Comparing Self-Stabilizing Dining Philosophers through Simulation*

Jordan Adamek[1], Mikhail Nesterenko[1], and Sébastien Tixeuil[2]**

[1] Kent State University
[2] UPMC Sorbonne Universités & IUF

July 18, 2013

Techical report: TR-KSU-CS-2013-01
Department of Computer Science
Kent State University

Abstract. We evaluate the performance of five of the most well-known self-stabilizing dining philosophers algorithms in a read/write atomicity model. The algorithms are evaluated under interleaving, power-set and synchronous execution semantics. For all algorithms, we compute latency and throughput of critical section access, as well as the number of safety violations. These metrics are computed under various load and fault scenarios. On the basis of the evaluation, we propose a combined algorithm that switches between different algorithms to achieve optimal performance.

1 Introduction

A self-stabilizing algorithm [9, 23] eventually resumes correct operation even if it starts from an arbitrary state. Thus, regardless of the nature and extent of a fault, once the fault's influence stops, the self-stabilizing algorithm recovers. This property makes self-stabilization an attractive fault-tolerance technique.

The need to recover from every possible system state makes designing self-stabilizing algorithms a challenge. One approach is to design such algorithms in an execution model with limited number of states and then convert the designed algorithm into a more realistic model. Naturally, such conversion has to preserve correctness as well as self-stabilization of the original algorithm.

One of the most popular transformation methods is to decrease the atomicity of actions in the execution model of the self-stabilizing algorithm. The common source model is combined atomicity (*a.k.a.* state model, or high-atomicity model) and the target model is read/write atomicity (*a.k.a.* low-atomicity model). In the combined model, the algorithm is allowed to read the variables of its neighbors and update its own variables in a single atomic action. In the read/write atomicity model, the process can either

* This paper is eligible for best student paper award.
** This work was supported in part by LINCS.

read a single neighbor's variable (and update its own private variables) or write its public variables.

A low-atomicity self-stabilizing solution to the dining philosophers problem [6, 8] can be used for such a transformation. In dining philosophers, neighbor processes compete for critical section (CS) access and only one process at a time is allowed to execute this critical section. If the execution of a high-atomicity action is carried out as the critical section, the combined model program correctly executes in the low-atomicity model.

With such usefulness of self-stabilizing dining philosophers algorithms, a large number of them is presented in literature [2–5, 7, 13, 15, 17, 19, 21, 22]. With such a variety of algorithms, it is unclear which one would be most suitable for a particular transformation. The research in self-stabilization usually focuses on formally proving correctness of the algorithms and estimating theoretical performance bounds. The number of practical performance evaluation studies in self-stabilization is relatively limited [1, 11, 12, 16, 18, 20, 25]. Probabilistic model checking [11] can be used to assess the performance evaluation of self-stabilizing protocols with a fine grained view, at the cost of scalability (only systems of a few nodes can be evaluated). In this work, we follow the more classical (and scalable) path of simulationbased evaluation [1, 12, 16, 18, 20, 25].

In this study, we set out to compare the performance of five better known self-stabilizing read/write atomicity dining philosophers algorithms. For the purposes of presentation we call them LRA [5], Fuzzy [15], Transformation [19], Alternator [17], and Refinement [22]. The first three algorithms did not quite operate as advertized in the read/write model so we had to modify them to conform to it.

With performance studies, selecting the simulation environment is tricky. What we want to achieve is a comparison of the algorithms themselves. Hence, the environment should be generic not to favor a particular algorithm yet realistic enough so that the comparison is meaningful for practical system design.

For algorithm comparison, we selected random graph topology. We executed algorithms in three popular action execution semantics: interleaving (serial), power-set (distributed) and synchronous. We injected faults of varying extent and also varied the load, *i.e.* the number of processes requesting critical section access. We evaluated the algorithms along the following metrics: safety violations — the number of neighbor processes executing the CS concurrently, latency — the number of actions it takes the algorithm from submission of CS access till its execution, and throughput — the number of CS accesses per action. The first metric shows the algorithms' robustness while the last two are the functional properties.

Our measurements indicate that besides LRA, the robustness of the algorithms was adequate. Yet Transformation, Fuzzy and LRA were found to violate safety even if no faults were injected. This was in part due to the need to adapt the algorithms to the read/write atomicity model. Considering the functional properties, there was no clear winner among the remaining two algorithms: Alternator and Refinement. The throughput of Refinement was better under low load while Alternator outperformed it under high load.

Observing this dichotomy, we propose a combined algorithm that estimates the load and, depending on the load, selects the appropriate algorithm to execute. We propose a load estimation mechanism to be used by this combined algorithm and evaluate its performance.

2 Model

Execution model. An algorithm consists of a set of n processes and a neighbor relation N between them. Processes P_i and P_j are *neighbors* if the pair (P_i, P_j) belongs to N. Each process P_i has a unique identifier i. Each process contains a set of variables and a set of actions. A variable ranges over a fixed domain of values, such as boolean, integer, etc. A variable can be updated, i.e. written to, only by the process that contains it. A variable may be read by its neighbors. An action (guarded command) contains a guard and a command. A *guard* is a boolean predicate over local and neighbor variables. A *command* is a sequence of assignment statements updating the values of local variables.

