Logical Clocks Are Not Fair: What Is Fair? A Case Study of High-level Language and Optimization

Yanhong A. Liu
Computer Science Department, Stony Brook University
Stony Brook, New York, USA
liu@cs.stonybrook.edu

ABSTRACT

This paper describes the use of a high-level, precise, and executable language, DistAlgo, for expressing, understanding, running, optimizing, and improving distributed algorithms, through the study of Lamport's algorithm for distributed mutual exclusion. We show how a simplified algorithm, reached by several rounds of better understanding and improvement of the original algorithm, leads to further simplification and improved understanding of fairness. This allows us to use any ordering for fairness, including improved fairness for granting requests in the order in which they are made, over using logical clock values. This leads to the discovery that logical clocks are not fair in general.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; **Distributed programming languages**;

KEYWORDS

high-level language, logical clock, mutual exclusion, fairness

ACM Reference Format:
Yanhong A. Liu. 2018. Logical Clocks Are Not Fair: What Is Fair? A Case Study of High-level Language and Optimization . In *ApPLIED '18: Advanced tools, programming languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems, July 27, 2018, Egham, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3231104.3231109

1 INTRODUCTION

Distributed algorithms carry out the logic of distributed systems. From distributed control such as distributed consensus to distributed data such as distributed hash tables, the underlying algorithms dictate the correctness and efficiency of distributed systems.

For better understanding, evaluation, and improvement of distributed algorithms, their precise implementation and execution (or specification and simulation) are not only important, but essential, in dealing with the myriad of complex interactions that can arise in real-world distributed systems.

Many languages and tools have been proposed, for actual implementations, e.g., Argus [17], Emerald [4], and Erlang [16], as well as for formal specifications, e.g., TLA [15] and IOA [11, 26]. A recent language, DistAlgo [23], combines the advantages of existing languages: high-level as in pseudocode languages, precise as in specification languages, and directly executable as in programming languages.

This paper describes the use of DistAlgo for expressing, understanding, running, optimizing, and improving distributed algorithms, through the study of Lamport's algorithm for distributed mutual exclusion. We show how a simplified algorithm, reached by several rounds of better understanding and improvement of the original algorithm, leads to further simplification and improved understanding of fairness. This allows us to use any ordering for fairness, including improved fairness for granting requests in the order in which they are made, over using logical clock values. This leads to the discovery that logical clocks are not fair in general.

DistAlgo has been used to implement a wide variety of well-known distributed algorithms and protocols in our research, as well as the core of many distributed systems and services in dozens of different course projects by hundreds of students; some of these are summarized previously [23]. However, this does not mean that algorithm designers see the value of using DistAlgo in algorithm design. The main contribution of this paper is showing, through a case study, that

1) distributed algorithms can be expressed precisely, at the same high level as English descriptions or pseudocode, in an executable language and be run directly,
2) clear and precise algorithm specifications and language optimizations can lead to improved algorithms, and
3) the improvement led to the discovery that ordering using logical clock values is not fair when fairness requires that requests be granted in the order in which they are made.

Logical clocks are proposed in Lamport's seminal paper [13], which also describes the use of logical clocks with an algorithm for distributed mutual exclusion. Logical clocks are used for ordering of events in distributed systems, to overcome the lack of synchronized real-time clocks or physical clocks. Distributed mutual exclusion is for multiple processes to access a shared resource mutually exclusively. Lamport's algorithm was designed to guarantee that access to the resource is granted in the order in which requests are made, and the order was determined using logical clock values.

Both logical clocks and distributed mutual exclusion have since been studied extensively, especially with use of logical timestamps for ordering of events and guaranteeing fairness, as discussed in Section 7. To the best of our knowledge, no prior work presented improved fairness for the required request ordering, or showed that using logical timestamps is not fair for the required ordering.

ApPLIED '18, July 27, 2018, Egham, United Kingdom

ACM ISBN 978-1-4503-5775-3/18/07…$15.00
https://doi.org/10.1145/3231104.3231109

In particular, we show that use of DistAlgo allows us to easily remove unnecessary use of logical times, and then remove logical clocks altogether, and use instead any desired ordering directly and exactly. As a side result of this, we show that logical clocks are not fair in general. These results are built on top of earlier optimization and improvement of Lamport's algorithm that led us to remove unnecessary enqueue and dequeue of each process itself and to replace use of queues needing complex dynamic data structures with use of sets needing only simple counts and bits [24].

