Dining Philosophers that Tolerate Malicious Crashes

Mikhail Nesterenko [*]
Department of Computer Science,
Kent State University, Kent, OH 44242,
mikhail@cs.kent.edu

Anish Arora [†]
Dept. of Computer and Information Science,
Ohio State University,
Columbus, OH 43210,
anish@cis.ohio-state.edu

Abstract

We present a solution to the problem of dining philosophers. Our solution tolerates malicious crashes. In a malicious crash *the failed process behaves arbitrarily for a finite time and then ceases all operation undetectably to other processes. The tolerance of our solution is achieved by the combination of stabilization and crash failure locality. Stabilization allows our program to recover from an arbitrary state. Crash failure locality ensures that only a limited number of processes are affected by a process crash. The crash failure locality of our solution is optimal. Finally, we argue that the malicious crash fault model and its extensions are worthy of further study as they admit tolerances that are not achieved under stronger fault models and are unnecessary under weaker fault models.*

1 Introduction

Malicious crashes. A fault due to a component malfunction or environmental influence frequently results in arbitrary behavior of the affected part of the distributed system. The faulty part of the system either eventually recovers or ceases to operate. An arbitrary behavior of a faulty component of the system is traditionally modeled as *Byzantine failure* [16] where the faulty process may behave arbitrarily. Yet Byzantine failures are a relatively expensive type of failures to counter. For example, in all Byzantine tolerant solutions the number processes that can become Byzantine is restricted to a certain fraction of the total number of processes in the system.

Halting failure [13], on the other hand, is a common failure semantics assumed in studies of process faults. *Halting failure* specifies that the failed process does not do anything. If the other processes know when the process fails this halting failure is *fail-stop*; if not – it is *crash.* There is a special case of crash failure – initially dead process. An *initially dead process* does not do anything throughout the operation of the system. To counter crash failures the program has to have a finite failure locality [6]. A program is *m-failure local* if the distance between the processes affected by the crash and the crashed process is at most m (m is called *locality constant.*) This type of failure can be handled relatively inexpensively: there is no limit on the number of processes that can fail.

A transient failure is another type of failure that is relatively easy to counter. A *transient failure* perturbs the state of the system for a finite amount of time and leaves the system in arbitrary state. Stabilization is used to counter transient failures. A *stabilizing program* is able to start from an arbitrary incorrect state, arrive at a legitimate state, and continue correct operation thereafter.

We combine the latter two failure types to model malicious crashes. We define *malicious crash* to be a crash in which the faulty process makes a finite number of arbitrary steps before halting. Malicious crashes approach Byzantine failures in their generality. Yet, we demonstrate that this failure type can be handled effectively and inexpensively. We call non-malicious crashes *benign.*

Tolerating malicious crashes. A problem specifies a set of properties to which the program solving this problem has to conform. Consider a problem whose specification can be relativized with respect to any subset of the processes of the system. For example, we state the leader election problem $\mathcal{LE}$ as follows. Prob-

*This research was supported in part by DARPA contract OSU-RF #F33615-01-C-1901.

†This research was supported in part by DARPA contract OSU-RF #F33615-01-C-1901, NSF grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, and a grant from Microsoft Research.

lem $\mathcal{LE}$ for a subset P has two properties: (1) termination – every process in P finishes; (2) election – upon termination all processes in P select one process (leader) in P. Note that our specification of $\mathcal{LE}$ is identical to the classical specification of the leader election problem if P contains all processes in the system.

Let P be the subset of processes where each member's distance to the crashed processes is no less than the locality constant m. Given a problem $\mathcal{A}$ and a locality constant m independent of the system size we define the *malicious crash tolerance problem* $\mathcal{MCA}$ as follows. A program is a solution to $\mathcal{MCA}$ if for any set of crashed processes, the properties of $\mathcal{A}$ are eventually satisfied for P. The following proposition states a sufficient condition for a program to be a solution to $\mathcal{MCA}$.

