Localizing Failures in Distributed Synchronization

Manhoi Choy and *Ambuj K. Singh*
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, California 93106

Abstract

We investigate the fault-tolerance of distributed algorithms in asynchronous message passing systems with undetectable process failures. We introduce *failure locality* as a measure of the fault-tolerance of distributed algorithms. This measure captures the general idea of allowing part of a system to continue to function despite the failure of other parts of the system. We concentrate on two problems, the *dining philosophers problem* and the *committee coordination problem*, and present algorithms with an optimal failure locality.

Index Terms: Concurrency, Distributed Algorithms, Fault-tolerance, Lower Bounds, Synchronization

1 Introduction

The ability to tolerate failures is an important design requirement of computer systems in general, and of distributed systems in particular as the detection of failures becomes difficult in such systems. We investigate the effects of failures on distributed synchronization problems that require the co-operation of a set of independent processes. The specific problems we study assume an underlying undirected graph that defines the neighborhood of the processes. Inter-process communication is through message-passing and is only possible between neighboring processes. The goal is to design solutions for which the effect of the failure of a process can be confined to its immediate neighborhood. In other words, we are interested in solutions where processes are shielded from the effects of non-local failures. We assume the *fail-stop* model in which failures are undetectable and failed processes are indistinguishable from processes that are very slow.

We choose the problems of dining philosophers and committee coordination as representatives of the class of distributed synchronization problems. The dining philosophers problem is a generalization of the mutual exclusion problem and can be used to solve many other synchronization problems. The committee coordination problem occurs in languages such as CSP [4] and Ada [1] that are based on synchronous message communication. We use *failure locality* [6] to measure the degree of fault-tolerance. The failure locality of an algorithm denotes the size of the neighborhood that gets affected by a failure. Thus, if an algorithm has a failure locality of m then any process for which there are no failures within a distance of m in the underlying graph executes as if no failures occurred. In other words, any such process continues to meet the specified safety and progress properties.

The main contribution of this paper is the presentation of tight lower bounds on the failure locality of solutions to the dining philosophers and the committee coordination problem. This is achieved through the presentation of lower bounds and the design of solutions that achieve them. As a part of the optimal algorithms, we also introduce the idea of a *bounded doorway*. This doorway has the interesting property that when coupled with an algorithm ensuring the absence of global starvation, the doorway ensures the absence of local starvation *without* adding to the failure locality of the algorithm. This is in contrast to other kind of doorways [6, 13] that add to the failure locality of the underlying algorithm.

The rest of the paper is organized as follows. In Section 2 we present some of the relevant background. In Section 3, we introduce the idea of bounded doorways. In Section 4, we solve the dining philosophers problem with an optimal failure locality. In Section 5, we solve the committee coordination problem again with an optimal failure locality. A brief discussion appears in Section 6 and an appendix contains the proofs of some theorems used in the main body of the paper.

2 Relevant Background

Locality of distributed computation has been studied by many authors [2, 14, 19, 20]. Linial [14] considered the problem of computing functions that do not require global communication. To compute a function, each process starts with some local data. During the computation, information is exchanged between neighboring processes. Finally, each of the processes terminates and returns a value. A function is said to be computed *locally* [2] if it can be computed in a time less than the diameter of the network. Some of these functions include finding a maximal independent set [2, 11, 14, 15] or computing nodes and edge colorings of a network [2, 10]. In this paper, we investigate the locality of distributed synchronization problems that do not require processes to terminate and return results. In particular, we concentrate on the *dining philosophers problem* [8, 16] and the *committee coordination problem* [5].

The dining philosophers problem consists of a set of processes and a set of resources that are modeled by a conflict graph. Each process is mapped to a node of the conflict graph and each resource is mapped to an edge. Many solutions to the dining philosophers problem appear in the literature [3, 5, 6, 16, 20]. Styer and Peterson [20] are perhaps the first to consider the issue of locality for this problem. They measure the locality of an algorithm by the length of the longest waiting chain of processes that may be formed in the system. We formalize this idea of waiting chains to the notion of *failure locality*. For the dining philosophers problem, failure locality of an algorithm is defined as the smallest number m such that any process, for which there are no failures within a distance of m in the conflict graph, is free from starvation. The algorithms obtained by Styer and Peterson achieve a failure locality of 3. Recently, we considered the problem of achieving a good failure locality in conjunction with the problem of achieving a good response time and a good message complexity [6]. We presented solutions that achieved a failure locality of 4 while maintaining a quadratic (in the degree of the conflict graph) response time and a quadratic message complexity. The paper also contains an algorithm that achieves a failure locality of 3. However, the question of the best achievable failure locality remained open.

The committee coordination problem consists of a set of interactions (or committees) and a set of processes. The problem initially arose out of rendezvous-based communication in languages like CSP and Ada [1, 4] where a subset of processes needs to exchange information in a synchronous manner. A conflict graph can be constructed for this problem by mapping each interaction to a node and placing an edge between interactions that share a common process. We concentrate on the binary version of the problem in which every interaction has two members. Both the binary and the general versions of the problem have been solved by many researchers [4, 5, 6, 12, 18, 19]. Sistla [19] calls a solution to the problem to be *real time* if its complexity is independent of the size of the communication network. He also investigates the formation of waiting chains and then uses this to derive a lower bound on the response time. We define the failure locality of solutions to the committee coordination problem to be the smallest number m such that any continuously ready interaction, for which there are no failures within a distance of m in the conflict graph, will be eventually committed. Recently, we presented solutions to the committee coordination problem with a failure locality of 2 [6]. But, once more the question of the optimal failure locality remained open.

3 Bounded Doorways

Solutions to problems requiring the absence of local starvation are often composed of two parts [6, 13, 20]. The first part satisfies all the requirements of the problem except that it ensures the absence of global starvation (i.e., no deadlocks). The second part of the solution, usually called a *doorway*, translates the absence of global starvation of the underlying solution into the absence of local starvation (i.e., progress with respect to each process) for the composite solution. In other words, a doorway D takes a program F ensuring the absence of global starvation, and produces a composite program consisting of F and D such that local starvation does not occur.