The rest of this paper is organized as follows. Section 2 describes Lamport's logical clocks and distributed mutual exclusion algorithm. Section 3 summarizes DistAlgo and its use in specification and simplification of Lamport's distributed mutual exclusion algorithm. Section 4 describes elimination of unnecessary use of logical times. Section 5 presents different notions of fairness using different orderings. Section 6 shows why logical clocks are not fair in general. Section 7 discusses related work and concludes.

2 LOGICAL CLOCKS AND DISTRIBUTED MUTUAL EXCLUSION

Lamport [13] proposes logical clocks and describes an algorithm for distributed mutual exclusion.

Lamport's logical clocks

A system consists of a set of processes. Each process consists of a sequence of events. Sending or receiving a message is an event.

The observable ordering of events in a system is captured by the "happened before" relation, $\rightarrow$, the smallest relation satisfying:

1) if a and b are events in the same process, and a comes before b, then $a \rightarrow b$,
2) if a is the sending of a message by one process and b is the receipt of the message by another process, then $a \rightarrow b$, and
3) if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Logical clocks implement the "happened before" relation by defining a clock C_i for each process P_i such that, for any events a in P_i and b in P_j, if $a \rightarrow b$, then C_i's value of a is smaller than C_j's value of b. Two implementation rules are used:

1. Each process P_i increments C_i's value between any two successive events.
2. When sending a message m by process P_i, m contains a timestamp that is C_i's value of the sending event. Upon receiving m by process P_j, C_j's value is set to be greater than or equal to its current value and greater than the timestamp in m.

A total order of events is obtained by ordering pairs of logical timestamp of an event and process id where the event happens, in lexical order. That is, process ids are used to break ties in logical clock values.

Lamport's distributed mutual exclusion

The problem is that n processes access a shared resource, and need to access it mutually exclusively, in what is called a critical section, i.e., there can be at most one process in a critical section at a time. There is a fairness requirement, stated in [13] as:

> Different requests for the resource must be granted in the order in which they are made. (⋆)

Lamport's algorithm assumes that communication channels are reliable and first-in-first-out (FIFO).

Figure 1 contains Lamport's original description of the algorithm, except with the notation $<$ instead of $\Rightarrow$ in rule 5 (for comparing pairs of logical time and process id using lexical ordering: `(t,p)` $<$ `(t2,p2)` iff `t` $<$ `t2` or `t` $=$ `t2` and `p` $<$ `p2`) and with the word "acknowledgment" added in rule 5 (for simplicity when omitting a commonly omitted [9, 26] small optimization mentioned in a footnote). This description is the most authoritative, is at a high level, and uses the most precise English we found.

> The algorithm is then defined by the following five rules. For convenience, the actions defined by each rule are assumed to form a single event.
>
> 1. To request the resource, process P_i sends the message T_m:P_i *requests resource* to every other process, and puts that message on its request queue, where T_m is the timestamp of the message.
> 2. When process P_j receives the message T_m:P_i *requests resource*, it places it on its request queue and sends a (timestamped) acknowledgment message to P_i.
> 3. To release the resource, process P_i removes any T_m:P_i *requests resource* message from its request queue and sends a (timestamped) P_i *releases resource* message to every other process.
> 4. When process P_j receives a P_i *releases resource* message, it removes any T_m:P_i *requests resource* message from its request queue.
> 5. Process P_i is granted the resource when the following two conditions are satisfied: (i) There is a T_m:P_i *requests resource* message in its request queue which is ordered before any other request in its queue by the relation $<$. (To define the relation $<$ for messages, we identify a message with the event of sending it.) (ii) P_i has received an acknowledgment message from every other process timestamped later than T_m.
>
> Note that conditions (i) and (ii) of rule 5 are tested locally by P_i.

Figure 1: Original description in English.

The algorithm is safe in that at most one process can be in a critical section at a time. It is live in that some process will be in a critical section if there are requests. It is fair in that requests are granted in the order of $<$, on pairs of logical time and process id, of the requests. Its message complexity is $3(n-1)$ in that $3(n-1)$ messages are required to serve each request.

3 LANGUAGE AND OPTIMIZATION

Liu et al. [23, 24] proposes DistAlgo, a language for high-level, precise, executable specifications of distributed algorithms, and studies its use for specification, implementation, optimization, and simplification, with Lamport's distributed mutual exclusion as an example.

