Simulating Few by Many: Limited Concurrency = Set Consensus (Extended Abstract)

Eli Gafni, UCLA

Rachid Guerraoui, EPFL

May 5, 2009

Abstract

Perfection is the enemy of the good. Most backoff schemes wait until a single contending candidate survives. But it is “the last mile,” getting contention down from few to one, which is usually the most costly: backoff systems which wait on some fixed $k > 1$ perform better than those that require perfection, i.e. $k = 1$. This paper asks what tasks are read-write solvable “k-concurrently ” i.e. tasks where progress can be guaranteed when contention goes below $k+1$. That is, what good is backoff to $k > 1$? While backoff to $k = 1$ is indeed omnipotent and can guarantee progress on any task, what set of tasks can be guaranteed progress with $k > 1$? We show that the set of these tasks is exactly the set of tasks that are solvable wait-free with the availability of k-set consensus. It is now for the system designer to decide whether good is good enough.

1 Introduction

Well designed systems rarely experience contention. Two concurrent codes that contain conflicting sections are hopefully scheduled such that the need for explicit mutual exclusion is never experienced. This is a virtuous cycle: With no contention a section that has started finishes fast. Thus by the time of another potentially conflicting section starts, the older section has terminated. On the flip side, when contention does arise, the cycle becomes vicious: Contention resolution takes time, and by the time contention is resolved, more conflicting sections have started. Thus, it makes a big difference whether contention resolution algorithms have to kick-in once two conflicting section are concurrent, or contention only arises when $k > 1$ are concurrent. If conflicting sections "start to conflict" only when more than k threads are concurrent, there is a potential for big performance gains.

In this paper we consider "conflicting sections" to be the collection of section codes at each processor that solve some task. Think of a section as a read-write implementation of a subroutine "invoke-return" that is invoked with some parameter and returns a result i.e. a task [10]. We investigate and characterize what tasks are read-write solvable as long as concurrency does not exceed some $k > 1$. The quest to answer this question has led us to design a novel simulation technique, which we call *simulation-by-value*, and which will be outlined shortly, after we highlight the result and some of its ramifications. Aside from the technical contribution of the "by-value simulation," we have a software-engineering contribution. We combine the "by-value simulation" it with the Extended BG-agreement [3] in a way which is, to us, an engineering marvel.

Our main tangible result is that the tasks that are read-write solvable when no more than k processors are concurrent, i.e. the number of participating processors which did not output is never more than k, are exactly the tasks that are wait-free solvable when k-set consensus is available [6]. As a consequence we show that these are also the tasks that can be solved *k-obstruction-free* [16]. (A task is solvable obstruction-free, if there is a read-write code for it that is safe - it never allows a wrong output, and it guarantees progress, if eventually there is no contention.)

Underlying our result lies a new simulation technique, which we call *simulation-by-value*, and which generalizes the Extended-BG simulation [3], itself an extension of the BG simulation [2, 5].

To get the idea underlying our simulation, consider n processors that want to solve a task T. Assume these processors have a read-write shared memory wait-free code to solve T. In a standard execution model, the processors post their input, execute their code to solve T, get an output and depart. Assume now that, instead of executing their codes themselves, the processors, acting as clients, simply post their input in shared memory. A bank of other s processors, acting as servers, are now going to execute the code on behalf of the clients, and eventually post an output for them. The servers execute asynchronously, communicate among themselves through shared memory and may fail by stopping. We would like the servers to give each client an output wait-free as long as at least one server did not crash. This precludes the allocation of a code per server, as the server allocated to a client might crash and cannot be distinguished from a merely slow one. Why are we interested in this execution model? If the number of servers is k then by each server choosing a code to work on and complete it, before choosing another code to work on, they execute at most k codes concurrently. Can a method be designed to accomplish this?

Unfortunately no wait-free method can accomplish it as it will tantamount to solving consensus, which is impossible [7]. In this model of execution each live server may know all the outputs for clients that arrived. The essence of wait-free solvability is that a processor cannot insist on knowing the output of another processor. In our case processors may pretend to be clients, post their ids, execute snapshot codes, and since they know all snapshots for participating servers, they can elect the lowest id server that appears in the smallest size snapshot. Later snapshots, by servers that arrive afterwards will be of larger size than the smallest observed already, hence election/consensus.

