2021 IEEE Global Engineering Education Conference (EDUCON) | 978-1-7281-8478-4/20/$31.00 ©2021 IEEE | DOI: 10.1109/EDUCON46332.2021.9453906

Basic Distributed *Algorithms* Visual Simulations for Distributed *Systems* Students

Alexander Richard
Dept. of Computer Science
University of Calgary
Calgary, Canada
alexander.richard@ucalgary.ca

Jesse Francis
Dept. of Computer Science
University of Calgary
Calgary, Canada
jesse.francis1@ucalgary.ca

Jalal Kawash
Dept. of Computer Science
University of Calgary
Calgary, Canada
jalal.kawash@ucalgary.ca

***Abstract*—An introductory course on distributed systems typically exposes students to some basic synchronization distributed algorithms. This is often the first exposure for these students to the topic of distributed algorithms. In addition, in a systems course, these algorithms are typically covered in an informal way, avoiding proofs of correctness or complexity analysis. Hence, this first exposure can be challenging to students. Visualization of these algorithms can help alleviate some of these challenges. We present a suite to visualize six basic algorithms on total ordering, critical sections, and leader election.**

***Index Terms*—Computer Science Education; Tools for Education; Distributed Algorithms; Distributed Systems; Visualization**

I. Introduction

A first course on distributed systems intends to expose students to the fundamentals of building such systems. One of the most challenging subjects in such courses is distributed algorithms. Unlike a course on distributed *algorithms*, a distributed *systems* course is of the applied nature focusing on how to practically build distributed systems. Hence, only a minimal exposure to the topic of distributed algorithms is required. Distributed algorithms is a complex subject on its own and requires rigor to prove the correctness of algorithms and analyze their complexity, but such rigor is normally beyond the scope of a distributed systems course. Hence, the coverage of the algorithms topic in a systems course tends to be informal (see Tanenbaum and Steen's textbook [15] for example.) The distributed algorithms community and the larger algorithms community rightfully warn of the dangers of shying away from formalism. One of the authors of this paper taught distributed systems for two decades and also published theory papers about distributed algorithms. While we believe that such "in-formalism" may be necessary in this context, it aggravates the students confusion and misunderstanding of these algorithms. This author's experience manifests that visualizing these algorithms can diminish these pedagogical challenges. The remaining authors are undergraduate students who also concur with this observation based on their experience in their first course on distributed systems. The development of the visualization suite presented in this paper was motivated by their personal needs as students as well as the needs of their peers to better understand these algorithms.

In this paper, we present a suite to visualize six algorithms chosen from Tanenbaum and Steen's textbook [15]. These are totally-ordered multicast or broadcast [3], token-ring mutual exclusion [15], leader-based mutual exclusion [15], timestamp-based mutual exclusion [11], [13], ring leader election [7], and bully leader election [8]. We reproduce these algorithms in a unified framework using clear pseudo-code, alleviating some of the confusion resulting form the narrative approach taken in the textbook. Then, an interactive visual simulation for each of the algorithms is presented, bringing the students closer to understanding these algorithms without wondering into complex formal discussions (which we emphasize is necessary in a distributed algorithms course.) This suite is available at the URL: `https://pages.cpsc.ucalgary.ca/~jkawash/DAsim.html`.

Algorithm visualization has been widely recognized as an effective approach to improve instruction, and has been positively received by educators and students alike (for example, see [6], [9], [12], [14]). Hence, there is a sizable body of research on this subject, but it is largely in the context of introductory data structures and (sequential) algorithms. There has been also some work on visualizing multithreaded behavior [1], [4], [5]. Visualization of classical synchronization problems involving concurrent algorithms has been addressed as well [2]. To our knowledge, there is barely any published work on visualizing distributed algorithms for educational purposes. The work of Koldehofe et al. [10] is aimed at students in a distributed algorithms course, where complexity (time and space) of algorithms as well as basic invariants to establish their correctness are the focus. Our work is aimed at students in a distributed systems course, where complexity measures and proofs of correctness are not required.

The remainder of the paper has two sections. In the next section, the algorithms and their corresponding visualizations are presented. Conclusions are in the last section.