Atomicity. If the algorithm is *high-atomicity*, the guards and commands may freely mention neighbor variables. A *low-atomicity* algorithm restricts the variables use as follows. Variables are declared as either private or public. A *public* variable can be read by neighbor processes as well as by its own process. A *private* variable can be read only by the process that contains it. In effect, the process action can either update its private variables on the basis of the local variables and neighbor's public variables, or update its public variables on the basis of its own variables. In other words, in a low-atomicity algorithm, a single action is not allowed to read the neighbor's public variables and update the process' own public variables.

Execution semantics. An algorithm *state* is an assignment of values to variables of all processes. An action whose guard evaluates to **true** is *enabled*. An algorithm *computation* is a sequence of steps such that for each state s_i, the next state s_{i+1} is obtained by executing the command of an action enabled in s_i. This disallows the overlap of action execution. That is, the action execution is *atomic*.

Multiple actions may be enabled in a single state. The action selection for execution determines its *execution semantics* also called *scheduler* or *daemon* [10]. In *interleaving* (*centralized*) execution semantics, an arbitrary single enabled action is executed. In *powerset* (*distributed*), an arbitrary subset of enabled actions in non-neighbor processes are executed. That is, in powerset semantics, if actions are enabled in two neighbor processes in a single state, either one or the other action may be selected. In (*maximally*) *synchronous* semantics, every non-neighbor enabled action is selected for execution.

Problem Statement. The problem of *Dining Philosophers* [6, 8] (*Local Mutual Exclusion*) is defined as follows. Any process P_i may request access to the *critical section* (CS) portion of code by setting a read-only boolean $\texttt{request}_i$ to **true**. A solution to the dining philosophers problem ensures that (i) no two neighbor processes have actions executing the CS enabled in the same state; and (ii) a process requesting the CS is eventually allowed to execute such action.

Stabilization. An algorithm is a (*self-*)*stabilizing* solution to the Dining Philosophers Problem if, regardless of the initial state, it eventually satisfies the problem requirements. Due to the arbitrary initial state, the algorithm may allow processes to violate the safety of the Dining Philosophers. For example, the algorithm may start from a state where two neighbors are already in the CS. However, there has to be a finite number of such violations. After the algorithm stabilizes, it has to conform to the problem requirements.

3 Algorithms.

process P_i
parameter $j : (P_i, P_j) \in N$
variables
  public
    ts_i : integer // in case of LRA, range is $(0..n)$
    $ready_i$: boolean
  private
    $\texttt{request}_i$: boolean,
    $ready_i.j$: boolean
    $ts_i.j$: integer // in case of LRA, range is $(0..n)$

actions
  $\texttt{request}_i \wedge \neg ready_i \longrightarrow$
    $ready_i :=$ **true**,
    $ts_i := max(ts_i.k \mid k \in N) + 1$
  $ready_i \wedge (\forall k \in N : ready_i.j =$ **true** $: ts_i \prec ts_i.k) \longrightarrow$
    // critical section
    $ready_i :=$ **false**
    $ts_i := 0$ // LRA action
  $(ts_i.j \neq ts_j) \vee (ready_i.j \neq ready_j) \longrightarrow$ // added to conform to model
    $ts_i.j := ts_j$,
    $ready_i.j := ready_j$

Fig. 1. Transformation and LRA algorithms.

Transformation and LRA. *Transformation* [19] (see Figure 1), uses timestamps to order CS access. In this algorithm, processes enter the CS in the order of their timestamps. Each process P_i has an integer variable ts_i that stores the timestamp of this process. Once the process receives the request for CS access ($\texttt{request}_i$ is **true**), the process selects the timestamp to be the largest among its neighbors and indicates that it is ready to enter the CS by setting $ready_i$ to **true**. When the process determines that it has the smallest timestamp among the ready processes, it executes the CS. In timestamp comparison (denoted $\succ$), in case the integer counters are equal, process identifiers are used to break the tie.

In Transformation, the integer timestamp counter may grow without bound. Such unbounded variables in self-stabilizing algorithms present unique implementation challenges. Since they have to be implemented as finite counters, a corrupted initial state may be such where the counter is close to overflowing. Such overflowing may result in

incorrect algorithm operation. Algorithm *LRA* [5], modifies the behavior of Transformation to eliminate the need for infinite counters. In Figure 1, we outlined the modification introduced by LRA to Transformation. Specifically, once the process executes the CS, the algorithm resets the counter to zero. This way, the counter value never exceeds the total number of processes in the system n. To make it easier to compare with Transformation, we modified the presentation of LRA from its description in the original paper. In particular, we combined entering the CS and exiting the CS into a single action.