Yet, if we somewhat relax the requirement on the delivery of outputs, the client-server execution model above can be implemented. For $d < s$ if d servers crash we do not require progress, even though there are live servers, once the number of missing outputs is less or equal to d. Nevertheless, we still cannot allocate codes to servers since when the number of clients pending output is less than $2s-1$ there is no read-write wait free algorithm to accomplish the allocation [8,19]. How then do we then solve the relaxed problem? The BG simulation [2,5] does that. All servers simulate all codes by agreeing on the outcome of each *read* in each code. To agree on the outcome of a *read* is to reach consensus. The BG simulation employs simultaneous "blocking agreement-protocols." A processor participates at one agreement at a time, thus it can block only a single agreement on a single *read* value. Correspondingly, each blocked agreement prevents further simulation of a single code. Thus at most d codes will not progress but all the rest will indeed obtain an output.

But, the servers are pedagogical "fiction." What we will really have are clients simulating servers working on behalf of clients! i.e., clients simulating the codes of few clients as we want to simulate few by many. It is easy to see that if we we had a resiliency condition that at most d participating clients may crash, then the clients could simulate $d+1$ servers. They can also solve $d+1$-set consensus [6]. What if instead of the resiliency condition clients work wait-free but are given $d+1$-set consensus? We know that the condition that a participating client will not crash (i.e. $d=0$) is equivalent to clients work wait free but are given consensus (i.e. $d+1=1$). Does this equivalence generalize to any d, i.e., beyond 0.

This paper answers this question on the affirmative. The way we would use consensus to simulate a single server is by the state-machine method [18]. So what this paper accomplishes is a generalization of the state-machine method when we are given k-set consensus rather than consensus. Here comes handy the equivalence between k-set consensus and k parallel independent consensuses introduced in [12]. It allows us to generalize the state-machine method such that each state-machine is a server and as a result we will get the "by-value" simulation. This simulation allows clients with k-set consensus to simulate k BG servers working wait-free. We simulate BG servers rather than original codes of clients, since the code of the BG servers tells them which client to simulate next etc. Thus we relegate the dynamics of the simulation to a fixed set of BG servers each of which is a fixed state-machine.

Our new "by-value" simulation gives an alternative method to accomplish what the BG simulation does, since when there are just k or less clients they do not need k-set consensus, as they can reach it trivially. Thus we can conceive of a proof of [**?**], by taking 2 simulators and let them by-value simulate a code of 100 processors 1-resiliently. Our "by-value" simulation can be extended to this situation without emulating BG-agreement protocols. But, it will be a "mess." It was tried before. Thus, as said, here, the BG comes handy to handles the dynamic nature of the problem. This intermediate BG simulation serves as a "compiler." Hence the engineering marvel.

To show that with k-set consensus we can simulate a k-concurrent execution, the processors simulate as if they are k BG servers. If the number of codes available to be simulated is larger or equal to k then a BG server that can make progress on one of the codes that was already started to be simulated stick to one of these codes. Thus, there will be at most k codes simulated at a time. When some codes terminate, the simulators start a new code available for simulation. Since we have k servers, at any point in time, the number of codes started minus the number of codes terminated is bounded by k, i.e. a k concurrent execution.

But we still face a problem. When the number of codes available for simulation minus the number of codes terminated goes below k, then the live BG simulators cannot proceed if some other simulators crashed in the simulation. Thus the processors associated with these codes cannot obtain an output. Luckily, in our application, when the servers will be clients that pretend to be servers, we notice that the number of processors that simulate the BG simulator, i.e. the processors with pending codes, is also less then k.

Thus, if the number of processors is less than k, say $j < k$, they can obtain j-set consensus rather than k-set consensus and with this stronger consensus, they can simulate the first j BG simulators [12]. The simulators which are not simulated now are just a burden, possibly blocking some codes. Thus, we need a way to check-point and restart the BG simulation without these dead simulator blocking codes. This check-point and restart is an addition to the BG simulation recently introduced [3]. With it, the problem is resolved and each processor can obtain an output provided a processor that obtains an output for its code, departs, and does not interfere with the simulation any more.