II. Visualization Suite

All the described algorithms assume a system of n processes with unique ids. Processes can communicate via point-to-point, reliable connections. All algorithms utilize the following blocking system calls to send and receive messages:
send(m, p): send message m to process p.
receive(): receive and return a message.

Some algorithms utilize (FIFO or priority) queues of processes or messages. A queue Q has the following functions:
Q.head(): returns the element at the head of the queue without removing it.
Q.remove(e): removes and returns the element e.
Q.add(e): adds the element e.
Q.isEmpty(): returns *true* if the queue is empty. Returns *false* otherwise.

A. Totally-Ordered Broadcast

The totally-ordered broadcast algorithm [3] uses logical timestamps [11]. Logical time is incremented each time a local event or a single message is sent. It is also updated if the received message contains a timestamp that is larger than the local logical time [11]. These details are omitted from the presented pseudo-code.

In the totally-ordered broadcast algorithm, a group of processes (called *tob* processes) coordinate to enforce a global total ordering on updates to a fully-replicated database (each site has a complete copy of the database). The algorithm specifies how updates are dealt with. Database queries (reads), can be simply satisfied by the local replica without any coordination with the other replicas. To enforce a total ordering on the updates, each process maintains a priority queue of messages. The priority is established using logical timestamps, which are assumed to be unique (pairing the timestamps with the unique process id will do the trick.) Given a message m, m.ts refers to the timestamp stored in m. The algorithm uses two types of messages:
μ[update, t]: an update to the distributed database with piggybacked logical time t
μ[ack, t]: an acknowledgment for receiving an update message with piggybacked logical time t.

Each process maintains the following local variables:

variables
mQueue: a priority queue for messages m sorted by m.ts
ts: $n \times n$ matrix of timestamps; initially zero
//$\text{ts}_{i,j}$ is the timestamp recorded by process i for process j

Whenever a process i receives an update u from the local application layer, the tob process executes the **broadcastUpdate**(u) function:

broadcastUpdate(update u) {
$\text{ts}_{i,i} \leftarrow \text{ts}_{i,i} + 1$
u.ts $\leftarrow \text{ts}_{i,i}$
send(u, j), $\forall j \neq i$
}

Each process i has two daemon threads: one indefinitely executes **receiveMessage**() to receive update and ack messages from other processes and the other indefinitely executes **applyUpdate**() to apply updates to the local database replica.

receiveMessage() {
$m \leftarrow$receive() from process j
$\text{ts}_{i,j} \leftarrow m$.ts
mQueue.add(m)
if m.ts $> \text{ts}_{i,i}$ **then**
$\text{ts}_{i,i} \leftarrow m$.ts
send(μ[ack, $\text{ts}_{i,i}$], k), $\forall k \neq i$
end-if
}

applyUpdate() {
$u \leftarrow$mQueue.head()
if $\forall j$, u.ts $\leq \text{ts}_{i,j}$ **then**
mQueue.remove(u)
apply u to the local database replica
end-if
}

The simulator for this algorithm is shown in Figure 1. To illustrate the need for totally ordering updates, the simulation assumes a simple banking application. Processes can invoke one of two updates on an account object: withdrawing $100 or applying a 1% interest rate to the balance.

The simulation options allow the user to run the simulation with and without totally ordering the updates, resulting in inconsistent replicas in the latter case. They also allow the user to choose the number of processes (up to 4) and the speed of simulation (fast, slow, or step-by-step). The visualization area shows the processes, their queues of updates, the local logical time, and the value of the account balance in the local replica. Update requests and acknowledgments arrows are also color coded.

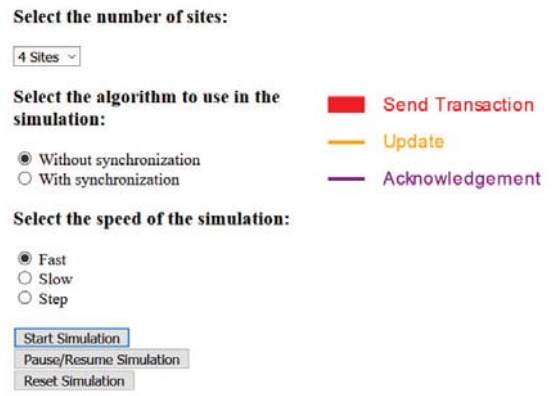
B. Critical Sections

Critical sections are a mechanism to achieve mutual exclusive locking of shared resources. Processes participating in the critical section problem have the following structure:

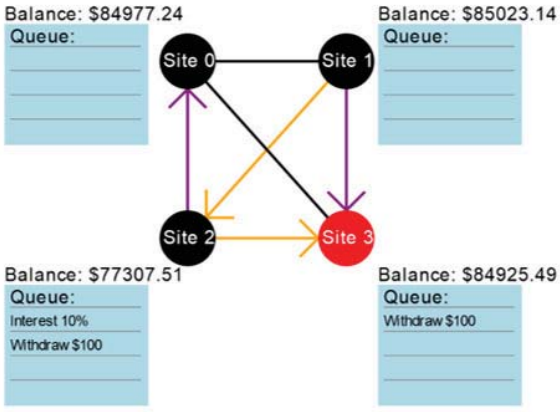
repeat
enter()
criticalSection()
exit()
remainder()
until *done*

The **remainder**() function contains non-critical-section code. The three algorithms described next provide implementations for the **enter**(), **exit**(), and **remainder**() functions. Where appropriate, **criticalSection**() will also be modified.

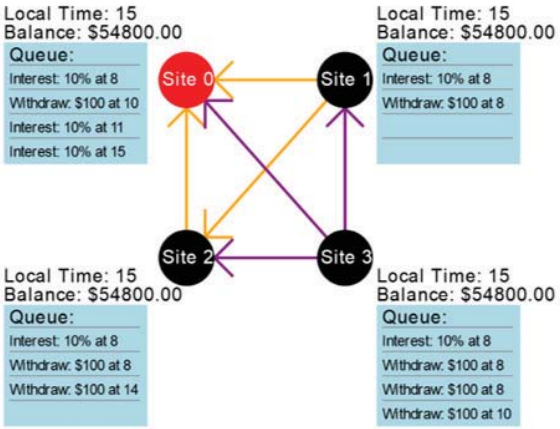
1) Ring Algorithm: The ring critical section algorithm [15] assumes that processes are organized in a uni-directional logical ring and uses one message type: μ[token]. One process



(a)



(b)



(c)

Fig. 1. The Totally-Ordered Broadcast Algorithm Simulator (a) Simulation options (b) Simulation without total ordering (c) Simulation with total ordering

creates a single token message and passes it to its successor in the ring. The token circulates the ring. To enter the critical section, a process waits for the token message and when it exits, it releases the token. The following code is for process i.

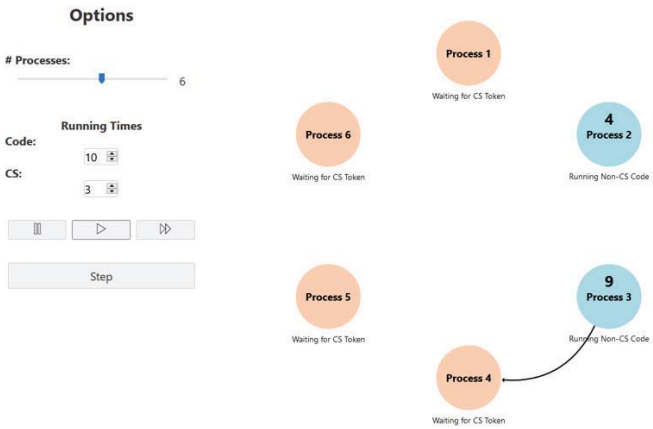


Fig. 2. The Ring Critical Section Algorithm Simulator

enter() {
  wait for μ[token]
}

exit() {
  send(μ[token], $successor(i)$)
}

remiander() {
  if μ[token] is received **then**
    send(μ[token], $successor(i)$)
  end-if
}

The visual simulator for the ring critical section algorithm is shown in Figure 2. The slider bar allows the user to specify the number of processes up to 9 processes. The **Running Times** allow the user to specify the length of remainder code and critical section code in clock ticks. The simulation can be run continuously with a slow (using the "play" button) and fast speed (indicated with the "fast forward" button). It can be also run step-by-step using the "step" button.

The visualization area shows the ring of processes. A process representation changes color based on its state (in critical section, in remainder, or waiting to enter critical section); the state is also written underneath the bubble representing a process. Passing of a token message is indicated with an arrow from the sender to the receiver process.