DistAlgo, a language for distributed algorithms

For expressing distributed algorithms at a high level, DistAlgo supports four main concepts by building on an object-oriented programming language, Python: (1) distributed processes that can send messages, (2) control flow for handling received messages, (3) high-level queries for synchronization conditions, and (4) configuration for setting up and running. DistAlgo is specified precisely by a formal operational semantics [23].

(1) Distributed processes that can send messages. A type P of processes is defined by

`process` P`:` *stmt*

The body *stmt* may contain, among usual definitions,

- a `setup` definition for setting up the values used the process,
- a `run` definition for running the main flow of the process, and
- `receive` definitions for handling received messages.

A process can refer to itself as `self`. Expression `self.`*attr* (or *attr* when there is no ambiguity) refers to the value of *attr* in the process.

- ps `:=` n `new` P creates n new processes of type P, and assigns the new processes to ps.
- ps`.setup(`*args*`)` sets up processes ps using values of *args*.
- ps`.start()` starts `run` of ps.

`new` can have an additional clause, `at` *node*, specifying remote nodes where the created processes will run; the default is the local node.

A process can easily send a message m to processes ps:

`send` m `to` ps

(2) Control flow for handling received messages. Received messages can be handled both asynchronously, using `receive` definitions, and synchronously, using `await` statements.

- A `receive` definition is of the following form:

  `receive` m `from` p`:` *stmt*

  It handles, at yield points, un-handled messages that match m from p. A yield point is of the form `--` l, where l is a label. There is an implicit yield point before each `await` statement, for handling messages while waiting. The `from` clause is optional.

- An `await` statement is of the following form:

  `await` $cond_1$`:` $stmt_1$ `or ... or` $cond_k$`:` $stmt_k$ `timeout` t`:` *stmt*

  It waits for one of $cond_1$, ..., $cond_k$ to be true or a timeout after period t, and then nondeterministically selects one of $stmt_1$, ..., $stmt_k$, *stmt* whose conditions are true to execute. Each branch is optional. So is the statement in `await` with a single branch.

(3) High-level queries for synchronization conditions. High-level queries can be used over message histories, and patterns can be used to match messages.

- Histories of messages sent and received by a process are kept in `sent` and `received`, respectively. `sent` is updated at each `send` statement, by adding each message sent. `received` is updated at the next yield point if there are un-handled messages, by adding un-handled messages before executing all matching `receive` definitions.

  Expression `sent` m `to` p is equivalent to m `to` p `in sent`. It returns true iff a message that matches m `to` p is in `sent`. The `to` clause is optional. Expression `received` m `from` p is similar.

- A pattern can be used to match a message, in `sent` and `received`, and by a `receive` definition. A constant value, such as `"release"`, or a previously bound variable, indicated with prefix `=`, in the pattern must match the corresponding components of the message. An underscore `_` matches anything. Previously unbound variables in the pattern are bound to the corresponding components in the matched message.

  For example, `received("release",t3,=p2)` matches every triple in `received` whose first component is `"release"` and third component is the value of `p2`, and binds `t3` to the second component.

A query can be an existential or universal quantification, a comprehension, or an aggregation over sets or sequences.

- An existential quantification and a universal quantification are of the following two forms, respectively:

  `some` v_1 `in` s_1`, ...,` v_k `in` s_k `has` *cond*
  `each` v_1 `in` s_1`, ...,` v_k `in` s_k `has` *cond*

  They return true iff for some or each, respectively, combination of values of variables that satisfies all v_i `in` s_i clauses, *cond* holds.

- A comprehension is of the following form:

  {e`:` v_1 `in` s_1`, ...,` v_k `in` s_k`,` *cond*}

  It returns the set of values of e for all combinations of values of variables that satisfy all v_i `in` s_i clauses and condition *cond*.

- An aggregation is of the form *agg* s, where *agg* is an aggregation operator such as `count` or `max`. It returns the value of applying *agg* to the set value of `s`.

- In all query forms above, each v_i can be a pattern.

Other operations on sets can also be used, e.g., s_1 `+` s_2 returns the union of sets s_1 and s_2.

(4) Configuration for setting up and running. Configuration for requirements such as use of logical clocks and use of reliable and FIFO channels can be specified in a `main` definition. For example, `configure clock = Lamport` specifies that Lamport's logical clocks are used; it configures sending and receiving of a message to update the clock value, and defines a function `logical_time()` that returns the clock value.

Specification, execution, optimization, and simplification

Distributed algorithms can be expressed in DistAlgo precisely and at a high level, and be executed directly. The executable specifications are actual implementations that can be drastically optimized using a systematic method based on incrementalization [18, 20, 21, 29]. Precise high-level specification and systematic incrementalization have allowed us to discover simplifications and even higher-level specifications of distributed algorithms [22, 23].