Various kinds of doorways have been defined and used in the literature. A basic kind called the *first-come-first-served* doorway was defined and used by Lamport in [13] in the context of the mutual exclusion problem. This doorway consists of a finite sequence of non-waiting statements and enforces a first-come-first-served behavior. Though useful for shared variable systems, this doorway cannot be used for message based systems as the doorway would have to consist of waiting statements in order to enforce a first-come-first-served behavior. A variant of this doorway was used by Styer and Peterson to solve the dining philosophers problem in a message based model similar to ours [20]. In their scheme neighboring processes executes a P operation on a common semaphore (implemented by passing messages) while entering the doorway and a V operation while exiting the doorway. Two other kinds of doorways called *single doorway* and *double doorway* were introduced in [6] in order to avoid unbounded overtaking. All the above doorways require processes outside the doorway to wait for processes inside the doorway and this increases the failure locality of the composite solution by at least one.

In this paper, we introduce a new kind of a doorway that does not increase the failure locality of the underlying computation. This doorway, referred to as a *bounded doorway*, is implemented by dynamically assigning a set of distinct IDs to the processes. The IDs are distinct and used for resolving conflicts that occur during the execution of the algorithm. We assume that the underlying solution resolves conflicts in favor of the process with a smaller ID. In the doorway algorithm, a process that has just made some progress selects a neighbor with a higher ID to perform an ID exchange. This exchange results in the neighbor having a smaller ID (i.e. higher priority) than before. An ID exchange involves sending an *exchg* message to the selected neighbor, waiting for an acknowledgement to be returned, and a broadcast of the new ID to other processes. The process receiving an *exchg* message agrees to the exchange of IDs only if this will lead to a smaller ID and the process is not currently participating in another exchange. Each process maintains a queue of neighbors and chooses the candidate process for ID exchange by scanning the queue for a process with a higher ID. (No exchange is needed if all neighbors have a lower ID.) In order to ensure fairness, the process so chosen is placed at the end of the queue after the exchange of IDs. Thus, a process that makes progress infinitely often while its neighbors are starving will eventually have a higher ID (and consequently a lower priority) than all its neighbors. This will allow the neighbors to make progress and avoid starvation.

4 The Dining Philosophers Problem

4.1 Problem Description

The problem consists of a number of processes each of which may be in one of three states: *thinking*, *hungry*, or *eating*. Predicate $thinking_i$ denotes that process i is thinking. Predicates $hungry_i$ and $eating_i$ are defined similarly. Initially every process is thinking. A thinking process becomes hungry when it needs to access the resources on the adjoining edges. A process stays hungry until it is allowed to eat by the dining philosophers algorithm. A process accesses the resources during its eating period, which is assumed to be finite. After that an eating process transits to the thinking state. A dining philosophers algorithm should satisfy two requirements: *mutual exclusion* which requires that neighboring processes do not eat at the same time and *no starvation* which requires that every hungry process eventually gets a chance to eat.

A process may fail at any time during a computation. The failure of a process i is modeled by an auxiliary predicate $failed_i$. This predicate is initially false and is set to true by the failure of process i. Once it is true, it remains true forever. We assume the fail-stop mode of failures in which the failure of a process is undetectable. In particular, this implies that the predicate $failed_i$ cannot be used in the code of an algorithm. (The predicate can however be used in the proofs of correctness.) As stated earlier, we measure the fault-tolerance of an algorithm by its failure locality. Formally, the failure locality of an algorithm is defined as the smallest number m such that any process, for which there are no failures within a distance of m in the conflict graph, is free from starvation. We require that failures have no effect on the safety properties (such as mutual exclusion) of an algorithm.

For example, consider the conflict graph of 7 processes shown in Figure 1. If the failure locality of an algorithm is at most 1 then failures do not have any effect on the progress of processes outside the immediate neighborhood of the failed process. In other words, the failure of process 1 can only starve processes 2,3, and 4, the failure of process 4 can only starve processes 1,2,3,5, and 6, and so on. If an algorithm has a failure locality of 2 then the failure of process 1 cannot starve process 7 and vice-versa.

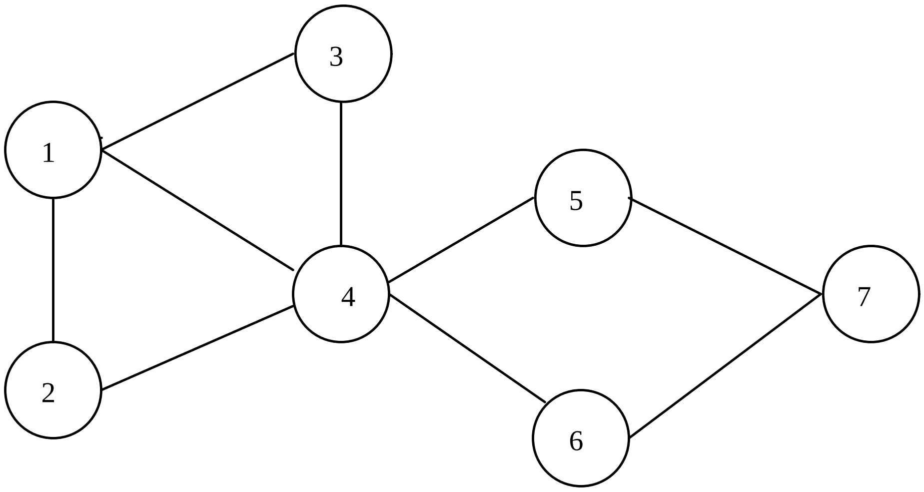

Figure 1: Example of a Conflict Graph

4.2 Impossibility Result

It is easy to see that the a failure locality of 0 (i.e., the failure of a process does not starve any other processes) is unachievable. This is because a process may fail while eating and that means that the immediate neighbors, unaware of this failure, may starve. The following theorem states that a failure locality of 1 is also unachievable.

Theorem 1 *There does not exist a solution to the dining philosophers problem with a failure locality of 1.*

Proof: Suppose there exists a solution with a failure locality of one. Consider the problem instance of 3 processes i, j, and k where i and k are neighbors of j but i and k are not neighbors. Consider the following execution history in which none of the processes fail.