Neither Transformation nor LRA explicitly use the atomicity model we describe in this paper. To make it comparable with the other algorithms, we modified Transformation and LRA to differentiate private and public variables as well as actions that update them. Specifically, for each neighbor P_j , we introduced private variables $ready_i.j$ and $ts_i.j$ that store at P_i copies of $ready_j$ and ts_j respectively. We also introduced an action that copies from originals to the local copies if their values differ.

Note that in our model, Transformation and LRA violate correctness. Specifically, each algorithm may infinitely often allow two neighbors to enable their CS executing actions in one state. For example, consider a system of two processes P_a and P_b where $a < b$. Their timestamps are zero. That is, $ts_a = 0$ and $ts_b = 0$. Both processes request to enter the CS. Process P_a selects its timestamp $ts_a = 1$. So does P_b by setting $ts_b = 1$. At this point, the actions that execute the CS are enabled at both processes. The actions remain enabled even if Process P_a copies the new values from P_b. In this case $ready_a.b = true$ and $ts_a.b = 1$. However, since P_a has higher priority than P_b, its CS executing action remains enabled. Despite these correctness issues, we evaluated the performance of both Transformation and LRA to compare them with the other algorithms.

process P_i
parameter $j : (P_i, P_j) \in N$
variables
  public
    $c_i.j$: boolean, link-bit in a color subgraph for process P_j
  private
    $c_i.ji$: boolean, copy of P_j's link-bit
actions
  $(\forall k \in N : k < j : c_i.ki = c_i.k) \wedge (\forall k \in N : k > j : c_i.ki \neq c_i.k) \longrightarrow$
    // critical section
    $(\forall k \in N : c_i.k = \neg c_i.i)$
  $c_i.ji \neq c_j.i \longrightarrow$ // added to conform to model
    $c_i.ji = c_j.i$

Fig. 2. Fuzzy algorithm.

Fuzzy. The *Fuzzy* [15] solution to the Dining Philosophers uses the following idea. In a tree, regardless of edge orientation, there exists at least one sink. The algorithm imposes a read-only set of trees on the system such that every link is covered by at least one tree. Each tree may be disconnected, i.e. it may be a forest. See Figure 2 for illustration. For simplicity, we illustrate the operation of the algorithm on a single tree. Every edge of

the tree has a basic orientation. In our illustration, the edges are oriented towards lower-id processes. Each process P_i, for each neighbor P_j maintains a boolean variable $c_i.j$. The value of this variable and the corresponding neighbor variable $c_j.i$ determine the orientation of link (P_i, P_j). The orientation is selected such that either P_i or P_j may be able to change it by flipping the value of its variable. In our illustration, for neighbor P_k with identifiers lower than j, i.e. $k < j$ the edge is considered oriented towards j if $c.i.k$ is equal to $c.k.i$. When a process determines that it is a sink, i.e. all edges are oriented towards it, the process executes the CS and flips all its link-bits to give priority to its neighbors. No explicit request for the CS is considered. In a multi-tree algorithm, the trees have static priorities. To avoid deadlock, each process collects higher priority tree sinks first while giving up lower-priority sinks without executing the CS.

Just like Transformation and LRA, Fuzzy does not explicitly use the atomicity of the execution model of this paper. Similarly to the previous two, we modified the algorithm to differentiate public and private variables. We introduced a variable $c_i.ji$ that keeps a private copy of the neighbor's link bit variable $c_j.i$. We also added an action that maintains the original and the copy in synch. Just like the other two algorithms, Fuzzy violates correctness in our execution model.

Indeed, consider the system of two processes P_a and P_b such that $a < b$. In the initial state of the computation, let their link-bits $c_a.b$ and $c_b.a$ be set to **false**, while $c_a.ba$ = **false** and $c_b.ab$ = **true**. In this state, the CS-executing actions of both P_a and P_b are enabled. Assume that the execution semantics is interleaving. Consider the following action execution. Process P_b executes the CS. This sets $c_b.a$ to **true**. Next, P_b copies the value of $c_a.b$ and sets $c_b.ab$ to **false**. Then P_a executes the CS and sets $c_a.b$ to **true**. Then, P_a synchronizes the value of $c_a.ba$ and sets it to **true**. Observe that in this state, the CS executing actions of both P_a and P_b are enabled again. This computation may proceed in a similar manner indefinitely.

Alternator. *Alternator* [17] algorithm is shown in Figure 3. In Alternator, each process P_i maintains a bounded counter x_i which signifies the priority of this process. The process executes the CS when it has the lowest priority among neighbors. Process P_i examines the priority of each of its neighbors in sequence. It maintains a private variable v that stores which neighbor to check next. Once that particular neighbor's priority falls behind (signified by **beh**), P_i moves to the next. The priority comparison is modulo bound B. This bound B is set large enough to keep neighbors priority in range. In case of incorrect initial state, neighbor priorities may diverge too far for valid comparison. If Alternator detects that, it sets v to a special value $d+1$ which forces priority reset. Like Fuzzy, Alternator does not use explicit request for CS access. Instead, each process is given the opportunity to execute the CS regardless of need.

Unlike Transformation, LRA, or Fuzzy, Alternator distinguishes between public and private variables and operates in our atomicity model correctly without modifications.