For Lamport's algorithm for distributed mutual exclusion in Figure 1, repeated improvements to high-level specification and systematic incrementalization led to the following results:

Original. The original algorithm can be expressed in DistAlgo at the same high level as Lamport's English description in Figure 1, except that operations of both P_i and P_j are expressed as operations of a process P.

Send-to-self. It is easy to see that, in rules 1 and 3 in Figure 1, P_i need not enqueue or dequeue its own request, but just send request and release messages to all processes including itself. The enqueue and dequeue are taken care of by rules 2 and 4 when it receives a message from itself.

Inc-with-queue. Expensive conditions (i) and (ii) in rule 5 in Figure 1 can be optimized by incrementally maintaining their truth values as messages are sent and received, including using a dynamic queue data structure for (i) for comparison with earliest of other requests.

Ignore-self. Discovered in the result of Inc-with-queue, in rules 1 and 3 in Figure 1, P_i need not enqueue or dequeue its own request or send request and release messages to itself, but just send to others. Condition (i) in rule 5 compares only with other requests anyway.

Inc-without-queue. Expensive condition (i) in rule 5 in Figure 1 can be better optimized, when incrementally maintaining its truth value, by using just a count of requests earlier than the process's own request and using a bit for each process if messages can be duplicated.

Simplified. Discovered with both Inc-with-queue and Inc-without-queue, condition (i) in rule 5 in Figure 1 can just compare with any request for which a corresponding release has not been received, omitting all updates of queue in rules 1-4, yielding a higher-level specification than Original.

The precise programs for Original, Inc-with-queue, Inc-without-queue, and Simplified are given in [23]. The program for Simplified is shown in Figure 2, including configuring and running 50 processes.

```
1 process P:
2   def setup(s):                 # take set of other procs
3     self.s := s

4   def mutex(task):
5     -- request
6     self.t := logical_time()                      # 1
7     send ("request", t, self) to s                # 1
8     await each received("request", t2, p2) has    # 5(i)
            (not some received("release",t3,=p2) has t3>t2
             implies (t,self) < (t2,p2))
9           and each p2 in s has                    # 5(ii)
              some received("ack", t2, =p2) has t2 > t
10    task()
11    -- release
12    send ("release", logical_time(), self) to s   # 3

13  receive ("request", _, p2):                     # 2
14    send ("ack", logical_time(), self) to p2      # 2

15  def run():                    # main flow of proc
16    def task(): output(self)    # define task for mutex
17    mutex(task)                 # use mutex to do task

18 def main():                    # main of application
19   configure clock = Lamport    # use Lamport clock
20   configure channel = {reliable, fifo}
                                  # use reliable FIFO channels
21   ps := 50 new P               # create 50 P procs
22   for p in ps: p.setup(ps-{p}) # pass other procs to each
23   for p in ps: p.start()       # start run of each proc
```

Figure 2: Simplified algorithm (lines 4-14) in a complete program in DistAlgo. The comments on lines 6-9 and 12-14 indicate the rule numbers and conditions in Figure 1.

4 REMOVING UNNECESSARY USE OF LOGICAL TIMES

Use of logical clock times requires calls to `logical_time()`. Excessive use of logical times may make an algorithm more complex than necessary. We show how incrementalization allows us to remove unnecessary uses.

Consider the program Simplified in Figure 2. Systematic incrementalization can derive from it the same optimized programs Inc-with-queue and Inc-without-queue as from Original. However, systematic incrementalization can allow one to easily see that incrementalization would be simpler and yield simpler optimized programs if logical times are not used for acknowledgment and release messages.

In particular, in Figure 2, a release message corresponding to a request message is recognized by having a larger timestamp (`t3` such that `t3 > t2`, on line 8), instead of having simply the same timestamp (`t2`). So is an acknowledgment message (by having `t2` such that `t2 > t`, on line 9, instead of simply `t`). The latter expresses condition (ii) in rule 5 in Figure 1. The former imitates the latter for condition (i) in Simplified.

Therefore, the program Simplified can be further simplified: a release message sent on line 12 can use the same time, `t`, as the request message; and an acknowledgment on line 14 can use the same time as the request received on line 13, instead of not using that time, as described in rule 2 of Figure 1. Precisely, Figure 2 can be simplified as follows.

