Generalized FLP Impossibility Result for t-resilient Asynchronous Computations *

(Extended Abstract)

Elizabeth Borowsky
(borowsky@cs.ucla.edu)

Eli Gafni
(eli@cs.ucla.edu)

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024
U.S.A.

Abstract

Demarcation of the border between solvable and unsolvable distributed tasks under various models is the holy grail of the theory of distributed computing. One of the most celebrated of these results is [6] (FLP) which established the impossibility of asynchronous consensus that can tolerate a single undetected fail-stop processor. This paper generalizes FLP to multiple faults. It establishes that k-set consensus proposed by Chaudhuri is impossible, if the protocol is to tolerate k failures, while there exists a protocol that tolerates $k-1$ failures.

Our proof technique is completely different than the one employed in [6]. We introduce a new model of computation, the *immediate-atomic-snapshot.* We fully characterize the graph of waitfree views within the model. Applying a variant of Sperner Lemma to this graph establishes the impossibility of $k+1$ processors achieving waitfree k-set consensus.

Finally, we introduce a new notion of *non-blocking-busy-wait* agreement protocol. With this protocol we construct a read/write waitfree simulation technique by which $k+1$ processors can produce a k-faulty execution of n processors protocol. Using the simulation we establish the impossibility of n processors protocol that achieves k-set consensus and tolerates k failures, by reducing it to the waitfree case.

*Work supported by NSF Presidential Young Investigator Award under grant DCR84-51396 .

1 Introduction

One of the most celebrated results in distributed computing is [6] which established the impossibility of asynchronous consensus that can tolerate a single undetected fail-stop processor. That is, from some point on a processor stops taking steps. Undetected fail-stop failure can be used to model a slow process. Given a task the question is then in what way a single slow process can affect the execution of the task. Essentially, the FLP result says that in an asynchronous system using message passing or read/write shared memory for interprocess communication, even a simple synchronization task like single shot mutual exclusion (a.k.a test-and-quit) can be executed only as fast as the slowest processor. In the limit, if a single processor stops taking steps at some critical time, the failure may prevent any other processor from either entering the critical section or deciding to give up.

FLP has given rise to a lively research of the asynchronous model. The authors of [10] investigated the "amount of synchrony" that has to be introduced into the system in order to reach consensus. A number of problems are raised in [3] with a subsequent investigation into whether they are solvable or not. In [4] a combinatorial characterization is given for the input/output relation of the tasks which are solvable in spite of a single failure. Yet, no meaningful problem was shown to be solvable with $k-1$ failures but unsolvable with k failures. Such a problem and its impossibility proof would lead the way to characterization of this class of problems, in as much as FLP lead to [4]. In particular such a problem will aid in characterization of which tasks n processors can execute waitfree. that is in spite of $n-1$ failures. Waitfree computation is investigated and partially characterized in [8] given a stronger interprocessor communication than is provided by message passing or read/write model.

To our knowledge, Chaudhuri in [5] made the first serious attempt to provide the FLP counterpart in case of $k > 1$ failures. She proposed the k-set (agreement) consensus problem. That is, each processor, which holds an initial private value, must halt with an output value which is one of the initial values, and the total number of distinct output values of the different processors is to be bounded by k. She conjectured that there is no protocol that achieves k-set consensus in spite of k failures, and has given an algorithm to solve it when there are at most $k-1$ failures. In an attempt to follow the FLP steps, she successfully generalized some of the FLP proof steps, but was unable to generalize others. Yet in her attempt, she observed something very insightful: namely, that FLP makes repeated use of the one dimensional Sperner Lemma, and therefore it stands to reason that the generalization of FLP to $k > 1$ should use higher dimensional versions of the Lemma.

We proceed along the steps of [5]. Many of our ideas can be traced to [5] where they appear in an embryonic form. Yet we concluded that the FLP proof is too "procedural," referring to low level notions like critical states and their valency. Following [8] we first concentrated on the waitfree case, that is a protocol for $k+1$ processors that can tolerate k failures. There, what Chaudhuri worked so hard to obtain, the "$(k+1)$-valency" of the initial state, is given for free. Our first step is to prove that $k+1$ processors cannot waitfree achieve k-set consensus.

