Algoritmos e Sistemas Distribuídos

João Carlos Antunes Leitão
NOVA Laboratory for Computer Science and Informatics (NOVA LINCS)
and
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade NOVA de Lisboa

20^{th} September 2019

1 Introduction

This document provides a reference for the contents discussed in the Lab 1 of Algoritmos e Sistemas Distribuídos(*Algoritmos e Sistemas Distribuídos*). The goal of this document is not to completely cover all details discussed in the class (in particular, it fully omits the correctness arguments on the algorithms that materialize the link abstractions presented in Section 3). It however provides a complete list of the operators and notations used to write pseudo-course in the context of the class, with the link abstraction algorithms serving as practical examples

2 Writing Pseudo-Code

This section covers the essential aspects, in an (almost) exhaustive way, regarding the pseudo-code notation being employed in the context of the course. For completeness we now provide a brief discussion of why it is relevant to use a concrete notation when writing pseudo-code.

2.1 Motivation

Pseudo-code is a typical representation used to describe algorithms, and particularly distributed algorithms. The prime advantage of using pseudo-code lies in both in conciseness (short descriptions) and its completeness (the complete algorithm can be specified). Pseudo-code, when written in a clear and adequate way allows for experts to look at the algorithms and build either informal or formal proofs of their correctness (which is based on specific properties that the algorithm should strive to meet, as well as any properties of the underlying algorithm if one exists[1].
There is no *standard* or *globally acceptable* notation for writing pseudo-code. Different authors (and hence different pieces of literature) do tend to use relatively different notations. With some authors even using multiple variants of their own particular style of notation. However, in the context of an advanced course where pseudo-code will be often presented to and written by students, it becomes relevant to have a standard unified notation, as to simplify the readability of algorithms by students (and by professors).

2.2 Basic Structure

In Algoritmos e Sistemas Distribuídoswe are using a notation based on events. So most of your pseudo-code will be writing event handlers, which in a nut-shell, specify what is the behavior of the algorithm when an event happens. Furthermore we assume that event handlers are atomic, in the sense that even if another event is triggered locally, its processing will only begin after finishing the processing of the event being currently processed. Notice that although event handlers are processed atomically, a process can fail during it.

[1] Your professor is actually one such person, that tends to use different notations among different documents, see for instance [4, 3, 5, 1]. Some liberties are sometimes acceptable in the name of conciseness and readability.

Algorithm 1: Basic Structure of an Algorithm

```
//The interface of the algorithm
```

Interface:

This is an optional section of an algorithm that should be present when the interface of the algorithm is not specified somewhere else, or when the algorithm has multiple requests *and* indications.

Bellow, we provide an example of a (generic) request and indication each with two arguments: arg_1 *and* arg_2 *respectively.*

Requests:

request (arg_1, arg_2)

Indications:

indication (arg_1, arg_2)

```
//State of the algorithm
```

State:

Here is where you define variables (§ 2.4) that maintain the state of the algorithm. If the algorithm does not require to maintain state, this section can be omitted.

Usually each variable is defined by providing a name followed by a comment to explain its goal, for instance to define a variable msgs *that maintains a collection of delivered messages you would write:*

msgs `//Set of messages that were already delivered.`

```
//Now you define the event handlers:
```

Upon Init do:

Event handlers (§ 2.3) are denoted by the special keyword ***Upon*** *followed by the name of the event being handled, and any arguments that the event might have associated (referenced between parenthesis) and finally the special keyword* ***do*** *followed by two dots.*

The special ***Init*** *event is assumed to be automatically triggered when the algorithm starts. This is where usually you will initialize variables or setup timers(§ 2.8) that are used by the algorithm independently of any external event.*

```
//You also might have some procedures
```

Procedure exampleProcedure ( )

Procedures (§ 2.7) are functions that you can call at any point of your pseudo-code that usually manipulate the state of the algorithm. Their goal is to reduce the number of lines that you have to write to describe your algorithm.

