A distributed algorithms simulator in the interleaving model

Tomohiro Yasuda*, Hiroaki Karasawa†, Tadafumi Yoshida‡, Fumito Nakamura§, and Yukihiro Hamada¶

Abstract: In a distributed system, processes do their tasks asynchronously by computing locally and exchanging messages with other processes. Due to the asynchrony of processes and a communication delay, it is difficult to observe the behaviour of a distributed algorithm and to verify the correctness of the algorithm. To this end, various distributed algorithms simulators have been developed. The one we developed consists of two components, a descriptor and a simulator. Using the descriptor, a user defines processes and messages. A process is defined by states, variables, events, and parameters. A message is defined by name and variable type. Then, the descriptor outputs skeleton codes written in Java. Based on the skeleton codes, the user describes distributed algorithm codes in Java using any editor. The descriptor complies the distributed algorithm codes. The simulator is in the interleaving model. It simulates message delivery with a communication delay and process failure. It runs both in a manual mode and in an automatic mode. It also saves a simulation both in a text file and in a format that can be run repeatedly in the simulator. Furthermore, it partly provides a function deadlock detection.

Keywords: distributed algorithm, interleaving model, descriptor, simulator, process, message delivery, event

1 Introduction

A distributed algorithm is a collection of algorithms such that each algorithm is executed by a distinct process, and that all processes achieve their common purpose. The algorithm of a process consists of several events. Each event is classified into one of three types; send event, receive event, and internal event. The execution order of events by a process may vary according to the state of the process. All processes execute events asynchronously.

Due to the asynchrony of event execution and the communication delay of a sent message, it is difficult to observe the behaviour of a distributed algorithm and to verify the correctness of the algorithm. To this end, various distributed algorithms simulators have been developed [3, 8].

A distributed algorithms simulator in [3] was written in Java. It provides GUI and enables a user to construct any communication network topology. It simulates a distributed algorithm that the user wrote in Java. However, it cannot simulate the communication delay of a sent message and the failure of a process. The number of processes is limited to 11.

A distributed algorithms simulator DAJ in [8] was also written in Java. DAJ runs in a standalone mode with (or without) visualization and as an applet embedded in a Web page. DAJ provides GUI and enables a user to construct any communication network topology. DAJ simulates a distributed algorithm that the user wrote in Java. Execution of events is in a round robin fashion or by the schedule that the user defined. DAJ simulates the communication delay of a sent message. A link between two processes is unidirectional and it delivers messages only in the FIFO mode. DAJ simulates losing and duplicating a message.

We developed a distributed algorithms simulator H-DAS. You can download H-DAS in the near

*The author is currently with none.
†Department of Information and Communication Engineering, The University of Tokyo.
‡The author is currently a freelance Web engineer.
§Department of Computational Intelligence and System Science, Tokyo Institute of Technology.
¶Department of Electrical and Computer Engineering, National Institute of Technology, Akashi College, Akashi-city, 674–8501 Japan. hamada@akashi.ac.jp

future. H-DAS consists of two components, a descriptor and a simulator. Using the descriptor, a user defines processes and messages. A process is defined by states, variables, events, and parameters. A message is defined by name and variable type. Then, the descriptor outputs skeleton codes written in Java. Based on the skeleton codes, the user describes distributed algorithm codes in Java using any editor. The descriptor complies the distributed algorithm codes. The simulator is in the interleaving model and simulates at most 1000 processes. It provides GUI and simulates message delivery with a communication delay and process failure. It runs both in a manual mode and in an automatic mode. In the automatic mode, the execution order of events is determined randomly. It also saves a simulation both in a text file and in a format that can be run repeatedly in the simulator. Furthermore, it partly provides a function deadlock detection.

The rest of this article is organized as follows. In Section 2, we describe fundamental knowledge of a distributed algorithm and how to use H-DAS. In Section 3, we explain the descriptor. In Section 4, we explain the simulator. In Section 5, we summarize H-DAS and describe future development.

2 Preliminaries

2.1 Process and Event

In general, a distributed system is modelled by the set of processes and the set of communication channels. A process is an abstraction of a sequential program that is executed on a computer. A channel connects two processes. We denote the number of processes in a distributed system by N.