Refinement. *Refinement* algorithm [22] is shown in Figure 4. In this algorithm, synchronization between each pair of neighbor processes P_i and P_j is achieved through a self-stabilizing sequence of variables: $a_i.j$, $b_j.i$, $c_j.i$ and $d_i.j$. First and last are in process P_i, the other two are in process P_j. Once P_i learns that there is a request for

process P_i
parameter $j : (P_i, P_j) \in N$
constants
$B \geq n^2 + 1$, where n is the system size
$d = |N|$, number of neighbors
variables
public
$x_i : (0..B-1)$
private
$v : (0..d+1)$
operators
$(x_i \textbf{ beh } x_j) \equiv ((0 < ((x_j - x_i) \textbf{ mod } B) \leq n) \vee (x_i = x_j \wedge j < i))$
$(x_i \textbf{ far } x_j) \equiv (\neg(x_j \textbf{ beh } x_i) \wedge \neg(x_i \textbf{ beh } x_j))$
actions
$(x_i \textbf{ beh } x_j) \wedge (j = v) \wedge (v < d) \longrightarrow$
$v := v + 1$
$v = d \longrightarrow$
//critical section
$x_i := x_i + 1 \textbf{ mod } B$,
$v := 0$
$(\exists k : 0 \leq k < d : ((x_i \textbf{ far } x_k) \wedge (x_i > x_k))) \longrightarrow$
$v := d + 1$
$v = d + 1 \longrightarrow$
$x_i := 0$,
$v := 0$

Fig. 3. Alternator algorithm.

CS access (`request` $=$ **true**), it increments $a_i.j$. This increment is propagated along the sequence of variables. Piggybacked on this propagation is synchronization of public variables of the neighbors with the local private copies at each process. Once $d_i.j$ holds the same value as $a_i.j$, process P_i learns that the neighbor has its up-to-date values.

The lower identifier processes have higher priority. The process P_i enters the CS once it has synchronized with all of its neighbors while higher priority processes are not requesting CS access ($\neg r_i.k$). To ensure fairness, after executing the CS, a process lets its lower priority neighbors enter the CS once before requesting CS again. Process P_i uses variable $y_i.j$ to keep track of its lower priority neighbor P_j CS access.

Similar to Alternator, Refinement operates correctly in our atomicity model without modifications.

4 Simulation Model

Topologies. We used the program by Viger and Latapy [24] to generate random connected graph topologies for our system simulation. This program generates graphs that are sufficiently diverse, yet do not favor a particular algorithm or computing environment. The parameters for graph generation were as follows. The edge distribution is by power-law with minimum node degree 1, maximum node degree 15, average degree 3 and $\alpha = 2.5$. The number of nodes in a graph is 100. Figure 5 shows an example generated graph.

Computations, faults, load. The computations for all of the evaluated algorithms were $1,000$ states. To fairly represent initial state for the recovery, we started with a "zero"

process P_i
parameter $j : (P_i, P_j) \in N$
variables
public
$ready_i$: boolean,
$a_i.j, c_i.j : (0..3)$
private
$\texttt{request}_i$: boolean,
$r_i.j, y_i.j$: boolean,
$b_i.j, d_i.j : (0..3)$
actions
$\texttt{request}_i \wedge \neg ready_i \wedge (\forall k : a_i.k = d_i.k) \wedge (\forall k > i : \neg y_i.k) \longrightarrow$
$ready_i := \textbf{true}$,
$(\forall k > i : y_i.k := r_i.k,\ a_i.k := (a_i.k + 1)\ \textbf{mod}\ 4)$

$ready_i \wedge (\forall k : a_i.k = d_i.k) \wedge (\forall k < i : \neg r_i.k) \longrightarrow$
// critical section
$ready_i := \textbf{false}$,
$(\forall k < i : a_i.k := (a_i.k + 1)\ \textbf{mod}\ 4)$

$c_i.j \neq b_i.j \longrightarrow$
$c_i.j := b_i.j$

$r_i.j \neq ready_j \vee (b_i.j \neq a_j.i) \vee (d_i.j \neq c_j.i) \vee (j > i \wedge \neg ready_j \wedge y_i.j) \longrightarrow$
$r_i.j := ready_j$,
$b_i.j := a_j.i$,
$d_i.j := c_j.i$,
if $j > i \wedge \neg ready_j \wedge y_i.j$ **then** $y_i.j := \textbf{false}$ **fi**

Fig. 4. Refinement algorithm.

state for each algorithm: each variable holds the lowest value of its range. We then picked a random state among the $1,000$ states of the computation and introduced a single fault. A fault is a random perturbation of values of the variables for a particular process. The extent of the fault varied from zero processes to all 100 modeled processes. In the case of all processes being subject to the fault, the initial state is in effect random. We varied the number of CS requests from 1 to 100 and randomly assigned the requests to processes. For each particular data point, we ran $1,000$ simulated computations. The computations were done in three execution semantics: interleaving, powerset and synchronous. For interleaving semantics, we randomly selected one of the enabled actions. For powerset and synchronous, the action selection procedure is more involved as algorithms are proven correct only for the interleaving semantics of execution. Therefore, the actions of two neighbor processes cannot be executed concurrently. The enabled action selection is as follows. We randomly choose one of such action. After that, we eliminate the neighbor process actions from consideration and repeat the selection. For powerset semantics the number of selected enabled actions is chosen at random between 1 and the total number of enabled actions. For synchronous semantics we continue the selection until no more actions remain.