In breaking with the FLP technique, we argue directly about execution sequences, and views of processors therein. We define the graph of views of all possible waitfree computations. Nodes in this graph are views of processors after they reach a "decision" state. Two views of processors are neighbors if they correspond to the same execution. In the extreme views of this graph, processors see only themselves and thus must decide their own input value as output. Similarly, a processor may decide only on the input value of a processor that appears in its view. Using a variant of Sperner's Lemma we prove the existence of a single run where the view of every processor in that run causes each processor to decide on his own input value as output, establishing the impossibility required.

Then, in a step which is less technically involved but perhaps even more novel, we extend the result to show that for any number of processors n, $n > k$, there is no read/write protocol to implement a k-set consensus in spite of k failures. This is done by reduction to the previous problem. We let $k+1$ processors waitfree simulate the protocol and show they can then wait-free solve the k-set consensus problem.

The simulation is facilitated by the idea of an agreement protocol with a property of *non-blocking busy-wait*. The idea is to design an agreement protocol whose code is partitioned into a straight-line part followed by a busy-wait part. A processor which is busy-waiting does not block the agreement. Only a processor executing the straight-line part does. While busy-waiting a processor may "time-share" and participate in another agreement. Thus each processor that participates in multiple agreements is at each point of time executing at the most the straight-line part of a single agreement protocol, and consequently can block only a single agreement. To simulate a concurrent read/write protocol, the simulating processors have to agree on the outcome of a read step of a simulated code. Since the simulation is wait-free at most k processor may fail, blocking the simulation of at most k codes, and thus giving rise to an execution which is at most k faulty.

Finally we show that the k-set consensus task is fundamental by defining the new problems of k-leader election and k-set test-and-set and reducing them to consensus.

In other papers in this proceedings [12] has independently proven the impossibility of the wait free k-set consensus in spite of k failures, while [13] has supplied the counterpart of [4] to t-resilient computations.

The paper is organized as follows. First we define precisely our model. Afterwards we show how the impossibility of the $k+1$ processors wait-free achieving k-set consensus implies the impossibility of the general k-resilient version of the problem. Then we prove the impossibility of a waitfree $k+1$ processor k-set consensus. And finally we define and reduce various problems to the set consensus problem.

2 Model

The equivalence among various distributed computation models has played an important role in our choice of a model. The equivalence of shared-memory and message passing is established in [2]. That is, any task computable with no more than t faults ($t < n/2$) in the message passing model is computable in the shared memory model and vice versa. The equivalence between various shared memory models is established in [9] and [1] and numerous other papers. Further more, to establish impossibility one may choose the strongest model in which the impossibility holds. We introduce the model of a single-writer / immediate-atomic-snapshot-reader shared memory. This allows us to ignore details like message queue discipline, or which memory cell is being written or read, and summarize an execution as a sequence of sets of processors.

An n-processor single writer/immediate atomic snapshot reader shared memory, is a system of n processors communicating by atomically writing into and reading from shared memory. The memory is divided into n cells, C_i, $i = 1, ..., n$. Processor P_i can only write to C_i, but reads all of the memory in one atomic step. Processors submit write-read requests. That is they have computed a value to write, they wish it to be written and they request a snapshot of the memory in order to compute the next value to be written. The execution of requests is arbitrated by a scheduler. The scheduler chooses a set of requesting processors to first simultaneously write and then simultaneously read. We call each set a *concurrency class* since the order of writes and reads within the set can not be distinguished. The state of the memory and processors after each write and read step changes in the obvious way.

An *execution* or *run* of the system is a sequence of global states (memory plus processors), starting at some initial state. One state can be reached from the previous by specifying the concurrency class that leads from one to the other. For deterministic processors, as the ones dealt with here, this sequence of global states is fully specified by the initial state and the sequence of sets of processors chosen by the scheduler. Since we deal with computability we assume w.l.o.g. that a processor never overwrites what it has written before. When something new is to be written it rewrites whatever was written there before, together with the new information to be written. We assume that each processor has local states in which it is *decided.* A decision is associated with an *output value.* A decided state together with the value associated is required to be a stable property.

Thus, we do not interpret what a processor writes, i.e. we deal with a *full-information* protocol [7]. Since the evolution of a run is determined by the sequence of concurrency classes, we follow the evolution of the knowledge of this sequence of classes by the participating processors. A processor that reads sees all the writes in the run that preceded its read. Each write by a processor corresponds to this processor being scheduled in some concurrency class. Yet the processor that reads the writes has some uncertainty as to how these writes are partitioned into concurrency classes. We later precisely define this uncertainty. We define the view of a processor after a read to be the set of all the runs that do not violate this uncertainty.