In H-DAS, we model a distributed system only by the set of processes as follows. A process is defined by the set of states and the set of events. The state of a process is determined by the values of local variables, a send buffer, and a receive buffer. We assume that each process can send a message directly to every other process. The send buffer is a sequence of messages that was sent but not delivered yet. The receive buffer is a sequence of messages that was delivered but not received yet.

An event is a short sequential program that is executed by a process and it may change the state of the process. All processes execute events asynchronously. Each event is classified into one of three types; send event, receive event, and internal event. We assume that in a send event, a process can put a message or the same messages whose destinations are distinct to its send buffer, but it cannot put different messages to the send buffer. Each message is assigned with a finite message delay randomly. After the message delay (the concept of global time is described in the following subsection), the message is removed from the send buffer and it is put the receive buffer of the destination process. We also assume that in a receive event, a process can remove and get exactly one message from its receive buffer.

2.2 Model of a Computation

In order to observe the behaviour of a distributed algorithm, three models of a computation are used. They are interleaving model, happened before model, and potential causality model [2]. Each model is defined formally as a binary relation on the set of all events that all processes execute. Let P_i denote a process and S_i the set of all events that P_i executes. Let $S = S_0 \cup S_1 \cup \cdots \cup S_{N-1}$.

The interleaving model is a binary relation such that a total order is defined on S. The happened before model is a binary relation such that a total order is defined on each S_i and that a partial order is defined on S. The potential causality model is a binary relation such that a partial order is defined on each S_i. Figure 1 and Figure 2 illustrate a run in the happened before model and a run in the potential causality model, respectively. Observe that a run in the potential causality model is equivalent to a set of runs in the happened before model. Similarly, a run in the happened before model is equivalent to a set of runs in the interleaving model. H-DAS simulates a run in the interleaving model.

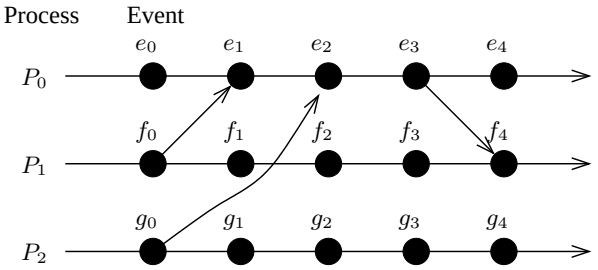

Figure 1: A run in the happened before model.

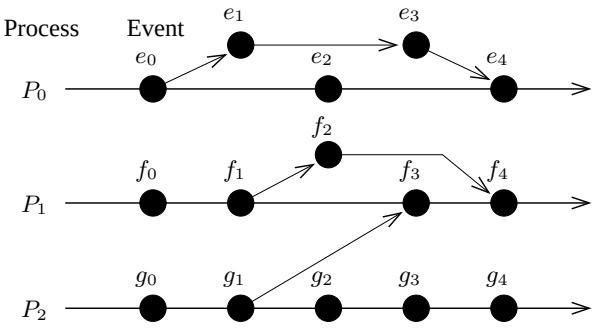

Figure 2: A run in the potential causality model.

2.3 Logical Clock

A logical clock was devised by Lamport [4]. It is a mapping from the set of all events to the set of natural numbers. It enables us to determine a total order on the set of all events.

H-DAS uses a kind of vector clock called direct-dependency clock [2]. A vector clock is a mapping from the set of all events to the set of N-dimensional vectors whose component is a natural number. It enables us to determine a partial order on the set of all events. Note that in a real distributed system, only a partial order on the set of all events can be observed. H-DAS simulates a run in the interleaving model i.e., a total order on the set of all events such that the run is consistent with the partial order on the set of all events.

We now present a direct-dependency clock algorithm. It is described from the point of view of local states. We denote processes by $P_0, P_1, \cdots, P_{N-1}$. Let s denote a local state of a process and $s.p$ the subscript of a process on which state s occurs. Let $s.v[\,]$ denote the vector of a local state s.

