A Pedagogical Presentation of the BG-Simulation

Daniel Lister

January 28, 2013

Abstract

The BG-simulation is a well known algorithm that allows the simulation of an algorithm for m processes by $n < m$ processes in an asynchronous environment while preserving the number of crash failures. While originally presented using snapshot objects, we focus our presentation on registers. The goal of this thesis is to simplify the understanding of the BG-simulation and its proof of correctness. We do so by first presenting a proof in the simplified model where all processes are correct and are able to access consensus objects. We then remove these assumptions and correspondingly modify proof.

Acknowledgements

I would like to thank my advisers Sam Toueg and Vassos Hadzilacos for their dedication, encouragement, feedback, and suggestions. Without them this thesis would not exist. Thank you to my friend Laura Sutula and my grandfather Jack Canfield for their help editing. A special thanks goes to my parents for their unending love and support.

Contents

1 Introduction

In this work, we are concerned with asynchronous systems where processes are subject to crash failures and communicate using shared registers or snapshot objects. Intuitively a snapshot object is a shared memory object that allows for the simultaneous reading of multiple registers.

1.1 The BG-Simulation and the Goals of this Thesis

The BG-simulation was originally introduced in 1993 by Elizabeth Borowsky and Eli Gafni in their paper "Generalized FLP Impossibility Result for t-resilient Asynchronous Computations" [3]. They used it to prove that, in an asynchronous system, there is no algorithm for n processes to solve the k-set agreement problem (where $k < n$) that tolerates up to k process crashes.[1] This was proven by (a) first showing that there is no algorithm for $k+1$ processes to solve the k-set agreement problem that tolerates up to k process crashes, and (b) then using a simulation to reduce the case of $n > k$ processes where at most k of which can crash, to the case of $k+1$ processes where at most k of which can crash.

The BG-simulation was later formalized and proved by Borowsky, Gafni, Lynch, and Rajsbaum in their paper "The BG distributed simulation algorithm" [4]. They also showed how the BG-simulation could be generalized to problems other than k-set agreement by introducing the notion of fault-tolerant reducibility between decision problems. Gafni later developed the extended BG-simulation which could then be applied to a wider range of problems[6].

Our goal here is to simplify the understanding and the proof of the BG-simulation. To do so we present this proof incrementally, in two stages as we now explain. The BG-simulation uses safe agreement objects — an ingenious implementation of consensus that may block some non-faulty processes if some process crashes in an "unsafe" part of this implementation. In the first stage, we simplify the BG algorithm and its proof by replacing the safe agreement objects that are used in the actual BG-simulation, with simple consensus objects where no process can be blocked. This allows us to understand and prove the safety components of the BG-simulation in a much simpler setting. In the second stage, we restore the safe agreement objects in place of consensus objects, and use their properties to prove the

[1]In the k-set agreement problem, each process has an input value and must output one of the input values so that there are at most k different output values. For $k = 1$, this problem coincides with the well-known consensus problem.

liveness component of the BG-simulation; the proof of the safety components remains unchanged from the first stage. We think that this modular approach to proving the correctness of the BG-simulation is pedagogically effective. Additionally, to the best of our knowledge, this approach is novel.

Another contribution of this thesis is the presentation of the BG-simulation in a very compact, and easy to understand pseudo-code that is informal yet precise. This pseudo-code is derived from the presentation of the BG-simulation in [9].

Finally, in [3], the BG-simulation was presented using snapshot objects only. In [4] the focus is also on using snapshot objects, although the authors also discuss how to extend the BG-simulation and its proof for a system with registers instead of snapshot objects. In this work we take the opposite approach. We first present and prove the BG-simulation for a system with plain registers since this is a simpler, more basic model. We then modify the BG-simulation to work in a system with snapshot objects, and prove its correctness in that system. It turns out that the presentation and the proof of the register version is essentially the same as, and it is just as simple as, the snapshot version. As such, the BG-simulation can be presented and understood without prior knowledge of snapshots and their complex implementations from registers.

1.2 Introduction to Simulations and the BG-Simulation

A simulation is when one or more "real" processes execute the steps of an algorithm designed for a possibly different number of "virtual" processes, and use the results of that execution to solve a problem among themselves. The states of the virtual processes are stored and manipulated in the memory of the real processes. If the simulated algorithm calls for shared objects such as registers, then the real processes must manage those objects for the virtual processes. The inputs to the simulated algorithm are also provided by the real processes, and are usually derived from their own inputs.

The BG-simulation allows n real processes to simulate the execution of an algorithm $\mathcal{A}$ for m virtual processes, each of which owns one SWMR register. The BG-simulation preserves the number of crashes. More precisely, if f real processes crash during this simulation, then at most f of the virtual processes crash in the simulated execution of $\mathcal{A}$. The real processes provide the inputs to the simulated algorithm based on their own inputs, and they use the outputs of the virtual processes to derive their own outputs. In this way, n processes, t of which may crash, can solve a task for n processes by simulating the execution of an algorithm that tolerates t crashes and solves a task for m processes.

For example, suppose we are given a 5-resilient algorithm $\mathcal{A}$ (that is an algorithm that tolerates up to 5 process crashes) that uses registers and solves k-set agreement among 100 processes. We want a 5-resilient algorithm that uses registers and solves k-set agreement among 6 processes. Note that the latter task appears to be harder, since an algorithm that solves it can rely on at most one process to be correct, while $\mathcal{A}$ can rely on up to 95 processes to be correct. To solve the seemingly harder task, the 6 real processes use the BG-simulation to simulate an execution of $\mathcal{A}$ on 100 virtual processes. The input to each of the 100 virtual processes will be chosen arbitrarily from among the inputs to the real processes. The real processes will use the output of the virtual processes as their own output. More generally, a t-resilient k-set agreement algorithm for n processes that uses registers can be created by simulating a t-resilient k-set agreement algorithm for m processes that uses registers.

1.3 Organization of this Thesis

In section 2 we present our model. In section 3 we give a simplified version of the BG-simulation for k-set agreement, which uses registers and consensus objects, and prove its correctness. In section 4 we show how to modify the simplified version so that it uses only registers and prove the correctness of the BG-simulation, similar to the correctness proof of the simplified version. In section 5 we give a version of the BG-simulation that uses snapshot objects and prove its correctness. In section 6 we first give a brief overview of the extended BG-simulation and then describe an open problem for future work.

2 Model

In this work we are concerned with asynchronous distributed systems in which processes communicate by reading and writing shared variables and are subject to crash failures. A system consists a set of processes $1, 2, \ldots, n$ and a set of single-writer/multi-reader (SWMR) shared registers.

Each process carries out its computation by executing one of three possible steps: read a register, write to a register, or output a value. The model captures local computation as changes in a process's state.

We assume that each process has a single SWMR register. This assumption can be made without loss of generality because it is possible to modify an algorithm that uses multiple registers per process to use only a single register per process. This is accomplished by representing multiple registers as fields of a single register.

An *algorithm* describes the behaviour of each process. More precisely, we specify an algorithm $\mathcal{A}$ by giving the following for each process $1 \leq i \leq n$ in the system:

(1) the initial value of i's register,

(2) the set S_i of i's states,

(3) the set I_i of i's inputs, and

(4) the following three functions:

  (i) $\textsc{init\_state}_i(x)$, where $x \in I_i$, returns the initial state of i when i's input is x,

  (ii) $\textsc{nextop}_i(S)$, where $S \in S_i$, returns the next operation of i. More precisely, it returns one of the following three tuples: $(\textsc{read}, i')$ where i' is a process, in which case the next step of i is to read the register of i'; $(\textsc{write}, v)$ where v is a value, in which case the next step of i is to write the value v in its own register; or $(\textsc{output}, y)$ where y is an output value, in which case the next step of i is to output y.

  (iii) $\textsc{trans}_i(S, \mathit{ret})$, where $S \in S_i$ and *ret* is a return value of the operation $\textsc{nextop}_i(S)$, returns the next state S' of i after an operation that returns *ret*.

Formally we define a *step* of an algorithm $\mathcal{A}$ as a tuple (i, S, ret) where i is the process that takes the step, S is the state of i before the step is taken, and *ret* is the return value of the step. We require the following:

(1) $1 \leq i \leq n$

(2) $S \in S_i$

(3) If $\textsc{nextop}_i(S)$ is $(\textsc{write}, v)$ or $(\textsc{output}, y)$, then $\mathit{ret} = \textsc{done}$;
if $\textsc{nextop}_i(S) = (\textsc{read}, i)$, then *ret* is a possible value of the register of process i'.

Consider any step (i, S, ret) of algorithm $\mathcal{A}$:

(1) If $\textsc{nextop}_i(S) = (\textsc{read}, i')$, then (i, S, ret) is a read step of i that reads *ret* from the register of i'.

(2) If $\textsc{nextop}_i(S) = (\textsc{write}, v)$, then (i, S, ret) is a write step of i that writes v into its own register.

(3) If $\textsc{nextop}_i(S) = (\textsc{output}, y)$, then (i, S, ret) is an output step of i that outputs y.

A *history* H of $\mathcal{A}$ models an execution of $\mathcal{A}$. Formally it is a sequence of steps of $\mathcal{A}$ that satisfies the following properties:

Local consistency For every process i, if (i, S, ret) and $(i, S', -)$ are consecutive steps of i in H, then $S' = \textsc{trans}_i(S, \mathit{ret})$.

Read consistency if (i, S, ret) is a read step that reads *ret* from the register of process i', then *ret* is the value written by the last write step of process i' that precedes (i, S, ret) in H, if such a step exists, and *ret* is the initial value of the register of i' otherwise.

Valid initialization For each process i, if (i, S, ret) is the first step of process i in H, then $S = \textsc{init\_state}_i(x)$ for some $x \in I_i$.

Process i is *correct* in history H if it takes an infinite number of steps in H (i.e., H has an infinite number of steps of the form $(i, -, -)$); it is *faulty* otherwise.

Note that the above definitions reflect the fact that we are concerned with asynchronous systems prone to crash failures. This is because in a history H, there is no bound on the number of steps of other processes that can appear between successive steps of a process (asynchrony); and at any point a process can permanently stop executing steps (crash failures).

A *history H of $\mathcal{A}$ with input $x = (x_1, x_2, \ldots, x_n)$* is a history of $\mathcal{A}$ such that, for each process i, if (i, S, ret) is the first step of process i in H, then $S = \textsc{init\_state}_i(x_i)$.

2.1 One-shot tasks

A (one-shot) task T is a relation between inputs and outputs. More precisely, a task for n processes is a triple $(\mathcal{I}, \mathcal{O}, \Delta)$, where $\mathcal{I}$ is a set of input vectors of size n (one value for each process), $\mathcal{O}$ is a set of output vectors of size n (one value for each process), and $\Delta \subseteq \mathcal{I} \times \mathcal{O}$ is a relation such that for every $x \in \mathcal{I}$ there is at least one $y \in \mathcal{O}$ such that $(x, y) \in \Delta$. Intuitively, for every $x \in \mathcal{I}$ and every $y \in \mathcal{O}$, $(x, y) \in \Delta$ iff when the input is x (i.e., for every process $1 \leq i \leq n$, the input of i is $x[i]$) the output y (i.e., for every process $1 \leq i \leq n$, the output of i is $y[i]$) is acceptable for task T.

An algorithm $\mathcal{A}$ is *a t-resilient algorithm for the task* $T = (\mathcal{I}, \mathcal{O}, \Delta)$ if, for every input $x \in \mathcal{I}$, every infinite history H of $\mathcal{A}$ with input x such that at most t processes are faulty in H satisfies the following:

(1) Every process outputs a value at most once.

(2) Every correct process i in H outputs a value.

(3) There is a vector $y \in \mathcal{O}$ such that

 (i) $(x, y) \in \Delta$, and
 (ii) every process i that outputs a value in H, outputs $y[i]$.

We will now define some tasks that are of interest to us in this work.

The *binary consensus task* $T_C = (\mathcal{I}, \mathcal{O}, \Delta)$ for n processes is defined as follows: $\mathcal{I} = \{0, 1\}^n$, $\mathcal{O} = \{0^n, 1^n\}$, and $(x, y) \in \Delta$ if and only if:

(a) if $y = 0^n$ then there is an i such that, $x[i] = 0$ and

(b) if $y = 1^n$ then there is an i such that, $x[i] = 1$.

From the definition of solving a task it is easy to see that a f-resilient algorithm $\mathcal{A}$ solves the binary consensus task T_C for n processes, if every infinite history H of $\mathcal{A}$ with up to f faulty processes satisfies the following:

Termination Every process outputs a value at most once, and every correct process outputs a value.

Agreement No two processes output different values.

Validity If a process outputs a value v, then $v \in 0, 1$ and is the input to some process.