Metrics. For the simulated algorithms, we computed two functional and one fault-tolerance metric. The functional metrics are: *latency* — average number of states between request and corresponding CS access, and *throughput* — average number of CS access per action. Latency quantifies how long it takes the algorithm to respond to a CS

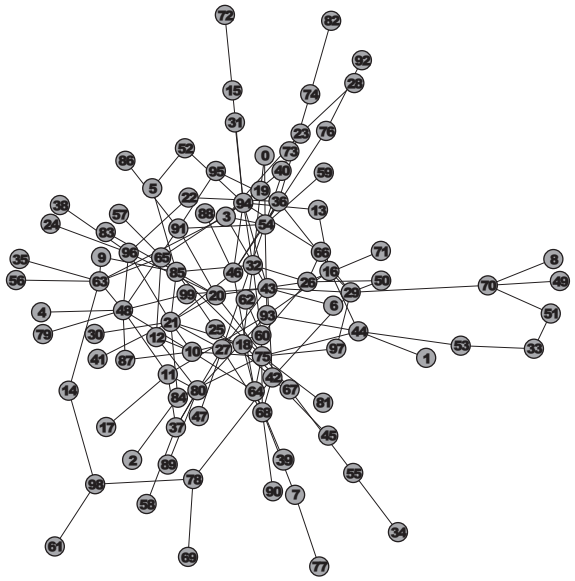


Fig. 5. Example random graph.

request, while throughput shows how much computer resources are required to satisfy a CS request. Two algorithms, Fuzzy and Alternator, do not accept requests for CS. For these algorithms, we assumed that request arrives at a particular process and then counted the number of steps for the next CS access by this process. In other words, we discounted unsolicited CS accesses.

Usually, the fault-tolerance for self-stabilizing algorithms is measured by the time it takes the algorithm to return to a legitimate state. A state is legitimate if a computation never violates specification requirements after it arrives at this state. However, Transformation, LRA and Fuzzy were modified to accomodate read/write atomicity model. Hence, the legitimate states defined in the paper where they were presented no longer apply. Moreover, three algorithms violate safety requirements of the Dining Philosophers problem even if no faults are present in the system. That is, none of the states of these algorithms in our model are legitimate.

Therefore, to make a fair comparison, we counted the number of safety violations for each computation. For that we computed the number of states where at least two distinct neighbors have their CS executing actions enabled. Depending on the execution semantics, a certain pair of neighbors may retain their enabled actions across several consequent states. We counted such a pair as a single safety violation. However, if the same pair gets one of the CS executing actions disabled and then re-enabled, it is counted as a separate safety violation.

Algorithm configuration. Alternator requires bound B to be at least $n^2 + 1$. We selected it to be exactly equal to this value. Transformation uses unbounded timestamp counters. To make it comparable to the other algorithms, we used the same bound B

for these counters. Fuzzy requires a set of trees to be configured for each topology. We generated the trees as follows. We randomly selected a spanning tree. We then removed the selected edges and randomly selected a spanning tree of the remaining subgraph. We continued the process until all the edges were covered.

5 Simulation Results

Presentation. First we show the tolerance properties of the algorithms. We plot the number of safety violations depending on the load and number of injected faults. Figure 6 demonstrates the number of safety violations with a constant fault rate of 50% and varying load, i.e. the number of processes requesting CS access. The figure show the algorithm's performance under interleaving, powerset and synchronous execution semantics.

Figure 6 demonstrates safety properties of the algorithm in a different dimension: safety violations are plotted under varying the extent of the fault while load is held constant.

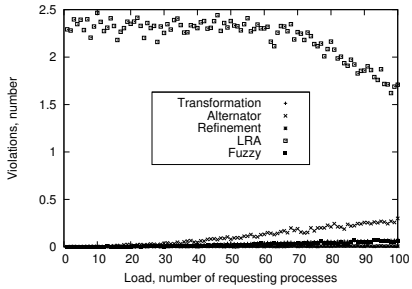
Figures 7 and 8 compare the functional properties of the algorithms. This time we did not introduce faults, varied load and counted latency and throughput for each algorithm. Latency is counted as the average number of action executions from request till CS access. Throughput is the number of CS accesses per action execution. Latency measures the synchronization delay induced by the algorithm. Throughput measures the algorithm's ability to serve CS requests with maximum efficiency.

Analysis. The safety properties of the five algorithms are comparable except for LRA which allows significantly greater number of safety violations. The reason is that the CS access in LRA is based on timestamps that are reset to zero after each CS access. Hence, the selected timestamps seldom increase past 1 or 2. A fault can relatively easily perturb processes to have the same timestamps and hence allow the processes to concurrently enter the CS.