Consequently, in a full information protocol the only thing determined by the protocol is which views are decided and which are not. That is, a protocol in this model is a mapping of views to a decided/undecided together with an output value that is associated with the decision.

We now define several variants of "t-resilient" protocols. An execution of a protocol partitions processors into four sets. Processors that took infinite number of steps - *infinite-step set.* Processors that took finite number of step but are decided by their last step - *finite-step-decided set.* Processors that took finite number of steps but are undecided by their last step - *finite step-undecided set.* Processors that did not take any step - *sleeping set.* A protocol is *t-resilient* if in any execution the existence of an infinite-step undecided processor implies that the number of faulty processors is greater than t. An execution in which no more than t processors fail is *t-faulty execution.* We have four ways to define a set of faulty processors. First by whether we consider a sleeping processor as faulty or not, and secondly whether we consider a finite-step-decided processor to be faulty or not.

A protocol for n processors is *wait-free* if it is $(n-1)$ resilient. It is easy to see that all models coincide on the wait-free case in the sense that no justification exists for an infinite undecided run. Thus if we halt a processor after it decides then a wait-free protocol does not have an infinite execution. For each processor, given an input, there exists some number of times it may be scheduled after which the processor must decide, and then halts without blocking other decisions.

A *task* is a partial point to set mapping which for each set of processors with given input associates what possible combination of outputs the processors may decide on. A protocol t-resilently implements the mapping if in a t-faulty execution in which only a set S of processors participates. i.e. the complement set of S is a sleeping set, they decide on a combined output that agrees with the mapping for S.

A task is t-solvable if there exists a t-resilient protocol that implements it. A task A is t-reducible to task B, if there exist a t-resilient protocol for A, which on top of writing and reading, may submit inputs values to versions of B and obtain compatible output values. Two tasks are t-equivalent if they are t-reducible to each other. Obviously if two objects or tasks are t-equivalent they are $t-1$-equivalent.

We thus obtain four fault models. The strongest model, in the sense that every task t-solvable in it is solvable in all the other models, is the one in which processors can "wait" on the least number of other processors. This model is the one in which a sleeping processor or a finite-step-decided processor are not considered faulty. i.e. only finite-step-undecided processor is considered faulty. The weakest model is the one in which any non-infinite-step processor is considered faulty. We mention in passing without a proof that the only material distinction is whether a sleeping processors is considered faulty or not. When programming processor i, since we must take into account the execution in which a decided processor j may only appear together with i at the same concurrency class, after j has decided, we may as well let j halt without considering it faulty, since it in no way "helps" processor i.

Our impossibility result holds for all these models. We prove that in none of them can we solve k-set consensus k-resilently, while in all of them we can solve it $k-1$-resilently.

3 Wait-Free Simulation of t-resilient Computation

3.1 Non-blocking-Busy-Wait Agreement Protocol

We first design a read-write agreement protocol. The protocol is 0-resilient but it has the property that waiting is done only in the last step. A wait is a loop of reads that terminates when the read returned complied with some condition. Thus the code of the agreement protocol consist of a straight line code ending with a single read loop. When all participating processors have reached the wait statement they can all decide. Thus if a processor at the wait statement can not make a decision, we can attribute it to at least one other processor which is in the middle of the straight-line part of the code.

The agreement protocol we use is a one-shot mutual-exclusion algorithm. Here we state it in the generality of l-exclusion, with all processors observing when any processor decides to enter the critical section. The algorithm presented in figure 1, is a simple variant of the FIFO l-exclusion algorithm proposed in [11]. First take l to be 1. All the variables are single writer multi-reader (i.e. none are private, remember that any read or write is done atomically, although reading and writing many variables).

To agree on a value each processor writes his proposed value into the shared memory then enters

```
Protocol for Processor $i$:
Initially:
$x_i$=false,  $S_i = \{1, \ldots, n\}$, $label_i = (0, i)$.

$x_i := true$;
read
$label_i := (Max\{y | (y, j) = label_j\} + 1, i)$;
$S_i := \{j | x_j = true\}$;
read until there exists $j$ such that:
$|\{k \in S_j | label_k \leq label_j\}| \leq l$
(* let $\leq$ denote the lexicographic ordering *)
choose: $j$
```

Figure 1: Agreement Protocol

the agreement protocol. The value of the winner is then taken.