A direct-dependency clock algorithm

P_i ::

var v : array$[0..N-1]$ of integer

initially $(\forall j : j \neq i : v[j] = 0)$ and $(v[i] = 1)$

send event $(s, send, t)$:

// $s.v[t.p]$ is appended to a message.

$t.v[t.p] := s.v[t.p] + 1$;

receive event $(s, rcv[u], t)$:

// the message was sent in state u.

$t.v[t.p] := max\{s.v[t.p], u.v[u.p]\} + 1$;

$t.v[u.p] := max\{s.v[u.p], u.v[u.p]\}$;

internal event $(s, internal, t)$:

$t.v[t.p] := s.v[t.p] + 1$;

A sample execution of the direct-dependency clock algorithm is shown in Figure 3.

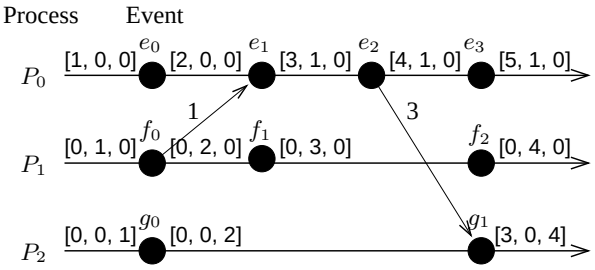

Figure 3: A sample execution of the direct-dependency clock algorithm.

2.4 How to Use H-DAS

2.4.1 Install

Download a tar file. You can download the tar file in the near future. Extract files from the tar file. The structure of extracted directories is partly shown in Figure 4.

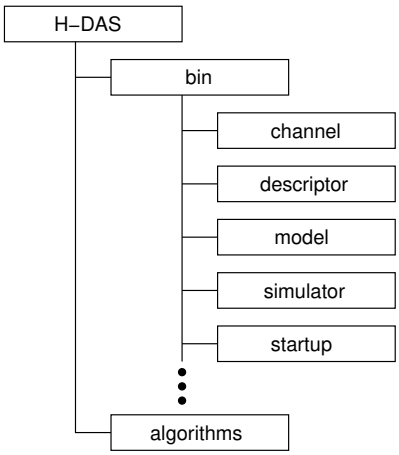

Figure 4: The structure of directories.

2.4.2 Startup H-DAS

Move to directory "bin", and execute "java startup.Controller". Then, the startup screen appears as shown in Figure 5.

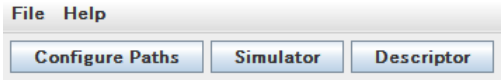

Figure 5: Startup screen.

When you use H-DAS for the first time, it is required to configure the path to file "tools.jar" and the path to the directory in which a system directory is saved. A file "tools.jar" contains classes used by Java. A system directory stores distributed algorithm codes and a das file. A das file is created by the descriptor. Configuring paths can be done using the dialogue as shown in Figure 8. The dialogue appears when button "Configure Paths" in Figure 5 is clicked.

2.4.3 Online Help

H-DAS has an online help that is referred with a browser. The items of online help are "Configure paths", "Descriptor", "Simulator", and "About H-DAS". You can refer the online help from the startup screen, descriptor window, and simulator window.

3 Descriptor

3.1 Overview

The descriptor appears by clicking button "Descriptor" in Figure 5. It is shown in Figure 9. Using the descriptor, we define processes and messages. A process is defined by states, variables, events, and parameters. A message is defined by name and variable type.

After defining processes and messages, we can output several skeleton codes written in Java in the pre-specified directory. A das file is also output. The skeleton codes are algorithmnameProcess.java, algorithmnameState.java, and messagenameMessage.java.

Based on the skeleton codes, we describe distributed algorithm codes in Java using any editor.

After saving distributed algorithm codes, we execute "build" operation. The build operation consists of the following four steps.

1. Create a system file (systemnameSystem.java) from distributed algorithm codes.
2. Check whether all algorithm files were saved after file "systemname.das" had been saved.
3. Check whether each message model is used in some algorithm code and whether each event model deals with valid message models.
4. Call Java compiler.

4 Simulator

4.1 Overview

The simulator appears by clicking button "Simulator" in Figure 5. At first, the simulator is as shown in Figure 6.