The k-set agreement task $T_S = (\mathcal{I}, \mathcal{O}, \Delta)$ for n processes is defined as follows: $\mathcal{I} = \mathbb{N}^n$, $\mathcal{O} = \{v \in \mathbb{N}^n : |\{v[i] : 1 \leq i \leq n\}| \leq k\}$ (i.e., the set of vectors in $\mathbb{N}^n$ with at most k distinct elements), and $(x, y) \in \Delta$ if and only if for each $1 \leq i \leq n$ there exists some i', such that $y[i] = x[i']$. Note that consensus is the special case of k-set agreement when $k = 1$. A f-resilient algorithm $\mathcal{A}$ solves the k-set agreement task if every infinite history H of $\mathcal{A}$ with up to f failures satisfies the following:

Termination Every process outputs a value at most once, and every correct process outputs a value.

Agreement At most k distinct values are output.

Validity If a process outputs a value v, then v is the input to some process.

2.2 Extending the Model

The above model is for a system where each process owns a single SWMR register that it writes and all processes can read. In the BG-simulation it is convenient to assume that the simulated algorithm has this property: In this way, each real process only needs to keep track of one register for each virtual process it simulates. However, when presenting the BG-simulation itself (i.e., the algorithm that the real processes follow to effect the simulation of the virtual processes), it is convenient for each process to own several SWMR registers. We therefore modify the model to allow for multiple registers per process. To do so we need only modify the definition of the function $\textsc{nextop}_i()$ to specify *which* register is read or written. Thus, $\textsc{nextop}_i(S)$ now returns one of the following three tuples:

(1) $(\textsc{read}, i', R')$ where i' is a process and R' is a register owned by i',

(2) $(\textsc{write}, v, R)$ where v is a value and R is a register owned by process i,

(3) $(\textsc{output}, y)$ where y is an output value.

So, in a step (i, S, ret) of algorithm $\mathcal{A}$:

(1) If $\textsc{nextop}_i(S) = (\textsc{read}, i', R')$, then (i, S, ret) is a read step of i that reads *ret* from register R' belonging to i'.

(2) If $\textsc{nextop}_i(S) = (\textsc{write}, v, R)$, then (i, S, ret) is a write step of i that writes v into i's register R.

(3) If $\textsc{nextop}_i(S) = (\textsc{output}, y)$, then (i, S, ret) is an output step of i that outputs y.

The other definitions, including the history of an algorithm $\mathcal{A}$, remain the same.

3 The Simplified BG-Simulation

We will begin by presenting a simplified version of the BG-simulation. This version is based on the strong simplifying assumptions stated below, and its purpose is purely pedagogical. It can be used to explain the key ideas behind the BG-simulation in a simple setting. We will see how to remove the simplifying assumptions in section 4.

The assumptions we will make for the simplified BG-simulation are: (a) all processes are correct, and (b) the real processes have access to consensus objects. Intuitively, a consensus object allows the real processes to propose values and reach agreement on one of the proposed values. More precisely, a consensus object CO has one operation, DECIDE(v), that takes a single value v as an argument and returns a value. There are three properties that the consensus object guarantees.

Agreement

If two invocations of DECIDE return v_1 and v_2 then $v_1 = v_2$.

Validity

If some invocation of DECIDE returns v then some process invoked DECIDE(v).

Termination

All processes return from any invocation of DECIDE.

In Section 4 we will replace the consensus objects used by the real processes with safe agreement objects, which can be implemented using registers.

3.1 Algorithm

The code for this simplified version of the BG-simulation is presented in Algorithm 1 and an explanation of how it functions follows. Each real process simulates all of the m virtual processes independently from the other real processes, using consensus objects only to agree on the outcome of certain operations. Since the system is asynchronous the real processes may disagree on the progress of each of the virtual processes. However, as we will see, they will agree on the outcome of each step.

Each real process runs m independent threads, one for each virtual process (line 3). Each of these threads carries out the simulation for one of the virtual processes. We will now take a closer look at thread j of real process i. It will simulate steps of virtual process j. Each real process attempts to use its own input as the input of virtual process j in the simulated execution. For the real process to agree on the same execution of virtual process j, however, they must use the same input value. To ensure this, the real process use consensus object $CO[j][0]$ to agree on the input for virtual process j. Each real process proposes its own input value to this object and uses the value returned by the object as the input to virtual process j (line 4). Next, on line 5, the state of virtual process j is initialized using the function INIT_STATE$_j$ that is part of the specification of the simulated algorithm $\mathcal{A}$.

Table 1: Example of Shared Memory Used by the BG-Simulation

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
$i = 1$	$(a, 0)$	$(f, 4)$	$(e, 1)$	$(g, 0)$
$i = 2$	$(z, 9)$	$(c, 7)$	$(e, 1)$	$(g, 0)$
$i = 3$	$(f, 3)$	$(c, 7)$	$(d, 5)$	$(g, 0)$

This state is local to process i: each real process creates and maintains a copy of its own.

After initialization, the main loop on line 7 is executed. Iteration k of this loop simulates the k-th step of virtual process j. The simulation of read, write, and output steps each require different actions by the real process, so the simulation of a step is divided into different branches of an if statement depending on the type of step.

When the simulated step of virtual process j is one in which j writes v to its register, the new value v must be shared with the other real processes so that when a subsequent simulated read step involves a virtual process reading j's register, an appropriate value can be chosen as the value read in that step. The array of registers *MEM* contains what every real process believes the shared memory of the simulated system currently contains, along with timestamps indicating the step during which each value was written. In particular, $\mathit{MEM}[i][j].\mathit{val}$ contains the value that real process i believes is in j's register, and $\mathit{MEM}[i][j].\mathit{steps}$ contains the timestamp (step number) at which j wrote $\mathit{MEM}[i][j].\mathit{val}$. See Table 1 for an example of the contents of *MEM*.

When real process i simulates a write operation of virtual process j, it writes the new value and timestamp into $\mathit{MEM}[i][j]$ (line 9).

When the simulated step of virtual process j is one in which j reads the register belonging to virtual process j', the value that was read must be determined. To do this, the real process that is simulating the step of j will first scan $\mathit{MEM}[-][j']$ (lines 12-13) to find out what each real process believes the value of the requested register contains. It will then determine the most up-to-date value by checking the timestamp of each value using the "vector maximum" VMAX function (line 14). This function takes an array of pairs and returns the pair with the largest value in the second component. Because each real process that simulates this read will do so at a different time, the real processes may disagree on what value is read. To ensure that the real processes agree on the value read by step k of virtual process j, the real processes use consensus object $\mathit{CO}[j][k]$. Each real process proposes to $\mathit{CO}[j][k]$ the value it believes the read should return, based on the value selected by the VMAX function from among the pairs it read when it scanned $MEM[-][j']$, and uses the value returned by $CO[j][k]$ as the value read by the k-th step of virtual process j (line 15). In essence, the first real process to simulate step k of process j determines the outcome of that step. Every other real process simply adopts the already determined outcome.

Finally, when real process i simulates an output step of virtual process j, i will output the same value that j does (line 18). In each case, the state of the virtual process j is advanced using the TRANS_j function (lines 9, 15, and 19).

Algorithm 1 The Simplified BG-Simulation for k-set Agreement
Code for process i, where $i = 1..n$

```
Shared Memory
$MEM$, an $n$ by $m$ array of registers. For each $i$,$j$, $MEM[i][j]$
    contains a pair $(val, steps)$:
    $MEM[i][j].val$, initially the initial value of $j$'s register in $\mathcal{A}$
    $MEM[i][j].steps \in \mathbb{N}$, initially 0
    For each $i$ $MEM[i][-]$ can be written to by $i$
$CO$, an $m$ by infinite array of consensus objects
Process Local Variables
$input \in I_i$, initially the input of real process $i$
$state$, $steps$ arrays of size $m$, initially arbitrary
$decided$, a boolean, initially arbitrary
Thread Local Variables
$k', i', j, j', \ell, steps \in \mathbb{N}$, initially arbitrary
$val, v, w, o$, variables, initially arbitrary
$my\_mem$, an array of $n$ variables, initially arbitrary
procedure DECIDE($input$)
    $decided \leftarrow false$
    parallel for $j = 1..m$
        $w \leftarrow CO[j][0].\text{DECIDE}(input)$
        $state[j] \leftarrow \text{INIT\_STATE}_j(w)$
        $steps[j] \leftarrow 1$
        repeat forever
            if $\text{NEXTOP}_j(state[j]) = (\text{WRITE}, v)$ then
                $MEM[i][j].\text{WRITE}\Big((v, steps[j])\Big)$
                $state[j] \leftarrow \text{TRANS}_j(state[j], \text{DONE})$
            else if $\text{NEXTOP}_j(state[j]) = (\text{READ}, j')$ then
                for $i' = 1..n$ do
                    $my\_mem[i'] \leftarrow MEM[i'][j'].\text{READ}()$
                $(val, steps) \leftarrow \text{VMAX}(my\_mem)$
                $(w, k') \leftarrow CO[j][steps[j]].\text{DECIDE}\Big((val, steps)\Big)$
                $state[j] \leftarrow \text{TRANS}_j(state[j], w)$
            else if $\text{NEXTOP}_j(state[j]) = (\text{OUTPUT}, o)$ then
                if $\neg decided$ then output $o$; $decided \leftarrow true$
                $state[j] \leftarrow \text{TRANS}_j(state[j], \text{DONE})$
            $steps[j] \leftarrow steps[j] + 1$
        end
    end
```

3.2 Why it Works

In this section we give an informal overview of the proof that Algorithm 1 solves k-set agreement. The actual proof is given in the next section.

A Modified Version of the Given Algorithm

In the proof of the BG-simulation, we construct a history of a k-set agreement algorithm. The natural choice would be to construct a history of $\mathcal{A}$, since that is the k-set agreement algorithm we are given. However, since the BG-simulation writes a value/timestamp pair whenever the simulated algorithm writes a value, and the timestamp is frequently referenced in the proof, it is more convenient to construct a history of a modified version of $\mathcal{A}$ that writes pairs instead of values.

We create an algorithm $\mathcal{A}'$ that behaves the same as $\mathcal{A}$ except it writes a timestamp along with every value. To do so each process will simply maintain a count of the number of steps it has executed. When writing a value, the modified algorithm will also write the value of the counter. When reading, the modified algorithm will ignore the extra information stored in the register. The code of such a modified algorithm is provided in Algorithm 2.

Algorithm 2 The "Modified Algorithm" $\mathcal{A}'$

Code for process j, where $j = 1..m$

```
    Shared Memory
    $MEM$ an array of $m$ registers, for each $j$, $MEM[j]$ can be
        written to by $j$, and is initialized to $(v_j, 0)$ where $v_j$ is the initial
        values of $j$'s register in $\mathcal{A}$.
 1: procedure DECIDE($input$)
 2:     $state \leftarrow$ INIT_STATE$_j(input)$
 3:     $steps \leftarrow 1$
 4:     repeat forever
 5:         if NEXTOP$_j(state) = ($WRITE$, v)$ then
 6:             $MEM[j]$.WRITE$\Big((v, steps)\Big)$
 7:             $state \leftarrow$ TRANS$_j(state,$ DONE$)$
 8:         else if NEXTOP$_j(state) = ($READ$, j')$ then
 9:             $(w, k') \leftarrow MEM[j']$.READ$()$
10:             $state \leftarrow$ TRANS$_j(state, w)$
11:         else if NEXTOP$_j(state) = ($OUTPUT$, o)$ then
12:             output $o$
13:             $state \leftarrow$ TRANS$_j(state,$ DONE$)$
14:         $steps \leftarrow steps + 1$
15:     end
```

Formally the specification of $\mathcal{A}'$ is obtained form the specification of $\mathcal{A}$ as follows.

- The register belonging to j is initialized to $(v_j, 0)$, where v_j is the initial value of j's register in $\mathcal{A}$.
- The states of $\mathcal{A}'$ are of the form (S, k) where S is a state of $\mathcal{A}$ and k is a counter (that tracks the number of steps executed).
- The inputs of $\mathcal{A}'$ are the same as the inputs of $\mathcal{A}$.
- The functions $\textsc{init\_state}'_j$, $\textsc{nextop}'_j$, and $\textsc{trans}'_j$ are defined below.