2) Timestamp Algorithm: The timestamp critical section algorithm [13] employs logical time stamps [11] and requires a process to obtain clearance from all other processes before entering a critical section. It uses two types of messages:
μ[request, p, t]: a request from process p to enter the critical section at time t.
μ[acquire, q, p]: a clearance from process q to process p for the critical section.

In addition, each process maintains the following local variable.

variables
waitQueue: FIFO queues of processes.

Similar to the totally ordered broadcast algorithm, logical time is incremented each time a local event or a single message is sent, and it is updated if the received message contains a timestamp that is larger than the local logical time. These details are omitted from the presented pseudo-code. We use the function **time**(i) to return the current local logical time for process i. In addition, it is assumed that timestamps are unique.

```
enter(){
    thread:
        send(μ[request, i, time(i)], j), ∀j ≠ i
        wait for μ[acquire, j, i], from every j ≠ i
    end-thread
    thread: repeat while in enter()
        μ[request, j, t'] ← receive()
        if time(i) ¿ t' then send(μ[acquire, j, i], j)
        else waitQueue.add(j)
    end-thread
}

exit() {
    for each j in waitQueue do
        send(μ[acquire, i, j], j)
        waitQueue.remove(j)
    end-for
}

remiander() {
     if μ[request,j, t] is received then
        send(μ[acquire, i, j], j)
     end-if
}

criticalSection() {
    thread:
        ... critical section code
    end-thread
    thread: repeat while in criticalSection()
            μ[request, j, t] ← receive()
            waitQueue.add(j)
    end-thread
}
```

The simulator for this algorithm is shown in Figure 3. The simulation options allow the user to choose the number of processes (up to 4) and the speed of simulation (fast, slow, or step-by-step). The visualization color-codes the states of processes (in enter, critical section, exit, or reminder sections). It also shows the queues maintained by each process. The request and acquire (ACK) messages are indicated using color-coded arrows.

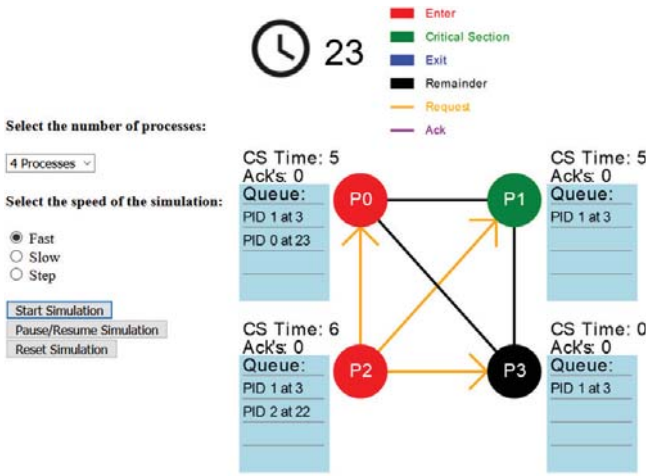


Fig. 3. The Timestamp Critical Section Algorithm Simulator

3) Leader Algorithm: The leader critical section algorithm [15] assumes a leader has been elected. The leader is responsible for permitting other processes to enter the critical section and keeps track of the processes requesting to enter the critical section. It uses three types of messages:
μ[request, p]: a request from process p for the critical section.
μ[release, p]: a release by process p of the critical section.
μ[acquire, p]: a message from the leader process for process p to acquire the critical section.

This is an asymmetric algorithm that requires different code for the leader and non-leader processes. The enter and exit functions for non-leader process i are as follows (remainder section is irrelevant):

```
enter() {
    send(μ[request, i], leader)
    wait for μ[acquire] from the leader
}

exit() {
    send(μ[release, i], leader)
}
```

The leader process maintains the following local variables.

variables
waitQueue: a FIFO queue of waiting processes for the critical section.

mutex: Boolean mutex variable, initially *false*; it is *true* when the critical section is acquired.