Proposition 1 *Given a problem $\mathcal{A}$, to solve the malicious crash tolerance problem $\mathcal{MCA}$ with locality constant m it is sufficient for the program to guarantee the following. Starting from an arbitrary state and arbitrary set of initially dead processes, the program ensures that for the processes whose distance from the crashed processes is at least m the properties of the original problem $\mathcal{A}$ are eventually satisfied.*

Note that this proposition excludes the necessity to consider the case that a malicious crash occurs when the system is recovering from another such crash. However, the specification of $\mathcal{MCA}$ is trivially satisfied when each process in the system has at least one crashed process in the distance no greater than m. Hence, either the crashes eventually stop allowing the system to recover or there are no processes unaffected by the crashes. Hence the proposition. In this paper we define a malicious crash dining-philosophers problem and present a solution to it.

Dining-philosophers problem and solution. The dining philosophers problem (or diners for short) [8] is defined as follows. A set of processes is joined by an arbitrary neighbor relation. A process can be either *thinking*, *hungry*, or *eating*. A solution to the problem satisfies the following properties: *safety* – no two neighbors are eating in the same state; *liveness* – every hungry process eventually eats provided that no process eats indefinitely.

Choy and Singh [6] prove that the minimum crash failure locality (for benign crashes) for the diners problem is 2. Hence, the malicious crash dining-philosopher problem with $m < 2$ does not have a solution. The program we present in this paper solves the problem with $m = 2$ which is optimal. Naturally our program tolerates transient faults and benign crashes occurring separately. In addition our program masks benign crashes outside of crash failure locality. That is, the program continue to operate correctly when a benign crash occurs.

Related work. Classical stabilization does not handle process crashes in the asynchronous system model. In an asynchronous system it is impossible to deterministically distinguish a crashed process from a slow one [10]. The only class of permanent failures a classic stabilizing program can admit is fail-stop failures. Such a failure is treated as a system topology update from which the system stabilizes.

Research in FTSS (fault tolerance and stabilization) [3, 4] explores the idea of joining stabilization and 0-failure locality: every non-faulty process eventually behaves correctly. However, due to the restrictive definition few problems are FTSS-solvable. In particular the diners problem is not FTSS-solvable.

A few non-stabilizing solutions to the diners problem with optimal failure locality are known [7, 17, 18]. A number of stabilizing solutions are published as well [1, 2, 11, 12, 14, 15]. However, to the best of our knowledge no solution combines failure locality and stabilization.

Solution ideas. Our solution is based on a well-known idea of maintaining a partial order of priority among processes [5]. A link between each pair of neighbor processes is assigned a direction (priority) such that the resultant priority graph is acyclic. The priority graph defines ancestors and descendants for every process. A hungry process is allowed to eat only if its direct ancestors are not hungry. A deadlock is not possible since the graph is acyclic and it is guaranteed that at least one process eventually eats. After eating, the process changes its priority to become the descendant of its neighbors. This operation keeps the graph acyclic and ensures liveness property if all processes make progress.

The dependency graph can potentially form arbitrary long chains of waiting processes which jeopardizes liveness in case one of the waiting processes crashes. To ensure crash failure locality we use dynamic threshold preemption mechanism [7, 18]. To maintain *dynamic threshold* a process yields to its direct descendants when there is a hungry direct ancestor.

Liveness property can also be violated if the dependency graph contains cycles. To guarantee stabilization, our program eventually breaks every cycle. To break a cycle each process maintains the distance to its farthest descendant. A cycle is detected if this distance exceeds the system's diameter. In this case the process that detects the cycle becomes the descendant of all its neighbors thus breaking the cycle. We assume that the diameter of the system is known to every process.

To simplify the presentation we describe our solution in the shared-memory model where a process can read its neighbors' variables. Each pair of neighbors also shares a variable either of them can update in a restricted manner. In the conclusion we discuss a way of implementing our program in the message-passing model.

Organization of the paper. We describe our program in Section 2 and prove it correct in Section 3. In Section 4 we summarize the results presented in the paper, describe how to transform our program to the message-passing model and discuss possible extensions of the malicious crash fault model.

2 Program description

Model. A *program* consists of a set of processes and a binary reflexive symmetric relation N between them. Processes p and q are *neighbors* if $(p, q) \in N$. A process consists of a set of variables and actions. Process' variables are *local* to this process. An *action* consists of a guard and a command. A *guard* is a predicate containing local and neighbor variables. A *command* is a sequence of statements assigning new values to local variables.