Definition 1.

$$\textsc{init\_state}'_j(\mathit{input}) = \Big(\textsc{init\_state}_j(\mathit{input}), 1\Big)$$

$$\textsc{nextop}'_j\Big((S,k)\Big) = \begin{cases} \Big(\textsc{write}, (v,k)\Big), & \text{if } \textsc{nextop}_j(S) = (\textsc{write}, v) \\ \textsc{nextop}_j(S), & \text{if } \mathit{otherwise} \end{cases}$$

$$\textsc{trans}'_j\Big((S,k), \mathit{ret}\Big) = \begin{cases} \Big(\textsc{trans}_j(S,w), k+1\Big), & \text{if } \mathit{ret} = (w, -) \\ \Big(\textsc{trans}_j(S,\mathit{ret}), k+1\Big). & \text{if } \mathit{ret} = \textsc{done} \end{cases}$$

Observation 2. *If $\mathcal{A}$ is an f-resilient algorithm for a task T, then the modified algorithm $\mathcal{A}'$ is also an f-resilient algorithm for task T.*

Informal Description of the Proof

The basic idea of the proof is to show that, given any history of Algorithm 1, we can construct a history of the simulated algorithm $\mathcal{A}'$. We then appeal to the fact that $\mathcal{A}'$ is a k-set agreement algorithm to argue that the given history of Algorithm 1 "inherits" from the constructed history of $\mathcal{A}'$ the properties of k-set agreement.

Recall that each real process i simulates a sequence of steps for each virtual process j — one step in each iteration of the loop on lines 7-21 of thread j. Let $\sigma^{i,j}$ denote the sequence of steps that real process i simulates for virtual process j.

It is easy to see that each $\sigma^{i,j}$ satisfies the local consistency property of a history. This is because the state of the process is updated using the $\textsc{trans}_j$ function in each iteration. Similarly it is easy to see that each $\sigma^{i,j}$

has a valid initialization. This is because the state of j is initialization using the INIT_STATE$_j$ function on line 5

Next, we argue that the different real processes simulate the *same* sequence of steps for every virtual process. That is, for each pair of real processes i and i', $\sigma^{i,j} = \sigma^{i',j}$. This follows by a straightforward induction, given that real processes agree on the input of virtual process j (by using the consensus object $CO[j][0]$) and they agree on the value read by each read step of j (by using the consensus object $CO[j][k]$, where the read step in question is the k-th step of j). We denote this sequence σ^j, and the k-th step in σ^j, i.e., $\sigma^j[k]$, as σ^j_k.

We now have sequences of steps of the individual virtual processes, but we still have to combine them into a single sequence and show that the sequence is a history of the simulated algorithm $\mathcal{A}'$. To combine the individual sequences we will assign to every step σ^j_k an "execution time", denoted $T(\sigma^j_k)$, and order the steps by their execution times into a sequence σ. To ensure that the resulting sequence is a history of $\mathcal{A}'$, we must guarantee that (a) the steps of each virtual process appear in σ in the same order as in σ^j (thus preserving the local consistency and valid initialization of σ^j); and (b) σ satisfies read consistency.

To define the execution time of σ^j_k we first define the time t^j_k when σ^j_k "begins to be simulated." This is the time of the first execution of the first line in iteration k of the main loop in thread j by any real process. We will assign $T(\sigma^j_k)$ to a value in the interval $[t^j_k, t^j_{k+1})$, i.e., between the times when σ^j_k and the next step of j, σ^j_{k+1}, begin to be simulated. This ensures that $T(\sigma^j_k) < T(\sigma^j_{k+1})$ so the steps of j are ordered in σ as in σ^j.

With this in mind, we can now assign execution times to write and output steps. If σ^j_k is a write step that writes (w, k) we set $T(\sigma^j_k)$ to be the first time any real process writes (w, k) to $MEM[-][j]$. This is clearly in interval $[t^j_k, t^j_{k+1})$ since no real process can begin iteration $k+1$ of thread j without first writing a value for step σ^j_k. If σ^j_k is a output step then we set $T(\sigma^j_k)$ to be t^j_k.

It remains to assign execution times to read steps. Consider a read step σ^j_k that reads value (w, k') from the register of virtual process j'. This value was written by step $\sigma^{j'}_{k'}$. Let $\sigma^{j'}_{k''}$ be then next write step of j', which overwrites (w, k') by a new pair, (w', k'') in j''s register, for some value w' and some timestamp $k'' > k'$. To satisfy read consistency, the execution time of σ^j_k, $T(\sigma^j_k)$, must fall between the execution times of the write steps $\sigma^{j'}_{k'}$ and $\sigma^{j'}_{k''}$.

By the above definition of the execution times of write steps, we have that $T(\sigma^{j'}_{k'})$ is the earliest time when any real process writes (w, k') into $MEM[-][j']$, and $T(\sigma^{j'}_{k'})$ is the earliest time when any real process writes (w', k'') into $MEM[-][j']$. We call this interval, between when (w, k') is first written and the time it is first over-written in $MEM[-][j']$, the interval during which (w, k') is "available." Note that the real process that first overwrites (w, k') may be different from the real process that first writes (w, k').

So there are two requirements that $T(\sigma^j_k)$ must satisfy: It must be in the interval $[t^j_k, t^j_{k+1})$ and it must be in the interval (t_1, t_2) during which (w, k'), the pair read by σ^j_k, is available. For this to be possible, these two intervals must intersect. We now argue that this is indeed the case.

Consider the simulation of σ^j_k, a read step that reads (w, k') from the register of virtual process j', by any real process. That is, we consider the execution of lines 11-16 in iteration k of thread j. We call the execution of the loop on lines 12-13 *the scan*, and the pair (v, s) that i proposes to the consensus object $CO[j][k]$ on line 15 the value that the scan *yields* for j' (the process whose register σ^j_k reads). Note that scans of different real processes simulating σ^j_k may yield different values — these will be reconciled through the use of the consensus object $CO[j][k]$.

Since σ^j_k reads the pair (w, k') from the register of virtual process j', it is clear from the algorithm that *some* real process's scan when it simulates σ^j_k yields (w, k'), and proposes (w, k') to $CO[j][k]$. Consider the scan of the first real process, say i, to do so; by the specification of the consensus object, this is the first real process to access $CO[j][k]$. This scan occurred during a subinterval of $[t^j_k, t^j_{k+1})$: It started no earlier then any process started simulating σ^j_k, and it ended before any real process accessed $CO[j][k]$ (because, by definition, i is the first real process to access $CO[j][k]$) and therefore before any real process started simulating σ^j_{k+1}.

We claim that this scan overlaps the interval (t_1, t_2) in which (w, k') is available. For, if the scan ended before the start of (t_1, t_2), it could not yield (w, k') for j' since (w, k') is first written in $MEM[-][j']$ at t_1; and if the scan started after the end of (t_1, t_2), it would yield some pair with timestamp greater then k', since at time t_2 some real process writes in $MEM[-][j']$ a pair with timestamp greater then k'.

So, we have shown that a scan that occurs during a subinterval of $[t^j_k, t^j_{k+1})$ also overlaps (t_1, t_2). Therefore $[t^j_k, t^j_{k+1})$ intersects (t_1, t_2), as wanted, and we can choose a time in the intersection of these two intervals for $T(\sigma^j_k)$, thereby satisfying both requirements. Figure 3 illustrates

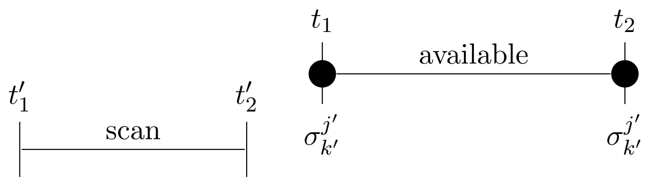


Figure 1: The scan preceding the availability interval.

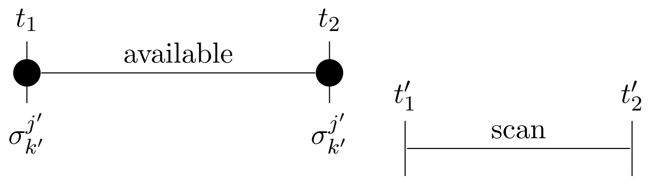
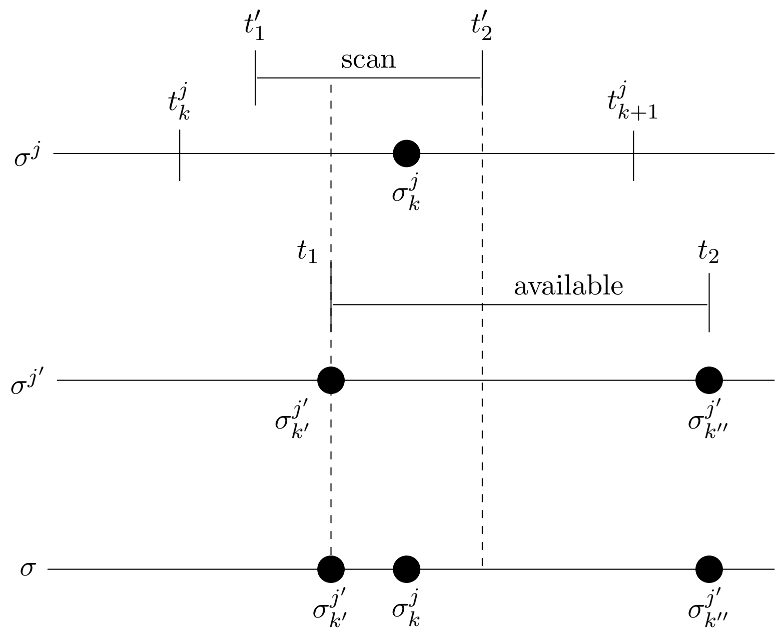


Figure 2: The scan following the availability interval

how $T(\sigma^j_k)$ is chosen when σ^j_k is a read step.

We now know that we can organize the steps simulated by the real processes into a history σ of $\mathcal{A}'$. We can therefore rely on the agreement, validity, and termination properties of the simulated k-set agreement algorithm to prove the agreement, validity, and termination properties for the algorithm executed by the real processes. Each real process simulates infinitely many steps of every virtual process, so it will eventually simulate a step in which a virtual process outputs a value (by the termination property of σ) and it will also output a value. Since the input used in the simulation of each virtual process is the input of some real process, by the validity property of σ, the value that any real process outputs was the input of some real process. Finally, since each values output by a real processes is a value output by a virtual process in V, by the agreement property of σ, the real processes output at most k different values.



- ∘ σ^j_k is a read step that reads the value written by virtual process j' in step k'.
- ∘ $[t'_1, t'_2]$ is the interval of a scan for step σ^j_k.
- ∘ $\sigma^{j'}_{k'}$ is a write step of j'.
- ∘ $\sigma^{j'}_{k''}$ is the next write step of j'.
- ∘ (t_1, t_2) is the interval during which the value written in step $\sigma^{j'}_{k'}$ is available.
- ∘ The dashed lines indicate the intersection of the scan and the availability interval.
- ∘ The solid black dots indicate the execution time of the steps.

Figure 3: The placement of a read step.

3.3 Proof of the Correctness of the Simplified BG-Simulation

Given any k-set agreement algorithm $\mathcal{A}$, we now describe how to simulate its counterpart $\mathcal{A}'$ (which, by Observation 2 also solves k-set agreement). Our goal is the following: given an arbitrary history of Algorithm 1, construct a history of $\mathcal{A}'$ that is used by the real processes to solve k-set agreement. We begin with some simple definitions.

Definition 3. *Let I be the set of real processes and J be the set of virtual processes. $|I| = n$, $|J| = m$.*

Throughout this section i and i' denote real processes in I, and j and j' denote virtual processes in J. Also throughout this section we will consider an arbitrary history $\mathcal{R}$ of Algorithm 1 by the real processes in I. All mentions of the steps of *real* processes refer to their steps in $\mathcal{R}$. We now define what it means for a real process to simulate a step of a virtual process.

Definition 4. *Consider iteration k of thread j by real process i. Let S be the value of state$[j]$ at the beginning of this iteration. There are three possible cases:*

- $\textsc{nextop}(S) = (\textsc{write}, v)$. *If real process i executes line 9 in iteration k, we say that process i simulates step $(j, (S, k), \textsc{done})$ of $\mathcal{A}'$; we also say that this is a write step in which virtual process j writes (v, k).*
- $\textsc{nextop}(S) = (\textsc{read}, j')$. *If real process i executes line 15 in iteration k and gets (w, k') from consensus object $CO[j][k]$, we say that process i simulates step $(j, (S, k), (w, k'))$ of $\mathcal{A}'$; we also say that this is a read step in which virtual process j reads (w, k') from virtual process j'.*
- $\textsc{nextop}(S) = (\textsc{output}, o)$. *If real process i executes line* DECIDE *in iteration k we say that process i simulates step $(j, (S, k), \textsc{done})$ of $\mathcal{A}'$; we also say that this is an output step in which virtual process j outputs o.*

Definition 5. *Let $\sigma^{i,j}$ be the sequence of steps of $\mathcal{A}'$ that real process i simulates for virtual process j, in the order that i simulates them.*

Lemma 6. *For each real process i and each virtual process j, $\sigma^{i,j}$ is infinite.*

Proof. For every i, the loop on lines 7 to 21 contains no blocking statements. Under the assumption that the threads created on line 3 are scheduled in a fair way, for each j, the main loop (line 7) in thread j belonging to real process i will be executed an infinite number of times. Therefore i will simulate an infinite number of steps of j. Thus $\sigma^{i,j}$ is infinite. □

Lemma 7. *For each real process i and each virtual process j, $\sigma^{i,j}$ is locally consistent with respect to $\mathcal{A}'$.*

Proof. Consider any two consecutive steps $\sigma^{i,j}[k]$ and $\sigma^{i,j}[k+1]$ of $\sigma^{i,j}$. By Definition 4,

- $\sigma^{i,j}[k] = (j, (S, k), \mathit{ret})$ where S is the value of $\mathit{state}[j]$ at the start of iteration k of thread j by real process i, and ret is the pair $(w, -)$ returned by $CO[j][k]$ (if $\sigma^{i,j}[k]$ is a read step) or DONE (if $\sigma^{i,j}[k]$ is a write or output step); and
- $\sigma^{i,j}[k+1] = (j, (S', k+1), -)$, where S' is the value of $\mathit{state}[j]$ at the start of iteration $k+1$ of thread j by real process i.