3.2 The Simulation

We simulate the execution of n pieces of read/write code by $k+1$ processors. We use an agreement protocol for each read statement simulated so that processors can agree on what read value is returned in every read step for each of the n codes. Each code has to be simulated sequentially thus no processor can simulate past a read statement without an outcome to the agreement protocol for that statement. However, a processor waiting on the outcome of one simulated read may join or start simulating a read in another code. Thus a processor may be in the straight-line part of the agreement protocol for only a single simulated read. Since at most k simulating processors may fail, their failure may block the simulation of at most k pieces of code. All remaining codes are guaranteed to proceed producing k-faulty execution.

We now apply this simulation to the Set-Consensus problem. The simulation will imply that if the (n, k)-Set-Consensus Task is k-solvable, then the $(k+1, k)$-Set-Consensus task is wait-free solvable, which is proved impossible in the next section.

The (n, k)-Set-Consensus Task [5]: Each of n processors holds a private value initially. Each processor decides on a value from among the initial values of the participating processors, such that the number of distinct values in the output is at most k.

Before simulating a code the simulating processors will agree on which value of their initial conditions to associate with an input to the code (using an agreement protocol). They then produce a k faulty execution. Since the code is k-resilient some simulated codes must decide. A processor that observes a decided code adopts its decision value as its output value and stops. Since the simulated code will not decide on more than k initial values, we are done.

3.3 General Ramifications of the Simulation

We now use this simulation in a more general way. First we give an explicit proof that an object of consensus number $k+1$ is strictly stronger than one of consensus number k. This result was derived independently in [14] in a less direct way. Without loss of generality [8] we can assume that a protocol that uses k-consensus number objects actually use k-consensus objects. This use is accomplished by allowing the code to have a line invoke(consensus object, value), followed by a line return(value), where for each consensus object used this pair of lines appears in at most k codes. We show that if there is a protocol for $k+1$ processors to reach consensus using only k-consensus objects, then 2 processors can reach consensus by read/write. In the simulation we now have two simulating processors and one agreement protocol for each k consensus object. A slow simulating processor can block at most the progress of k codes (through one k-consensus object), leaving at least one waitfree code free to proceed. Thus the faster processor can simulate this remaining code waitfree and terminate with a valid consensus value.

Using similar reasoning we can infer that five processors using 2-consensus cannot wait free solve the 2-set consensus problem, otherwise three processors can wait free read/write solve 2-set consensus. Notice here that two of the three simulating processors can block at most four codes through two separate 2-consensus objects. Thus, again a single code remains to be simulated without chance of blocking.

Similarly, nine processors using objects of $(4, 2)$-set consensus cannot reach 4-set-consensus. Now the idea is to use the l-exclusion agreement protocol with $l = 2$. Under these constraints, in order to block four processors we will need two processors to failstop. That is, we can reduce this to five processor 4-consensus read/write which is impossible.

This leads to the general formula that if the number of processors is greater than the number of the set consensus times the n/k value of the object, then we have impossibility.

4 Impossibility of Wait Free $(k+1,k)$-Set-Consensus

4.1 Set-Up for Applying the Sperner Lemma

Consider the wait-free $(k+1,k)$-Set-Consensus problem and assume w.l.o.g. that the initial value of a processor is its own unique ID. A trivial solution by which a processor is programmed to decide an initial pre-agreed ID is precluded by the requirement that an initial value of a processor P_i may be a decision value of some processor only if P_i is participating. Thus we have a single input problem. Consequently, for each processor P_i there exists some number of steps N_i after which it must decide. We consider executions in which processor fails after these N_i steps. Thus we consider the set S to be all the possible executions of the full information protocol in the single-writer/immediate-atomic-snapshot-reader model, in which each processor P_i takes exactly N_i steps and stops. In these executions all processors decide on some input value. We consider the graph of halting views of S.

Nodes in this graph will be views by processors. A view V_i by P_i represents a set of runs in S, compatible with the view, i.e. the set of runs in which the halting view of P_i is V_i. Two views are neighbors if they share a run in S. We will show that the existence of a decision function represents a restricted coloring of this graph. We will argue by Sperner Lemma that such a coloring is impossible.