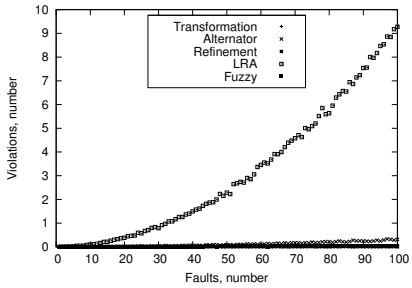
Let us discuss the functional properties of the algorithm. Alternator exhibits the highest latency among the algorithms followed by Fuzzy. In Alternator, before accessing the CS, every process has to sequentially examine the state of each neighbor. This induces the considerable delay. In Fuzzy, each process enters the CS only after it is a sink in every color tree; this tree-based synchronization induces the delay.

Throughput plot demonstrates interesting properties of the algorithms. For Alternator and Fuzzy the throughput grows linearly with load. This is due to the fact that the two algorithms execute CS actions regardless of requests. That is, they keep serving CS access whether the application program needs it or not. We only counted solicited CS accesses. Hence, as the load increases, the operation of the two algorithms becomes more efficient since more of the produced CS accesses are needed.

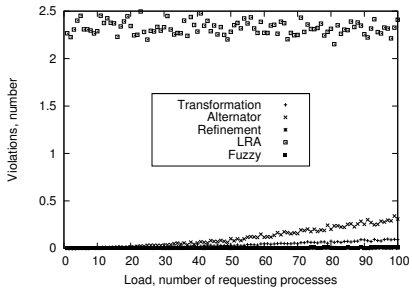
The performance of LRA and Transformation is interesting. Even though the difference in their code is minor, the results that the two algorithms demonstrate differ significantly. While Transformation's throughput grows linearly, LRA's stays nearly stable as the load increases. Let us discuss the performance of Transformation first. In this algorithm, to enter the CS, each process selects a new timestamp and waits until



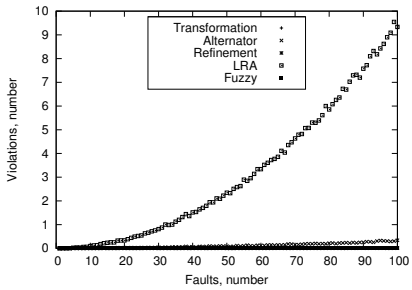
(a) Interleaving semantics (Constant fault rate of 50%)



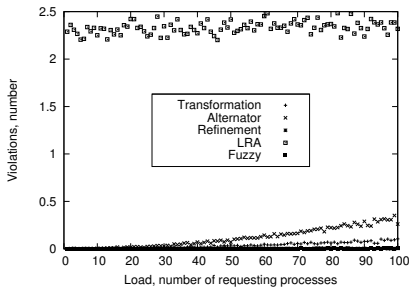
(b) Interleaving semantics (Constant load rate of 50%)



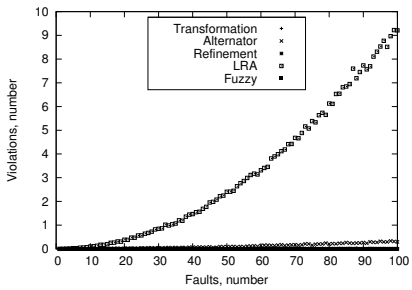
(c) Powerset semantics (Constant fault rate of 50%)



(d) Powerset semantics (Constant load rate of 50%)



(e) Synchronous semantics (Constant fault rate of 50%)



(f) Synchronous semantics (Constant load rate of 50%)

Fig. 6. Safety violations.

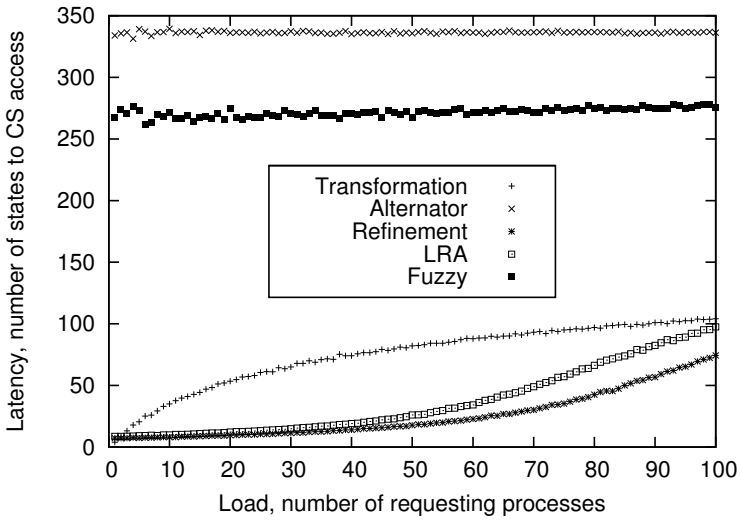


Fig. 7. Latency under interleaving semantics and no faults.

it is the smallest among neighbors. In low atomicity, for each CS access, the neighbor processes have to copy the newly selected timestamp to their local variables. As the load increases, the neighbors may be able to delay this copy and thus spread it across multiple CS accesses. This leads to a linear increase in performance with load increase.

