. Despite this, if the caller waits for the result, the general function design could be just fine. If this is not the case, we have at least two designs:

- Call and pick up the response (figure 14). This case is, in fact, doing a request and polling for the response.

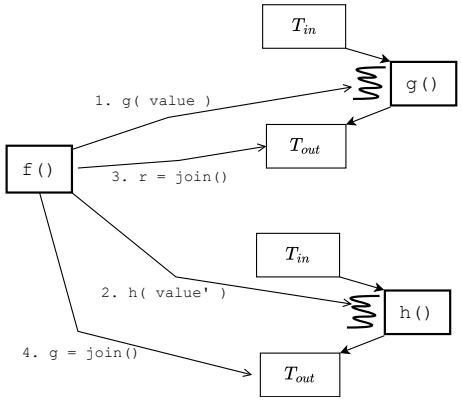


Figure 13: Joining threads.

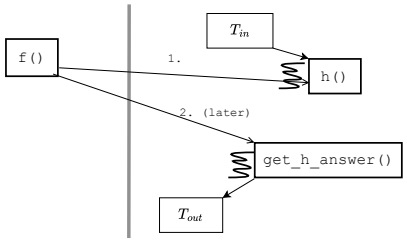


Figure 14: Call and pick up the response.

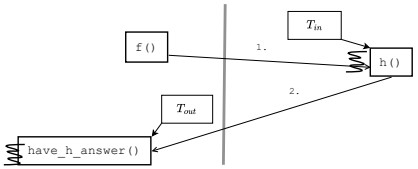


Figure 15: Call and wait for the response.

- Call and wait for the response (figure 15). This design can represent either the case that the communication is a request-response scheme with a callback to let the caller know when the response is available or the case that the callee can address the caller and call it to push the response.

2.2 Remote APIs

From just a client-server application to a microservices-based one, there are always remote APIs that must be modeled in a distributed system (figure 16).

This design is comunication agnostic. There are two main ways to implement it, taking into account communications. The first is to develop a proxy of the logic for the client, figure 17.

If the logic's algorithms are complex, this approach is advantageous because they will be implemented once. On the other hand, the proxy logic must be rewritten for each client's programming language, but it is much simpler since it only deals with communication.

A service's logic often relies on another piece, such as a database, an event store, or some other middleware, which offers a programming interface where

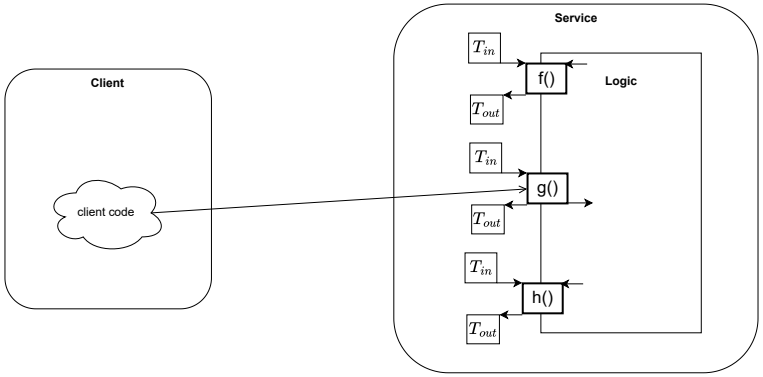


Figure 16: Remote API.

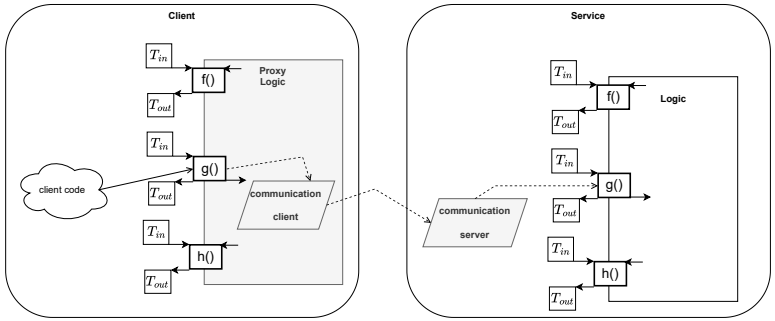


Figure 17: Proxy of a Remote Logic

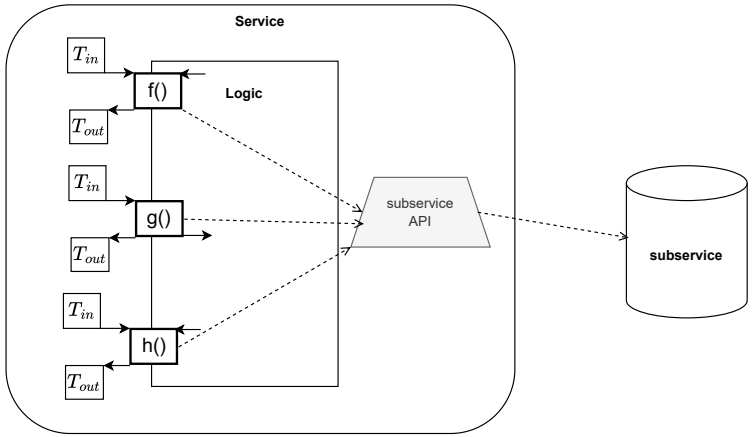


Figure 18: Logic Based on a Sub-service

communications are hidden (figure 18).