We must prove that $(S', k+1) = \textsc{trans}'_j\Big((S,k), \mathit{ret}\Big)$.

By Definition 66,

$$\textsc{trans}'_j\Big((S,k), \mathit{ret}\Big) = \begin{cases} \Big(\textsc{trans}_j(S, w), k+1\Big), & \text{if } \mathit{ret} = (w, -) \\ \Big(\textsc{trans}_j(S, \textsc{done}), k+1\Big), & \text{if } \mathit{ret} = \textsc{done} \end{cases}$$

By lines 10, 16, and 19,

$$S' = \begin{cases} \textsc{trans}_j(S, w), & \text{if } \mathit{ret} = (w, -) \\ \textsc{trans}_j(S, \textsc{done}), & \text{if } \mathit{ret} = \textsc{done} \end{cases}$$

Therefore, $\textsc{trans}'_j\Big((S,k), \mathit{ret}\Big) = (S', k+1)$, as wanted. □

We will now show that the real processes simulate the *same* sequence of steps for each virtual process.

Lemma 8. *For each pair of real processes i and i', each virtual process j, and each $k \in \mathbb{N}$, $\sigma^{i,j}[k] = \sigma^{i',j}[k]$.*

Proof. Let i and i' be arbitrary real processes and j be an arbitrary virtual process. We prove the lemma by induction on k.

BASE CASE: $k = 1$. Let $\sigma^{i,j}[1] = (j, (S, 1), \mathit{ret})$, and $\sigma^{i',j}[1] = (j, (S', 1), \mathit{ret}')$. At the beginning of the first iteration of the main loop in thread j by any real process, $\mathit{state}[j] = \textsc{init\_state}_j(w)$ (line 5). Since w is the value returned by the DECIDE operation of consensus object $CO[j][0]$ (line 4), we can conclude that $S = S'$. If $\textsc{nextop}_j(S) = (\textsc{read}, -)$ then $\mathit{ret} = \mathit{ret}' = (w, k')$, since (w, k') is the value returned by the DECIDE operation of consensus object $CO[j, 1]$ (line 15); otherwise $\mathit{ret} = \mathit{ret}' = \textsc{done}$. Therefore $\sigma^{i,j}[1] = \sigma^{i',j}[1]$.

INDUCTION STEP: Suppose $\sigma^{i,j}[k] = \sigma^{i',j}[k] = (j, (S_k, k), ret_k)$. We will prove that $\sigma^{i,j}[k+1] = \sigma^{i',j}[k+1]$. Let $\sigma^{i,j}[k+1] = (j, (S, k+1), ret)$, and $\sigma^{i',j}[k+1] = (j, (S', k+1), ret')$. From the code (lines 10, 16, and 19) we see that for each real process, at the beginning of the $k+1$-th iteration of the main loop in thread j, $state[j] = \text{TRANS}_j(S_k, ret_k)$. Therefore $S = \text{TRANS}_j(S_k, ret_k) = S'$. As in the base case, if $\text{NEXTOP}_j(S) = (\text{READ}, -)$ then $ret = ret' = (w, k')$, since (w, k') is the value returned by the DECIDE operation of consensus object $CO[j, k+1]$ (line 15); otherwise $ret = ret' =$ DONE. Therefore $\sigma^{i,j}[k+1] = \sigma^{i',j}[k+1]$. □

By Lemma 8, every real process simulates the same sequence of steps for each virtual process. Thus we can define the sequence of steps of each individual virtual process independent from the real process performing the simulation.

Definition 9. *For each virtual process j, and each real process i, $\sigma^j = \sigma^{i,j}$.*

Thus, $\sigma^j[k]$ is the k-th step in the sequence of steps of virtual process j. We denote $\sigma^j[k]$ simply as σ^j_k; so $\sigma^j = \sigma^j_1\sigma^j_2\sigma^j_3\ldots$.

Observation 10. *For each virtual process j, σ^j is locally consistent with respect to $\mathcal{A}'$.*

Recall that $\mathcal{R}$ is the history of Algorithm 1 under consideration.

Definition 11. *A step of history $\mathcal{R}$ of Algorithm 1* occurs at time t *if it is the t-th step of $\mathcal{R}$, i.e., it is step $\mathcal{R}[t]$.*

Definition 12. *Consider the simulation, by some real process i, of a step in which some virtual process reads the register of virtual process j' (see Definition 4). Before i simulates that step (by executing line 15), it executes the loop in lines 12-13. The execution of this loop is called a* scan (of j')*. We say that this scan* yields (w, k') for j' *if $(w, k') = \text{VMAX}(M)$ where M is the value of array my_mem at the end of the scan. Finally we say that the scan* occurs during interval $[t_1, t_2]$, *if t_1 and t_2 are the times of the execution of the first and last read steps of this scan.*

If a pair (w, k') appears in $MEM[-][j']$ during real history $\mathcal{R}$, we define the interval of time during which it is "available" from j'. Intuitively this interval starts when (w, k') is first written to $MEM[-][j']$ and ends when a pair with a higher timestamp is written to $MEM[-][j']$. More precisely,

Definition 13. *Let j' be any virtual process and (w, k') be any pair that appears in $MEM[-][j']$. We say that (w, k')* is available from virtual process j' during interval (t_1, t_2) *where t_1 and t_2 are defined as follows.*

Case 1. $k' = 0$.

In this case, $t_1 = 0$; t_2 is the time of the first write to MEM$[-][j']$ by any real process, if such a write exists, and ∞ otherwise.

Case 2. $k' > 0$.

In this case, t_1 is the time of the first write of (w, k') to MEM$[-][j']$ by any real process; t_2 is the time of the first write of some value $(-, k'')$ to MEM$[-][j']$ by any real process where $k'' > k'$, if such a write exists, and ∞ otherwise.

The next observation follows immediately by inspection of the code of Algorithm 1.

Observation 14. *For each real process i, and each virtual process j, if MEM$[i][j]$ contains (w, k) and later contains (w', k'), then $k' \geq k$.*

The next observation follows from Definition 13, Observation 14, and Lemma 8.

Observation 15. *If a pair appears in MEM then the interval during which that pair is available is not empty.*

Lemma 16. *Suppose a scan yields (w, k') for virtual process j'. If*

(i) this scan occurs during some interval $[t'_1, t'_2]$, and

(ii) (w, k') is available from j' during interval (t_1, t_2),

then $[t'_1, t'_2]$ intersects (t_1, t_2).

Proof. Suppose, for contradiction, that a scan yields (w, k') for virtual process j', and (i) and (ii) hold, but $[t'_1, t'_2] \cap (t_1, t_2) = \emptyset$. Thus, interval $[t'_1, t'_2]$ is before or after (t_1, t_2), i.e., either $t'_2 \leq t_1$ or $t_2 \leq t'_1$.

Case 1. $t'_2 \leq t_1$. See Figure 1. From (i), the scan reads (w, k') from *MEM*$[-][j']$ at some time $t \leq t'_2$. Clearly, $t'_2 \geq 1$, and so $t_1 \geq 1$. Thus, from (ii) and Definition 13, $k' > 0$ and the pair (w, k') is first written in *MEM*$[-][j']$ at time t_1. Since $t_1 \geq t'_2$, we have $t < t_1$. So (w, k') is read from *MEM*$[-][j']$ at time t, before this pair is first written at time t_1 — a contradiction.

Case 2. $t_2 \leq t'_1$. See Figure 2. Since t'_1 is finite, t_2 is finite. Thus, from (ii) and Definition 13, some real process i wrote a pair $(-, k'')$ with $k'' > k'$ in *MEM*$[i][j']$ at time t_2. Thus, by Observation 14, the scan (which starts after time t_2) reads some pair $(-, k^*)$ such that $k^* \geq k'' > k'$ from *MEM*$[i][j']$. Since $k^* > k'$, the scan does *not* yield (w, k') for j' — a contradiction to (i).

□

We now define the time t_k^j that marks the beginning of the simulation of step σ_k^j (recall that σ_k^j is the k-th step of virtual process j).

Definition 17. *Let t_k^j be the earliest time when any real process executes the first step of iteration k of the main loop of thread j.*

The next fact follows immediately from the above definition and Lemma 8.

Observation 18. *For each pair of steps σ_k^j and $\sigma_{k'}^j$ by the same virtual process j, if $k < k'$ then $t_k^j < t_{k'}^j$.*

Lemma 19. *Suppose a step σ_k^j reads (w, k') from virtual process j', and let (t_1, t_2) be the interval during which (w, k') is available from j'. Then:*

(i) There is a scan that yields (w, k') for j'.

(ii) This scan occurs during an interval of time I such that:

- *(a) I is a subinterval of (t_k^j, t_{k+1}^j).*
- *(b) I intersects interval (t_1, t_2).*

Proof. Suppose a step σ_k^j reads (w, k') from virtual process j'. Then the consensus object $CO[j][k]$ must return (w, k') to at least one real process by some time $t < t_{k+1}^j$. So at least one real process, say process i, invokes $CO[j][k].\text{DECIDE}\Big((w, k')\Big)$ by time $t < t_{k+1}^j$; this must occur in iteration k of thread j of real process i. Consider the scan that process i does in iteration k of thread j. Clearly this scan yields (w, k') for virtual process j'. This proves (i).

Moreover, this scan must start after time t_k^j, and it must end before process i invokes $CO[j][k].\text{DECIDE}\Big((w, k')\Big)$, i.e., before time $t < t_{k+1}^j$. So this scan occurs during some sub-interval I of (t_k^j, t_{k+1}^j), proving part (a) of (ii). Since the scan yields (w, k') for virtual process j', by Lemma 16, the interval I of the scan intersects the interval (t_1, t_2) during which (w, k') is available from virtual process j', proving part (b) of (ii). □

The above lemma immediately implies the following corollary:

Corollary 20. *If a step σ_k^j reads (w, k') from virtual process j', then there is a time $t \in (t_k^j, t_{k+1}^j) \cap (t_1, t_2)$, where (t_1, t_2) is the interval during which (w, k') is available from virtual process j'.*

To order the virtual steps of the simulated processes, we now assign an "execution time" to each one.

Definition 21. *For each step σ^j_k, the* execution time of σ^j_k, *denoted $T(\sigma^j_k)$, is defined as follows:*

Case 1. *σ^j_k is a step that writes (w, k). Then $T(\sigma^j_k)$ is the time of the first write of (w, k) to $MEM[-][j]$ by any real process.*

Case 2. *σ^j_k is a step that reads some pair (w, k') from virtual process j'. By Corollary 20, there is a time $t \in (t^j_k, t^j_{k+1}) \cap (t_1, t_2)$, where (t_1, t_2) is the interval during which (w, k') is available from virtual process j'. Then $T(\sigma^j_k) = t$.*

Case 3. *σ^j_k is an output step. Then $T(\sigma^j_k) = t^j_k$.*

Observation 22. *For each step σ^j_k, $T(\sigma^j_k) \in [t^j_k, t^j_{k+1})$.*

Observation 18 and the above observation immediately implies:

Observation 23. *For each pair of steps σ^j_k and $\sigma^j_{k'}$ by the same virtual process j, if $k < k'$ then $T(\sigma^j_k) < T(\sigma^j_{k'})$.*

Lemma 24. *Suppose step σ^j_k reads (w, k') from virtual process j'.*

(i) If (w, k') is the initial value $(-, 0)$ then there is no write step $\sigma^{j'}_\ell$ of j' such that $T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$.

(ii) If (w, k') is not the initial value $(-, 0)$ then there is a write step $\sigma^{j'}_{k'}$ of j' that writes (w, k') such that $T(\sigma^{j'}_{k'}) < T(\sigma^j_k)$, and there is no write step $\sigma^{j'}_\ell$ of j' such that $T(\sigma^{j'}_{k'}) < T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$.