For LRA, the situation is even more curious. There, the timestamps are reset to zero once the process is done accessing the CS. Thus, if the load is low, a neighbor may not notice timestamp change at all and thus save on the action execution. This operation offsets the gradual efficiency increase with load that is similar to Transformation. Overall, the throughput of LRA appears not to change significantly with load increase.

6 Combined Algorithm

Among the algorithms that do not violate safety without faults, our simulation results indicate that there is no single algorithm that always outperforms the others. For the throughput metric, Refinement performs better than Alternator under low load. Then, as the load increases, Alternator becomes more efficient. The break-even load point is slightly above 60%.

To take advantage of these differences in performance, we propose a combined algorithm that switches from Refinement to Alternator and back depending on the load. For this algorithm to work, the system has to be able to estimate the load.

We propose a simple local load estimation procedure. Every process maintains a boolean variable $vote$ to determine whether the load in the system is high. Each process determines whether itself and its neighbors are requesting the CS. If the number of CS requests in the neighborhood is greater than the threshold point of 60%, the process sets $vote$ to **yes**. Otherwise, the process sets it to **no**. This vote is averaged across the

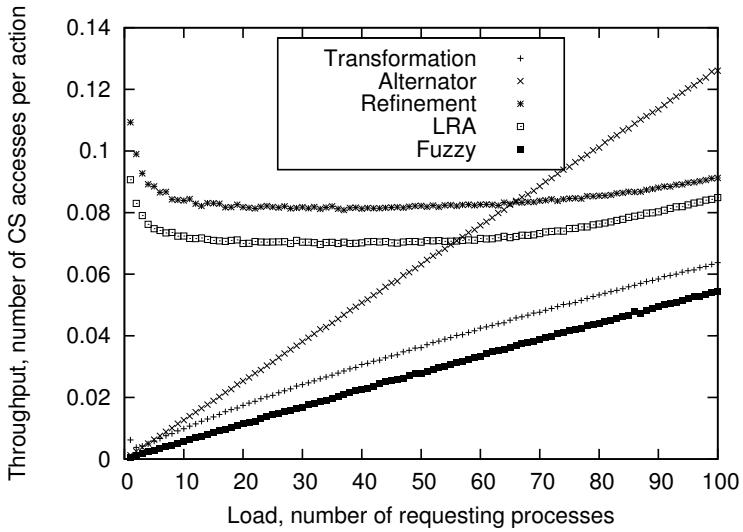


Fig. 8. Throughput under interleaving semantics and no faults.

computation of a particular algorithm. Figure 9 plots the number of votes depending on the actual system load for Alternator and Refinement. Our simulation results indicate that such voting can be used to predict the system load quite accurately. For example, for 60% load, the number of processes voting **yes** for Alternator and Refinement is 15% and 5% respectively. Hence, this voting procedure can be used for algorithm switching in the combined algorithm.

Note that to prevent jitter, there should be an insensitivity range around the break-even point and, depending on the specifics of the system, the combined algorithm should switch to Alternator somewhere above the threshold load of 60%, for example at 65%, and switch back to Refinement somewhere below the threshold, for example at 55%. Note that efficiency gains of using combined algorithms are offset by the overhead of gathering the vote information and switching between algorithms. The actual implementation of vote gathering and reset is left for future research.

7 Conclusions

In conclusion, we would like to stress the importance of performance evaluation for self-stabilizing algorithms as it quantifies their robustness and efficiency in a practical setting. Moreover, it often exposes algorithm features that are not covered in theoretical studies. For example, as it was observed for LRA, the limited variable range may lead to greater possibility of a fault making the corrupting state appear legitimate and leading to safety violations.

The efficiency evaluations have lead us to propose the combined algorithm that switches between two best performing algorithms depending on the load. However, this

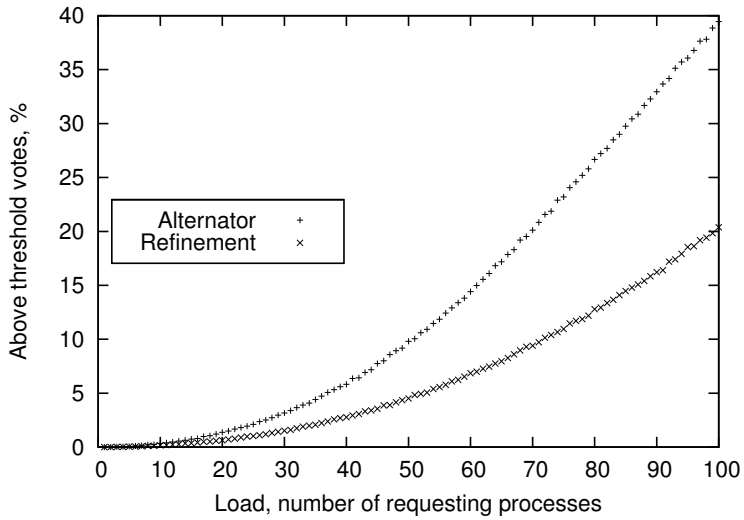


Fig. 9. Threshold votes under interleaving semantics and no faults.

combined algorithm implementation is not complete. The mechanism of load information gathering and algorithm switching is to be developed.