The leader process code is as follows:

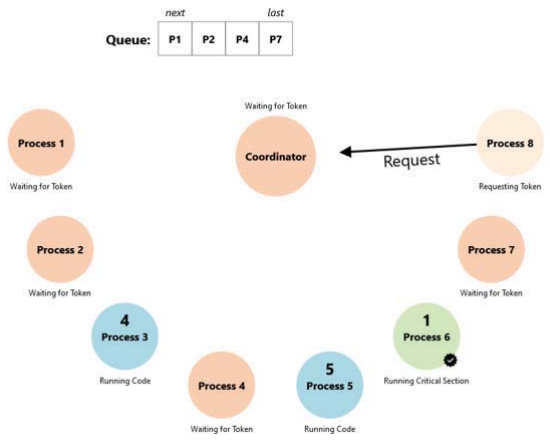


Fig. 4. The Leader-based Critical Section Algorithm Simulator

$m \leftarrow$ receive()
case m is μ[request,j] **then**
  if mutex **then** waitQueue.add(j)
  else
    mutex $\leftarrow$ *true*
    send(μ[acquire, j], j)
  end-else
end-case
case m is μ[release, j] **then**
  if waitQueue.isEmpty() **then**
    mutex $\leftarrow$ *false*
  end-if
  else
    $k \leftarrow$ waitQueue.head()
    waitQueue.remove(k)
    send(μ[acquire, k], k)
  end-else
end-case

The simulator for this algorithm is shown in Figure 4. The options are the same as the ring critical section algorithm and are omitted from the figure. In the visualization area, the queue of processes maintained by the leader process is also shown.

C. Leader Election

The purpose of leader election is to achieve census among all n processes in the system designating a single process as the leader. The leader is the process with the highest id. Both leader election algorithms described in this section assume each process with id i has the following local variables.

variables
  running$_i$:Boolean variable, initially *false*; set to *true* if process i is running for election
  leader$_i$: contains the id of the current leader

Both algorithms also use the following two types of messages:

μ[election, p]: signifies an election is going on with process p as a candidate.
μ[leader, p]: announces the end of an election with process p as the elected leader.

In both algorithms, any process can initiate an election through **initiateElection**() and all (live) processes execute **particpateInElection**() in a daemon thread. Several processes may initiate elections concurrently.

1) Ring Algorithm: The ring leader election algorithm covered in this class is the $O(n^2)$ algorithm [7]. Processes are organized in a unidirectional logical ring, with the following functions: predecessor(p) returns the id of the predecessor of p in the ring and successor(p) returns the id of the successor of p in the ring. A process creates an election message, writes its id in it, and sends it to its successor. A process that receives an election message from its predecessor checks if the process id in the message is smaller than its own. If not, it passes the message to it successor. Otherwise, it either kills the election message if it already participated in this election round or replaces the id in the message with its own and passes it to its successor.

The code for process i is given next.

initiateElection(process i) {
  running$_i$ $\leftarrow$ *true*
  send(μ[election, i], $successor(i)$)
}

Recall that function **particpateInElection**( ) runs in a process as a *daemon* thread; that is, it runs all the time.

particpateInElection(process i) {
  $m \leftarrow$ receive() from predecessor(i)
  case m is μ[election, j] **then**
    if $j > i$ **then**
      send(μ[election, j], $successor(i)$)
    end-if
    if $j < i$ **and not** running$_i$ **then**
      send(μ[election, i]], $successor(i)$)
      running$_i$ $\leftarrow$ *true*
    end-if
    if $j = i$ **then**
      send(μ[leader, i], $successor(i)$)
    end-if
  end-case
  case m is μ[leader, j] **then**
    leader$_i$ $\leftarrow$ j
    running$_i$ $\leftarrow$ *false*
    if $j \neq i$ **then**
      send(μ[leader, j], $successor(i)$)
  end-case
}

Process crash failures and recovery require the reconstruction of the ring. If a process realizes that its successor has crashed, it replaces its successor(i) with