Figure 6: Initial screen of simulator.

By clicking button "New" in Figure 6, file selector as shown in Figure 7 appears.

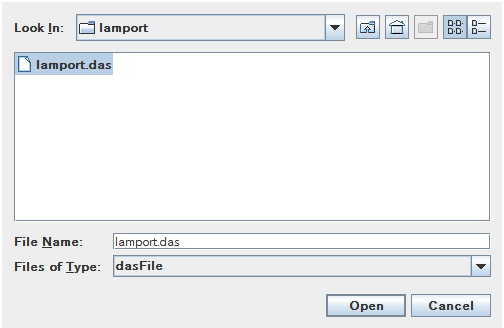

Figure 7: Selecting a das file.

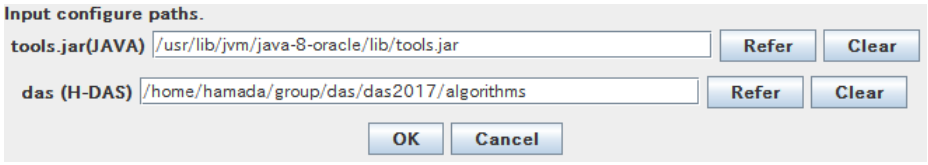

Figure 8: Configuring paths.

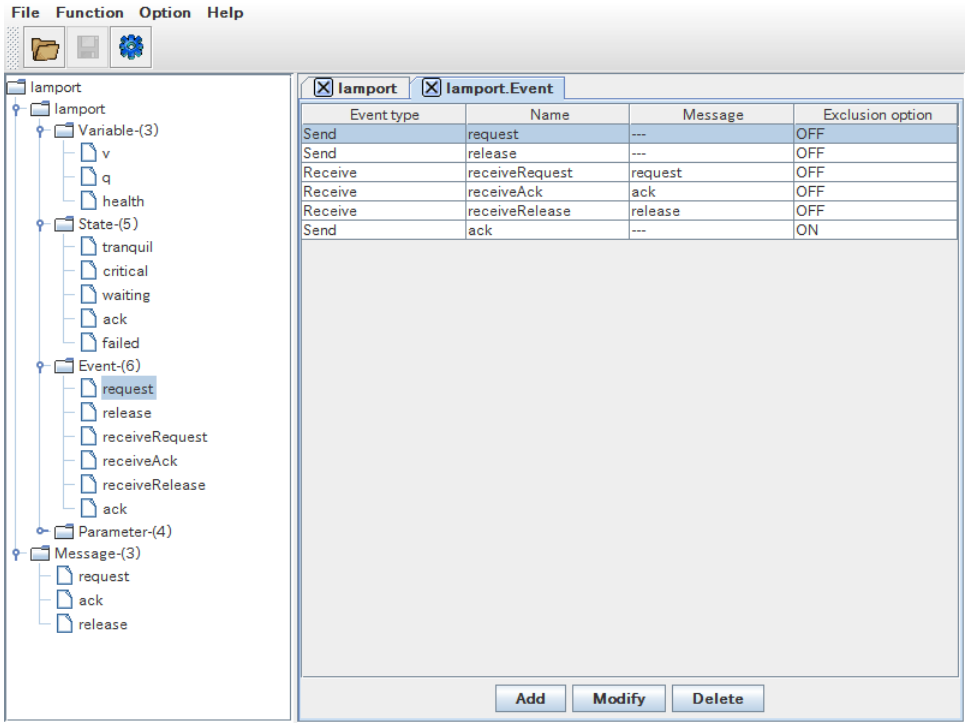

Figure 9: Descriptor.