For the sake of completeness we also outline the structure of the graph of view $G(N_1,\ldots,N_{k+1})$ in which P_i halts after taking N_i steps. Our impossibility proof does not require the knowledge of this full structure but rather some properties of the structure.

4.2 Runs and Views

For a fixed input a run r is a sequence of concurrency classes $C_1, C_2, \ldots, C_r$. Each concurrency class C_j is a set of labeled processors $P_{i,k}$, with the intended meaning that the k'th appearance of P_i occurs in concurrency class C_j. If $P_{i,k} \in C_m$ and $P_{i,k+1} \in C_n$ then $m < n$. In the set of runs S the last appearance of P_i is P_{i,N_i}.

A *view* of processor is a set of runs together with the name of the processor. Given r we now define $V_{P_{i,k}}(r)$, the view associated with $P_{i,k}$ in r. Define $P_{i,k}$ *after* $P_{j,m}$ in r if $P_{j,m}$ is in the same concurrency class with $P_{i,k}$ or in a concurrency class that precedes it in r. Define $P_{i,k}$ *not-after* $P_{j,m}$ in r, otherwise. $V_{P_{i,k}}(r)$ are all the runs $r' \in S$ such that:

1. $P_{i,k}$ after $P_{j,m}$ in r iff $P_{i,k}$ after $P_{j,m}$ in r'
2. $\forall m > 1, j$ if $P_{i,k}$ after $P_{j,m}$ in r then $r' \in \bigcap V_{P_{j,m-1}}(r)$

Lemma 1 *The after/not-after relation between every pair of processor appearance in a run defines the run.*

Lemma 2 *The view $V_{P_{i,j}}$ defines for each appearance of $P_{i,m}$, $m \leq j$ exactly which processor appearance $P_{i,j}$ is after.*

Lemma 3 *$V_{P_{1,N_1}}(r)$ through $V_{P_{k+1,N_{k+1}}}(r)$ define r.*

Proof: By lemma 2, the views imply the relations between all processor appearances and therefore by lemma 1 it determines the run uniquely. □

From here on we will concentrate on views of the type P_{i,N_i} for all $i = 1,,,k+1$.

Definition: Two views are *adjacent* if they share a run in S.

Lemma 4 *A view of P_i is not adjacent to any other view of P_i.*

Lemma 5 *If a set of views is pairwise adjacent then they all share a run in S.*

Definition: A *j-clique* is set of j pairwise adjacent views.

Definition: A k-clique is a *boundary* clique if all the views in the clique share a single unique run in S.

Lemma 6 *A k-clique is boundary clique iff there is no appearance of the missing processor in any of the views in the clique.*

Definition: A k-clique is an *interior* clique if it is not a boundary clique.

Lemma 7 *Views of an interior k-clique share exactly two runs in S.*

Sketch of Proof: Let P_j be the processor whose view is not in the clique. Let $P_{j,m}$ be the latest appearance of P_j in any of the k views. It is easy to see that the relation of being in the same concurrency class on all the appearances of the k processors in the view and all the appearances of P_j are determined, aside from $P_{j,m}$. The concurrency class C that immediately precedes its appearance is determined. Thus the only two possibilities are to put it in a singleton concurrency class following C, or to join it to the concurrency class the follows C in its absence.

4.3 Recursive Outline of the Graph of Views

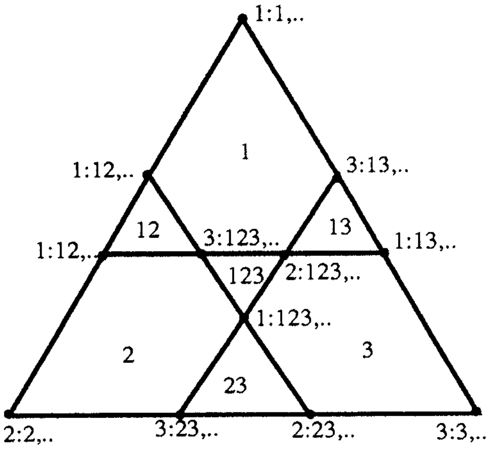

Figure 2: One round $G(N_1, N_2, N_3)$ subdivision.