Proof. Suppose step σ^j_k reads (w, k') from j', and let (t_1, t_2) be the interval during which (w, k') is available from j'. Note that by the definition of T, $T(\sigma^j_k) \in (t_1, t_2)$ so $t_1 < T(\sigma^j_k) < t_2$.

(i) Suppose $(w, k') = (-, 0)$. Since $(-, 0)$ is available from j' during (t_1, t_2), $t_1 = 0$, and there is no real process that writes in $MEM[-][j']$ before time t_2 (*). Suppose, for contradiction, that there is a write step $\sigma^{j'}_\ell$ such that $T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$. Thus $T(\sigma^{j'}_\ell) < t_2$. Since $\sigma^{j'}_\ell$ is a write step, by the definition of T, some real process writes $(-, \ell)$ in $MEM[-][j']$ at time $T(\sigma^{j'}_\ell)$. Since $T(\sigma^{j'}_\ell) < t_2$, this contradicts (*).

(ii) Suppose $(w, k') \neq (-, 0)$. Since $(w, k') \neq (-, 0)$ and (w, k') is available from j' during (t_1, t_2), some real process first writes (w, k') into $MEM[-][j']$ at time t_1. So there is a step $\sigma^{j'}_{k'}$ of j' that writes (w, k') and $T(\sigma^{j'}_{k'}) = t_1$. Thus, $T(\sigma^{j'}_{k'}) < T(\sigma^{j}_{k})$.

It remains to show that there is no write step $\sigma^{j'}_{\ell}$ of j' such that $T(\sigma^{j'}_{k'}) < T(\sigma^{j'}_{\ell}) \leq T(\sigma^{j}_{k})$. Suppose, for contradiction, that such a step $\sigma^{j'}_{\ell}$ exists. Thus, $T(\sigma^{j'}_{k'}) < T(\sigma^{j'}_{\ell}) \leq T(\sigma^{j}_{k})$. First note that by Observation 23, $\ell > k'$. Furthermore, since $T(\sigma^{j'}_{k'}) = t_1$ and $T(\sigma^{j}_{k}) < t_2$, we have $t_1 < T(\sigma^{j'}_{\ell}) < t_2$. So some real process writes a pair $(-, \ell)$ with $\ell > k'$ in $MEM[-][j']$ after time t_1 and before time t_2. This contradicts the fact that (w, k') is available from j' during interval (t_1, t_2). □

Definition 25. *The sequence of steps (of $\mathcal{A}'$) σ is the sequence consisting of all the steps in the sequences $\sigma^1, \sigma^2, \ldots, \sigma^j, \ldots, \sigma^m$, ordered by increasing step execution times (with process ids resolving any ties).*

Lemma 26. *The sequence of steps σ is a history of $\mathcal{A}'$ that has input $(x_1, x_2, \ldots, x_m)$, where for each j, x_j is the input to some real process in $\mathcal{R}$.*

Proof. By Observation 23, for each virtual process j, the sub-sequence of σ consisting of the steps of virtual process j is $\sigma^j_1\sigma^j_2\sigma^j_3\ldots = \sigma^j$. Thus, from Observation 10, σ is locally consistent.

By Lemma 24, σ is also read consistent.

From Definition 4, line 4, line 5, and Observation 23 it is clear that, for each virtual process j, if $(j, (S, 1), -)$ is the first step of j in σ, then $S = \text{INIT\_STATE}_j(w)$ where w is the input to some real process. Therefore, by Definition 66, $(S, 1) = \text{INIT\_STATE}'_j(w)$. □

Lemma 27. *If d is the output of some real process i in $\mathcal{R}$, then d is the output of some virtual process j in the history σ.*

Proof. Let i be an arbitrary real process that outputs d. Then i simulates an output step, $(j, (S, k), ret)$, of some virtual process j that outputs d (see line 18). By Definitions 4 and 9, $\sigma^j_k = (j, (S, k), ret)$. By Definition 25, σ^j_k is also a step in σ and therefore d is the output value of virtual process j in σ. □

Lemma 28. *The real processes output at most k different values in $\mathcal{R}$.*

Proof. Let V be the set of values output by real processes, and let V' be the set of values output by virtual processes. By Lemma 27, $V \subseteq V'$. By Lemma 26, and the agreement property of $\mathcal{A}'$, $|V'| \leq k$. Therefore $|V| \leq k$ and the real processes output at most k different values in $\mathcal{R}$. □

Lemma 29. *If some real process i outputs d, then d is the input of some real process i'.*

Proof. Let i be an arbitrary real process that outputs d. By Lemma 27, d is an output of some virtual process j in history σ. By Lemma 26, and the validity property of $\mathcal{A}'$, d is the input to some real process. □

Lemma 30. *Each real process i outputs exactly one value in $\mathcal{R}$.*

Proof. By Lemma 6 each $\sigma^{i,j}$ is infinite. Therefore, by Definitions 9 and 25, each virtual processes takes infinitely many steps in history σ. Since, by Lemma 26, σ is a history of $\mathcal{A}'$, the termination property of $\mathcal{A}'$ implies that all virtual processes output a value in σ.

Since i simulates an infinite number of steps of each virtual process j and j outputs in σ, i must simulate an output step of j. When i simulates such a step of for the first time, i will output a value (line 18). □

Theorem 31. *Algorithm 1 is a k-set agreement algorithm.*

Proof. Follows by Lemmas 28, 29, and 30. □

4 The Full BG-Simulation

We will now present the complete BG-simulation that allows for crashes and uses only registers. To do so, we must first introduce an object derived from registers that will be used to replace the consensus objects used in Algorithm 1. Such an object was first described in [3].

4.1 Safe Agreement

A safe agreement object is used by n processes to agree on some value in much the same way as consensus. It provides two operations PROPOSE(v) and DECIDE(): PROPOSE(v) receives a value as an argument and has no return value; DECIDE() has no arguments and returns the agreed upon value. A process must invoke PROPOSE before invoking DECIDE, and can invoke PROPOSE at most once. The following properties must hold for a safe agreement algorithm.

Agreement

If two invocations of DECIDE return v_1 and v_2 then $v_1 = v_2$.

Validity

If some invocation of DECIDE returns v then some process previously invoked PROPOSE(v).

Termination

(a) Every correct process that invokes PROPOSE returns from its invocation.

(b) If no process crashes during its invocation of PROPOSE, then every correct process returns from its invocation of DECIDE.

We now present (in Algorithm 3) an implementation of a safe agreement. The implementation presented here was first shown in [4] in the I/O automata formalization and in [8] in pseudo-code. Both previous implementations were in terms of snapshot objects. We slightly modify the the implementation to be in terms of registers. It works as follows. We say a process is at level $i \in \{1, 2\}$ if i is the value in its corresponding $LEVEL$ register. When a process i invokes PROPOSE(v), it places v in shared register $V[i]$ so that other processes may know what value it is proposing. It then announces that it is participating by setting its level to 1. It then checks to see if any process has reached level 2. If no process is at level 2 then i moves itself to level 2; otherwise it moves itself to level 0.

When invoking DECIDE, a process waits until no process is at level 1 and then it finds process p with the minimum id that is at level 2 and returns the value that p proposed.

Algorithm 3 Safe Agreement

Code for process i, where $i = 1..n$

```
    **Shared Memory**
    $V$ an array of $n$ registers each initialized to $\perp$
    $LEVEL$ an array of $n$ registers each initialized to $\perp$
    For each $i$, $V[i]$ and $LEVEL[i]$ can be written to by $i$
    **Process Local Variables**
    $v$, a variable, initially the input to $i$
    $j, k, m \in \mathbb{N}$, initially arbitrary
    $level$ an array of $n$ variables, initially arbitrary
 1: procedure PROPOSE(v)
 2:     $V[i] \leftarrow v$
 3:     $LEVEL[i] \leftarrow 1$
 4:     for $j = 1..n$ do
 5:         $level[j] \leftarrow LEVEL[j]$
 6:     if $\exists 1 \leq k \leq n, level[k] = 2$ then
 7:         $LEVEL[i] \leftarrow 0$
 8:     else
 9:         $LEVEL[i] \leftarrow 2$
10: procedure DECIDE( )
11:     repeat
12:         for $j = 1..n$ do
13:             $level[j] \leftarrow LEVEL[j]$
14:     until $\forall 1 \leq k \leq n, level[k] \neq 1$
15:     $m \leftarrow$ MINIMUM($\{k | level[k] = 2\}$)
16:     output $V[m]$
```

We will now prove that Algorithm 3 is a safe agreement algorithm.

Observation 32. *Once a process sets its LEVEL register to 0 or 2 its LEVEL register never changes.*

Lemma 33. *By the time any process completes an invocation of* PROPOSE *there is at least one process j that set $LEVEL[j]$ to 2.*

Proof. Assume some process i completes an invocation of PROPOSE. Either i read 2 from some $LEVEL[i']$ with $i' \neq i$ on line 5 or it set $LEVEL[i]$ to 2 on line 9. $\square$

Lemma 34. *Algorithm 3 satisfies the Validity property of safe agreement.*

Proof. Suppose process i returns a value v on line 16. By assumption when process i invokes DECIDE it has already completed an invocation of PROPOSE and therefore, by Lemma 33, there must be at least one process j that set $LEVEL[j]$ to 2. Therefore whenever process i executes the loop on line 12 it will read $LEVEL[j]$ as 2. This means that when process i executes line 15 of DECIDE the set $\{k|LEVEL[k] = 2\}$ is nonempty, and so m is an index such that $1 \leq m \leq n$ and $LEVEL[m] = 2$. Before process m sets $LEVEL[m] = 2$ on line 9 of PROPOSE, it sets $V[m] = v$ on line 2, where v is the argument of the call to PROPOSE that m invoked. Therefore process i returns a value v such that process m previously invoked PROPOSE(v). $\square$

Lemma 35. *Algorithm 3 satisfies the Agreement property of safe agreement.*

Proof. Suppose two processes i and i' output two different values v and v' respectively. By Validity v and v' were proposed by some processes j and j' respectively. Assume, without loss of generality, that $j < j'$.

Claim 35.1. *During i''s execution of the final iteration of the loop on line 12 of* DECIDE *it must read j''s level as 2 and j's level as $\perp$.*

Proof of Claim 35.1. If j''s level was not 2 then i' would not choose j' on line 15 and therefore, would not return v' on line 16. If j's level was 2 then i' would not choose j' on line 15 since $j < j'$. If j's level was 1 then i' would fail the test on line 14 and it would not be the final execution of the loop. If j's level was 0 then i would not choose j on line 15 because process j invokes PROPOSE only once; in that invocation, it sets $LEVEL[j]$ to 0, and no other process writes $LEVEL[j]$, so $LEVEL[j]$ is never set to 2. $\square$

Claim 35.2. *Some process is at level 2 before j executes line 3 of* PROPOSE.

Proof of Claim 35.2. By Claim 35.1, i' reads j's level as $\perp$ on line 13 of DECIDE and by assumption, i' must complete its invocation of PROPOSE before invoking DECIDE. Therefore i' must have completed its invocation of PROPOSE before j executes line 3 of PROPOSE. By Lemma 33, by the time i' executed PROPOSE there is some process at level to 2. □

From Claim 35.2, when j executes the loop on line 4 of PROPOSE it will find some process at level 2 and will therefore set itself to level 0 on line 7 and will never set $LEVEL[j]$ to 2. Since $LEVEL[j]$ is never set to 2, process i will not select k on line 15 of DECIDE and will not output v. This is a contradiction. □

Lemma 36. *Algorithm 3 satisfies the Termination property of safe agreement.*

Proof. (a) By inspection of the code, PROPOSE does not block.

(b) Suppose no process crashes during its invocation of PROPOSE. Then there is a time t after which all processes' $LEVEL$ variables are not 1. This is because (i) each process calls PROPOSE at most once (by assumption), and (ii) for any i, $LEVEL[i] = 1$ only between the time process i executes line 3 and the time it completes PROPOSE. So every correct process that invokes DECIDE will eventually exit the loop on line 12 of DECIDE, and thus return from that invocation.

□

Theorem 37. *Algorithm 3 is an implementation of a safe agreement object.*

Proof. By Lemmas 34, 35, and 36, Algorithm 3 satisfies the Validity, Agreement, and Termination properties of safe agreement. □

4.2 Algorithm

We now explain how to modify the simplified BG-simulation discussed earlier (Algorithm 1) so that the resulting algorithm (a) uses only registers (and no consensus objects), and (b) tolerates process crashes.