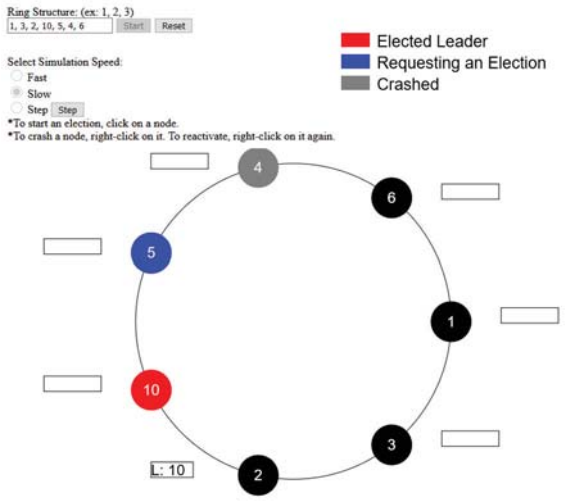


Fig. 5. The Ring Leader Election Algorithm Simulator

successor(successor(i)) in a recursive way. The restarting of a previously crashed process restores the original successor relation. A process p also invokes **initiateElection**(p) when p is first introduced to the system, re-introduced after a crash, or when it realizes that the leader has crashed.

The simulation for the ring leader election algorithm is shown in Figure 5. In the simulation options, the user chooses the ring organization, the number of processes can be selected up to 9 processes, and the simulation speed. The visualization area shows the processes and their states, and the message being sent. Clicking a bubble results in the corresponding process to initiating an election. In addition, the simulation allows a user to crash/restart a process by clicking its bubble. When a process crashes, its predecessor will initiate a new election. In the simulator, the user can decide which processes are crashed. Any crashed process can be re-introduced to the ring. Giving the user the ability to choose the ring organization, allows students to experiment with different ring organizations and to study the number of messages each organization generate.

2) Bully Algorithm: The bully leader election algorithm [8] also assumes that process failures can be detected. Any process can initiate an election, and at the end of any election, the process with the highest id that is still alive is elected. A process that initiates election sends and election message to all processes with a higher id. Those that are alive *bully* that process and send election messages to processes with higher ids, until a single process bullies all running-for-office processes. This algorithm has a third type of messages:
μ[bully]: bullies the receiver to quit an election.

Also it needs an extra local variable to synchronize internal threads with each process i.

variables

leaderAnnounced$_i$:initially *false*; it is set to *true* when a winner is announced

Similar to the ring algorithm, a process p also invokes **initiateElection**(p) when p is introduced or re-introduced to the system or when it realizes that the leader has crashed. In our simulator, it is left to the user to decide which process initiates elections and when to initiate them. Also, similar to the ring algorithm, the user can decide which processes to crash and which crashed process to re-introduce to the system. Code for process i is given next.

```
initiateElection(process i) {
   running_i ← true
   if i is the highest id then
        send(μ[leader, i], j), ∀j ≠ i
   end-if
   else
        send(μ[election, i], j), ∀j > i
   end-else
   wait for t time units
   if no response is received then
        leader_i ← i
        send(μ[leader, i], j), ∀j ≠ i
   end-if
   else        //a μ[bully] message must have been received
        wait for t' time units
        if not leaderAnnounced_i then
          initiateElection(i)
        end-if
   end-else
}

particpateInElection(process i) {
    m ← receive()
    case m is μ[leader, j] then
          leader_i ← k
          running_i ← false
          leaderAnnounced_i ← true
    end-case
    case m is μ[election, j] then
          if i > j then
            send(μ[bully], j)
            if not running_i then
              initiateElection(i)
            end-if
          end-if
    end-case
}
```

The simulation for the ring leader election algorithm is shown in Figure 6. The options are similar to the ring leader election algorithm. The visualization area shows the state of processes including crashed ones. A process can be crashed or "uncrashed" by clicking its corresponding bubble.

III. Conclusions

Distributed algorithms is an inherently complex subject that requires formalism to prove their correctness and analyze their

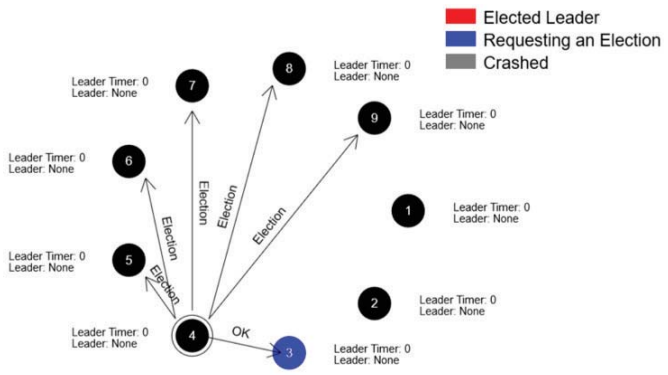


Fig. 6. The Bully Leader Election Algorithm Simulator

complexity. However in an introductory course on distributed systems, a minimal exposure to the subject of distributed algorithms is needed without the required rigor, and as as result, such algorithms are presented in an informal way. This creates a challenge to educators and students alike to navigate this difficult subject without resorting to formal methods. Visualization of algorithms has been utilized to alleviate some of the difficulties of teaching and learning algorithms and data structures, and we believe that it can accomplish the same objective in the context of distributed algorithms.