Program *state* is an assignment of values to variables of all processes of the program. An action is *enabled* at some program state if its guard is **true** at this state. A *computation* is a maximal fair sequence of states such that for each state s_i the state s_{i+1} is obtained by executing the command of an action that is enabled at s_i. Maximality of a computation means that the computation is either infinite or it terminates in a state where none of the actions is enabled. We assume that action execution in a computation is *weakly fair*. That is, if an action is enabled in all but finitely many states of an infinite computation then this action is executed infinitely often.

A state *conforms* to a predicate if this predicate is **true** in this state; otherwise, the state *violates* the predicate. By this definition every state conforms to predicate **true** and none conforms to **false**. Let R and S be predicates on the states of the program. Predicate R is *closed* with respect to the program actions if every state of the computation that starts in a state conforming to R also conforms to R. Predicate R *converges* to S if R and S are closed and any computation starting from a state conforming to R contains a state conforming to S. The program stabilizes to R iff **true** converges to R.

$$
\begin{array}{ll}
 & \textbf{process } p \\
 & \textbf{const } \; D = \text{diameter of the system} \\
 & \textbf{var} \\
 & \qquad \mathit{needs}().p : \text{boolean}, \\
 & \qquad \mathit{state}.p : \{T, H, E\}, \\
 & \qquad (\forall q :: \mathit{priority}.p.q \equiv \mathit{priority}.q.p : \{p, q\}), \\
 & \qquad \mathit{depth}.p : \text{integer} \\
 & *[\\
\mathit{join}: & \qquad \mathit{needs}().p \wedge (\mathit{state}.p = T) \wedge \\
 & \qquad (\forall q : \mathit{priority}.p.q = q : \mathit{state}.q = T) \longrightarrow \\
 & \qquad\qquad \mathit{state}.p := H \\
 & \;\; [] \\
\mathit{leave}: & \qquad (\mathit{state}.p = H) \wedge \\
 & \qquad (\exists q : \mathit{priority}.p.q = q : \mathit{state}.q \neq T) \longrightarrow \\
 & \qquad\qquad \mathit{state}.p := T \\
 & \;\; [] \\
\mathit{enter}: & \qquad (\mathit{state}.p = H) \wedge \\
 & \qquad (\forall q : \mathit{priority}.p.q = q : \mathit{state}.q = T) \wedge \\
 & \qquad (\forall q : \mathit{priority}.p.q = p : \mathit{state}.q \neq E) \longrightarrow \\
 & \qquad\qquad \mathit{state}.p := E \\
 & \;\; [] \\
\mathit{exit}: & \qquad (\mathit{state}.p = E) \vee (\mathit{depth}.p > D) \longrightarrow \\
 & \qquad\qquad \mathit{state}.p := T, \\
 & \qquad\qquad \mathit{depth}.p := 0, \\
 & \qquad\qquad (\| q :: \mathit{priority}.p.q := q) \\
 & \;\; [] \\
\mathit{fixdepth}: & \qquad (\exists q : \mathit{priority}.p.q = p : \\
 & \qquad \mathit{depth}.p < \mathit{depth}.q + 1 \longrightarrow \\
 & \qquad\qquad \mathit{depth}.p = \mathit{depth}.q + 1) \\
 & \;\;]
\end{array}
$$

Figure 1. Solution to dining-philosophers with malicious crashes

Variables and actions of our program. The variables and actions of each process p are shown in Figure 1. Constant D holds the diameter of the system (i.e. the maximum distance between two processes in it). Function $\mathit{needs}().p$ signifies whether p wants to eat; the function evaluates to **true** arbitrarily. Process p records if it is thinking, hungry, or eating in variable *state.p* as values T, H, or E respectively. To specify the order in which the neighbors eat, each pair of neighbor processes p and q maintain a shared variable *priority.p.q* (which is the same variable as *priority.q.p*). This variable holds the identifier of either p or q. Process p can check the value of *priority.p.q* or set it to q. If $\mathit{priority}.p.q = q$ we assume that the edge between p and q is directed towards p. Thus, *priority* assigns direction to the neighbor relation and forms a directed *priority graph*. A process that can be reached from p in the priority graph is a *descendant* of p. A process from which p can be reached is an *ancestor* of p. Variable $depth.p$ holds the distance to p's farthest descendant. It is used to break cycles in the priority graph.