We divide the views (in this section thought of as labels of nodes in a graph) in $G(N_1, \ldots, N_{k+1})$, with $N_i > 0\ \forall i$, into $\sum_{i=1}^{k+1}\binom{k+1}{i}$ subsets. Each subset is labeled by a different nonempty subset s_* of the $k+1$ processors. The subset of views S_{s_*} associated with s_* is all the views whose corresponding set of runs includes a run in which the first concurrency class is s_*. This grouping does not partition the set of views. Due to the ambiguity of which

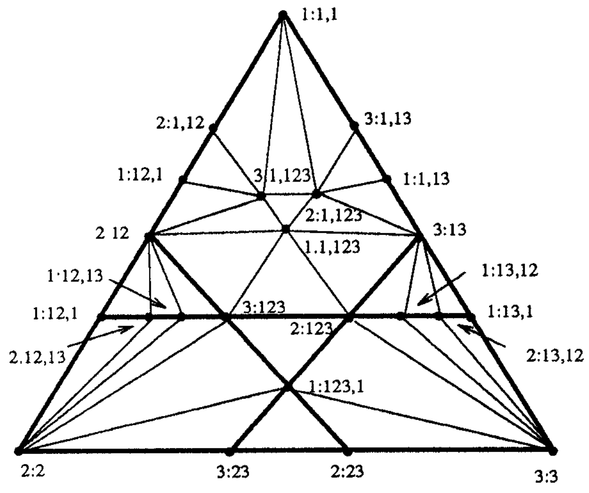

Figure 3: Labeled graph of $G(2,1,1)$.

is the concurrency class of the last appearance of another processor in a view, a given view may have runs with different initial concurrency class.

With respect to the labeled subset of views S_{s_*} of $G(N_1, \ldots, N_{k+1})$ we identify the graph $G(N'_1, \ldots, N'_{k+1})$ where $N'_i = N_i - 1$ if $P_i \in s_*$ and $N'_i = N_i$ otherwise. The views of S_{s_*} of $G(N_1, \ldots, N_{k+1})$ can be mapped in one-to-one correspondence onto views of $G(N'_1, \ldots, N'_{k+1})$ by removing the initial concurrency class s_*. Thus we consider the subset of views S_{s_*} to be the graph $G(N'_1, \ldots, N'_{k+1})$ *prefaced* by s_*, and continue dividing recursively. This builds up a sequence of concurrency classes labeling each subset. This sequence is a common prefix of all runs yielding views all in the subset.

The base of this recursion occurs when $N_i = 0$ for some i. At this point all appearances of processor P_i in the run have been accounted for in the recursion and this view of P_i is compatible with all the remaining views of its subset. Since P_i does not appear in the suffix of any of the remaining views, these views may be subdivided as the graph $G(N_1, \ldots, N_{i-1}, N_{i+1}, \ldots, N_{k+1})$. The view of P_i is adjacent to every view of this graph. Eventually, the graph $G(0_1, \ldots 0_m)$ is simply m empty views which are pairwise adjacent, and the graph $G(N_i)$ is the view of processor P_i in which it writes and reads alone N_i times.

This graph is properly triangulated due to the fact that there are exactly one or two views compatible with each of k pairwise adjacent views. To provide help in understanding the structure of this graph, a one round subdivision for the case of three processors is presented in Figure 2, and the graph G(2,1,1) is presented in full in Figure 3.

4.4 Application of the Sperner Lemma

Definition: A *j-k-clique* of a graph G is a k-clique which is contained in exactly j distinct $k+1$-cliques of G.

Lemma 8 (Variant of Sperner's Lemma) *Given a graph G colored by colors $1, ..., k+1$, such that every node is contained in a $k+1$-clique, and every k-clique is a 1-k-clique or a 2-k-clique, then if the number of 1-k-cliques in G colored by colors 1 through k is odd, then the number of $k+1$-cliques in G colored by colors 1 through $k+1$ is odd. In particular, there must be at least one such $k+1$-clique.*

Lemma 9 *The subgraph of $G(N_1, \ldots, N_{k+1})$ induced by views which do not contain any appearance of processor $k+1$ is $G(N_1, \ldots, N_k)$.*