Initially all the processes are thinking. Process i becomes hungry first. Since thinking processes may remain thinking forever, there exists a future state at which i starts eating while j and k are still thinking. Now, assume that while i is eating both j and k become hungry. It is possible that process i fails undetectably while eating and therefore in order to ensure a failure locality of one, process k should be allowed to eat. Thus, there exists a future state at which i and k are eating and j is hungry. Now, suppose that i finishes eating and becomes hungry again. By the same argument as before, there exists a future state at which k remains eating, j remains hungry, and i starts eating. Now, let process k stop eating and become hungry again. This leads us back to a state that had existed earlier in the history: process i is eating and processes j and k are hungry. Note that process j remains hungry in all the intermediate states. This scenario may repeat continuously, giving us the desired contradiction as process j remains hungry forever though none of its neighbors have failed. $\square$

Initially:	/* The main procedure */
$ID_i = i \ \wedge \ thinking_i \ \wedge$	**repeat**
$\langle \forall j : j \in N_i : ID_{ij} = ID_j \ \wedge \ fork_{ij} = (ID_i < ID_j) \rangle \ \wedge$	$\langle$*non-critical section*$\rangle$;
$ack_i \ \wedge \ (Q_i =$ processes in N_i in any order$) \ \wedge$	**wait** until ack_i;
$\langle \forall j : j \in N_i : \neg requested_{ij} \rangle \ \wedge$	$state_i := hungry$;
$pending_i = \{\}$	*request-forks*;
	wait until *all-forks*$_i$;
Define:	$state_i := eating$;
all-forks$_i \ \equiv \ \langle \forall j : j \in N_i : fork_{ij} \rangle$	$\langle$*critical section*$\rangle$;
all-low-forks$_i \ \equiv \ \langle \forall j : j \in N_i \ \wedge \ ID_{ij} < ID_i : fork_{ij} \rangle$	$state_i := thinking$;
$release_i \ \equiv \ (thinking_i \ \vee \ \neg$*all-forks*$_i)$	*release-forks;*
release-high$_i \ \equiv \ (thinking_i \ \vee \ \neg$*all-low-forks*$_i)$	*release-id*;
	forever

Figure 2: Algorithm for Process i

4.3 An Optimal Algorithm

We now construct an optimal algorithm with a failure locality of 2. The algorithm consists of two parts: a bounded doorway as discussed in Section 3, and a scheme to ensure mutual exclusion and absence of global starvation. Mutual exclusion is ensured by associating forks with edges and maintaining the invariant that an eating process has all the neighboring forks. In order to ensure the absence of global starvation, conflicting requests to the same fork are resolved in favor of the process with the smaller ID. Since this may lead to the formation of long waiting chains (which in turn results in poor failure locality), processes break such long chains by selectively releasing forks. However, the process with the smallest ID does not release any forks. This ensures the absence of global starvation. The ID exchange mechanism due to the bounded doorway ensures that a continuously hungry process will eventually have a smaller ID than its competing neighbors and will be able to eat. The composite algorithm thus also ensures freedom from local starvation.

The algorithm is shown in figures 2 and 3. For any process i, the set N_i defines its neighbors. Given a process $j \in N_i$ with $ID_i < ID_j$, we refer to process j as a *high* neighbor of process i and refer to process i as a *low* neighbor of process j. Similarly, the fork between the processes is referred to as a *high* fork at process i and as a *low* fork at process j. Process i uses a number of local variables. Variable ID_i refers to the ID of process i and variable ID_{ij} refers to the ID of process j as known at process i. Boolean $fork_{ij}$ is true iff the fork between processes i and j is at process i. Initially the fork is at the process with the smaller ID. The presence of a process inside the bounded doorway is indicated by a variable ack_i. This variable is false iff process i is executing the code of the bounded doorway, i.e., exchanging its ID with a neighbor. A queue Q_i is used in order to select the next process

On *receiving message* $\langle request \rangle$
from j such that $ID_{ij} > ID_i$:
  $pending_i := pending_i \cup \{j\}$;
  if $(fork_{ij} \wedge release\text{-}high_i)$ **then**
    send-fork(j) **fi**

On *receiving message* $\langle request \rangle$
from j such that $ID_{ij} < ID_i$:
  $pending_i := pending_i \cup \{j\}$;
  if $(fork_{ij} \wedge release_i)$ **then**
    send-fork(j);
    release-high-forks **fi**

On *receiving message* $\langle$*fork, flag*$\rangle$ *from j:*
  $fork_{ij} := true$;
  $requested_{ij} := false$;
  if *flag* **then** $pending_i := pending_i \cup \{j\}$**fi**;
  if *all-low-forks*$_i$ **then** *request-high-forks*
  elsif $flag \ \wedge \ ID_{ij} > ID_i$ **then** *send-fork(j)* **fi**

On *receiving message* $\langle$*exchg, ID*$\rangle$ *from j:*
  if $(ack_i \wedge ID_i > ID)$ **then**
    send $\langle exchg\text{-}yes,\ ID_i \rangle$ to j;
    for each $(k \in N_i)$ **do**
      send $\langle newid$, ID$\rangle$ to k **od**
    $ID_{ij} := ID_i$;
    $ID_i := ID$;
    if (*all-low-forks*$_i \wedge$ *hungry*$_i$) **then**
      request-high-forks **fi**
  else send $\langle exchg\text{-}no \rangle$ to j **fi**

On *receiving message* $\langle$*exchg-yes, ID*$\rangle$ *from j:*
  for each $(k \in N_i)$ **do**
    send $\langle newid$, ID$\rangle$ to k **od**
  $ID_i := ID$;
  $ack_i := true$

On *receiving message* $\langle$*exchg-no*$\rangle$ *from j:*
  $ack_i := true$

On *receiving message* $\langle$*newid,ID*$\rangle$ *from j:*
  $ID_{ij} := ID$

Procedure *request-forks*
  for each $j \in N_i$ **do**
    if $\neg fork_{ij} \wedge \neg requested_{ij}$ **then**
      send $\langle request \rangle$ to j;
      $requested_{ij} := true$ **fi od**
end *request-forks*

Procedure *request-high-forks*
  for each $(j \in N_i \ \wedge \ ID_{ij} > ID_i)$ **do**
    if $\neg fork_{ij} \ \wedge \ \neg requested_{ij}$ **then**
      send $\langle request \rangle$ to j;
      $requested_{ij} := true$ **fi od**
end *request-high-forks*

Procedure *release-forks*
  for each $(j \in pending_i \ \wedge \ fork_{ij})$ **do**
    send-fork(j) **od**
end *release-forks*

Procedure *release-high-forks*
  for each $(j \in pending_i \ \wedge \ fork_{ij} \ \wedge \ ID_{ji} > ID_i)$ **do**
    send-fork(j) **od**
end *release-high-forks*