The paper is organized as follows. Definitions and background material appear in section 2 and 3. Section 2 is model and definitions, while Section 3 recalls the BG and Extended-BG simulation for the paper to be self-contained. Section 4 is the heart of the paper and outlines the new simulation-by-value technique, by which n processors with k-set consensus simulate k Extended-BG simulators. At the end of the section we explain how all the pieces fit together, which is the magic of the paper. We then outline ramifications, and conclude.

2 Model

2.1 Shared Memory

We consider a Single-Writer Multi-Reader (SWMR) shared memory (SM). At times we will assume Atomic-Snapshot [1]. We are concerned with one-shot tasks, which are defined just an input output relationship [10]. A *run* in the Atomic-Snapshot model is a sequence of processors ids, where processors alternate between writing and snapshoting. The first appearance of a processor is interpreted as a write, the second a read, the third a write, etc. A snapshot in the sequence returns for each processor the latest value written by the processor before the position of the snapshot in the sequence. At any point in a run, the set of participating processors are the processors that appear at least once before the point.

A *model* is a set of runs. The model is a contract between the programmer and an adversary that delineates in what kind of sequences the programmer's program must produce a (correct) output. In sequences outside the model the programmer has no obligation for an output (notice that w.l.o.g "bad" output can always be prevented, by snapshoting tentative outputs and checking compliance with possible valid outputs). A programmer violates her obligation if in a run of the model there is a processor that appears infinitely often without output. A task is solvable in a model, if the programmer can produce a code, one for each processor, such that she will not violate her he obligation in runs of the model. The wait-free model is the set of all runs.

Some Models

The k-obstruction-freedom model is the set of all infinite runs in which at most k processors appear infinitely often. The $k - concurrent$ model is the set of runs such that if $m > k$ processors appear in a prefix then at least $m - k$ of them will not appear any more. Notice that k-obstruction-freedom is “eventually k-concurrent.” Each k-concurrent run is necessarily k-obstruction-freedon run. Thus, this paper shows, that k-concurrent and k-obstruction-free are equivalent in the sense of implementing each other wait-free. Does any liveness property have the equivalent safety property? We conjecture that this is the case. The k-active resiliency model is the set of runs such that if $m > k$ processors participate then at least $m - (k - 1)$ of them will appear infinitely often.

k-set Consensus and k Independent Consensuses Tasks

The k-set consensus task is for each processor to output an id of a participating processor such that the cardinality of the set of outputs is no more than k. The task k-set consensus is generalization of consensus, or more precisely here - election, which is the case when $k = 1$. It is known that like consensus [7], for $k < n$ the k-set consensus task is not wait-free solvable [2, 10, 11].

The k independent consensuses task, is a task in which a processor output an index j, $1 \leq j \leq k$ and and id of a participating processor. All processors that output the same index, output the same id.

Models Extended with Tasks

A model extended with a task T is the regular SM model only that now an appearance of a processor can be an invocation of an instance of T and the next appearance of the processor is the return of the task. Invocations and returns of the same instance satisfy the task. The programmer is allowed only finitely, yet unbounded number, of invocation of a task. The notion of solvability extend naturally.

It was shown in [12] that the two tasks above are equivalent. That's it the wait-free model extended with k-set consensus solves the k independent consensuses task, and vice versa. Moreover, it was shown that for participating set of size $less < k$, then with k-set consensus, there is a solution to the k independent consensuses task in which processors output an index which is less or equal to $less$.

The Commit-Adopt (CA) Task

The commit-adopt task is as follows:

1. If all participating processors propose the same value then they all commit to that value,

2. if a processor commits to a value then all other commit or adopt that value.

The CA task can be solved in SWMR SM wait-free [15].

3 The BG Simulation: How Few Simulate Many

The BG-Simulation was introduced in [2], and analyzed in detail in [5]. The Extended-BG simulation was recently proposed in [3] and attend to the case when the number of simulators s is dynamic. The simulation allows s simulators in SM wait-free to execute codes on behalf of clients. The simulation guarantees the requirement that as long as at least one simulator is alive then eventually the number of clients missing an output will be less than s.

3.1 A Brief Reminder of BG

How does the BG simulation guarantee the requirement?

Clients post their input in SM for the simulators to see. The BG simulation allows all simulators to simulate any client. When more than one simulator simulates the same client code we may have a coordination difficulty. Some simulators may want to simulate the same *read* command in a code concurrently. While they "read" on behalf of *read*, other simulators may asynchronously simulate *writes* of other codes. Thus the simulators may have different proposals for the simulated *read*. They cannot resolve this disagreement wait-free [7].