Algorithm 1 denotes the fundamental components of an algorithm. There is an initial (and many times omitted) component where the *interface* of the algorithm is specified. This implies providing the information for the name (and arguments if any) of all requests and indications of the algorithm. You can omit this section when the interface of the algorithm is simple and obvious, or when the algorithm addresses a well known problem (e.g, any variant of the *broadcast* problem).
Then the *state* component, where state variables for the algorithm are listed (and commented). This section should be omitted if the algorithm has no need to maintain state (although this is unlikely with the exception of the most simple algorithms).
Finally, you define the logic of the algorithm, through a set of event handlers (of which, the *Init* event is a special case) and, if required, a set of *procedures* that are used by your event handlers.

2.3 Upon and Trigger

Event handlers serve to specify the logic of an algorithm when an event is triggered locally. These events can have multiple sources. They can be a request from a some other algorithm, or they can be the indication of an algorithm being employed in the design of the algorithm. They can also represent the reception of a message from the Network (when algorithms are designed to interact directly with the Network layer without resorting to any abstraction (see § 3), or they can serve to handle an internal timer event used by the protocol (besides the special event *Init* which is triggered when the algorithm is initialized).
A special event that can also be relevant in the design of algorithms under very concrete fault models and synchrony model, is the **Crash** event, that notifies an algorithm that a remote process has crashed.
The start of an event handler is denoted, generally, using the following form (where the number of arguments can be any positive number n):

Upon EventName (arg$_1$, ... arg$_n$) do:

Conversely, an algorithm might need to trigger events itself. The events triggered by an algorithm can be requests on other algorithms being employed in its design, its own indications, or even its own requests. When designing an algorithm that interacts directly with the Network layer, one can also trigger the transmission of a message (equivalent to asking the operative system to send a message through the `socket` interface for instance).
Generally, the trigger of an event is denoted in the following way (where arguments should match the ordering of the arguments of the event handler):

Trigger EventName (arg$_1$, ... arg$_n$);

Below you can find a list of all types of event handlers signatures and their corresponding triggers (in these examples we assume that there are always two arguments arg_1 and arg_2).

Special Init event:

Upon Init (arg$_1$, arg$_2$) do:

(Notice that there is no trigger associated with this event, since this is assumed to be performed by an evolving runtime when the algorithm is initialized).

Algorithm requests and indications:

Upon EventName (arg$_1$, arg$_2$) do:

Trigger EventName (arg$_1$, arg$_2$);

Message Reception and Transmission:

Upon Receive (MessageType, s, arg$_1$, arg$_2$) do:

Trigger Send (MessageType, p, arg$_1$, arg$_2$);

In this case, there is a special meaning for the arguments s and p. For the receive event, s will denote the identity of the process that originally sent the message whereas in the send event p denotes the identity of the process that should receive the message.

Timer events:

Upon Timer TimeName ($\mathbf{arg_1}$, $\mathbf{arg_2}$) do:

(Timer are usually not explicitly triggered, they are setup at some point by the algorithm as shown in § 2.8.)

Crash events:

Upon Crash (p) do:

In this particular case the crash notification will not (in general) contain any other information with the exception of the identity of the process that was detected as crashed (denoted by p), and hence p will be the only argument of this type of event.

2.4 Variables and Special Operators

Variables can be denoted by any name. State variables that store state known to the algorithm among different handlers must be listed in the *State* section of the algorithm, their scope is the whole algorithm. In contrast, variables that are not listed in the *State* section will have a scope (i.e, be valid) only within the handler or procedure where they are first referred.
While variables do not explicitly have a type (a common practice of most programming languages with known exceptions such as Javascript), it is a good practice to have variables representing the same information in their relevant scopes, and hence maintain a single *type*.
To modify the value of a variable, one usually uses an arrow that point from the value to be attributed to the variable to the variable whose value is being defined. For instance, to attribute the value 3 to variable v you should write:

$$v \longleftarrow 3;$$

A special value can be attributed to any variable that indicated that the variable has no value defined. This value (which is semantically equivalent to the *null* value in programming languages such as C or Java) is denoted by the symbol $\perp$ (known as bottom). As an example, imagine that you want variable v to have an undefined value. In that case, you should write:

$$v \longleftarrow \perp;$$

A variable can be a set (i.e, a collection of values). A set is usually initialized being attributing the empty set value to a variable:

$$set \longleftarrow \{\}\,;$$

A set can also be used as a map. This is common for instance when you want to maintain information (imagine a set of message identifiers) for each process in the system[2]. To do so, you initialize a set as described above, and then refer to the value of each entry independently. For instance to atribute an empty set on the entry of a map relative to process p you would do:

$$map[p] \longleftarrow \{\}\,;$$

Whereas to refer to that value you would use:

$$map[p];$$

It is common to assume that any entry of a map that was not previously defined has the special value $\perp$.