Proof: For every run of $G(N_1, \ldots, N_k)$ there is a boundary clique of $G(N_1, \ldots, N_{k+1})$ which can be completed to form a $k+1$ processor run by adding N_{k+1} concurrency classes of P_{k+1} after processor 1 through k halt. Notice that the views of $G(N_1, \ldots, N_k)$ map in one-to-one correspondence to the views of $G(N_1, \ldots . N_{k+1})$ containing no appearance of processor $k+1$. □

Corollary 1 *The k-cliques of $G(N_1, \ldots, N_k)$ are in one-to-one correspondence with the boundary cliques of $G(N_1, \ldots, N_{k+1})$, no view of which contains an appearance of P_{k+1}.*

We now assume the existence of decision function that complies with the requirements of the $(k+1, k)$-Set-Consensus task. Without loss of generality, let processors initial private values be their own IDs. Such a function is a $k+1$ coloring of $G(N_1, \ldots, N_{k+1})$. Any boundary clique in $G(N_1, \ldots, N_{k+1})$ not associated with $G(N_1, \ldots, N_k)$ cannot be colored by exactly $1, ..., k$. If it were then some processor must have chosen the initial value of a processor it sees no appearance of, which is not allowed. Thus if inductively we prove the the number of k-cliques in $G(N_1, \ldots, N_k)$ colored by exactly $1, ..., k$ is odd, we can apply the variant of the Sperner Lemma to get a $k+1$ clique in $G(N_1, \ldots, N_{k+1})$ which is colored by exactly $1, ..., k+1$, which leads to the required contradiction. The base case of the induction is to notice that there are only two boundary cliques to $G(N_1, N_2)$, namely, the two views in which a processors observes only itself. One of the two boundary cliques is colored by 1, and the other by 2, establishing that the number of boundary cliques colored by 1 is odd.

5 Wait-Free Reductions Among Various Tasks

5.1 Equivalence of the Set-Consensus Task and Multiple Leader Election Task

The (n, k)-Set-Consensus Task [5]: Each of n processors holds a private value initially. Each processor decides on a value from among the initial values such that the number of distinct values in the output is at most k.

The (n, k)-Leader-Election Task: Each processor has his own distinct ID as an initial condition. A processor decides an ID of a participating processor. The total number of distinct IDs decided on is at most k.

We will show that the tasks are wait-free equivalent.

A solution for the leader election easily implies solution to the set-consensus. Each processor writes its private value in shared memory before starting the election protocol. After deciding on a leader in the election protocol, a processor decides on the private value of that leader. This can now be read from shared memory since by the Leader-Election specification the leader is participating in the election protocol.

Set-Consensus solves the leader problem as well, though the construction is more involved. First let the private value of each processor be its own distinct name, and apply the Set-Consensus protocol. Since the input set may be large it is easy to see that a decided ID must belong to a participating processor. We get a variant of election in which a processor elected by others might not elect itself. To remedy this we will add two wait free phases following the set consensus protocol. In the first phase we will break "cycles" of decided values. In the second phase, we will force every processor whose name was decided to decide itself.

In the first phase after a processor i decides a name j in the consensus protocol it raises an (i,j)-flag (i.e. writes (i,j) into shared memory). It then reads other processors flags. If it sees a $(*,i)$-flag its first phase decision is i, and otherwise the first phase decision is j. Observe a cycle of choices in the consensus protocol. The last processor in that cycle to raise a flag will choose its own name as the choice of the first phase. thereby breaking the cycle. Since each name chosen in the first phase was already a consensus choice, the size of the decision set has not increased.

We add now a second phase to remove chains. The input to the second phase is the choice of the first phase. Processor i that chose j in the first phase now raises a second phase (i,j)-flag. If it sees any first or second phase $(*,i)$-flag it chooses itself, otherwise let l be at the end of the longest maximal path of second phase flags of the type $(i,j_1),(j_1,j_2),...,(j_m,l)$. By the acyclicity of the first phase choices such simple path must exist. The choice of i in the second phase is the processor at the end of the chain it observes. It can be easily seen that since reading is done in snapshot, that the chosen processor l must see a flag pointing to it and will choose itself in the second phase. The second phase choices are the final choices that now satisfy the specification of the multiple leaders task. Obviously a chosen processor must be participating since otherwise its initial value would not have been written and thus cannot be decided on in the consensus task.