Each process contains five actions to solve the diners problem. There is also an implicit action to model process crash. This action stops the process from executing any other action for the rest of the computation. A process is *dead* if it crashed, it is *live* otherwise.

When process p wants to eat ($needs().p = \mathbf{true}$) and p as well as its direct ancestors is thinking, p becomes hungry by executing action *join*. If there is a direct ancestor who is hungry then p executes *leave* and continues thinking. This mechanism allows p's descendants to proceed while p's ancestors are eating or hungry and block p's progress. When p is hungry, its direct ancestors are thinking and its direct descendants are not eating, p starts eating by executing *enter*. Process p finishes eating by executing *exit*. By executing *exit*, p sets $priority.p.q$ for each neighbor to q thus giving q higher priority.

Action *fixdepth* is used to break cycles in the priority graph. If one of p's direct descendants q has $depth.q$ set higher than $depth.p - 1$, then p executes *fixdepth* adjusting $depth.p$. If there is a cycle in the graph where all processes are live, at least one of these process has *fixdepth* enabled. The cycle processes keep executing *fixdepth* until $depth$ in one of the processes p exeeds D. This enables *exit* in p. The execution of *exit* breaks the cycle.

Example operation. The example in Figure 2 shows a fragment of a computation of the program. In the first state process a dead. Since a is eating, its neighbors: b and c are forever prevented from eating (they are blocked.) Process b remains hungry and c – thinking. Process d is hungry, yet it has a direct ancestor b which is blocked in hungry state. This prevents d from eating. We use dynamic threshold: d executes *leave* and yields to its direct descendant e. Thus, the effect of a's crash is contained within the distance of 2.

In the first two states e, f and g form a cycle in the priority graph. If this cycle is not broken, these processes can forever alternate between hungry and thinking without ever eating. However, in the second state g detects that $depth.g$ is 4 which is greater than the system's diameter: 3. Process g executes *exit* breaking the cycle and allowing e to eat after the third state.

3 Correctness proof

To demonstrate the correctness of the program we need to show that it is crash failure local in the presence of transient faults. We state a certain predicate I that is closed with respect to the program actions. This invariant is parameterized: it is defined for live processes only. In Subsection 3.2 we demonstrate that I guarantees safety and liveness properties of the diners problem for any live process whose distance to every dead process is no less than 2. In Subsection 3.1 we show that the program stabilizes to I in the absence of process crashes. According to Proposition 1 it is sufficient to prove correctness. In Subsection 3.2 we also demonstrate that the safety and liveness properties of the processes outside of crash failure locality are not affected by benign process crashes. Hence, our program masks them. Throughout this section we use "crash" to denote a benign crash.

3.1 Stabilization

We show that the program stabilizes to the predicates stating the following three properties: priority graph is acyclic (except for the cycles containing dead processes) – Lemma 1; the processes cannot change priority graph topology without eating first – Lemma 3; live neighbors cannot eat simultaneously – Lemma 4. We define the program invariant I to be the conjunction of these predicates.

Priority graph acyclic.

Lemma 1 *The program stabilizes to the following predicate:*

If the priority graph contains a cycle,
at least one process in the cycle is dead. $\quad (NC)$

Proof: The only action that manipulates the topology of the priority graph is *exit*. This action directs all edges incident to p towards p. Redirecting edges in this manner does not create cycles. Hence, no new cycles are created during any computation of the program and NC is closed.