To eliminate the consensus objects used in the simplified BG-simulation, we replace each instance of a consensus object $CO[j][k]$ with a safe agreement object $SA[j][k]$ and each call to $CO[j][k]$.DECIDE(v) with a pair of calls to $SA[j][k]$.PROPOSE(v) and $SA[j][k]$.DECIDE().

This change alone eliminates the use of consensus objects but the resulting algorithm is not fault tolerant. To see this, we first note that a real process that crashes while executing the operation PROPOSE on a safe agreement object may cause other real processes executing the DECIDE operation on the same safe agreement object to never terminate. This will happen if the real process executing PROPOSE crashes after it has set its *LEVEL* variable to 1 (see line 3 in Algorithm 3) and before it has set that variable to 0 or 2 (see lines 7 and 9).

Therefore, a real process that crashes while executing multiple concurrent calls to PROPOSE applied to multiple safe agreement objects, $SA[j_1][k_1]$, $SA[j_2][k_2], \ldots, SA[j_k][k_k]$, while executing threads $j_1, j_2, \ldots, j_k$ (see lines 3-22 in Algorithm 1) may end up blocking forever other real processes that are trying to complete invocations of DECIDE on the safe agreement objects $SA[j_1][k_1], SA[j_2][k_2], \ldots, SA[j_k][k_k]$. In fact, the crash of a single real process may block all other real processes from simulating steps of the virtual processes. This would be catastrophic, as it would cause the execution of the simulated algorithm to "freeze."

To avoid this problem, we simply require each real process to execute PROPOSE operations on safe agreement objects (in its different threads) in mutual exclusion, i.e., one at a time. In this manner, the crash of a real process can prevent at most one thread, i.e., at most one virtual process, from making progress. Thus, each crash of a real process during the execution of the simulation algorithm results in a crash of a single virtual process in the simulated execution. If the simulated algorithm tolerates f crashes, the algorithm using it also tolerates f crashes.

Therefore, to obtain the full BG-simulation we simply replace every line of the form $r \leftarrow CO.\text{DECIDE}(v)$ in Algorithm 1 with the four lines given below.

Algorithm 4 Replacing Consensus Objects with Safe Agreement Objects

```
1: $safe\_agreement\_lock.\text{LOCK}()$
2: $SA.\text{PROPOSE}(v)$
3: $safe\_agreement\_lock.\text{UNLOCK}()$
4: $r \leftarrow SA.\text{DECIDE}()$
```

Note that *safe_agreement_lock* is a mutual exclusion object between threads of a single real process and requires no shared memory to implement.

The fully modified algorithm is presented in Algorithm 5.

Algorithm 5 The BG-Simulation for k-set Agreement
Code for process i, where $i = 1..n$

```
**Shared Memory**
$\mathit{MEM}$, an $n$ by $m$ array of registers. For each $i$,$j$, $\mathit{MEM}[i][j]$
    contains a pair $(\mathit{val}, \mathit{steps})$:
    $\mathit{MEM}[i][j].\mathit{val}$, initially the initial value of $j$'s register in $\mathcal{A}$
    $\mathit{MEM}[i][j].\mathit{steps} \in \mathbb{N}$, initially 0
    For each $i$ $\mathit{MEM}[i][-]$ can be written to by $i$
$\mathit{SA}$, an $m$ by infinite array of safe agreement objects
**Process Local Variables**
$\mathit{input} \in I_i$, initially the input of real process $i$
$\mathit{state}$, $\mathit{steps}$, arrays of size $m$, initially arbitrary
$\mathit{decided}$, a boolean, initially arbitrary
$\mathit{safe\_agreement\_lock}$, a mutual exclusion object for $m$ threads
**Thread Local Variables**
$k', i', j, j', \ell, \mathit{steps} \in \mathbb{N}$, initially arbitrary
$\mathit{val}, d, v, w, o$, variables, initially arbitrary
$\mathit{my\_mem}$, an array of $n$ variables, initially arbitrary
```

```
procedure DECIDE($input$)
    $decided \leftarrow false$
    parallel for $j = 1..m$
        $safe\_agreement\_lock$.LOCK()
        $SA[j][0]$.PROPOSE($input$)
        $safe\_agreement\_lock$.UNLOCK()
        $w \leftarrow SA[j][0]$.DECIDE()
        $state[j] \leftarrow$ INIT_STATE$_j(w)$
        $steps[j] \leftarrow 1$
        repeat forever
            if NEXTOP$_j(state[j]) = ($WRITE$, v)$ then
                $MEM[i][j]$.WRITE$\Big((v, steps[j])\Big)$
                $state[j] \leftarrow$ TRANS$_j(state[j],$ DONE$)$
            else if NEXTOP$_j(state[j]) = ($READ$, j')$ then
                for $i' = 1..n$ do
                    $my\_mem[i'] \leftarrow MEM[i'][j']$.READ()
                $(val, steps) \leftarrow$ VMAX$(my\_mem)$
                $safe\_agreement\_lock$.LOCK()
                $SA[j][steps[j]]$.PROPOSE$\Big((val, steps)\Big)$
                $safe\_agreement\_lock$.UNLOCK()
                $(w, k') \leftarrow SA[j][steps[j]]$.DECIDE()
                $state[j] \leftarrow$ TRANS$_j(state[j], w)$
            else if NEXTOP$_j(state[j]) = ($OUTPUT$, o)$ then
                if $\neg decided$ then output $o$; $decided \leftarrow true$
                $state[j] \leftarrow$ TRANS$_j(state[j],$ DONE$)$
            $steps[j] \leftarrow steps[j] + 1$
        end
    end
```

4.3 Proof of the Correctness of the BG-Simulation

The proof in this section is very similar to the proof in section 3.3. The proofs of the safety properties are essentially the same, with only changes being in the small details such as line number references. The only significant changes are to the proofs of the liveness properties. To be precise only Lemma 41, Definition 44, and Lemma 64 contain significant changes. These definitions and lemmas are marked with a *.

Given any k-set agreement algorithm $\mathcal{A}$, we now describe how to simulate its counterpart $\mathcal{A}'$ (which, by Observation 2 also solves k-set agreement). Our goal is the following: given an arbitrary history of Algorithm 5, construct a history of $\mathcal{A}'$ that is used by the real processes to solve k-set agreement. We begin with some simple definitions.

Definition 38. *Let I be the set of real processes and J be the set of virtual processes. $|I| = n$, $|J| = m$.*

Throughout this section i and i' will denote real processes in I, and j and j' will denote virtual processes in J. Throughout this section we will consider an arbitrary history $\mathcal{R}$ of Algorithm 1 by the real processes in I. All mentions of the steps of *real* processes refer to their steps in $\mathcal{R}$. We now define what it means for a real process to simulate a step of a virtual process.

Definition 39. *Consider iteration k of thread j by real process i if it exists. Let S be the value of state$[j]$ at the beginning of this iteration. There are three possible cases:*

- $\text{NEXTOP}(S) = (\text{WRITE}, v)$. *If real process i executes line 12 in iteration k, we say that process i simulates step $(j, (S, k), \text{DONE})$ of $\mathcal{A}'$; we also say that this is a write step in which virtual process j writes (v, k).*

- $\text{NEXTOP}(S) = (\text{READ}, j')$. *If real process i executes line 21 in iteration k and gets (w, k') from safe agreement object $SA[j][k]$, we say that process i simulates step $(j, (S, k), (w, k'))$ of $\mathcal{A}'$; we also say that this is a read step in which virtual process j reads (w, k') from virtual process j'.*

- $\text{NEXTOP}(S) = (\text{OUTPUT}, o)$. *If real process i executes line 25 in iteration k we say that process i simulates step $(j, (S, k), \text{DONE})$ of $\mathcal{A}'$; we also say that this is an output step in which virtual process j outputs o.*

Definition 40. *Let $\sigma^{i,j}$ be the sequence of steps of $\mathcal{A}'$ that real process i simulates for virtual process j, in the order that i simulates them.*

We will now show that each real process i simulates an infinite sequence of steps of at least $m - \hat{f}$ virtual processes, where $\hat{f}$ is the number of real processes that crash in real history $\mathcal{R}$.

***Lemma 41.** *Suppose $\hat{f}$ real processes crash in real history $\mathcal{R}$. For each correct real process i there are at least $m - \hat{f}$ virtual processes j such that $\sigma^{i,j}$ is infinite.*

Proof. The only blocking statements in the main loop (lines 10 to 27) are the decide operations on lines 7 and 21. By the Termination property of safe agreement, these lines will block only if some real process invokes PROPOSE on the same safe agreement object as the DECIDE operation, and crashes during this invocation. Since each instance of PROPOSE is guarded by the use of mutual exclusion object *safe_agreement_lock*, a real process may crash while at most one thread is executing one of lines 5 or 19. Therefore, for each crashed real process there is a single safe agreement object whose invocations of DECIDE may block. Since $\hat{f}$ real processes crash and no two threads use the same safe agreement object, at most $\hat{f}$ threads may be blocked.

Let i be any correct real process. Under the assumption that the threads are scheduled in a fair way, the main loop in at least $m - \hat{f}$ threads of i will be executed an infinite number of times. Therefore i will simulate an infinite number of steps for at least $m - \hat{f}$ virtual processes, which, by the definition of $\sigma^{i,j}$, implies that there are at least $m - \hat{f}$ virtual processes j for which $\sigma^{i,j}$ is infinite. □

Lemma 42. *For each real process i and each virtual process j, $\sigma^{i,j}$ is locally consistent with respect to $\mathcal{A}'$.*

Proof. Consider any two consecutive steps $\sigma^{i,j}[k]$ and $\sigma^{i,j}[k+1]$ of $\sigma^{i,j}$. By Definition 39,

- $\sigma^{i,j}[k] = (j, (S, k), \mathit{ret})$ where S is the value of *state*$[j]$ at the start of iteration k of thread j by real process i, and *ret* is the pair $(w, -)$ returned by $CO[j][k]$ (if $\sigma^{i,j}[k]$ is a read step) or DONE (if $\sigma^{i,j}[k]$ is a write or output step); and

- $\sigma^{i,j}[k+1] = (j, (S', k+1), -)$, where S' is the value of *state*$[j]$ at the start of iteration $k+1$ of thread j by real process i.

We must prove that $(S', k+1) = \textsc{trans}'_j\Big((S,k), \mathit{ret}\Big)$.

By Definition 66,

$$\textsc{trans}'_j\Big((S,k), \mathit{ret}\Big) = \begin{cases} \Big(\textsc{trans}_j(S,w), k+1\Big), & \text{if } \mathit{ret} = (w, -) \\ \Big(\textsc{trans}_j(S, \textsc{done}), k+1\Big), & \text{if } \mathit{ret} = \textsc{done} \end{cases}$$

By lines 13, 22, and 25,

$$S' = \begin{cases} \textsc{trans}_j(S,w), & \text{if } \mathit{ret} = (w,-) \\ \textsc{trans}_j(S, \textsc{done}), & \text{if } \mathit{ret} = \textsc{done} \end{cases}$$

Therefore, $\textsc{trans}'_j\Big((S,k), \mathit{ret}\Big) = (S', k+1)$, as wanted. □

We will now show that the real processes simulate the *same* sequence of steps for every virtual process.

Lemma 43. *For each pair of real processes i and i', each virtual process j, and each $k \in \mathbb{N}$, if steps $\sigma^{i,j}[k]$ and $\sigma^{i',j}[k]$ exist, then $\sigma^{i,j}[k] = \sigma^{i',j}[k]$.*

Proof. Let i and i' be arbitrary real processes. We prove the lemma by induction on k.

Base Case: $k = 1$. If $\sigma^{i,j}[1]$ or $\sigma^{i',j}[1]$ does not exist, we are done; otherwise let $\sigma^{i,j}[1] = (j, (S,1), \mathit{ret})$, and $\sigma^{i',j}[1] = (j, (S',1), \mathit{ret}')$. At the beginning of the first iteration of the main loop in thread j by any real process, $\mathit{state}[j] = \textsc{init\_state}_j(w)$ (line 8). Since w is the value returned by the $\textsc{decide}$ operation of safe agreement object $\mathit{SA}[j][0]$ (line 7), we can conclude that $S = S'$. If $\textsc{nextop}_j(S) = (\textsc{read}, -)$ then $\mathit{ret} = \mathit{ret}' = (w, k')$, since (w,k') is the value returned by the $\textsc{decide}$ operation of safe agreement object $\mathit{SA}[j][1]$ (line 21); otherwise $\mathit{ret} = \mathit{ret}' = \textsc{done}$. Therefore $\sigma^{i,j}[1] = \sigma^{i',j}[1]$.