Procedure *release-id*
  $j := next(Q_i)$;
  if $j \neq null$ **then**
    $ack_i := false$;
    send $\langle exchg,\ ID_i \rangle$ to j **fi**
end *release-id*

Procedure *send-fork(j)*
  $fork_{ij} := false$;
  $pending_i := pending_i - \{j\}$;
  if $(hungry_i \ \wedge \ \neg requested_{ij})$ **then**
    send $\langle fork, true \rangle$ to j;
    $requested_{ij} := true$
  else send $\langle fork, false \rangle$ to j **fi**
end *send-fork*

Figure 3: Algorithm for process i (continued)

for exchanging IDs. Initially ack_i is true and Q_i is set to any arbitrary ordering of the processes in N_i. Boolean $requested_{ij}$ is true iff process i has sent an outstanding fork request to process j. Set $pending_i$ contains the fork requests that have not yet been serviced.

When a process becomes hungry, it attempts to collect missing forks by sending request messages, as in procedure *request-forks*, to the neighbors. A process i receiving a request for a fork decides whether it is going to release the fork to requester j depending on its state and the relative order of the IDs. If $ID_{ij} > ID_i$ then the fork is released if process i is thinking or if it has not collected all the low forks (this condition is captured by the predicate *release-high*$_i$). If $ID_{ij} < ID_i$ then the fork is released (along with all captured high forks in order to avoid the formation of long waiting chains) if process i is thinking or if it has not collected all the forks (this condition is captured by the predicate *release*$_i$). Note from procedure *send-fork* that if the process releasing a fork is hungry then it informs the requesting process of this by setting a flag in the fork to true. If a fork is not released to a requesting process then the ID of the requester is saved in set $pending_i$ for future consideration.

If the reception of the fork completes the collection of all the low forks at a process then request for missing high forks are made by calling procedure *request-high-forks*. Otherwise, if the fork just received is a high fork with a true flag then the fork is returned to the sender.

Once a process gathers all its forks, it transits to the eating state and enters the critical section. Upon exiting from the critical section, it sets its state to thinking, releases all the forks in *pending*, and executes the bounded doorway code, i.e., procedure *release-id*, in order to exchange IDs. In this procedure, process i first invokes a function *next* to choose the next process with a higher ID from the queue Q_i. If such a process exists then its ID is returned and it is moved to the end of the queue Q_i, otherwise a special value *null* is returned. The implementation of the function is straightforward and not shown here. If function *next* returns a non-null value j then ack_i is set to false and an *exchg* message is sent to process j. Upon receiving this message, process j participates in an ID exchange iff it is outside the doorway (indicated by *ack* being true) and the ID exchange will lower its ID. In that case process j sends an *exchg-yes* message to i and also informs all neighbors of the new ID. Otherwise, process j responds with an *exchg-no* message to process i. Upon receiving a positive acknowledgement from process j, process i completes the ID exchange process and informs all of its neighbors of the new ID. If a negative acknowledgement is received then no exchange is carried out. In either case, ack_i is set to true and the next iteration of state transitions begins.

4.4 Proof of Correctness

We use the temporal operators $\Box$ and $\Diamond$ [17] in the following proof. Briefly, $\Box p$ means that the formula p holds at all states in the history beginning with the current state, and $\Diamond p$ means that the formula p holds at some future state in the history beginning with the current state. The following two lemmas are proved in the appendix.

Lemma 1 *If a non-failed process i starves then eventually for every neighbor j, the ordering of ID_i and ID_j becomes fixed. Formally,*

$$\Box(\Box(\neg failed_i \ \wedge \ hungry_i) \ \Rightarrow \ \langle \forall j : j \in N_i : \ \ \Diamond\Box(ID_i < ID_j) \ \ \vee \ \ \Diamond\Box(ID_i > ID_j)\rangle).$$

Lemma 2 *If a non-failed process i starves then eventually any low neighbor j either fails, or remains non-eating forever with $ID_{ji} = ID_i$. Formally,*

$$\Box(\Box(\neg failed_i \ \wedge \ hungry_i) \ \Rightarrow$$
$$\langle \forall j : j \in N_i : \ \Diamond\Box(ID_i < ID_j) \ \ \vee \ \ \Diamond\Box(ID_i > ID_j \ \wedge \ (failed_j \ \vee \ (\neg eating_j \ \wedge \ ID_{ji} = ID_i)))\rangle).$$

Definition 1 *We construct a waiting graph W_i for any starving non-failed process i as follows:*

- *Process i is in W_i.*
- *For any process j in W_i, apply Lemma 2. Add a neighbor k to W_i for which $\Diamond\Box(ID_j > ID_k \ \wedge \ \neg failed_k \ \wedge \ hungry_k \ \wedge \ ID_{kj} = ID_j)$ holds.*
- *Repeat until no more processes can be added to W_i.*

Informally, graph W_i includes process i and all starving non-failed processes that are reachable by a chain of decreasing IDs through other processes in W_i. The following lemma is proved in the appendix.

Lemma 3 *For any process j in W_i and any low neighbor k of j, eventually the fork between j and k either remains at j forever or remains at k forever. Formally,*

$$\Box\langle \forall j : j \in W_i : \langle \forall k : k \in N_j :$$
$$\Diamond\Box(ID_j < ID_k) \ \ \vee \ \ \Diamond\Box(ID_j > ID_k \ \wedge \ fork_{jk}) \ \ \vee \ \ \Diamond\Box(ID_j > ID_k \ \wedge \ \neg fork_{jk})\rangle\rangle.$$

Based on the above lemma, we can prove the following theorem about the failure locality of the algorithm.

Theorem 2 *If process i starves then there is a failed process at a distance at most 2 from it.*

Proof: For the sake of contradiction, assume that process i starves and all the processes within a distance of 2 from i do not fail. By Lemma 3, for any neighbor j of process i eventually either $\Box(ID_i < ID_j)$ or $\Box(ID_i > ID_j \ \wedge \ fork_{ij})$ or $\Box(ID_i > ID_j \wedge \ \neg fork_{ij})$ holds forever. Now, there are two cases to consider: either eventually process i is holding all its low forks forever or eventually process i is missing some low fork forever. In the former case, process i sends requests, if necessary, to its high neighbors to collect the high forks. Since these processes do not fail they will eventually release the forks, and thus process i will eventually collect all the forks and start eating. This contradicts our assumption that process i starves.