The paper presented interactive visual simulations for six basic distributed algorithms, typically covered in an introductory distributed systems course. These cover totally-ordered broadcast, critical sections, and leader election. Developed by students, this suite includes the features that the students identified as needed in order to help them and their peers understand these algorithms.

In this paper, there is no assessment of the suite. Two research questions remain open and will be addressed in future research: (1) How engaging do the students find the suite, and (2) How and in what measurable ways these visualizations improve the learning of students.

REFERENCES

[1] J. C. Adams, P. A. Crain, C. P. Dilley, S. M. Nelesen, J. B. Unger, and M. B. V. Stel. Seeing is believing: Helping students visualize multithreaded behavior. In C. Alphonce, J. Tims, M. E. Caspersen, and S. H. Edwards, editors, *Proceedings of the 47th ACM Technical Symposium on Computing Science Education, Memphis, TN, USA, March 02 - 05, 2016*, pages 473–478. ACM, 2016.

[2] J. C. Adams, E. R. Koning, and C. D. Hazlett. Visualizing classic synchronization problems: Dining philosophers, producers-consumers, and readers-writers. In E. K. Hawthorne, M. A. Pérez-Quiñones, S. Heckman, and J. Zhang, editors, *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019, Minneapolis, MN, USA, February 27 - March 02, 2019*, pages 934–940. ACM, 2019.

[3] H. Attiya and J. L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.)*. Wiley series on parallel and distributed computing. Wiley, 2004.

[4] M. Bedy, S. Carr, X. Huang, and C. Shene. A visualization system for multithreaded programming. In L. B. Cassel, N. B. Dale, H. M. Walker, and S. M. Haller, editors, *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, 2000, Austin, Texas, USA, March 7-12, 2000*, pages 1–5. ACM, 2000.

[5] S. Carr, J. Mayo, and C. Shene. Threadmentor: a pedagogical tool for multithreaded programming. *ACM J. Educ. Resour. Comput.*, 3(1):1, 2003.

[6] L. B. Cassel, editor. *Proceedings of the 1st Annual Conference on Integrating Technology into Computer Science Education, ITiCSE 1996, Barcelona, Spain, 2-6 June, 1996*. ACM, 1996.

[7] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, May 1979.

[8] Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, 1982.

[9] C. M. Kehoe, J. T. Stasko, and A. Talor. Rethinking the evaluation of algorithm animations as learning aids: an observational study. *Int. J. Hum. Comput. Stud.*, 54(2):265–284, 2001.

[10] B. Koldehofe, M. Papatriantafilou, and P. Tsigas. Distributed algorithms visualisation for educational purposes. In C. Erickson, T. Wilusz, M. Daniels, R. McCauley, and B. Z. Manaris, editors, *Proceedings of the 4th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 1999, Cracow, Poland, 27-30 June, 1999*, pages 103–106. ACM, 1999.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In D. Malkhi, editor, *Concurrency: the Works of Leslie Lamport*, pages 179–196. ACM, 2019.

[12] T. L. Naps, G. Rößling, V. L. Almstrum, W. Dann, R. Fleischer, C. D. Hundhausen, A. Korhonen, L. Malmi, M. F. McNally, S. H. Rodger, and J. Á. Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *ACM SIGCSE Bull.*, 35(2):131–152, 2003.

[13] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[14] C. A. Shaffer, M. L. Cooper, A. J. D. Alon, M. Akbar, M. Stewart, S. P. Ponce, and S. H. Edwards. Algorithm visualization: The state of the field. *ACM Trans. Comput. Educ.*, 10(3):9:1–9:22, 2010.

[15] A. S. Tanenbaum and M. van Steen. *Distributed systems - principles and paradigms, 2nd Edition*. Pearson Education, 2007.