Induction Step: If $\sigma^{i,j}[k+1]$ or $\sigma^{i',j}[k+1]$ does not exist, we are done. Otherwise $\sigma^{i,j}[k]$ and $\sigma^{i',j}[k]$ also exist and, by induction hypothesis we assume that $\sigma^{i,j}[k] = \sigma^{i',j}[k] = (j, (S_k, k), \mathit{ret}_k)$. We will prove that $\sigma^{i,j}[k+1] = \sigma^{i',j}[k+1]$. Let $\sigma^{i,j}[k+1] = (j, (S, k+1), \mathit{ret})$, and $\sigma^{i',j}[k+1] = (j, (S', k+1), \mathit{ret}')$. From the code (lines 13, 22, and 25) we see that for each real process, at the beginning of the $k+1$-th iteration of the main loop in thread j, $\mathit{state}[j] = \textsc{trans}_j(S_k, \mathit{ret}_k)$. Therefore $S = \textsc{trans}_j(S_k, \mathit{ret}_k) = S'$. As in the base case, if $\textsc{nextop}_j(S) = (\textsc{read}, -)$ then $\mathit{ret} = \mathit{ret}' = (w, k')$, since (w,k') is the value returned by the $\textsc{decide}$ operation of safe agreement object $\mathit{SA}[j][k+1]$ (line 21); otherwise $\mathit{ret} = \mathit{ret}' = \textsc{done}$. Therefore $\sigma^{i,j}[k+1] = \sigma^{i',j}[k+1]$. □

Lemma 43 justifies defining the sequence of steps of each individual virtual process independent from the real process performing the simulation.

***Definition 44.** *For each virtual process j, $\sigma^j = \sigma^{i,j}$ where i is the real process such that $\sigma^{i,j}$ is of maximum length.*

Thus, $\sigma^j[k]$ (if it exists) is the k-th step of the sequence of steps of virtual process j. We denote $\sigma^j[k]$ simply as σ^j_k; so $\sigma^j = \sigma^j_1\sigma^j_2\sigma^j_3\ldots$. Note that σ^j may be finite.

Observation 45. *For each virtual process j, σ^j is locally consistent with respect to $\mathcal{A}'$.*

Recall that $\mathcal{R}$ is the history of Algorithm 5 under consideration.

Definition 46. *A step of history $\mathcal{R}$* occurs at time t *if it is the t-th step of $\mathcal{R}$, i.e., it is step $\mathcal{R}[t]$.*

Definition 47. *Consider the simulation, by some real process i, of a step in which some virtual process reads the register of virtual process j' (see Definition 39). Before i simulates that step (by executing line 21), it executes the loop in lines 15-16. The execution of this loop is called a* scan (of j')*. We say that this scan* yields (w, k') for j' *if $(w, k') =$ VMAX(M) where M is the value of array my_mem at the end of the scan. Finally, we say that the scan* occurs during interval $[t_1, t_2]$, *if t_1 and t_2 are the times of the execution of the first and last read steps of this scan.*

If a pair (w, k') appears in *MEM*$[-][j']$ during real history $\mathcal{R}$, we define the interval of time during which it is "available" from j'. Intuitively this interval starts when (w, k') is first written to *MEM*$[-][j']$ and ends when a pair with a higher timestamp is written to *MEM*$[-][j']$. More precisely,

Definition 48. *Let j' be any virtual process and (w, k') be any pair that appears in MEM$[-][j']$. We say that (w, k')* is available from virtual process j' during interval (t_1, t_2) *where t_1 and t_2 are defined as follows.*

Case 1. $k' = 0$.

$t_1 = 0$. *Let t_2 be the time of the first write to MEM$[-][j']$ by any real process, if such a write exists, and ∞ otherwise.*

Case 2. $k' > 0$.

Let t_1 be the time of the first write of (w, k') to MEM$[-][j']$ by any real process. Let t_2 be the time of the first write of some value $(-, k'')$ to MEM$[-][j']$ by any real process where $k'' > k'$, if such a write exists, and ∞ otherwise.

The next observation follows immediately by inspection of the code of Algorithm 5.

Observation 49. *For each real process i, and each virtual process j, if $MEM[i][j]$ contains (w,k) and later contains (w',k'), then $k' \geq k$.*

Lemma 50. *Suppose a scan yields (w,k') for virtual process j'. If*

(i) this scan occurs during some interval $[t'_1,t'_2]$, and

(ii) (w,k') is available from j' during interval (t_1,t_2),

then $[t'_1,t'_2]$ intersects (t_1,t_2).

Proof. Suppose, for contradiction, that a scan yields (w,k') for virtual process j', and (i) and (ii) hold, but $[t'_1,t'_2] \cap (t_1,t_2) = \emptyset$. Thus, interval $[t'_1,t'_2]$ is before or after (t_1,t_2), i.e., either $t'_2 \leq t_1$ or $t_2 \leq t'_1$.

Case 1. $t'_2 \leq t_1$. See Figure 1. From (i), the scan reads (w,k') from $MEM[-][j']$ at some time $t \leq t'_2$. Clearly, $t'_2 \geq 1$, and so $t_1 \geq 1$. Thus, from (ii) and Definition 48, $k' > 0$ and the pair (w,k') is first written in $MEM[-][j']$ at time t_1. Since $t_1 \geq t'_2$, we have $t < t_1$. So (w,k') is read from $MEM[-][j']$ at time t, before this pair is first written at time t_1 — a contradiction.

Case 2. $t_2 \leq t'_1$. See Figure 2. Since t'_1 is finite, t_2 is finite. Thus, from (ii) and Definition 48, some real process i wrote a pair $(-,k'')$ with $k'' > k'$ in $MEM[i][j']$ at time t_2. Thus, by Observation 49, the scan (which starts after time t_2) reads some pair $(-,k^*)$ such that $k^* \geq k'' > k'$ from $MEM[i][j']$. Since $k^* > k'$, the scan does *not* yield (w,k') for j' — a contradiction to (i).

□

We now define the time t^j_k that marks the beginning of the simulation of step σ^j_k (recall that σ^j_k is the k-th step of virtual process j).

Definition 51. *If step σ^j_k exists let t^j_k be the earliest time when any real process executes the first step of iteration k of the main loop of thread j, and let $t^j_k = \infty$ otherwise.*

The next fact follows immediately from the above definition.

Observation 52. *For each pair of steps σ^j_k and $\sigma^j_{k'}$ by the same virtual process j, if $k < k'$ then $t^j_k < t^j_{k'}$.*

Lemma 53. *Suppose a step σ_k^j reads (w, k') from virtual process j', and let (t_1, t_2) be the interval during which (w, k') is available from j'. Then:*

(i) There is a scan that yields (w, k') for j'.

(ii) This scan occurs during an interval of time I such that:

(a) I is a subinterval of (t_k^j, t_{k+1}^j).

(b) I intersects interval (t_1, t_2).

Proof. Suppose a step σ_k^j reads (w, k') from virtual process j'. Then the safe agreement object $SA[j][k]$ must return (w, k') to at least one real process by some time $t < t_{k+1}^j$. So at least one real process, say process i, invokes $SA[j][k].\text{PROPOSE}\Big((w, k')\Big)$ by time $t < t_{k+1}^j$; this must occur in iteration k of thread j of real process i. Consider the scan that process i does in iteration k of thread j. Clearly this scan yields (w, k') for virtual process j'. Moreover, this scan must start after time t_k^j, and it must end before process i invokes $SA[j][k].\text{PROPOSE}\Big((w, k')\Big)$, i.e., before time $t < t_{k+1}^j$. So this scan occurs during some sub-interval I of (t_k^j, t_{k+1}^j). Since the scan yields (w, k') for virtual process j', by Lemma 50, the interval I of the scan intersects the interval (t_1, t_2) during which (w, k') is available from virtual process j'. □

The above lemma immediately implies the following corollary:

Corollary 54. *If a step σ_k^j reads (w, k') from virtual process j', then there is a time $t \in (t_k^j, t_{k+1}^j) \cap (t_1, t_2)$, where (t_1, t_2) is the interval during which (w, k') is available from virtual process j'.*

To order the virtual steps of the simulated processes, we now assign an "execution time" to each one.

Definition 55. *For each step σ_k^j, the* execution time *of σ_k^j, denoted $T(\sigma_k^j)$, is defined as follows:*

Case 1. *σ_k^j is a step that writes w. Then $T(\sigma_k^j)$ is the time of the first write of (w, k) to $MEM[-][j]$ by any real process.*

Case 2. *σ_k^j is a step that reads some pair (w, k') from virtual process j'. By Corollary 54, there is a time $t \in (t_k^j, t_{k+1}^j) \cap (t_1, t_2)$, where (t_1, t_2) is the interval during which (w, k') is available from virtual process j'. Then $T(\sigma_k^j) = t$.*

CASE 3. *σ^j_k is an output step. Then $T(\sigma^j_k) = t^j_k$.*

Observation 56. *For each step σ^j_k, $T(\sigma^j_k) \in [t^j_k, t^j_{k+1})$.*

Observation 18 and the above observation immediately implies:

Observation 57. *For each pair of steps σ^j_k and $\sigma^j_{k'}$ by the same virtual process j, if $k < k'$ then $T(\sigma^j_k) < T(\sigma^j_{k'})$.*

Lemma 58. *Suppose step σ^j_k in σ^j reads (w, k') from virtual process j'.*

(i) If (w, k') is the initial value $(-, 0)$ then there is no write step $\sigma^{j'}_\ell$ of j' such that $T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$.

(ii) If (w, k') is not the initial value $(-, 0)$ then there is a write step $\sigma^{j'}_{k'}$ of j' that writes w such that $T(\sigma^{j'}_{k'}) < T(\sigma^j_k)$, and there is no write step $\sigma^{j'}_\ell$ of j' such that $T(\sigma^{j'}_{k'}) < T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$.

Proof. Suppose step σ^j_k reads (w, k') from j', and let (t_1, t_2) be the interval during which (w, k') is available from j'. Note that by the definition of T, $T(\sigma^j_k) \in (t_1, t_2)$ so $t_1 < T(\sigma^j_k) < t_2$.
(i) Suppose $(w, k') = (-, 0)$. Since $(-, 0)$ is available from j' during (t_1, t_2), $t_1 = 0$, and there is no real process that writes in $MEM[-][j']$ before time t_2 (*). Suppose, for contradiction, that there is a write step $\sigma^{j'}_\ell$ such that $T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$. Thus $T(\sigma^{j'}_\ell) < t_2$. Since $\sigma^{j'}_\ell$ is a write step, by the definition of T, some real process writes $(-, \ell)$ in $MEM[-][j']$ at time $T(\sigma^{j'}_\ell)$. Since $T(\sigma^{j'}_\ell) < t_2$, this contradicts (*).
(ii) Suppose $(w, k') \neq (-, 0)$. Since $(w, k') \neq (-, 0)$ and (w, k') is available from j' during (t_1, t_2), some real process first writes (w, k') into $MEM[-][j']$ at time t_1. So there is a step $\sigma^{j'}_{k'}$ of j' that writes w and $T(\sigma^{j'}_{k'}) = t_1$. Thus, $T(\sigma^{j'}_{k'}) < T(\sigma^j_k)$.

It remains to show that there is no write step $\sigma^{j'}_\ell$ of j' such that $T(\sigma^{j'}_{k'}) < T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$. Suppose, for contradiction, that such a step $\sigma^{j'}_\ell$ exists. Thus, $T(\sigma^{j'}_{k'}) < T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$. First note that by Observation 57, $\ell > k'$. Furthermore, since $T(\sigma^{j'}_{k'}) = t_1$ and $T(\sigma^j_k) < t_2$, we have $t_1 < T(\sigma^{j'}_\ell) < t_2$. So some real process writes a pair $(-, \ell)$ with $\ell > k'$ in $MEM[-][j']$ after time t_1 and before time t_2. This contradicts the fact that (w, k') is available from j' during interval (t_1, t_2). □

Definition 59. *The sequence of steps (of $\mathcal{A}'$) σ, is the sequence consisting of all the steps in the sequences $\sigma^1, \sigma^2, \ldots, \sigma^j, \ldots, \sigma^m$, ordered by increasing step execution times (with process ids resolving any ties).*

Lemma 60. *The sequence of steps σ is a history of $\mathcal{A}'$ that has input $(x_1, x_2, \ldots, x_m)$, where for each j, x_j is the input to some real process in $\mathcal{R}$.*

Proof. By Observation 57, for each virtual process j, the sub-sequence of σ consisting of the steps of virtual process j is $\sigma_1^j \sigma_2^j \sigma_3^j \ldots = \sigma^j$. Thus, from Observation 45, σ is locally consistent.

By Lemma 58, σ is also read consistent.