In other possible future work, we would like to suggest testing the self-stabilizing dining philosophers solutions in a greater variety of specific execution models. One interesting testing model might be wireless radio communication model, for which existing probabilitically correct solutions [14] that are worth comparing using our metrics.

References

1. Jordan Adamek, Mikhail Nesterenko, and Sébastien Tixeuil. Evaluating practical tolerance properties of stabilizing programs through simulation: the case of propagation of information with feedback. In *Proceedings of the 14th international conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'12, pages 126–132, 2012.
2. G. Antonoiu and P.K. Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph that stabilizes using read/write atomicity. In *EuroPar'99*, volume 1685 of *LNCS*, pages 823–830. Springer-Verlag, 1999.
3. J. Beauquier, A.K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *12th International Symposium on Distributed Computing*, pages 223–237. Springer, 2000.
4. C. Boulinier, F. Petit, and V. Villain. When graph theory helps self-stabilization. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 150–159. ACM Press, 2004.
5. S. Cantarell, A.K. Datta, and F. Petit. Self-stabilizing atomicity refinement allowing neighborhood concurrency. In *6th International Symposium on Self-Stabilizing Systems*, volume 2704 of *LNCS*, pages 102–112. Springer, 2003.
6. K.M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

7. Praveen Danturi, Mikhail Nesterenko, and Sébastien Tixeuil. Self-stabilizing philosophers with generic conflicts. *ACM Trans. Auton. Adapt. Syst.*, 4(1):7:1–7:20, February 2009.
8. E. Dijkstra. *Cooperating Sequential Processes*. Academic Press, 1968.
9. Shlomi. Dolev. *Self-stabilization*. MIT Press, March 2000.
10. Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.
11. Narges Fellahi, Borzoo Bonakdarpour, and Sébastien Tixeuil. Rigorous performance evaluation of self-stabilization using probabilistic model checking. In *Proceedings of the International Conference on Reliable Distributed Systems (SRDS 2013)*, Braga, Portugal, September 2013. IEEE, IEEE Computer Society.
12. M Flatebo and AK Datta. Simulation of self-stabilizing algorithms in distributed systems. In *Proceedings of the 25th Annual Simulation Symposium*, pages 32–41, 1992.
13. M.G. Gouda and F. Haddix. The alternator. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 48–53. IEEE Computer Society, 1999.
14. Ted Herman and Sébastien Tixeuil. A distributed tdma slot assignment algorithm for wireless sensor networks. In *Proceedings of the First Workshop on Algorithmic Aspects of Wireless Sensor Networks (AlgoSensors'2004)*, number 3121 in Lecture Notes in Computer Science, pages 45–58, Turku, Finland, July 2004. Springer-Verlag.
15. S.T. Huang. The fuzzy philosophers. In J. Rolim et al., editor, *Proceedings of the 15th IPDPS 2000 Workshops*, volume 1800 of *LNCS*, pages 130–136, Cancun, Mexico, May 2000.
16. Colette Johnen and Fouzi Mekhaldi. Self-stabilization versus robust self-stabilization for clustering in ad-hoc network. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par (1)*, volume 6852 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2011.
17. Sandeep S. Kulkarni, Chase Bolen, John Oleszkiewicz, and Andrew Robinson. Alternators in read/write atomicity. *Information Processing Letters*, 93(5):207–215, March 2005.
18. Nathalie Mitton, Bruno Séricola, Sébastien Tixeuil, Eric Fleury, and Isabelle Guérin-Lassous. Self-stabilization in self-organized multihop wireless networks. *Ad Hoc and Sensor Wireless Networks*, 11(1-2):1–34, January 2011.
19. M. Mizuno and M. Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 1998.
20. N. Mullner, A. Dhama, and O. Theel. Derivation of fault tolerance measures of self-stabilizing algorithms by simulation. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 183 –192, april 2008.
21. M. Nesterenko and A. Arora. Dining philosophers that tolerate malicious crashes. In *22nd International Conference on Distributed Computing Systems (ICDCS)*, pages 191–198. IEEE, July 2002.
22. M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. *Journal of Parallel and Distributed Computing*, 62(5):766–791, 2002.
23. Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009.
24. Fabien Viger and Matthieu Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *COCOON: Annual International Conference on Computing and Combinatorics*, 2005.
25. Sally K. Wahba, Jason O. Hallstrom, Pradip K. Srimani, and Nigamanth Sridhar. *SFS*^{3}: a simulation *f*ramework for *s*elf-*s*tabilizing *s*ystems. In *SpringSim*, page 172, 2010.