1. Replace `logical_time()` on line 12 with `t`, and replace `some received("release", t3, =p2) has t3 > t2` on line 8 with `received("release", t2, p2)`.
2. Replace `_` on line 13 with a fresh variable, say `t2`, replace `logical_time()` on line 14 with `t2`, and replace `some received("ack", t2, =p2) has t2 > t` on line 9 with `received("ack", t, p2)`.

This yields the further simplified algorithm in Figure 3. This replaced two existential quantifications with two constant-time tests. Therefore, the conditions are not only simpler, but more efficient even if executed without optimization.

5 REQUEST ID, REQUEST ORDER, AND FAIRNESS

Simplified use of logical times makes the essence of the ordering clear. Understanding the essence of ordering allows easy use of different orderings and better orderings. More importantly, it allows better characterization of fairness.

Consider the further simplified algorithm in Figure 3. The request time `self.t := logical_time()` on line 6 is the only use of logical time. It is then easy to see that the request time is used for two different purposes:

1) comparison as part of < for ordering the requests, on line 8 in Figure 3, for condition (i) in rule 5 in Figure 1, and

```
1 process P:
2   def setup(s):                  # take set of other procs
3     self.s := s

4   def mutex(task):
5     -- request
6     self.t := logical_time()                          # 1
7     send ("request", t, self) to s                    # 1
8     await each received("request", t2, p2) has        # 5(i)
              (not [received("release", t2, p2)]
               implies (t,self) < (t2,p2))
9           and each p2 in s has                        # 5(ii)
              [received("ack", t, p2)]
10    task()
11    -- release
12    send ("release", [t], self) to s                  # 3

13  receive ("request", [t2], p2):                      # 2
14    send ("ack", [t2], self) to p2                    # 2
```

Figure 3: Further simplified algorithm in DistAlgo. The boxes indicate the changed parts from Figure 2. Definitions of `run` and `main` are as in Figure 2.

2) identifying a request on line 7, and its corresponding release and acknowledgment messages on lines 12 and 14, used in lines 8-9 for conditions (i) and (ii) in rule 5.

Any other measure that can fulfill these two purposes can be used to replace the request time to give a different order in which the requests are granted.

Request id for request order

For identifying a request, a request id can use any value that is different from previous request id values of the same process. It does not even have to be larger for later requests.

For ordering the requests using a request id, the id just needs to support an order-comparison operation. The order that requests are granted depends on whether they are sequential or concurrent, as discussed below.

Let R_1 and R_2 be requests made by two different processes P_1 and P_2, respectively.

Sequential requests. We say that R_1 and R_2 are sequential with R_1 before R_2 if R_1 is granted before R_2 is received by P_1.
We assert that R_1 is granted before R_2 is granted. This is because (1) as given, R_1 is granted before P_1 has received R_2, (2) by condition (ii) in rule 5, R_2 can only be granted after all processes, including P_1, have received R_2 and acknowledged it, and (3) by transitivity using (1) and (2), R_1 is granted before R_2 is granted.

Concurrent requests. We say that R_1 and R_2 are concurrent if they are not sequential.
We assert that concurrent requests are granted in the order of smaller request id (paired with process id) first. This is because (1) as given, R_1 and R_2 are concurrent, i.e., R_1 is not granted before P_1 has received R_2, and symmetrically, R_2 is not granted before P_2 has received R_1, (2) by rule 2, upon P_1 receiving R_2, R_2 is among pending requests at P_1, and symmetrically, upon P_2 receiving R_1, R_1 is among pending requests at P_2, (3) by (1) and (2), both R_1 and R_2 will be among pending requests at both P_1 and at P_2 before either is granted, and (4) by condition (i) in rule 5, when R_1 and R_2 are both among pending requests, only the one with the smaller request id (paired with process id) can be granted first.

That is, two sequential requests with R_1 before R_2 will be granted in the order of R_1 before R_2, regardless of the request id ordering. The request id (paired with process id) comparison is only used to order concurrent requests.

Ordering by per-process count

A simplest request id that also gives a simplest ordering is a local request count in each process, by (1) setting up `self.t` to be `0` in `setup`, i.e., adding the following after line 3 in `setup`:

```
self.t := 0
```

and (2) incrementing `t` before sending it in a request, i.e., replacing line 6 with the following:

```
t := t + 1
```

This ordering means that, among concurrent requests from different processes, a request from a process for which a smaller number of requests have been granted will be granted earlier, regardless of any order in which the requests are made.