The BG-simulation addresses this using a non-wait-free protocol as follows. A simulator s_j posts its id in SM. It then takes a snapshot of all posted ids to obtain a set $ids(s_j)$ whose cardinality is the number of ids in the snapshot. Recall that all snapshots of the same cardinality are identical. Simulator s_j then posts $ids(s_j)$ in SM. It now "waits" for all simulators whose ids appear in $ids(s_j)$, to post their respective snapshots. It outputs the lowest id that appears in the smallest $ids(s_i)$ snapshot, i.e for all s_l , $|ids(s_i)| \leq |ids(s_l)|,\ s_i, s_l \in ids(s_j)$, posted by some simulator $s_i \in ids(s_j)$.

The algorithm above results in blocking election. To solve blocking consensus with it processors just post their consensus proposals in SM before starting the BG election.

The protocol has the following properties:

1. If all simulators that appear in some snapshot reach the "wait-statement" then all simulators can output,
2. all simulators output the same proposal,
3. if a simulator at the "wait-statement" has no output then another simulator has started the protocol and is not yet at the "wait-statement," and,
4. with a minor change to the algorithm, if all simulators start with the same proposal and one of them posted a snapshot then they (wait-free) output that proposal.

What is the drawback of the BG simulation that later necessitated the Extended-BG simulation [3]? We said that at most $s-1$ client's code will not obtain an output. Suppose some "selfish" simulators which participated in the simulation depart without "cleaning-up after themselves," i.e. some of them are in the middle of a BG agreement blocking its resolution. Even though they have departed they will then still count to increase the number of clients that will not obtain an output. We would like a way to "abort" such a BG agreement and restart it without the simulators that left. The extended BG endows us with the facility the accomplish that as we briefly discuss below.

3.2 A Brief "Reminder" of Extended-BG

The Extended-BG [3] protocol uses a new agreement protocol that is a sequence of CA-appended BG-agreement protocols. A CA-appended BG-agreement protocol is a BG-agreement protocol followed by CA (commit-adopt) [15].

In the Extended-BG, the simulation scheme now becomes as follows: A simulator does the standard BG-agreement simulation trying to evaluate "reads" of simulated code in an order determined by the application. If the BG-agreement is resolved, a simulator outputs it only if it is successful in committing the proposal through the subsequent CA. If it commits a value then this value is its output of the "read" in the Extended-BG-agreement, else, it continues to the subsequent CA-appended BG-agreement, either with its adopted value, or with its previous proposal, if it did not adopt one.

What if it is blocked on a BG-agreement? As usual it tries other agreements in order determined by the application. When it is blocked on all codes either because a code has terminated or the agreement is blocked and there is no "permission" to start a code that has not started, it then proposes its last proposal to the appropriate version of CA succeeding the unresolved BG-agreement. It does this to all unresolved agreements. If it commits to a value the CA-appended BG-agreement is resolved, if it does not commit, it continues to the next copy of an agreement with the adopted proposal if it did adopt one, or its previous proposal, otherwise. A termination of an extended-BG-agreement is always only by committing in one of the CAs in it.

The advantage of the Extended-BG protocol is that if simulator s_j really died (for various reasons as for instance the simulator is run by a client that departed) in the middle of a BG-agreement section, by proceeding to the CA succeeding that BG agreement, the dead simulator is essentially kicked out of the simulation (at least until another client will proceed to run it).

4 Simulation-by-Value: How Many Simulate Few

This section is the crux of the paper. First we address the question of generalizing the state-machine approach, available when consensus is provided [18], to the case when k-set consensus is available. When consensus is available each processor proposes a command, one is chosen, to be the first, and advertised to all. Then commands are submitted again, and one is chosen as second, etc. All processor agree on a growing prefix of such a sequence.

We show a generalization in which we have k parallel independent consensuses. In each round processors submit a vector of commands, one for each consensus. A processor returns with the first command for some consensus. When commands are to resubmitted again, they have to be submitted to a "fresh copy" of a consensus. But for $k-1$ copies a processor may not know if a first command was decided by some processors or not. The second command to be submitted to such a consensus may need to be a function of the first command linearized, some processors do not know the first command! Also, how would it know if the next return is a second command or a first, and how does it know it did not create a conflict?