For this scenario, we should consider whether we are unnecessarily doing the work of implementing a proxy for our logic. This would be the case when all the client code is written in the same language or when our logic's algorithms are reasonably straightforward. Therefore, if we develop a library for the client code which comprises the actual logic we'll have a "fat client", figure 19. A benefit of this approach is that the clients carry out a significant part of the computations, improving the system's scalability.

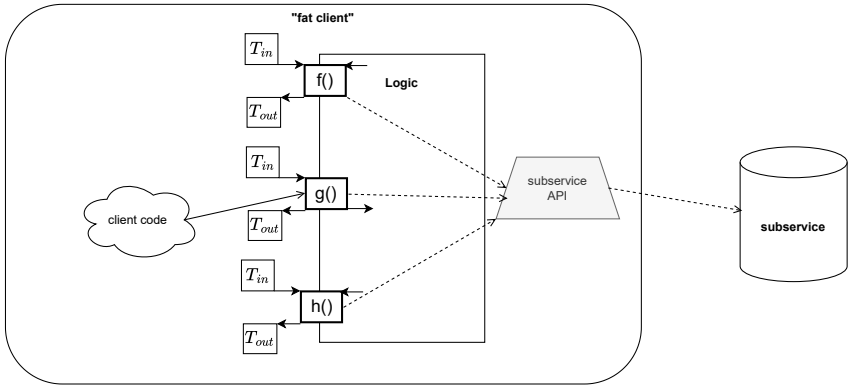


Figure 19: Fat Client

2.3 A Word on Pure Functions

A fundamental rule in programming is never to mix computations with input/output. This is, a function that performs a calculation must depend only on its input parameters and provide a result to its caller without any side-effect. Not following this principle leads to messes, confusion, and duplicate code. Distributed algorithms further aggravate it.

When we develop the logic of a service, which in turn is based on a sub-servic such as a database, we often fail to realize that the accesses to the sub-servic are actually input/out. Consequently, we should keep the algorithm for th computation separate from the accesses to the sub-service. Figure 20 show how this must be designed.

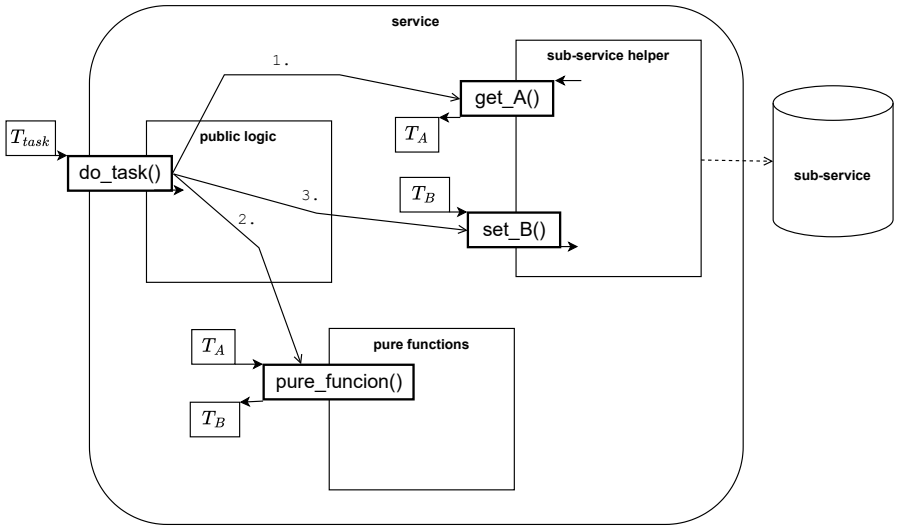


Figure 20: A Logic with Pure Functions

References

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, sep 1965.

[GZ89] R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3bsd. *IEEE Transactions on Software Engineering*, 15(7):847–853, 1989.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[RA81] Glenn Ricart and Ashok K Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[Wik23] Wikipedia contributors. Transition system — Wikipedia, the free encyclopedia, 2023. [Online; accessed 17-May-2024].

[Wik24a] Wikipedia contributors. Process — Wikipedia, the free encyclopedia, 2024. [Online; accessed 17-May-2024].

[Wik24b] Wikipedia contributors. Remote procedure call — Wikipedia, the free encyclopedia, 2024. [Online; accessed 17-May-2024].

[Wik24c] Wikipedia contributors. Thread — Wikipedia, the free encyclopedia, 2024. [Online; accessed 17-May-2024].