In the second case, let j be a low neighbor that holds on to a fork from process i forever. Since process j is holding a high fork, it must have collected all its low forks. Consequently, process j can start collecting any missing high forks. Since processes within a distance of 2 from i do not fail, processes within a distance of 1 from j do not fail. This implies that process j will be able to collect all the high forks and start eating. Eventually process j will finishes eating and release the fork to process i. This contradicts the assumption that the fork shared by processes i and j stays at process j forever. $\square$

5 The Committee Coordination Problem

5.1 Problem Description

The problem consists of a set of processes and a set of interactions. Each process has a predetermined set of interactions that it may participate in and each interaction has a predetermined set of processes that co-operate to execute it. Processes that may participate in the same interaction are physically close to each other and can communicate directly. A process may be in one of three states: *idle*, *ready*, and *commit*. Predicate $idle_i$ denotes that process i is idle. Predicates $ready_i$ and $commit_i$ are defined similarly. Initially all processes are *idle*. From time to time, a process becomes *ready* to take part in any of the interactions associated with it and it remains *ready* until one of the interactions is committed. Similar to a process, an interaction may also be in one of three states: *idle*, *ready*, or commit. An interaction is *ready* to be executed if all the processes associated with it are *ready*, an interaction is in the *commit* state if all the processes participating in it are committed to perform it, and an interaction is *idle* otherwise. During the execution of an interaction, the interaction and all the processes associated with the interaction are in the state *commit*. The commitment of each interaction eventually terminates, at which point the interaction and the processes become *idle* again.

We say that two interactions *conflict* with each other if they share a common process. Based on this conflict relation between interactions, we define a *conflict graph* as follows: we represent each interaction by a node and place an edge between two nodes if the corresponding interactions conflict. As mentioned earlier, the state transitions of a process occur in the order *idle* to *ready* to *commit* and back to *idle*. The state transitions of an interaction are similar except for the fact that an interaction may transit to state *idle* directly from the state *ready*. This is because two neighboring interactions may be ready at the same time and committing one changes the state of the other to *idle*. The time for which an interaction is committed is assumed to be finite. A solution to the committee coordination problem should satisfy two safety requirements, *synchronization* and *mutual exclusion*, and one progress requirement, *weak interaction fairness*. Synchronization requires that the commitment of an interaction is only started if it is ready. Mutual exclusion requires that no two interactions sharing a common process commit simultaneously. Weak interaction fairness requires that if an interaction is continuously ready then it will eventually be committed.

As in the dining philosophers problem, a process may fail at any time during a computation and this is modeled by an auxiliary predicate $failed_i$. We say that an interaction fails if any process belonging to it fails. The failure locality of an algorithm is defined as the smallest number m such that for any interaction, as long as interactions within a distance of m in the conflict graph do not fail, the requirement of weak interaction fairness is satisfied. Failures do not impact the requirements of synchronization and mutual exclusion as these are safety properties.

5.2 Impossibility Result

The following theorem shows that the effect of failure of a process cannot be limited just to the interactions in which it participates.

Theorem 3 *There does not exist a solution to the committee coordination problem that achieves a failure locality of 0.*

Proof: The proof is by reduction to the well-known result about the impossibility of distributed consensus with one faulty process [9]. Assume that there exists an algorithm that achieves a failure locality of 0. In other words, the algorithm supposedly ensures that as long as processes in an interaction are ready and do not fail, the interaction or one of its neighbors commit eventually. We use such an algorithm to solve the distributed consensus problem for three processes i, j, and k.

In the consensus problem every process has an initial value and all non-failed processes have to agree on a common value that was one of the initial values. For the case of three processes that we consider here, we form three interactions $\mathcal{I}= \{i,j\}$, $\mathcal{J}= \{j,k\}$, and $\mathcal{K}= \{k,i\}$ and devise a mechanism by which the commitment of an interaction (e.g. $\mathcal{I}$) implies that a consensus is reached on the initial value of the corresponding process (in this case i).

Every process first broadcasts its initial value to the other processes and then becomes ready to participate in any interaction of which it is a member. By the requirements of synchronization, mutual exclusion and weak interaction fairness, the underlying committee coordination algorithm commits exactly one of the interactions $\mathcal{I}$, $\mathcal{J}$, or $\mathcal{K}$. The processes that are participating in the committed interaction broadcast its identity to the third process. Finally, the processes agree on the initial value of process corresponding to the committed interaction.

Since processes broadcast their initial values before becoming ready, the value corresponding to a committed interaction is available at all the processes. Since at most one process fails, at least one interaction will be ready to be committed. By the assumed failure locality of the underlying committee coordination algorithm, at least one interaction commits. Therefore, at least one process is able to broadcast the identity of the committed interaction. As a result, all the non-failed processes agree on a common value that was one of the initial values. □

Initially:	/* The main procedure */
$ID_i = i \ \wedge\ idle_i$	**repeat**
$\langle \forall j : j \in N_i : ID_{ij} = ID_j \rangle \ \wedge$	$\langle idle\ region \rangle$;
$ack_i \ \wedge\ (Q_i =$ processes in N_i in any order$) \ \wedge$	**wait** until ack_i;
$\langle \forall j : j \in N_i : \neg requested_{ij} \rangle \ \wedge$	$state_i = ready$;
$pending_i = \{\} \ \wedge\ \neg bound_i$	*wakeup-neighbors*;
	request-commitment;
	wait until $commit_i$;
	$\langle perform\ interaction \rangle$;
	$state_i := idle$;
	$bound_i := false$;
	release-id
	forever

Figure 4: Algorithm for Process i

5.3 An Optimal Algorithm

We now construct an optimal algorithm with a failure locality of 1 for the binary version of the problem. As for the dining philosophers problem, we use a bounded doorway in which processes exchange IDs and another piece of code that ensures the two safety properties and global progress. For every interaction, the process with the smaller ID is designated as the manager and is responsible for committing the interaction. A process may possibly manage several interactions based upon the relative ordering of the IDs of the processes. For any process i, set N_i defines the set of processes that participate in some common interaction with process i. When process i becomes ready, it attempts to commit the interactions it manages by invoking procedure *request-commitment*. While executing this procedure, process i sends *request* messages to each process j for which $\{i, j\}$ is an interaction managed by process i. When process j receives a request, it replies with a yes message if it is ready and has not sent a yes message to another manager. Otherwise, it saves i in the set $pending_j$. A local variable $bound_j$ is set to true iff process j has sent a yes message to some manager. Upon receiving a yes message from process j, process i commits the interaction $\{i, j\}$ provided it is still ready and has not responded with a yes message to some other manager. Otherwise, manager i responds with a *release* message to process j. On receiving this release message, process j informs the neighboring managers to retransmit *request* messages by sending a *wakeup* message to each process in $pending_j$. On receiving a *wakeup* message from process j, these processes try to commit the corresponding interactions again.