Let us suppose the priority graph contains a cycle C such that none of the processes in C are dead. If at least one process in C executes *exit*, the cycle is broken. In any state of the system *fixdepth* is enabled in at least one process p in C. The execution of *fixdepth* increases $depth.p$. The only action which decreases $depth.p$ is *exit*. Thus, if *exit* is not executed by any process in C, in at least one process q in C, $depth.q$ eventually exceeds D which enables *exit*. This action stays enabled

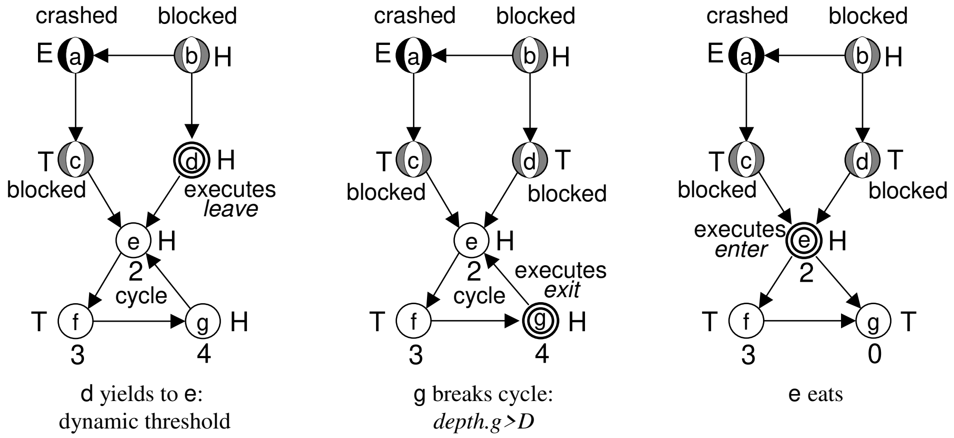

Figure 2. Example operation

until q executes it and breaks C. Since new cycles cannot be created during the program's computation, the program converges to NC. □

All processes shallow. We define shallow process p such that it neither can execute *exit* itself (due to *depth.p* exceeding D) nor cause its ancestors to execute *exit* by propagating a large value of *depth*. The latter is ensured by requiring that for every p's direct descendant q, either *depth.q* is small enough that it does not grow past D if propagated to p's live ancestors or p's *fixdepth* is disabled. Let $l.p$ for a process p be the length of the longest chain of live ancestors of p (including p itself). A process p is *shallow* if it conforms to predicate $SH.p$.

$$(p \text{ is dead}) \vee \\ ((depth.p \leq D) \wedge \\ (\forall q : priority.p.q = q : (depth.q + l.p \leq D) \vee \\ (depth.q + 1 \leq depth.p))) \qquad (SH.p)$$

A process is *deep* if it is not shallow. A shallow process may still become deep due to its descendants. Hence we define a stably shallow process. A process is *stably shallow* if it is shallow and it is either dead or all of its live descendants are shallow. A shallow process is *unstably shallow* otherwise.

Lemma 2 *If a process is stably shallow in the initial state of the computation, it remains stably shallow throughout the computation.*

Proof: To determine if a process p remains shallow, we need to show that p's status remains unaffected by the execution of *exit* and *fixdepth*. We shall consider the execution of the above actions by p itself, p's ancestors, p's descendants, and processes unrelated to p.

exit: When p executes *exit*, it sets *depth.p* to 0 and p becomes a sink in the priority graph. Since *depth.p* is thus less than D and p does not have any descendants after executing *exit*, p remains stably shallow.

When p's ancestor, p's descendant, or a process unrelated to p executes *exit* the process (i) severs its current relationship to p and (ii) may possibly become p's descendant. We denote these two events e' and e'' respectively and consider the effect of each event on the status of p separately. When p's ancestor executes e' it can affect the status of p by changing the value of $l.p$. Since p is shallow, $l.p$ is bounded from above by $D - depth.q$ where q is one of p's direct descendants. However, when p's ancestor executes e' the value of $l.p$ can only decrease. Thus, p remains stably shallow. When p's descendant executes e', some of the processes cease to be p's descendants. However, p remains shallow since the status of the remaining descendants does not change. When a process unrelated to p executes e', p's status is not affected.