2.5 If and Forall/Foreach Operator

As in any modern programming language you can use the *if*, *then*, *else* keywords to define branching decisions in your algorithm.
Algorithm 2 denotes the usage of these operators.

[2]Remember that on algorithms that assume the global membership of the system to be known, the set of all process identifiers is denoted by Π.

Algorithm 2: If, Then, Else Example

```
//Usage of If, Then, Else
```

If (*some boolean condition*) **then**
    Statements for the if branch.
Else If (*some other boolean condition*) **then**
    Statements for the else if branch.
Else
    Statements for the else branch.

Boolean conditions in these statements should resort to logical operators. In particular logical *and* is represented by $\wedge$ while the logical *or* is represented by $\vee$. Negation is achieved by the $\not{o}$perator. Equality is evaluated through the $=$ operator. Inequality can be expressed by either $\neq$ or $\neg$.
When iterating over a set of elements one resorts to either the *foreach* or *forall* operators. Both are equivalent and used in exactly the same way. Algorithm 6 provides an example of the use of this operator, where a variable containing a map is initialized and then, for each process in the system, an empty set is associated in the map variable.

Algorithm 3: Foreach Example

```
//Usage of Foreach operator
```

map $\longleftarrow \{\}$;
Foreach $p \in \Pi$ **do**:
    map[p] $\longleftarrow \{\}$;

2.6 Set Operators

Since many algorithms require the manipulation of sets for maintaining relevant information about the (local) state of the algorithm we now briefly remind the reader of the operator used to manipulate a set. To this end we will use the variable *set* (occasionally set_1 and set_2 when we need two sets) that represents the set being manipulated, and the variable x (and occasionally y) to represent an element.
The union between two sets is represented by:

$$set_1 \cup set_2;$$

The intersection between two sets is represented by:

$$set_1 \cap set_2;$$

To add an element to a set we write:

$$set \longleftarrow set \cup \{x\};$$

Sometimes it is useful to add multiple elements (what is called usually a tuple) to the set. Imagine that you want to add a tuple composed of x and y to a set. In that case we write:

$$set \longleftarrow set \cup \{(x, y)\};$$

To remove an element x from a set one writes:

$$set \longleftarrow set \backslash x;$$

To verify if an element x is within a set (also used to iterate across all element of a set) one writes:

$$x \in set;$$

Conversely to verify that an element x does not exists in a set:

$$x \notin set;$$

2.7 Procedures

In Section 2.2 we have already discussed how to define a procedure. We however have not discussed how to indicate in the pseudo-code when to execute a procedure. Notice that procedure execution happens within the atomic context of the handler that executes it.
Assuming that we have a procedure with the name *ProcedureName* with two arguments arg_1 and arg_2 respectively, and that the procedure does not returns any value (through the use of the **Return** command[3]) or that the return value is to be ignored, we execute a procedure by using the **Call** command as follows:

Call ProcedureName (arg_1**,** arg_2 **);**

If however the procedure does have a return value that should be stored in a variables, say var, then we denote the execution of the procedure as follows:

$var \longleftarrow$ **ProcedureName (** arg_1**,** arg_2 **);**

2.8 Timers

Timers are very useful constructs in the design of algorithms, since they allow algorithms to execute pre-defined tasks independently of external events (i.e, without requiring the reception of an indication form another algorithm or receiving a message from another process). For instance, if you consider the design of the TCP protocol, it uses internal timers to trigger the re-transmission of messages that were not acknowledged by a remote process and that might have been lost by the network.
While in Section 2.2 we have already discussed how to handle a timer in the context of an event handler (triggering of timers are themselves events), we have not yet discussed how to setup a timer nor cancel a timer.
First of all one should notice that for convenience (and to make pseudo-code listings shorter) we consider two different types of timers. Simple timers that only trigger once after some time t and then never trigger again (except if the algorithms explicitly setup the timer again within the context of the timer handler), and periodic timers, that trigger continually every time interval t until the timer is explicitly canceled.
Timers, similarly to any event that can be explicitly triggered by an algorithm, can have arguments that are defined when the timer is setup, and are then provided to the timer handler when it triggers. Additionally, when setting up a timer, an extra argument t denoting how long after the setup should the timer trigger is also provided. This value is often not part of the arguments in corresponding handler (although it can be if required).
To setup a simple timer (one that expires once) one uses the keywords *Setup Timer*. Conversely, to setup a periodic timer one uses the keywords *Setup Periodic Timer*. Since an algorithm can resort to multiple different timers to control different aspects of its operation, Timers have usually an associated name.
To illustrate the action of defining a timer, we now provide two examples of setting up a simple and a periodic timer named, respectively, *STimer* and *PTimer*. Both timers are configured with the same time t, and both receive two arguments: arg_1 and arg_2, respectively.

Setup Timer STimer (t**,** arg_1**,** arg_2 **);**

Setup Periodic Timer PTimer (t**,** arg_1**,** arg_2 **);**
In some cases, it might be useful to cancel a timer, ensuring that it will no longer trigger. To understand why think of the TCP example again. When a message is sent, a timer is setup to trigger the re-transmission of the message if

[3] We have omitted the discussion of this operator since its usage is the same as in well known programming languages.

no acknowledgement message is receive from the remote process. however, if such a message is received before the timer triggers, the re-transmission is no longer required, and hence the timer can be safely canceled.
To denote in pseudo-code that a timer is being canceled one uses the *Cancel* keyword, followed by the Timer name and, if multiple timers with the same name have been (or could have been) setup previously with different arguments, then the arguments should also be indicated as to clearly identify the timers being affected.
Considering the previous example, and assuming that the Periodic Timer was setup only once, once could represent that timer being canceled by writing:

Cancel Timer PTimer;
Since there was only a PTimer, this is enough to clearly denote what is the timer being canceled. However, not imagine that at different points of an algorithm multiple timers of type STimer were setup using different values for the arguments arg_1 and arg_2. In that case, to avoid ambiguity when canceling one particular timer, one would also indicate the values for those arguments, as in:

Cancel Timer STimer (arg_1**,** arg_2 **);**
Notice that as in the timer handler the special argument t is omitted. In fact, most of the times, t will only be relevant when setting up the timer.

3 Link Abstractions Algorithms

We now illustrate the use of pseudo-code to specify the algorithms that materialize the *stubborn point-to-point link* abstraction (§ 3.2) on top of the weaker abstraction of *fair loss point-to-point link* abstraction (whose interface and algorithm is presented in § 3.1).
Finally we show how to define the algorithm for materializing the abstraction of the *perfect point-to-point link* on top of the previous one.
The algorithms (and specifications) presented are adapted from [2].

3.1 Fair Loss Point-to-Point Link

Specification

FL1 (Fair-Loss): Considering two correct processes i and j; if i sends a message m to j infinite times, then j delivers m infinite times.

FL2 (Finite Duplication): Considering two correct processes i and j; if i sends a message m to j a finite number of times, then j cannot deliver m infinite times.

FL3 (No Creation): If a correct process j delivers a message m, then m was sent to j by some process i.

Algorithm 4: Fair Loss Point-to-Point Link

Interface:
Requests:
fp2pSend (p**,** m **)**
Indications:
fp2pDeliver (s**,** m **)**

Upon fp2pSend(p**,** m **) do:**
Trigger Send (DATA, p**,** m **);**

Upon Receive (DATA, s**,** m **) do:**
Trigger fp2pDeliver (s**,** m **);**

3.2 Stubborn Point-to-Point Link

Specification

SL1 (Stubborn Delivery): Considering two correct processes i and j; if i sends a message m to j, then j delivers m an infinite number of times.

SL2 (No Creation): If a correct process j delivers a message m, then m was sent to j by some process i.