The complete algorithm is shown in figures 4 and 5. Local variables $ID_i, ID_{ij}, requested_{ij}, Q_i$, and ack_i are as in the dining philosophers algorithm. The implementation of the bounded doorway, i.e., procedure *release-id*, is also as in the previous solution.

On *receiving message* $\langle request \rangle$ *from* j:
  if $(ready_i \;\wedge\; \neg bound_i)$ **then**
    send $\langle yes \rangle$ to j;
    $bound_i := true$;
  else $pending_i := pending_i \cup \{j\}$ **fi**

On *receiving message* $\langle yes \rangle$ *from* j
  if $(ready_i \;\wedge\; \neg bound_i)$ **then**
    send $\langle commit \rangle$ to j;
    $state_i := commit$
  else send $\langle release \rangle$ to j **fi**;
  $requested_{ij} := false$

On *receiving message* $\langle release \rangle$ *from* j
  $bound_i := false$;
  wakeup-neighbors;
  request-commitment

On *receiving message* $\langle wakeup \rangle$ *from* j
  $requested_{ij} := false$;
  if $ready_i \;\wedge\; \neg bound_i \;\wedge\; ID_{ij} > ID_i$ **then**
    send $\langle request \rangle$ to j;
    $requested_{ij} := true$ **fi**

On *receiving message* $\langle exchg, ID \rangle$ *from* j:
  if $(ack_i \wedge ID_i > ID)$ **then**
    send $\langle exchg\text{-}yes, ID_i \rangle$ to j;
    for each $(k \in N_i)$ **do**
      send $\langle newid, ID \rangle$ to $k\rangle$ **od**
    $ID_{ij} := ID_i$;
    $ID_i := ID$;
    request-commitment;
  else send $\langle exchg\text{-}no \rangle$ to j **fi**

On *receiving message* $\langle commit \rangle$ *from* j:
  $state_i := commit$

On *receiving message* $\langle exchg\text{-}yes, ID \rangle$ *from* j:
  for each $(k \in N_i)$ **do**
    send $\langle$newid, ID$\rangle$ to $k\rangle$ **od**
  $ID_i := ID$;
  $ack_i := true$;

On *receiving message* $\langle exchg\text{-}no, ID \rangle$ *from* j:
  $ack_i := true$;

On *receiving message* $\langle newid, ID \rangle$ *from* j:
  $ID_{ij} := ID$

Procedure *request-commitment*
  for each $(j \in N_i \;\wedge\; ID_{ij} > ID_i)$ **do**
    if $\neg bound_i \;\wedge\; \neg requested_{ij}$ **then**
      send $\langle request \rangle$ to j;
      $requested_{ij} := true$ **fi od**
end *request-commitment*

Procedure *wakeup-neighbors*
  for each $(j \in pending_i)$ **do**
    send $\langle wakeup \rangle$ to j **od**
  $pending_i := \{\}$
end *wakeup-neighbors*

Procedure *release-id*
  $j := next(Q_i)$;
  if $j \neq null$ **then**
    $ack_i := false$;
    send $\langle exchg, ID_i \rangle$ to j **fi**
end *release-id*

Figure 5: Algorithm for process i (continued)

5.4 Proof of Correctness

The following two lemmas can be proved in a manner similar to Lemma 1 and Lemma 2.

Lemma 4 *If a non-failed process i remains ready forever then eventually for every neighbor j, the ordering of ID_i and ID_j becomes fixed. Formally,*

$$\Box(\Box(\neg failed_i \ \wedge \ ready_i) \ \Rightarrow \ \langle \forall j : j \in N_i : \ \ \Diamond\Box(ID_i < ID_j) \ \ \vee \ \ \Diamond\Box(ID_i > ID_j)\rangle).$$

Lemma 5 *If a non-failed process i remains ready forever then eventually any low neighbor of i either fails, remains non-committed forever. Formally,*

$$\Box(\Box(\neg failed_i \ \wedge \ ready_i) \ \Rightarrow$$
$$\langle \forall j : j \in N_i : \ \Diamond\Box(ID_i < ID_j) \ \vee \ \Diamond\Box(ID_i > ID_j \ \wedge \ (failed_j \ \vee \ (\neg commit_j \ \wedge \ ID_{ji} = ID_i)))\rangle).$$

Definition 2 *We construct a waiting graph W_i for any non-failed process i that is ready forever as follows:*

- *Process i is in W_i.*
- *For any process j in W_i, apply Lemma 5. Add a neighbor k to W_i for which* $\Diamond\Box(ID_j > ID_k \ \wedge \ \neg failed_k \ \wedge \ ready_k \ \wedge \ ID_{kj} = ID_j)$ *holds.*
- *Repeat until no more processes can be added to W_i.*

Informally, graph W_i includes process i and all non-failed processes that are ready forever and reachable by a chain of decreasing IDs through other processes in W_i. The following lemma is proved in the appendix.

Lemma 6 *For any process j in W_i, eventually $bound_j$ becomes stable. Formally,*

$$\Box\langle \forall j : j \in W_i : \ \ \Diamond\Box(bound_j) \ \ \vee \ \ \Diamond\Box(\neg bound_j)\rangle.$$

Based on the above lemma, we can prove the following theorem about the failure locality of the algorithm.