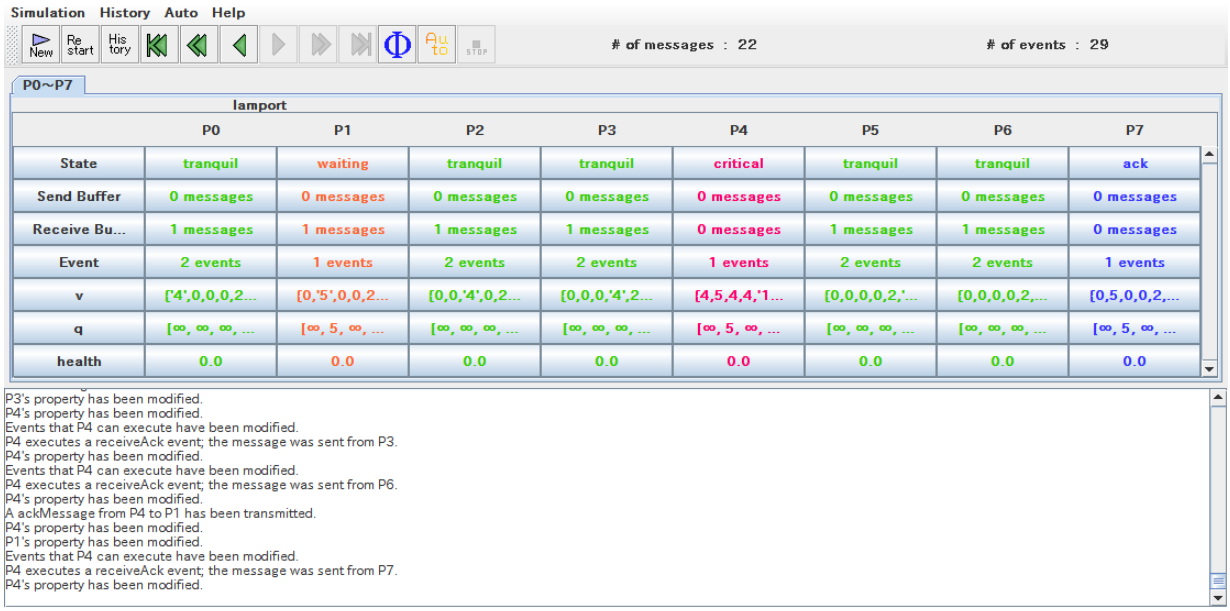

Figure 10: Simulator.

After selecting a das file, setting dialogue as shown in Figure 11 appears. In the setting dialogue, we specify the number of processes, a failure probability, an upper bound on communication delay, and channel mode. We also specify whether we use a coterie.

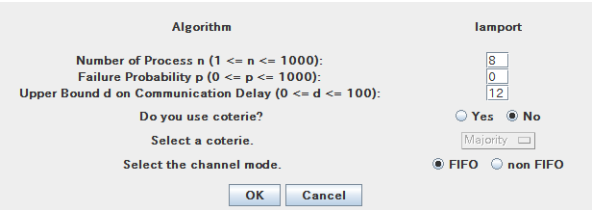

Figure 11: Specifying simulation settings.

The failure probability p is a threshold that each process fails if the value of variable "health" of the process is less than $p/1000$. The variable "health" of each process is assigned a random number initially and after every event.

Any two processes are assumed to be connected by a channel. We model a communication delay as follows. If a process u sends a message to a process v, then the message is assigned a communication delay d randomly, and put in the send buffer of u. After d events, the message is removed from the send buffer of u, and it is put the receive buffer of v. We can also select whether the channel mode is FIFO or not.

Let U denote the set of all processes. Let C denote a set of subsets Q_i's of U. $C = \{Q_0, Q_1, \cdots, Q_{m-1}\}$ is called coterie if $\forall i, j : i \neq j : \neg(Q_i \subseteqq Q_j)$ and $\forall i, j : Q_i \cap Q_j \neq \emptyset$. A coterie is used to implement mutual exclusion.

After we input all settings, we click button "OK" in Figure 11. Then, we have simulator window as shown in Figure 10.

4.2 Functions

4.2.1 Global Time

Since H-DAS simulates a run of a distributed algorithm in the interleaving model, there is a mechanism that indicates the global time. It is the number of events that is shown in the upper right side of Figure 10. Initially, the number of events is 0. If a process executes an event, then the number of events is incremented. In order to implement a communication delay, we introduce "NULL event". "NULL event" is an event in which no process executes an event and only the number of events is incremented.

4.2.2 FIFO Mode in a Channel

As described in Subsection 2.1, channels are modelled by send buffers and receive buffers of processes. We explain how FIFO mode in a channel is implemented.