Ordering after locally observable requests

A simplest request id that also gives a simplest ordering that attempts to follow the order in which requests are made among concurrent requests is to let a new request of a process be after all requests that the process has received, by (1) setting up the following, as above, after line 3 in `setup`:

```
self.t := 0
```

and (2) incrementing `t` to be after itself and the id values of all received requests, i.e., replacing line 6 with the following:

```
t := max ({t} + {t: received("requests",t,_)}) + 1
```

This ordering uses only id values of directly received requests. A request by a process P_1 might not have been directly received by P_2 before a request by P_2 is made, but the request by P_1 may have been received by a third process P_3 which then sends a non-request message that is received by P_2 before the request by P_2 is made. That is, the request by P_1 is made before the request by P_2 and should be granted first. Ordering after locally received requests does not obey such transitive ordering that needs to be observed globally.

Ordering after globally observable requests

In general, there may be a chain of events in between a request by P_1 and a request by P_2 to observe globally that the request by P_1 is made before the request by P_2. Precisely, the fairness requirement ($\star$) requires that

> if request R_1 "happened before" request R_2, i.e., $R_1 \rightarrow R_2$, then R_1 be granted before R_2, and as for logical clocks, process ids be used to break ties.

Note that this global ordering using $\rightarrow$ requires the use of all messages, even if it is to order only requests. To observe this ordering, non-request messages need to pass on the id values of request messages transitively, by sending any directly observed request id value

and the id values transitively received in all messages. To optimize, only the maximum of these id values need to be sent.

While this maximum id value gives the desired ordering, it conflicts with the simplification in Section 4 that acknowledgment and release messages use the id value of the corresponding request. There are two simple solutions to this:

1. Add a fourth component in messages (only needed for acknowledgment and release messages), so the two different values (maximum id value for request order comparison, and id value of corresponding request for identification) are in two different components.
2. Use only the maximum id value in messages, and recognize corresponding acknowledgment and release messages as having a larger id value, as in Figure 2.

We show the program for the second solution, for easy comparison with Figure 2; it is straightforward to construct the program for the first solution in a similar way. Precisely, in Figure 2, add the following in `setup` as before:

```
self.t := 0
```

replace line 6 with the following that increments `t` to be after itself and id values in all received messages, not just received requests:

```
t := max ({t} + {t: received(_,t,_)}) + 1
```

and replace `logical_time()` on lines 12 and 14 with the following that passes on the maximum id value:

```
max ({t} + {t: received(_,t,_)})
```

The expression for aggregation with `max` can be optimized by incrementally maintaining its value as messages are received [18, 23]. Precisely, in Figure 2, add the following in `setup` as before:

```
self.t := 0
```

replace line 6 with the following:

```
t := t + 1
```

replace `logical_time()` on lines 12 and 14 with `t`, and add the following `receive` definition:

```
received (_,t2,_):
  t := max(t,t2)
```

It is easy to see that `t` is a specialized logical time that increments only when sending a request. This ordering ensures that any request that is globally observable to be before another request is granted before the other request. Thus, ordering after globally observable requests paired with process ids satisfies the fairness requirement (★).

What is fair?

Any ordering on requests gives a kind of fairness in the sense of that ordering. We summarize the different kinds of fairness for the three orderings discussed: per-process counts, locally observable requests, and globally observable requests.

- All three orderings are fair in that sequential requests are granted in their sequential order, and concurrent requests are granted in the order of smaller request id first.
- The last two orderings are fair in that all requests are granted in the order of smaller request id first, and that a request made after receiving another request is not granted before the other request. The first ordering does not satisfy this.
- The last ordering is fair in that a request made after a chain of events starting with another request is not granted before the other request, which satisfies the fairness requirement (★). The first two orderings do not satisfy this.

In general, one may even have request ids be generated and compared by a separate protocol, supporting more complex fairness schemes, for example, priority by payment, or simply an oracle.

6 LOGICAL CLOCKS ARE NOT FAIR

Clearly, ordering after globally observable requests differs from ordering using logical clock values. Looking into the difference easily leads to the discovery that ordering by logical clock values is not fair in general.

The idea is that logical clocks can increment values at events where the clock value should not be incremented for a desired ordering requirement. Using such values for ordering may violate the desired ordering requirement, such as ordering by requests only. Note that these events must still be used to determine the request ordering by taking the maximum of id values and passing it on, but just that they should not increment the id value.

Consider a small example, with three processes P_1, P_2, and P_3, all starting at logical time 0, and P_1 has a smaller id than P_2.