5.2 The Relation Between Multiple Leader Election and Set-Test-and-Set Tasks

The (n,k) Set-Test-and-Set Task: Each of n processors has its distinct ID as the initial condition. Each processor decides *leader* or *follower*. The number of *leaders* is at least one and at most k, at least on of the participating processors decides *leader*.

When $k=1$ this is the standard one shot test-and-set task.

A solution to the (n,k) multiple leader election can be wait-free transformed into a solution to the (n,k) test-and-set. After executing the election protocol, let each leader decide *leader* and non-leader decide *follower*. Notice that this is easily accomplished due to the constraint that every leader must choose itself as leader.

A solution to the $(k+1,k)$ test-and-set can be wait-free transformed into a solution to the $(k+1,k)$ Multiple leader election. Processors that obtain *leader* in the Set-Test-and-Set choose their own ID. A Processor that obtains *follower* raises a flag. It then observes other flags. It chooses the ID of a participating processor which did not raise a flag as a leader. The first processor to raise a flag will not be chosen by anybody, and thus the total number of leaders is at most $k+1-1=k$. By the Consensus-Leader wait-free equivalence we can now guarantee that each chosen processor wait-free chooses itself.

Using $(2,1)$-Set-Test-and-Set tasks we can form a network to implement an $(n,1)$-Set-Test-and-Set task. Likewise, an $(n,1)$-Set-Test-and-Set task when used by only two processors implements $(2,1)$-Set-Test-ad-Set. Thus the two tasks are equivalent in power. Similarly we have the equivalence of $(2k,k)$-Set-Test-and-Set to (n,k)-Set-Test-and-Set $n \geq 2k$. We arrange a cascade of $n-2k$ of the $(2k,k)$ tasks. The first $2k$ processors attach to the first task and the rest one to each task. Out of the first task we get at most k *leaders*. These can wait-free rename [3] themselves to the $2k-1$ ports left free in the next task and continue in this way through the system.

Acknowledgment

We are in debt to Yehuda Afek for his encouragement and helpful discussions throughout this work, and to Bruce Rothschild for being a patient listener.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt and N. Shavit. "Atomic Snapshots of Shared Memory", *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 1-14, 1990.

[2] H. Attiya, A. Bar-Noy and D. Dolev, "Sharing Memory Robustly in Message Passing Systems", *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 363–375, 1990.

[3] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg, and R. Reischuk, "Achievable Cases in an Asynchronous Environment", *Proc. 28th IEEE Symp. on Foundations of Computer Science*, pages 337–346, 1987.

[4] O. Biran, S. Moran, and S. Zaks, "A Combinatorial Characterization of the Distributed Tasks Which Are Solvable in the Presence of One Faulty Processor", *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 263–275, 1988.

[5] S. Chaudhuri, "Agreement is Harder Than Consensus: Set Consensus Problems in Totally Asynchronous Systems", *Proc. 9th ACM Symp. on Principles of Distributed Computing*, pages 311–324, 1990.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", *Journal of the ACM*. Vol. 32, No. 2, pages 374–382, April 1985.

[7] G. N. Frederickson and N. A. Lynch, "Electing a Leader in a Synchronous Ring", *Journal of ACM*, Vol. 34, No. 1, pages 98–115, 1987.

[8] P. M. Herlihy, "Impossibility and Universality Results for Wait-Free Synchronization", *Proc. 7th ACM Symp. on Principles of Distributed Computing*, pages 276–290, 1988.

[9] L. Lamport, "On Interprocess Communication, Parts I and II", *Distributed Computing*, Vol. 1, pages 77–101, 1986.

[10] D. Dolev, C. Dwork and L. Stockmeyer, "On the Minimal Synchronization Needed for Distributed Consensus", JACM 34, January 1987.

[11] A. Afek, D. Dolev, E. Gafni, M. Merritt and N. Shavit,"First-in-first-Enabled l-Exclusion", *Proc. 4th International Workshop On Distributed Algorithms* , Bari, Italy 1990.

[12] F. Zaharoglou and M. Saks," Wait-Free k-set Agreement is impossible: The Topology of Public Knowledge", This Proceedings.

[13] M. Herlihy and N. Shavit," The Asynchronous Computability Theorem for t-Resilient Tasks", This Proceedings.

[14] P. Jayanti and S. Toueg, "Some Results on the Impossibility, Universality and Decidability of Consensus", private communication.