Let m denote a message that is sent from process P_i to process P_j. Let t denote the global time at which m is generated. Let d denote a communication delay that is assigned randomly to m and D denote an upper bound on communication delay. Let $t_m = t + d$. Note that d satisfies $1 \leqq d \leqq D$.

For the channel from P_i to P_j, we use a queue $QM_{i,j}$ and a priority queue $QT_{i,j}$ in which a smaller value has a higher priority. When message m is generated by P_i, m is inserted to $QM_{i.j}$ and t_m is inserted to $QT_{i,j}$. Assume that $QM_{i,j}$ is $\langle m_0, m_1, m_2 \rangle$ and $QT_{i,j}$ is $\langle 14, 20, 22 \rangle$. Note that t_{m_0} is 14 or 20 or 22. When the global time is 14, 14 is removed from $QT_{i,j}$, m_0 is removed from $QM_{i,j}$, m_0 is removed from the send buffer of P_i, and m_0 is put in the receive buffer of P_j.

4.2.3 Send Buffer and Receive Buffer

For each process P_i, there are button "Send Buffer" and button "Receive Buffer" in the simulator window. Each button always indicates the number of messages in the buffer. If button "3 messages" is clicked, then a small windows as shown in Figure 12 appears. It enables us to check the contents of a receive (send) buffer.

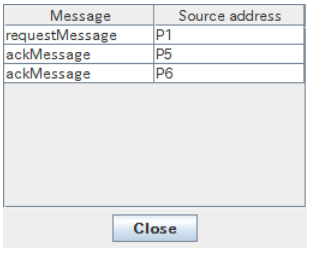

Figure 12: Checking a receive buffer.

4.2.4 Event

For each process P_i, there is button "Event" in the simulator window. This button always indicates the

number of events that P_i can execute. Assume that P_i can execute two events. If button "2 events" of P_i is clicked, then a small windows as shown in Figure 13 appears.

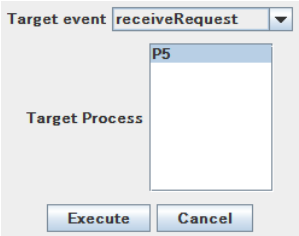

Figure 13: Selecting an event.

Since P_i can execute two events, we can select one of them using the drop-down list in the small window. If the selected event is a receive event and there are more than one message in the receive buffer, then we must select exactly one message among them.

After selecting a target event and a target process, we click button "Execute". Then, the specified event is executed.

When a send event is executed, we may have to execute "NULL event" repeatedly. This is because each message is assigned a communication delay randomly. "NULL event" is executed by clicking the button that is labeled with Φ in the upper side of Figure 10. In a send event, if process P_i sends the same message to several processes, then all messages are put in the send buffer of P_i. In this case, too, each message is assigned a communication delay randomly.

On the other hand, in a receive event, each process can receive exactly one message.

We can undo executed events repeatedly. This is implemented by that the simulator saves a sequence of all global states automatically from the initial global state until the current global state. There are three types of undo function. They differ in the number of events to be undone; one, a user specified number, and all.

The simulator can save a sequence of executed events both in a text file and in a format that can be executed again. We call this function "event history". In the lower side of the simulator, there is a text output area. It outputs about initial settings of a simulation and all executed events. Function "event history" saves these information.

In the upper side of Figure 10, there are 12 buttons. Among them, 6 buttons are labeled with one or more green triangles; 3 buttons are labeled with triangles that point right side and the remaining 3 buttons with triangles that point left side. The former 3 buttons are used when we simulate along "event history". Using them, we execute one event, the user specified number of events, or all events. The latter 3 buttons are used when we undo executed events and we go back along "event history".

Using the simulator, we can simulate a distributed algorithm both in a manual mode and in an automatic mode. In the manual mode, a user selects an event and execute it. In the automatic mode, the simulator selects an event randomly and execute it.

In order to simulate in the automatic mode, we click the button labeled with "Auto" in the upper side of the simulator. Then, a dialogue for the setting of automatic mode appears as shown in Figure 14.