Solving this dilemma in a manner that allows progress of at least one command sequence for some consensus it what is outlined next. We then take the algorithm verbatim and use it to implement k read-write threads which are free of "wait" statements, with a guarantee of progress of at least one thread. Finally, we consider the processors to be the ones to solve a task, with a code T that will progress k-concurrently. The k threads are those of Extended BG-simulators which execute the code of T. Of course, they cannot execute a thread of T, without the corresponding processor participating. The corresponding processor departs when the simulation provides an output for its code.

4.1 The Round-By-Round Structure

State-Machine Implementation from k-set Consensus

We have k-set consensus which is equivalent to k independent consensuses [12]. We have k state-machines we want to implement with progress on at least one. We will employ a sequence each element of which is k parallel independent consensuses.

We associate a commit-adopt protocol $CA_{i,j}$ with each consensus i, $1 \leq i \leq k$ in sequence element $j, j = 1, 2, ...$, for all i and j.

Processor move through the sequence-elements in order - hence forth consequently referred to as "round." In each round j each processor participates in all $CA_{i,j}$ for all i in some particular "personalized" order.

When done with round $j-1$, inductively, processor q has for each state-machine stm,a sequence of commands where the last command $command_{stm,w}$ for some m, is either committed ot adopted. In round j, it posts in SM the committed sequence for each independent state machine and it proposes a command (may be different for different state-machines) to be passed for each state-machine. If the last command $command_{stm,w}$ for i is committed, then it proposes a new command bases of the state of the state-machine. If the last command $command_{stm,w}$ for stm is just adopted, then it proposes it as the next command to be passed at stm. It applies these commands to all the independent consensuses at round j and return with some command $command_{i,m}$ for some i and m.

Let $command(q)_{j,stm,m(stm)}$, $i = 1, ..., k$ be the vector of commands q proposes in round j. Next, q goes through $CA_{stm,j}$ to commit a command. It starts with $CA_{i,j}$ where it tries to commit it return $command_{i,m}$. It then marches through $CA_{l,j}$ for all l for which it command is an adopted command from previous round. It tries to commit it. Then, for state-machine z for which it does not have an adopted command or is not i, it proposes a special value $NotCommand$ to $CA_{z,j}$. Not command is not committed or adopted, but just serves to disable hopefully other processors from passing commands for this state-machine. If at any $CA_{stm,j}$ observes a higher command sequence $command(p)_{j,stm,l}$ with l larger then the sequence number of the command it proposed $m(stm)$, it considers $command(q)_{j,i,m(i)}$ to have been committed. Else, it either commit, adopt, or return nothing from $CA_{i,j}$ in which case it considers the last committed value for i to be $command(q)_{j,i,m(i)-1}$. it then proceed in any order to propose adopted commands $command(q)_{j,z,m(i)}$ to $CA_{z,j,z \neq i}$. This establishes what it returns for all the state-machines.

The last question to answer is why try to pass $NotCommand$ rather than the originally proposed command. As we will see, when s, the number of processors that started but has not departed changes dynamically and become $s < k$, we want the $k - s$ higher index state-machine not to progress. The s processors will return from the first s consensuses, as this is the property of the algorithm in [12]. Thus we will get progress even though, the last $s - k$ state-machine will not progress. When we make the state-machine into simulators code, these simulators will appear to have crashed. Crashed simulators will be automatically kicked out by the Extended-BG mechanism.

Correctness

We first argue the safety of the algorithm sketched above. This follows since the behavior of processors with respect to a single state-machine out of the k, is just of processors marching through a sequence of CA's, submitting an adopted value to the next CA. In the regular use of CA [15] committed processors halt. Here, they continue, but the adopting processors will commit the adopted value either because they will succeed in the CA, or because they will observe a participating committed processor.

To see progress, notice that once a command is committed by all the next commands proposed will be of a higher sequence number. At each round, if we zoom on the state-machine in which a processor ended the CA, in the most recent round, first, before any processor ended another CA in that round, then for this state-machine a new processor commits a command it did not commit before.