When a process q executes e'', as argued above for the case of p, q itself becomes stably shallow. However, q may affect the status of its new direct ancestors and transitively p. Let r be q's new direct ancestor. Since r is stably shallow $l.r$ is no greater than D. Thus, $depth.q + l.r \leq D$ and r, as well as p, remains stably shallow.

fixdepth: When a process executes *fixdepth*, its own status as well as the status of its direct ancestor can be affected. Let us consider the effect of *fixdepth* execution on the executing process itself. The status of a shallow process r may change if the execution of *fixdepth* sets $depth.r$ greater than D. Process r can execute *fixdepth* only if it has a direct descendant q such that $depth.q + 1 > depth.r$. If r is shallow, then $depth.q + l.r \leq D$. Since a live chain of ancestors includes the process itself, the value of $l.r$ is at least 1. Thus, after the execution of *fixdepth*, $depth.p$ is no greater than D and r remains shallow.

Let us now consider the effect of r's execution of *fixdepth* on r's shallow direct ancestor s. Process s is shallow if $depth.r + l.s$ is no more than D. After the execution of *fixdepth* by r, $depth.q = depth.r - 1$. Since l is the length of the maximum chain of live ancestors then $l.r \geq l.s + 1$. If r is shallow: $depth.q + l.r \leq D$. We obtain: $depth.r + l.s \leq D$. Hence, s remains shallow. Thus, the execution of *fixdepth* by a stably shallow process does not change the status of the shallow processes. □

Deep graph is a subgraph of the priority graph induced by the deep and unstably shallow processes.

Lemma 3 *The program stabilizes to the following predicate:*

$$\textit{All processes in the system are stably shallow.} \quad (ST)$$

Proof: Let us consider the deep graph. Due to Lemma 1 this graph is acyclic. By definition, each sink of the graph is deep. Notice that a deep process p has either *exit* or *fixdepth* actions enabled. The execution of either action makes p shallow. If p was a sink in the deep graph before the execution of this action it becomes stably shallow. Due to Lemma 2 the set of stably shallow processes is closed. Hence p will not join the deep graph during the rest of the computation. The Lemma follows from repeated application of this reasoning to the sinks of the deep graph. □

The corollary below follows from Lemma 3.

Corollary 1 *When Predicate ST holds, for every live process p the value of $depth.p$ is less than D.*

Live neighbors not eating.

Lemma 4 *The program stabilizes to the following predicate:*

$$\textit{Two neighbors are eating in the same state only if they are both dead.} \quad (E)$$

Proof: Predicate E is closed because a process cannot execute *enter* when its neighbor is eating. To demonstrate convergence let us consider a computation with the initial state where two neighbor processes are eating. A live neighbor has *exit* enabled in such a state. Such neighbor either eventually crashes or executes *exit* and becomes thinking. □

We define the invariant I to be: $NC \wedge ST \wedge E$. The theorem below follows from Lemmas 1, 3, and 4.

Theorem 1 *The program stabilizes to I.*

3.2 Liveness and Safety

When the state of the system conforms to the invariant I proving safety is straightforward – Theorem 3. To prove liveness we first identify the live processes that are not affected by the dead ones. We call them green. The blocked processes are red. We prove that a red process cannot change its color – Lemma 5; a green process with the highest priority either changes color or eats – Lemma 6; and on the basis of this we prove that every green process eventually eats – Lemma 7. The liveness property follows – Theorem 2.

Process p is *red* if Predicate $RD.p$ is **true**. The rest of the processes are *green*. The *green graph* is the subgraph of the live graph induced by the set of green processes.

$$\begin{array}{l}
(p \text{ is dead}) \vee \\
((state.p = T) \wedge \\
\quad (\exists q : priroity.p.q = q : RD.q \wedge state.q \neq T)) \vee \\
((state.p = H) \wedge \\
\quad (\forall q : priority.p.q = q : RD.q \wedge state.q = T) \wedge \\
\quad (\exists q : priority.p.q = p : RD.q \wedge state.q = E))
\end{array} \quad (RD.p)$$

Note that RD is non-decreasing with respect to the set containment order. That is, for any process p, $RD.p$ does not negatively depend on the value of $RD.q$ of any other process q. Note also that RD is well-founded since the dead processes are red. Thus, the set of red processes can be determined by iteratively applying RD until fixpoint is reached. Therefore, a system state unambiguously defines the color of each process.