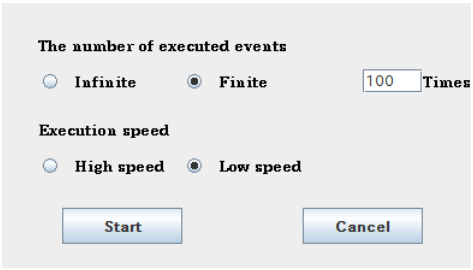

Figure 14: Setting automatic mode.

If we select "Infinite" for the number of events in the dialogue, the simulator executes events as many times as possible. An upper bound on the number of times depends upon the capacity of an auxiliary memory. This is because the simulator saves every 256 events from the main memory to the auxiliary memory. If we select "Low speed" for execution speed in the dialogue, we can catch up with event-executions.

4.2.5 Deadlock Detection

The simulator partly provides a function deadlock detection. If we describe a distributed algorithm code so that it inherits classes "WantResponse" and "PleaseResponseMessage", then deadlock detection is executed in the background. When a deadlock is

detected, the simulator notices it by a small message window.

4.3 Sample Algorithms

The following 5 sample algorithms are appended. You can run them using the simulator without describing a distributed algorithm.

- Centralized
  This algorithm solves a mutual exclusion problem. In this algorithm, P_0 is the coordinator process, and the other processes are client processes [2].
- Deadlock
  This algorithm shows a function deadlock detection. In this algorithm, for $0 \leqq i \leqq N-2$, P_i sends a message to P_{i+1}, and P_{N-1} sends a message to P_0. For $0 \leqq i \leqq N-2$, P_i requires a response message from P_{i+1}, and P_{N-1} requires a response from P_0.
- Lamport
  This algorithm solves a mutual exclusion problem. It uses timestamped messages; they are request messages, release messages, and acknowledge messages [4].
- Leader_Election
  This algorithm solves a leader election problem. Processes are logically arranged so that they form a cycle. For $0 \leqq i \leqq N-2$, P_i sends messages to P_{i+1}, and P_{N-1} sends messages to P_0. There are two types of messages; $(election, i)$ and $(leader, i)$ [1].
- Maekawa
  This algorithm solves a mutual exclusion problem. Processes are logically arranged so that they form a two dimensional grid. This algorithm uses a coterie. The simulator provides coteries "Majority", "Singleton", and "Crumbling Walls" [6, 7].

5 Concluding Remarks

We developed a distributed algorithms simulator H-DAS. It consists of a descriptor and a simulator. The descriptor helps us to describe a distributed algorithm. We simulate the distributed algorithm using the simulator to verify the correctness of the algorithm. Since 5 sample algorithms are appended, H-DAS may be useful in studying distributed algorithms. We would like to append more sample algorithms.

Unfortunately, there remain several bugs in H-DAS. For example, the simulation of sample algorithm "Maekawa" stops if we use a coterie "Singleton" or "Crumbling Wall". Therefore, we must fix known bugs.

We would like to add a new function to H-DAS such that if a user defines a predicate on the properties of processes, then the simulator always indicates the Boolean value of the predicate.

References

[1] E. J. H. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes", *Communications of the ACM*, Vol. 22, pp. 281–283, 1979.

[2] V. K. Garg, "Elements of Distributed Computing", John Wiley & Sons, 2002.

[3] O. Har-Tal, "A simulator for self-stabilizing distributed algorithms", `https://www.cs.bgu.ac.il/~projects/projects/odedha/html/`.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", *Communications of the ACM*, Vol. 21, pp. 558–565, 1978.

[5] N. A. Lynch, "Distributed Algorithms", Morgan Kaufmann Publishers, 1996.

[6] M. Maekawa, "A square root N algorithm for mutual exclusion in decentralized systems", *ACM Transactions on Computer Systems*, Vol. 3, pp. 145–159, 1985.

[7] D. Peleg and A. Wool, "Crumbling walls: a class of practical and efficient quorum systems", *Distributed Computing*, Vol. 10, pp. 87–97, 1997.

[8] W. Schreiner, "DAJ – a toolkit for the simulation of distributed algorithms in Java", `https://www.risc.jku.at/software/daj/`, 1998–2013.