1. Suppose P_3 requests first, is granted, and then sends a release to P_1 and P_2. Before receiving the release from P_3, and having each only received a request from P_3 and sent back an acknowledgment, P_1 and P_2 have the same logical time.
2. Suppose after P_3 is granted, P_1 and P_2 both request, and there is no chain of events between the two requests to tell their order. Because P_1 has a smaller id than P_2, P_1's request should be granted first. This is the order that satisfies the fairness requirement (★).
3. Suppose, however, that the release from P_3 is received by P_1 before P_1's request and is received by P_2 after P_2's request. Then P_1's request will have a larger timestamp than P_2's request according to logical clocks, meaning that P_2's request will be granted first. This conflicts with the order that satisfies the fairness requirement (★).

That is, the requests by P_1 and P_2 happened concurrently, but the logical timestamp of the release message to P_1 that does not respect the order of requests makes P_1 be treated unfairly.

In general, there can be many different kinds of events and messages. Fairness should be specified and implemented based on the desired requirements exactly. Blind use of logical clocks can be a source of not only inefficiency, but also unfairness.

7 RELATED WORK AND CONCLUSION

Beyond pseudocode notations or English, a wide spectrum of languages and tools have been developed and used for implementation and specification of distributed algorithms: from programming languages with networking libraries, messaging interfaces, remote calls, asynchronous support, built-in processes, etc., to formal specification languages based on state machines, process algebras, logic rules, and more. A few examples are given in Section 1. More extensive related works are discussed in [23]. DistAlgo is unique in that it allows algorithms to be expressed at the same high level as

pseudocode and English, yet is precise and directly executable, and thus allows much easier understanding of the algorithms.

Logical clocks and distributed mutual exclusion have been studied and discussed extensively, in books, e.g., [8–10, 12, 26, 32, 33], and formal specifications, e.g., [26, p. 646-649; 14; 28]. Since Lamport's algorithm, many other algorithms for distributed mutual exclusion have been developed, e.g., [10, 31, 34–36, 38], to remove the requirement of FIFO channels, reduce the number of messages, etc. Logical clocks have been used particularly to provide fairness, i.e., serving requests in the order in which they are made, e.g., [25, 34]. However, only the order of logical timestamps is used. Since Lamport's clock, more sophisticated logical clocks have been developed, e.g., [1, 2, 7, 27]. A scheme is also proposed to avoid unfairness in using pairs of logical timestamps and process ids, by dynamically changing the process ids, which they call node numbers [30]. To our knowledge, no prior work showed that using logical timestamps can be unfair.

Just like logical clocks are easy [3], seeing their being unfair is also easy, especially after simplification facilitated by precise high-level specifications. Such specifications have also helped us discover useless replies, unnecessary delays, and a main liveness violation that was previously unknown [19] in a more practical variant of Paxos for distributed consensus [37]. Methods for writing such high-level specifications are centered around expressing synchronization conditions as high-level queries over message histories [22]. Using message histories also yields simpler specifications and easier proofs [5] than otherwise [6].

In conclusion, DistAlgo is a language for expressing distributed algorithms precisely at a high level and running them directly. It has helped greatly in improving the understanding of distributed algorithms, including many algorithms for the core of distributed systems, taught in distributed systems courses and implemented in many course projects [23]. Future work includes further studies of important algorithms, by expressing them precisely at a high level, and continued improvements to the DistAlgo compiler and optimizations, as needed for maintenance of any modern language implementation, driven by constant changes in the underlying software and hardware technologies.

ACKNOWLEDGMENTS

This work was supported in part by NSF under grants CCF-1414078 and IIS-1447549 and ONR under grant N000141512208.