Theorem 4 *If interaction $\mathcal{I}$ remains ready forever then either $\mathcal{I}$ or some interaction conflicting with $\mathcal{I}$ fails.*

Proof: For the sake of contradiction, assume that interaction $\mathcal{I}$ remains ready forever and that all interactions conflicting with $\mathcal{I}$ do not fail. Let i and j be the two processes in $\mathcal{I}$. By Lemma 4, assume without loss of generality that $\Diamond\Box(ID_j < ID_i)$. Then both i and j belong to W_i and by Lemma 6, eventually $bound_i$ and $bound_j$ become fixed. If $bound_j$ becomes fixed to true then since neighboring interactions of $\mathcal{I}$ do not fail, process j will eventually send a yes message to a non-failed process. This process will eventually respond with a commit or a release message. In the former case, process j becomes non-ready and in the latter case, $bound_j$ is reset to false. Both of these are not possible by assumption. Therefore, it must be the case that $bound_j$ is fixed to false eventually. Similarly, it can be shown that $bound_i$ is fixed to false eventually. From the algorithm, process j sends a request message to process i each time $bound_j$ is reset to false. Consider a request message sent by process j after $bound_j$ becomes fixed to false. Process i responds to this message by either a yes or a wakeup message. In the former case, process j becomes non-ready and in the latter case, process j retransmits a request message. The former is not possible as process j remains continuously ready. Therefore, process j sends a request message to process i infinitely often. Since $bound_i$ is eventually fixed to false, process i will eventually respond with a yes message, thus committing interaction $\mathcal{I}$. This contradicts our assumption that $\mathcal{I}$ remains ready forever. □

6 Concluding Remarks

In this paper we proved that a failure locality of one cannot be achieved for the dining philosophers problem. The impossibility result was essentially based on the absence of global information in a totally asynchronous distributed system. Subsequently, we presented an algorithm with an optimal failure locality of two. The algorithm was based on the idea of a *bounded doorway* where processes exchanged IDs with their neighbors upon making progress. After that we considered the committee coordination problem and showed that a failure locality of 0 is unachievable for this problem. The proof was based on a reduction to the problem of distributed consensus. Finally, we presented an optimal algorithm for the binary version of the problem, once more based on the idea of a bounded doorway.

The general case of the committee coordination problem can be solved by a reduction to the dining philosophers problem [5]. If we use the optimal algorithm that is presented here then the resulting solution to the committee coordination problem has a failure locality of two. It is an open question whether a failure locality of one can be achieved for this problem.

References

[1] *Reference manual for the Ada programming language.* United States Department of Defense. Springer-Verlag, New York, 1983.

[2] B. Awerbuch, A. B. Goldberg, M. Luby, and S. A. Plotkin. Network decomposition and locality in distributed computation. In *Proceedings of the 30th IEEE Annual Symposium on Foundations of Computer Science*, pages 364–369, 1989.

[3] B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. In *Proceedings of the 31st Symposium on Foundations of Computer Science*, pages 65–74, 1990.

[4] G. N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM Trans. Program. Lang. Syst.*, 5(2):223–235, (April 1982).

[5] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley Pub. Co., Reading, Mass., 1988.

[6] M. Choy and A. K. Singh. Efficient fault tolerant algorithms for resource allocation in distributed systems. 1992, submitted to Journal. A preliminary version appears in *the Proceedings of the 21th Annual ACM Symposium on Theory of Computing*, 1992, pages 593–602.

[7] M. Choy and A. K. Singh. Adaptive algorithms for the mutual exclusion problem. unpublished manuscript.

[8] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.

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

[10] A. V. Goldberg and S. A. Plotkin. Parallel $(\delta + 1)$ coloring of constant-degree graphs. *Information Processing Let.*, 25:241–245, 1987.

[11] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 331–335, 1987.

[12] D. Kumar. An algorithm for N-party synchronization using tokens. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 320–327, 1990.

[13] L. Lamport. The mutual exclusion problem: Part I and II. *J. ACM*, 33(2):313–348, (April 1986).

[14] N. Linial. Distributive algorithms – global solutions from local data. In *Proceedings 28th IEEE Annual Symposium on Foundations of Computer Science*, pages 331–335, 1987.

[15] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15:1036–1052, 1986.

[16] N. Lynch. Fast allocation of nearby resources in a distributed system. In *Conference Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 70–81, 1980.

[17] Pnueli, A., and Z. Manna, How to Cook a Temporal Proof System for Your Pet Language, Proceedings of the 9th ACM Symposium on Principles of Programming Languages, 1983, 141–154.

[18] S. Ramesh. A new and efficient implementation of multiprocess synchronization. In *PARLE Parallel Architectures and Languages Europe. Lecture Notes in Computer Sciences 259*, pages 387–401, Springer-Verlag, Berlin, 1987.

[19] A. P. Sistla. Distributed algorithms for ensuring fair interprocess communications. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 266–277, (Vancouver, Aug. 1984).

[20] E. Styer and G. Peterson. Improved algorithms for distributed resource allocation. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 105–116, (Toronto, Aug. 1988).

7 Appendix

Observation 1 $(\neg ack_i \Rightarrow thinking_i) \ \wedge \ (hungry_i \Rightarrow ack_i)$.

Observation 2 *If* $ID_i = ID_j$ *then processes* i *and* j *are neighbors, and process* i *is thinking and is currently participating in a successful ID exchange with process* j*.*

Proposition 1 *If process* i *starves forever then eventually* ID_i *becomes fixed. Formally,*

$$\Box(\Box(hungry_i) \ \Rightarrow \ \langle \exists m :: \Diamond\Box(ID_i = m)\rangle).$$

Proof: The ID of process i increases only when it finishes eating and exchanges its ID with a high neighbor successfully. Since the range of IDs in the system is fixed, ID_i can not decrease infinitely often. $\Box$

Proposition 2 *If process* i *and its neighbor* j *do not fail and process* i *starves forever then eventually* $ID_{ji} = ID_i$*. Formally,*

$$\Box(\Box(\neg failed_i \ \wedge \ \neg failed_j \ \wedge \ hungry_i) \ \Rightarrow \ \Diamond\Box(ID_{ji} = ID_i)).$$

Proof: By Proposition 1, ID_i eventually becomes fixed. Process i sends a *new-ID* message to process j whenever ID_i changes, and in particular, when ID_i changes for the last time. Process j sets ID_{ji} to this final value upon receiving this message. $\Box$

Proof of Lemma 1: Consider a state s at which the temporal formula $\Box(\neg failed_i \ \wedge \ hungry_i)$ holds, i.e., $(\neg failed_i \ \wedge \ hungry_i)$ holds at state s and at all of its future states. By Proposition 1, ID_i becomes fixed at some future state t. Starting from state t, any further changes in the ordering of ID_i and ID_j are determined only by changes to ID_j. If process j fails at some future state then the proof follows as ID_j will eventually become fixed. So, assume that process j does not fail. Then, we apply Proposition 2 to conclude that eventually ID_{ji} and ID_i are equal and fixed forever. Also, by Observation 2, $ID_i \neq ID_j$ holds forever eventually.