From State-Machines to Read-Write Codes

Now, instead of k independent state-machines we have k read-write codes. We view a decision on the m'th *read* of code i as passing the m'th command for code i. The command to pass by each processor is now determined on-line. A committed *read* implies the value of the succeeding *write*. Thus, the *read* value a processor proposes is determined by the set of *read*s (commands) already committed by it. Since a *read* committed first by any processor at round j is committed by all at round $j+1$ a write can be perceived to happen atomically some time in round j. Some processors observed it while others read the previous value. In the next round they all read the new value or a later value. Thus we implement read-write atomic memory although our *read*'s implementation is not necessarily that of an atomic snapshot.

4.2 Executing a task on n processors k-concurrently using k-set consensus

Let $protocol(n)$ be read-write code with n threads for processors $p_1, .., p_n$ without a "wait" statement. Thread i is permitted to be executed only if p_i participates. When thread i terminates, p_i departs when it observes the termination of its code. The processors have k-set consensus available and $protocol(n)$ is to be executed, subject to executing a thread only when "permission arrives," such that the number of threads that started but did not terminate is bounded by k.

We consider k Extended-BG simulators that simulate a run of $protocol(n)$. The "environment" (arriving processors) intermittently may signal in SM additional threads that may be executed by the simulators. Magically, when the number of threads that can be simulated minus he number of threads that terminated, drops below k, say to the number *less*, then $k - less$ simulators stop taking steps. While waiting on a specific BG-agreements a simulator tries to move the program counter of treads that already started (Depth-First). Only when blocked on all agreements, it will start simulating a new thread which is permitted to be simulated by the environment. When blocked and no new thread to simulate is available it will try to "kick-out" other simulators through the mechanism provided by the Extend-BG simulation.

It is easy to see that with the "magic," all threads signaled by the environment eventually will terminate. The difference in the use of Extended-BG here to Extended-BG in [3] is that simulators here go "Depth First" wile simulators there proceed "Breadth First."

Obviously, the code of the simulation for the k Extended-BG simulators $protocol(k)$ is k treads of read-write codes, and these read-write code can be executed by processors using k-set consensus as explained in the preceding subsection.

Now, the "environment" is the processors signaling in SM that they have arrived and their threads are available for execution. The processors execute the code of the k Extended-BG simulators executing $protocol(k)$. The "magic" happens when the number of processors that arrived and not departed drops to *less* then only the first *less* BG simulators will be active and the rest will not move.

Obviously the BG simulators executing in Depth First execute *protocol*(n) k-concurrently.

Thus, the set of task solvable k-concurrently is subset of the set of tasks solvable with k-set consensus.

4.3 The Universality of the Simulation By-Value

Let T be a task on n processors that is solvable with k-set consensus. In this subsection we show that there exist a read-write code to be executed by our simulation that will solve T.

All we know is that there exist a read-write code that calls on k-set set consensus and solves T.

Let $protocol(n, k-set)$ be a read-write code that calls on k-set consensus. The code we will use in the simulation is the read-write code of $protocol(n, k-set)$, where we replace each call for k-set consensus with some read-write instructions.

Consider an execution of $protocol(n, k-set)$ k-concurrently without calling on k-set consensus. For each copy of k-set consensus invoked in $protocol(n, k-set)$ we keep a MWMR variable initialized to non-value null. An executing processor, upon encountering the command to apply to copy l of (n, k)-set consensus, reads the associated MWMR variable. If it sees a value there, it returns the value. Otherwise, it writes its own input in the MWMR variable and returns its own input.

Since, at most k processors may observe the MWMR variable with a null value, it is easy to see that the number of values returned will be at most k.

Since our simulation simulates k concurrent execution it establishes the reverse: Every task solvable with k-set consensus is solvable k-concurrently.

5 Ramifications

5.1 k-obstruction-free $=$ k-set consensus

To see the equivalence with k-obstruction-free, notice that if a task is solvable k-obstruction-free it is obviously solvable k-concurrently as a k-concurrent run is k-obstruction-free. On the other hand, to see that if a task is solvable k-concurrently then it is solvable k-obstruction-free, notice that k-set consensus is a task which is solvable k-obstruction-free; just do repeated k-converge [17] CA for k-set consensus until committing. When concurrency is down to less or equal k, the k-converge CA will commit. Now k-set is equivalent to k-concurrently.