Lemma 5 *If I holds in the initial state of the computation, the color of a red process does not change throughout the computation.*

Proof: The color of a process p is determined by the value of $state.p$ and p's priority relation to other red processes. These variables are manipulated by the first four actions of a process. When p is dead all actions are disabled. If p is live and red, *join*, *leave*, and *enter* are disabled. Action *exit* can be enabled only if $depth.p > D$. However, when I holds, for each live process p (including red processes) $depth.p$ is less than D. Thus, *exit* is also disabled in p. That is, when I holds, $state.p$ of each red process p and the priority relations between red processes remain fixed and, therefore, the color of any red process does not change. □

Lemma 6 *If the initial state of a computation conforms to I, and p is a not thinking green node such that all its direct ancestors are continuously thinking then eventually p either executes* exit *or changes color.*

Proof: Process p may have green and red direct ancestors. Let us discuss p's red direct ancestors first. Note that due to Lemma 5 none of p's red ancestors become green. Since p is live, all its red direct ancestors are thinking. Note also that green processes cannot become p's ancestors unless p executes *exit*. If p is hungry and every direct ancestor is thinking, *leave* is not enabled. Thus, *enter* (besides *fixdepth*) can be the only enabled action in p. However, if p's direct descendant q is eating, then *enter* is not enabled in p. Since p is live, so is q. Since q is eating and live, it has *exit* enabled. When q executes *exit* it becomes thinking. Furthermore, when p is hungry its direct descendants can not execute *enter* and become eating. Thus, after every eating direct descendant of p executes *exit*, p executes *enter* and becomes eating. When p is eating, *exit* is enabled in p. Thus, unless p changes color it eventually executes *exit*. □

Lemma 7 *If a computation starts from a state conforming to I and a green process p is such that either (a) p is thinking and wants to eat ($needs().p$ continuously evaluates to* **true***) or (b) p is not thinking, then p eventually either executes* exit *or changes color.*

Proof: Due to Lemma 1, when I holds the green graph is acyclic. Note also, that unless p executes *exit* the number of its green ancestors can only decrease. If p is green, it's red ancestors are thinking. p's green ancestors may or may not be continuously thinking. Let us consider these two cases.

In the first case p's join is enabled. When p executes *join* it becomes hungry. If green ancestors continue to think, according to Lemma 6, p eventually executes *exit* or changes color. In the second case there always exists a process q which is an ancestor of p who either does not have green ancestors or whose green ancestors are continually thinking. In either case, by Lemma 6, q either executes *exit* or changes color. Thus, eventually p loses all its non-thinking green ancestors. In this case p eventually executes *exit* or changes color. □

Theorem 2 (Liveness) *If a computation starts from a state conforming to I and a process p is at least two processes away from a crashed process in every state of the computation and p wants to eat, then p eventually eats.*

Proof: Note that process p is green in every state of the computation. By Lemma 7, p eventually executes *exit*. By Corollary 1, when I holds p can execute *exit* only when $state.p$ is E. Hence p eventually eats. □

Theorem 3 (Safety) *If a computation starts from a state conforming to I, during this computation the number of pairs of simultaneously eating neighbors does not increase.*

Proof: When I holds, two neighbors are eating in the same state only if they are both dead. The neighbors cannot first start eating and then fail in this computation. The neighbors cannot fail first and then start eating either. The theorem follows. □

4 Extension to message-passing and concluding remarks

One way to transform our program to message-passing is to use Chandy and Misra's fork collection [5]. This idea is used by Tsay and Bagrodia [18] and by Sivilotti et al [17]. However, we find that using fork collection mechanism for our program cumbersome. The approach used by Nesterenko and Arora [15] appears more applicable. In that paper we implement a diners solution DP in the low-atomicity shared memory model and then extend it to message-passing. To cope with low atomicity we use a stabilizing handshake mechanism based on Dijkstra's K-state token circulation protocol [9] to provide synchronization between neighbors To transform the program presented in this paper we can use the program's priority assignment scheme to ensure fairness and reuse the synchronization idea of DP.