Algorithm 5: Stubborn Point-to-Point Link

Interface:
Requests:
sp2pSend (p**,** m **)**
Indications:
sp2pDeliver (s**,** m **)**

State:
sent `//set of sent messages`

Upon Init do:
sent $\longleftarrow \{\}$;
Setup Periodic Timer (ReSend, t**);**

Upon sp2pSend(p**,** m **) do:**
Trigger fp2pSend (p**,** m **);**
sent $\longleftarrow$ sent $\cup \{(p, m)\}$;

Upon fp2pDeliver (s**,** m **) do:**
Trigger sp2pDeliver (s**,** m **);**

Upon ReSend ( ) do:
foreach $(p, m) \in$ Sent **do:**
Trigger fp2pSend (p**,** m **);**

3.3 Perfect Point-to-Point Link

Specification

PL1 (Reliable Delivery): Considering two correct processes i and j; if i sends a message m to j, then j **eventually**[4] delivers m.

PL2 (No Duplication): No message is delivered by a process more than once.

PL3 (No Creation): If a correct process j delivers a message m, then m was sent to j by some process i.

Algorithm 6: Perfect Point-to-Point Link

Interface:
Requests:
pp2pSend (p**,** m **)**
Indications:
pp2pDeliver (s**,** m **)**

State:
delivered `//set of delivered messages`

Upon Init do:
delivered $\longleftarrow \{\}$;

Upon pp2pSend(p**,** m **) do:**
Trigger sp2pSend (p**,** m **);**

Upon sp2pDeliver (s**,** m **) do:**
If $m \notin$ delivered **do:**
delivered $\longleftarrow$ delivered $\cup \{m\}$;
Trigger pp2pDeliver (s**,** m **);**

3.4 Making this Practical

Evidently, the presented algorithms are not practical, in the sense that it is hard to imagine that a process would be required to infinitely transmit a message just to ensure that it does reach the other process, or that a process must retain memory of all messages that it has observed and delivered at some point to avoid delivering the same message multiple times.
However, one should note that, in some (maybe non-trivial) sense, TCP that we use now-a-days without even thinking about it, do make use of some of these techniques to achieve its guarantees (which are not the ones provided by the perfect point-to-point link abstraction). TCP however deals with the limitations discussed above by employing what is actually sophisticated techniques.
In particular, and in a simplistic way that does not align with the precision that is often required in the context of Algoritmos e Sistemas Distribuídos, TCP avoids infinite re-transmissions by using a acknowledgment mechanism, that informs the sender of the messages that have already been successfully delivered by the received (and this even uses a piggyback mechanism for further efficiency).
While the mapping of the acknowledgment system with avoiding infinite re-transmissions might be more straightforward, the mechanism used by TCP to deal with the memory consumption on the received side to avoid delivery of the same message multiple times, and on the sender side to deal with the need to maintain messages in memory for

[4] Remember that in English, this means that something will happen for certain however at an unknown time in the future.

possible re-transmission in more nuanced. However, the clever reader will figure out that the use of transmission and reception windows by TCP actually deals (in part) with such problems.
Evidently, having a clear and correct algorithm to address a problem is an essential aspect of building a system, and to that end pseudo-code will be a very useful toll both to specify, reason about, and even explain your solution to someone. Implementing it however, will almost for certain require the use of tricks and dealing with practical aspects that are not the priority when designing the algorithm. This said, a well written pseudo-code is an invaluable tool when one needs to implement an algorithm, and with time, the materialization (implementation-wise) of some common algorithmic patterns will be obvious for the competent systems engineer.

References

[1] M. Ferreira, J. Leitão, and L. Rodrigues. Thicket: A protocol for building and maintaining multiple trees in a p2p overlay. In *Proceedings of the 29th IEEE International Symposium on Reliable Distributed Systems*, New Delhi, India, Oct. 2010.

[2] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[3] J. Leitão, J. Pereira, and L. Rodrigues. Epidemic broadcast trees. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 301 – 310, Beijing, China, Oct. 2007.

[4] J. Leitão, J. Pereira, and L. Rodrigues. Hyparview: a membership protocol for reliable gossip-based broadcast. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 419–429, Edinburgh, UK, June 2007.

[5] J. Leitão, R. van Renesse, and L. Rodrigues. Balancing gossip exchanges in networks with firewalls. In *Proceedings of the 9th International Workshop on Peer-to-Peer Systems (IPTPS '10)*, San Jose, CA, U.S.A., 2010.