For the sake of contradiction, assume that there does not exist a future state after which either $(ID_i < ID_j)$ or $(ID_i > ID_j)$ holds forever. Then the value of ID_j must be bouncing above and below that of ID_i infinitely often. This implies that ID_j is increased from a value smaller than ID_i to a value larger than ID_i infinitely often. According to the algorithm, this can happen only if process j eats with a smaller ID than ID_i and exchanges its ID after eating with some neighbor infinitely often. However, by the ID exchange mechanism of the doorway, process j will eventually exchange its smaller ID with process i. This will result in a change to ID_i and contradicts our assumption that ID_i is fixed. $\Box$

Proof of Lemma 2: Consider a state s at which the temporal formula $\Box(\neg failed_i \wedge hungry_i)$ holds. By Lemma 1, either $\Box(ID_i < ID_j)$ or $\Box(ID_i > ID_j)$ holds at some future state t. The proof obligation is met in the former case. So, assume that $\Box(ID_i > ID_j)$ holds at state t. If process j fails at some future state then once more the proof obligation is met. So, assume that process j does not fail. From Proposition 2, it follows that $\Diamond\Box(ID_{ji} = ID_i)$ holds at state t. Then, our proof obligation is to show that $\Diamond\Box(\neg eating_j)$ holds at state t. For the sake of contradiction, assume that $\Box(\neg eating_j)$ does not hold at any future state of t. Since all eating periods are finite, this implies that process j eats infinitely often. However, by the ID exchange mechanism of the doorway, process j will eventually exchange its smaller ID with process i. This will establish $ID_i < ID_j$ at some future state, contradicting our assumption that $\Box(ID_i > ID_j)$ holds at state t. $\Box$

Proof of Lemma 3: For any process j in W_i, define the distance of j as the length of the longest decreasingly chain of IDs from process i to process j in the graph W_i. We prove the lemma by induction on the distance of process j. The induction proceeds from the maximum distance down to distance 0.

Base Case: Let process j be at the maximum distance, i.e., process j is a leaf node in W_i. Let process k be any neighbor of process j. By Lemma 2, either $\Box(ID_j < ID_k)$ or $\Box(failed_k)$ or $\Box(thinking_k)$ or $\Box(ID_j > ID_k \wedge \neg failed_k \wedge hungry_k \wedge ID_{kj} = ID_j)$ holds eventually. The proof obligation follows trivially in the first case. In the next two cases, process k eventually stops sending out any requests and consequently, process j eventually stops sending out the fork between processes j and k. In the last case, process $k \in W_i$, contrary to our assumption that process j is a leaf node.

Induction Step: Consider a process j in W_i at a distance m. By Lemma 2, for any neighbor k, either $\Box(ID_j < ID_k)$ or $\Box(failed_k)$ or $\Box(thinking_k)$ or $\Box(ID_j > ID_k \wedge \neg failed_k \wedge hungry_k \wedge ID_{kj} = ID_j)$ holds eventually. The proof obligation follows as in the base case for the first three cases. In the last case, process k is at a distance greater than m in W_i. By the induction hypothesis, eventually each low fork of process k either remains at process k or remains away from process k forever. Now there are two cases to consider: either eventually process k is holding all its low forks forever, or eventually process k is missing some low fork forever. In the former case, process k sends requests, if necessary, to its high neighbors (including process j) to collect the forks. Since process k is a low neighbor of process j, process j will release the fork to process k. The fork stays at k forever because *all-low-forks*$_k$ holds forever. In the latter case, the condition *all-low-forks*$_k$ will be false forever and therefore, process k will eventually stop sending requests to process j. Consequently process j, which releases low forks only on request, will eventually stop sending the fork to process k. The proof follows. $\Box$

Proof of Lemma 6: For any process j in W_i, define the distance of j as the length of the longest decreasingly chain of IDs from process i to process j in the graph W_i. We prove the lemma by induction on the distance of process j. The induction proceeds from the maximum distance down to distance 0.

Base Case: Let process j be at the maximum distance, i.e., process j is a leaf node in W_i. Let process k be any neighbor of process j. Consider any state s. By Lemma 5, either $\Box(ID_j < ID_k)$ or $\Box(failed_k)$ or $\Box(idle_k)$ or $\Box(ID_j > ID_k \wedge \neg failed_k \wedge ready_k \wedge ID_{kj} = ID_j)$ holds eventually. In the first case, process k has a higher ID than process j and therefore, will eventually stop sending request messages. Process k will also eventually stop sending request messages in the next two cases since an idle or a failed process does not send any request messages. The last case is not possible for then process $k \in W_i$, contrary to our assumption that process j is a leaf node. Thus, process j eventually stops receiving request messages. Since $bound_j$ is changed to true only when a request message is received, this implies that eventually $bound_j$ becomes fixed.

Induction Step: Consider a process j in W_i at a distance m. Suppose for the sake of a contradiction that $bound_j$ does not become fixed. Then it must be set to true and reset to false infinitely often. This implies that process j sends yes messages to neighbors infinitely often. Let k be a process that receives yes messages from process j infinitely often and consequently, sends request messages to process j infinitely often. By Lemma 5, either $\Box(ID_j < ID_k)$ or $\Box(failed_k)$ or $\Box(idle_k)$ or $\Box(ID_j > ID_k \wedge \neg failed_k \wedge ready_k \wedge ID_{kj} = ID_j)$ holds eventually. In the first three cases, it can be shown as in the base case that process k will eventually stop sending request messages. Therefore, the fourth case must hold, i.e., process k is a successor of process j in W_i. From the induction hypothesis, $bound_k$ becomes fixed eventually. If $bound_k$ is fixed to true eventually then according to the algorithm process k eventually stops sending request messages. So, assume that $bound_k$ is fixed to false eventually. Since process j sends yes messages to process k infinitely often, process k will eventually receive a yes message and commit the interaction between j and k. This will make process j non-ready and thus, contradicts the assumption that process j is in W_i. $\Box$