REFERENCES

[1] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. 2008. Interval tree clocks. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*. Springer, 259–274.
[2] Carlos Baquero and Nuno Preguiça. 2016. Why logical clocks are easy. *Queue* 14, 1 (2016), 60.
[3] Carlos Baquero and Nuno Preguiça. 2016. Why logical clocks are easy. *Commun. ACM* 59, 4 (2016), 43–47.
[4] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. 2007. The Development of the Emerald Programming Language. In *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*. 11–1–11–51.
[5] Saksham Chand and Yanhong A. Liu. 2018. Simpler Specifications and Easier Proofs of Distributed Algorithms Using History Variables. In *Proceedings of the 10th NASA Formal Methods Symposium*. Springer, 70–86.
[6] Saksham Chand, Yanhong A. Liu, and Scott D. Stoller. 2016. Formal Verification of Multi-Paxos for Distributed Consensus. In *Proceedings of the 21st International Symposium on Formal Methods*. Springer, 119–136.
[7] Colin J. Fidge. 1988. Timestamps in Message-Passing Systems That Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*. 56–66.
[8] Wan Fokkink. 2013. *Distributed Algorithms: An Intuitive Approach*. MIT Press.
[9] Vijay K. Garg. 2002. *Elements of Distributed Computing*. Wiley.
[10] Sukhendu Kanrar, Nabendu Chaki, and Samiran Chattopadhyay. 2018. *Concurrency Control in Distributed System Using Mutual Exclusion*. Springer.
[11] Dilsun Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2010. *The Theory of Timed I/O Automata* (2nd ed.). Morgan & Claypool.
[12] A.D. Kshemkalyani and M. Singhal. 2008. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press.
[13] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.
[14] Leslie Lamport. 2000. Distributed Algorithms in TLA+. PODC 2000 Tutorial https://www.podc.org/podc2000/lamport.html.
[15] Leslie Lamport. 2002. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.
[16] Jim Larson. 2009. Erlang for Concurrent Programming. *Commun. ACM* 52, 3 (2009), 48–56.
[17] Barbara Liskov. 1988. Distributed Programming in Argus. *Commun. ACM* 31, 3 (Mar. 1988), 300–312.
[18] Yanhong A. Liu, Jon Brandvein, Scott D. Stoller, and Bo Lin. 2016. Demand-Driven Incremental Object Queries. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*. ACM Press, 228–241.
[19] Yanhong A. Liu, Saksham Chand, and Scott D. Stoller. 2017. Moderately Complex Paxos Made Simple: High-Level Specification of Distributed Algorithm. *Computing Research Repository* arXiv:1704.00082 [cs.DC] (2017).
[20] Yanhong A. Liu, Michael Gorbovitski, and Scott D. Stoller. 2009. A Language and Framework for Invariant-Driven Transformations. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*. ACM Press, 55–64.
[21] Yanhong A. Liu, Scott D. Stoller, Michael Gorbovitski, Tom Rothamel, and Yanni E. Liu. 2005. Incrementalization Across Object Abstraction. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 473–486.
[22] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. 2012. High-Level Executable Specifications of Distributed Algorithms. In *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer, 95–110.
[23] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. 2017. From Clarity to Efficiency for Distributed Algorithms. *ACM Transactions on Programming Languages and Systems* 39, 3 (May 2017), 12:1–12:41.
[24] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. 2012. From Clarity to Efficiency for Distributed Algorithms. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications*. 395–410.
[25] Sandeep Lodha and Ajay Kshemkalyani. 2000. A fair distributed mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems* 11, 6 (2000), 537–549.
[26] Nancy A. Lynch. 1996. *Distributed Algorithms*. Morgan Kaufman.
[27] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. North-Holland, 120–131.
[28] Stephan Merz. 2010. Lamport's algorithm. Email with Annie Liu.
[29] Robert Paige and Shaye Koenig. 1982. Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 402–454.
[30] Said K. Rahimi and William R. Franta. 1982. Fair Timestamp Allocation in Distributed Systems. In *Proceedings of the 1982 National Computer Conference*. 589–594.
[31] Kerry Raymond. 1989. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems* 7, 1 (Jan. 1989), 61–77.
[32] Michel Raynal. 1986. *Algorithms for Mutual Exclusion*. MIT Press.
[33] Michel Raynal. 1988. *Distributed Algorithms and Protocols*. Wiley.
[34] Glenn Ricart and Ashok K. Agrawala. 1981. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Commun. ACM* 24, 1 (1981), 9–17.
[35] PC Saxena and Jagmohan Rai. 2003. A survey of permission-based distributed mutual exclusion algorithms. *Computer standards & interfaces* 25, 2 (2003), 159–181.
[36] Ichiro Suzuki and Tadao Kasami. 1985. A Distributed Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems* 3, 4 (1985), 344–349.
[37] Robbert van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *Comput. Surveys* 47, 3 (Feb. 2015), 42:1–42:36.
[38] Martin G. Velazquez. 1993. *A Survey of Distributed Mutual Exclusion Algorithms*. Technical Report CS-93-116. Department of Computer Science, Colorado State University. http://www.cs.colostate.edu/pubserv/pubs/Velazquez-TechReports-Reports-1993-tr-116.pdf