In this paper we show that combining transient and crash failure tolerance provides an efficient and inexpensive solution to diners problem for a rich class of faults – malicious crashes. This class approaches Byzantine in its generality. This makes our program and the idea of combining stabilization and crash fault tolerance of practical importance and theoretical significance. Our investigation also demonstrates the weakness of the traditional fault classification. The existing fault models cannot effectively capture malicious crashes: benign crashes are not rich enough and Byzantine failures are too expensive to tolerate. In this respect we are interested in further exploring alternative fault classes.

In our view transient faults and Byzantine faults represent two orthogonal extremal modes. A transient fault is extremal in space – all system state is assumed to be corrupt. A Byzantine fault is extremal in time – the process is assumed to be corrupt infinitely long. A malicious crash limits the time of process corruption but not the scope of corruption propagation. Hence, the solution that tolerates malicious crashes only guarantees eventual correctness (as defined by the original fault-intolerant specification) outside of the failure locality. We may conceive of a stricter tolerance requirement: masking a malicious crash limits both the time and scope of the failure. A program that masks malicious crashes always operates correctly outside of failure locality during the crash. Masking tolerance makes such a program more attractive. Our latest research has yielded a solution to dining philosophers with masking tolerance of malicious crashes and we are planning to continue studying the subject.

References

[1] G. Antonoiu and P.K. Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph that stabilizes using read/write atomicity. In *Proceedings of EuroPar'99*, volume 1685 of *Lecture Notes in Computer Science*, pages 823–830. Springer-Verlag, 1999.

[2] J. Beauquier, A.K. Datta, M. Gradinariu, and F. Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *DISC00 Distributed Computing 14th International Symposium, Springer-Verlag LNCS:1914*, pages 223–237, 2000.

[3] J. Beauquier and S. Kekkonen-Moneta. Fault-tolerance and self-stabilization: impossibility results and solutions using self-stabilizing failure detectors. *International J. of Systems Science*, 28(11):1177–1187, 1997.

[4] J. Beauquier and S. Kekkonen-Moneta. On ftss-solvable distributed problems. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 64–79. Carleton University Press, 1997.

[5] K.M. Chandy and J. Misra. The drinking philosopher's problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[6] M. Choy and A.K. Singh. Efficient fault-tolerant algorithms for distributed resource allocation. *ACM Transactions on Programming Languages and Systems*, 17(3):535–559, May 1995.

[7] M. Choy and A.K. Singh. Localizing failures in distributed synchronization:. *IEEE Transactions on Parallel and Distributed Systems*, 7(7):705–716, July 1996.

[8] E.W. Dijkstra. *Hierarchical Ordering of Sequential Processes*, pages 72–93. Academic Press, 1972.

[9] E.W. Dijkstra. Self stabilizing systems in spite of distributed control. *Comm. ACM*, 17:643–644, 1974.

[10] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty processor. *J. of ACM*, 32(2):374–382, April 1985.

[11] M.G. Gouda and F. Haddix. The alternator. to appear in *J. of Parallel and Distoributed Computing.*

[12] S.T. Huang. The fuzzy philosophers. In J. Rolim et al., editor, *Proceedings of the 15th IPDPS 2000 Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 130–136, Cancun, Mexico, May 2000. Springer-Verlag.

[13] L. Lamport and N. Lynch. Distributed computing: Models and methods. In J. van Leewen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 19, pages 1157–1199. The MIT Press, NY, NY, 1990.

[14] 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.

[15] M. Nesterenko and A. Arora. Stabilization-preserving atomicity refinement. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 254–268, 1999.

[16] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. of ACM*, 27(2):228–234, April 1980.

[17] P.A.G. Sivilotti, S.M. Pike, and N. Sridhar. A new distributed resource-allocation algorithm with optimal failure locality. In *Proceedings of the 12th IASTED Internation Conference on Parallel and Distributed Computing and Systems*, volume 2, pages 524–529. IASTED/ACTA Press, November 2000.

[18] Y.K. Tsay and R.L. Bagrodia. An algorithm with optimal failure locality for the dining philosophers problem. In G.Tel and P.M.B. Vitányi, editors, *Distributed Algorithms, 8th International Workshop, WDAG '94*, volume 857 of *Lecture Notes in Computer Science*, pages 296–310, Terschelling, The Netherlands, 29 September–1 October 1994. Springer.