5.2 $k-1$-active resiliency $=$ k-concurrency

Recall Mutual-Exclusion (one-shot). It is solvable under the the assumption that processors which signal the desire to enter the critical-section will not die. It does not consider a non-contending processor as dead. Similarly with l-exclusion. At most $l-1$ contending processors may die, but processor which do not participate are not considered dead. In the same vain we may ask whether a task T is solvable under the assumption of $k-1$-*active resiliency*, that it at most $k-1$ processors that invoked the task may die, but processors cannot wait on processors that did not invoke the task.

It is easy to see through our simulation that the tasks which are solvable under $k-1$-active resiliency assumption are exactly the ones solvable k-concurrently: Under $k-1$-active resiliency one can solve k-set consensus. On the other hand, a k-concurrent execution is $k-1$-active resilient.

The substitution of k-set consensus with k-concurrency also make it simple to obtain explicit results for what is solvable with k-set consensus. It is easy to see that adaptive renaming of n processors [14] k-concurrently requires exactly $n+(k-1)$ slots, previously [13] this results for k-set consensus required some ingenuity.

6 Concluding Remarks

Our simulation-by-value promotes the thesis that all distributed computing models might just be a restricted form of the wait-free model. More specifically, our simulation shows that models that use k-set consensus or models that assume concurrency limited to k, can be though of as a wait-free model where, magically, disagreement is at most k.

Our simulation-by-value also puts some sense behind a question asked invariably by students taking a distributed algorithms class: "Do all programs, one for each processor, have to be the same?" This question should not be dismissed anymore as making no sense, even if the task is not anonymous. Our simulation shows that the id of a processor is important just to allow its code to be executed by all, and finally to allow it to depart the simulation once its code has terminated. In between, all processors "run the same code."

Acknowledgment Yehuda Afek helped with the writing of earlier drafts of this paper. The germination of this paper as well as [12] is from the first author's work with Petr Kuznetsov, where the 2-fronts agreement was realized - 2 processors read-write wait-free can resolve at least one of two independent simultaneous consensus. The mileage derived from such a simple realization is astonishing!

References

[1] Afek Y., H. Attiya, Dolev D., Gafni E., Merrit M. and Shavit N., Atomic Snapshots of Shared Memory. PODC 1990: pp. 1–13.

[2] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. STOC 1993: pp. 91-100.

[3] Gafni E., Extended BG. STOC 2009.

[4] Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-Free Computations. PDC 1997: pp. 189–198.

[5] Borowsky E., Gafni E., Lynch N. and Rajsbaum S., The BG Distributed Simulation Algorithm. *Distributed Computing,* 14(3):127–146, 2001.

[6] Chaudhuri S., More *Choices* Allow More *Faults:* Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation,* 105:132-158, 1993.

[7] Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985.

[8] Eli Gafni: Read-Write Reductions. ICDCN 2006: 349-354

[9] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on programming Languages and Systems*, 11(1):124-149, 1991.

[10] Herlihy M.P. and Shavit N., The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999.

[11] Saks, M. and Zaharoglou, F., Wait-Free k-Set Agreement is Impossible: The Topology of Public Knowledge. *SIAM Journal on Computing*, 29(5):1449-1483, 2000.

[12] Yehuda Afek, Eli Gafni, Sergio Rajsbaum, Michel Raynal, Corentin Travers, Simultaneous Consensus Tasks: A Tighter Characterization of Set-Consensus ICDCN06 2006: pp. 331–34.

[13] Eli Gafni, Renaming with k-Set-Consensus: An Optimal Algorithm into n + k - 1 Slots. OPODIS 2006: 36-44

[14] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, David Peleg, Rdiger Reischuk: Renaming in an Asynchronous Environment J. ACM 37(3): 524-548 (1990)

[15] Eli Gafni, Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony. PODC 1998: 143-152

[16] Maurice Herlihy, Viktor Luchangco and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. ICDCS 2003, p.522.

[17] Jiong Yang, Gil Neiger and Eli Gafni: Structured Derivations of Consensus Algorithms for Failure Detectors, PODC 1998: pp 297-306

[18] Fred B. Schneider: Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, ACM Computing Surveys,1990,volume = 22,pages = 299–319

[19] Eli Gafni, Achour Mostfaoui, Michel Raynal, Corentin Travers: From adaptive renaming to set agreement. Theor. Comput. Sci. 410(14): 1328-1335 (2009)