From Definition 39, lines 5, 7, 8, and Observation 57 it is clear that, for each virtual process j, if $(j, (S, 1), -)$ is the first step of j in σ, then $S = \text{INIT\_STATE}_j(w)$ where w is the input to some real process. Therefore, by Definition 66, $(S, 1) = \text{INIT\_STATE}'_j(w)$. $\square$

Lemma 61. *If d is the output of some real process i in $\mathcal{R}$, then d is the output of some virtual process j in the history σ.*

Proof. Let i be an arbitrary real process that outputs d. Then i simulates an output step, $(j, (S, k), ret)$, of some virtual process j that outputs d (see line 24). By Definitions 39 and 44, $\sigma_k^j = (j, (S, k), ret)$. By Definition 59, σ_k^j is also a step in σ and therefore d is the output value of virtual process j in σ. $\square$

Lemma 62. *The real processes output at most k different values in $\mathcal{R}$.*

Proof. Let V be the set of values output by real processes, and let V' be the set of values output by virtual processes. By Lemma 61, $V \subseteq V'$. By Lemma 60, and the agreement property of $\mathcal{A}'$, $|V'| \leq k$. Therefore $|V| \leq k$ and the real processes output at most k different values in $\mathcal{R}$. $\square$

Lemma 63. *If some real process i outputs d, then d is the input of some real process i'.*

Proof. Let i be an arbitrary real process that outputs d. By Lemma 61, d is an output of some virtual process j in history σ. By Lemma 60, and the validity property of $\mathcal{A}'$, d is the input to some real process. $\square$

***Lemma 64.** *If the simulated k-set agreement algorithm is f-resilient for m processes, at most f real processes crash in $\mathcal{R}$, and $f < m$, then all correct real processes output a value. Each real process outputs at most one value in $\mathcal{R}$.*

Proof. Let i be a correct real process. By Lemma 41 there are at least $m-f$ virtual processes such that $\sigma^{i,j}$ is infinite. Therefore, by Definition 44, and Definition 59, there are at least $m-f$ virtual processes that take infinitely many steps in history σ. Since $\mathcal{A}'$ is f-resilient and, by Lemma 60, σ is a history of $\mathcal{A}'$, by the termination property of $\mathcal{A}'$ all correct virtual processes in σ output a value.

Since i simulates an infinite number of steps of at least $m-f > 1$ virtual process, there must be a virtual process j such that i simulates and infinite number of steps of j. Therefore, j outputs in σ, and i must simulate a output step of j. When i simulates a output step of j, if i has not already outputted a value, i will output a value (line 24).

Clearly from line 24 each real process outputs at most one value. □

Theorem 65. *Algorithm 5 is an f-resilient k-set agreement algorithm for n processes.*

Proof. Follows by Lemmas 62, 63, and 64. □

5 Using Snapshots

The BG-simulation was originally formalized using snapshot objects instead of registers[4]. This does not change the power of the simulation because snapshot objects can be implemented from registers[1]. In this section we introduce snapshot objects, show how the algorithm changes when using them, and show the changes to the proof.

5.1 Snapshot Objects

A snapshot object is a shared memory object that allows for the simultaneous reading of multiple "registers." It consists of an array of fields, each of which can be written to by a different process. Process i writes a value v to its field of snapshot object MEM by invoking $MEM.\textsc{write}(i, v)$. A process may read the entire contents of a snapshot object in a single atomic step. To read the contents of snapshot object MEM a process invokes $MEM.\textsc{snapshot}()$. This returns to the process an array containing the current values in each of the fields of the snapshot object. We will use $MEM[i]$ as a shorthand to refer to the i-th field of MEM.

5.2 Snapshot Model

Now that we have introduced snapshot objects we will modify our model to account for them. We assume that the system has only a single snapshot object, with one field per process. This assumption can be made without loss of generality because it is possible to modify an algorithm that uses multiple snapshot objects so that only a single snapshot object is used by representing multiple snapshot objects as fields of a single snapshot object. First we modify the definition of the function $\text{NEXTOP}_i()$. We redefine $\text{NEXTOP}_i(S)$ so that it returns one of the following three values:

(1) SNAP,

(2) (WRITE, v) where v is a value,

(3) (OUTPUT, y) where y is an output value.

So, in a step (i, S, ret) of algorithm $\mathcal{A}$:

(1) If $\text{NEXTOP}_i(S) = \text{SNAP}$, then (i, S, ret) is a snapshot step of i that reads *ret*.

(2) If $\text{NEXTOP}_i(S) = (\text{WRITE}, v)$, then (i, S, ret) is a write step of i that writes v into i's field of the snapshot object.

(3) If $\text{NEXTOP}_i(S) = (\text{OUTPUT}, y)$, then (i, S, ret) is an output step of i that outputs y.

The other definitions remain the same except the definition of read consistency which becomes the following.

Read consistency if (i, S, ret) is a snapshot step that reads *ret*, then, for each $1 \leq i' \leq n$, $ret[i']$ is the value written by the last write step of process i' that precedes (i, S, ret) in H, if such a step exists, and $ret[i']$ is the initial value of the register of i' otherwise.

In a similar way that the single register per process model was extended to multiple registers per process, the single snapshot object per system model can be extended to multiple snapshots.

Table 2: Example of Per-Column Maximum

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
$i = 1$	$(a, 0)$	$(f, 4)$	$(e, 1)$	$(g, 0)$
$i = 2$	$(z, 9)$	$(c, 7)$	$(e, 1)$	$(g, 0)$
$i = 3$	$(f, 3)$	$(c, 7)$	$(d, 5)$	$(g, 0)$
PCMAX	$(z, 9)$	$(c, 7)$	$(d, 5)$	$(g, 0)$

5.3 Algorithm

The version of the BG-simulation that uses snapshots presented in Algorithm 6 is very similar to the register version presented in section 4.2. There are only two significant differences. First, since the real processes have access to snapshot objects, they are no longer required to scan the shared memory (lines 15-16 of Algorithm 5) but can simply perform a single snapshot operation instead (line 15 of Algorithm 6). Second, since the real processes must provide the simulated processes with a snapshot object instead of registers, when simulating a snapshot operation the most recently written value of each virtual process must be selected. This is done using the "per column maximum" PCMAX function defined here and used on line 16.

Let M be an $n \times m$ array of pairs of the form (v, k), where k is a natural number, with the property that (like MEM) any two pairs with the same second component on the same column are identical (i.e., for any i, i', j, v, v', k, if $M[i, j] = (v, k)$ and $M[i', j] = (v', k)$ then $v = v'$). Then PCMAX(M) is the array of pairs $V[1..m]$ where $V[j] = \text{VMAX}(M[-][j])$.

It is also convenient to define $V.val$ to be the array containing only the first value of each pair in V, so if $V = [(w_1, k_1), (w_2, k_2), \ldots, (w_m, k_m)]$ then $V.val = [w_1, w_2, \ldots, w_m]$.

Algorithm 6 The BG-Simulation for k-set Agreement using Snapshots
Code for process i, where $i = 1..n$

```
**Shared Memory**
$MEM$, a snapshot object with $n$ fields. For each $i$, the $i$-th field of
    $MEM$ can be written to by $i$, and is an array of $m$ variables.
    For each $j$, $MEM[i][j]$ contains a pair of the form $(val, steps)$,
    initially $(v, 0)$ where $v$ is the initial value of $j$'s field of
    the snapshot object in $\mathcal{A}$.
$SA$, an $m$ by infinite array of safe agreement objects
**Process Local Variables**
$input \in I_i$, initially the input of real process $i$
$state$, $steps$, arrays of size $m$, initially arbitrary
$mem\_local$, has the same components as $MEM[i]$, initially
    the same as $MEM[i]$
$decided$, a boolean, initially false
$safe\_agreement\_lock$, a mutual exclusion object for $m$ threads
**Thread Local Variables**
$i', j, j', \ell \in \mathbb{N}$, initially arbitrary
$d, v, o$, variables, initially arbitrary
$V$, $V\_proc$, arrays of $m$ pairs, initially arbitrary
$my\_mem$, an array of $n$ variables, initially arbitrary
```

```
procedure DECIDE($input$)
    parallel for $j = 1..m$
        $safe\_agreement\_lock$.LOCK()
        $SA[j][0]$.PROPOSE($input$)
        $safe\_agreement\_lock$.UNLOCK()
        $w \leftarrow SA[j][0]$.DECIDE()
        $state[j] \leftarrow$ INIT_STATE$_j(w)$
        $steps[j] \leftarrow 1$
        repeat forever
            if NEXTOP$_j(state[j]) = ($WRITE$, v)$ then
                $mem\_local[j] \leftarrow (v, steps[j])$
                $MEM$.WRITE($i, mem\_local$)
                $state[j] \leftarrow$ TRANS$_j(state[j],$ DONE$)$
            else if NEXTOP$_j(state[j]) =$ SNAP then
                $my\_mem \leftarrow MEM$.SNAPSHOT()
                $V\_proc \leftarrow$ PCMAX($my\_mem$)
                $safe\_agreement\_lock$.LOCK()
                $SA[j][steps[j]]$.PROPOSE($V\_proc$)
                $safe\_agreement\_lock$.UNLOCK()
                $V \leftarrow SA[j][steps[j]]$.DECIDE()
                $state[j] \leftarrow$ TRANS$_j(state[j], V.val)$
            else if NEXTOP$_j(state[j]) = ($OUTPUT$, o)$ then
                if $\neg decided$ then output $o$; $decided \leftarrow true$
                $state[j] \leftarrow$ TRANS$_j(state[j],$ DONE$)$
            $steps[j] \leftarrow steps[j] + 1$
        end
    end
```

5.4 The Modified Algorithm for Snapshots

As in section 3.3 when proving the correctness of Algorithm 6 we will construct a history not of $\mathcal{A}$ but of a modified version of $\mathcal{A}$ that writes a timestamp with each value. The code of the modified algorithm $\mathcal{A}'$ is presented in Algorithm 7

Algorithm 7 The "Modified Algorithm" $\mathcal{A}'$ for Snapshots
Code for process j, where $j = 1..m$

```
Shared Memory
$MEM$, a snapshot object with $m$ fields. For each $j$, the $j$-th field of
    $MEM$ can be written to by $i$, and is initialized to $(v_j, 0)$ where
    $v_j$ is the initial value of $j$'s field of the snapshot object in $\mathcal{A}$
procedure DECIDE($input$)
    $state \leftarrow$ INIT_STATE$_j(input)$
    $steps \leftarrow 1$
    repeat forever
        if NEXTOP$_j(state) = ($WRITE$, v)$ then
            $MEM$.WRITE$(i, (v, steps))$
            $state \leftarrow$ TRANS$_j(state,$ DONE$)$
        else if NEXTOP$_j(state) =$ SNAP then
            $V \leftarrow MEM[j']$.SNAPSHOT()
            $state \leftarrow$ TRANS$_j(state, V.val)$
        else if NEXTOP$_j(state) = ($OUTPUT$, o)$ then
            output $o$
            $state \leftarrow$ TRANS$_j(state,$ DONE$)$
        $steps \leftarrow steps + 1$
    end
```

Formally the specification of $\mathcal{A}'$ is obtained form the specification of $\mathcal{A}$ as follows.

- The field of the snapshot object belonging to j is initialized to $(v_j, 0)$, where v_j is the initial value of j's field of the snapshot object in $\mathcal{A}$.
- The states of $\mathcal{A}'$ are of the form (S, k) where S is a state of $\mathcal{A}$ and k is a counter (that tracks the number of steps executed).
- The inputs of $\mathcal{A}'$ are the same as the inputs of $\mathcal{A}$.
- The functions INIT_STATE$'_j$, NEXTOP$'_j$, and TRANS$'_j$ are defined below.

Definition 66.

$$\text{INIT\_STATE}'_j(\mathit{input}) = \Big(\text{INIT\_STATE}_j(\mathit{input}), 1\Big)$$

$$\text{NEXTOP}'_j\Big((S,k)\Big) = \begin{cases} \Big(\text{WRITE}, (v,k)\Big), & \text{if } \text{NEXTOP}_j(S) = (\text{WRITE}, v) \\ \text{NEXTOP}_j(S), & \text{if } \mathit{otherwise} \end{cases}$$

$$\text{TRANS}'_j\Big((S,k), \mathit{ret}\Big) = \begin{cases} \Big(\text{TRANS}_j(S, \mathit{ret}), k+1\Big). & \text{if } \mathit{ret} = \text{DONE} \\ \Big(\text{TRANS}_j(S, V.\mathit{val}), k+1\Big), & \text{if } \mathit{ret} = V \end{cases}$$

Observation 67. *If $\mathcal{A}$ is an f-resilient algorithm for a task T, then the modified algorithm $\mathcal{A}'$ is also an f-resilient algorithm for task T.*

5.5 Proof of the Correctness of the BG-Simulation with Snapshots

The proof in this section is very similar to the proof in section 4.3. The only changes, aside from minor things like references to line numbers, have to do with the differences between registers and snapshot objects and the selection of execution times. The following definitions and lemmas contain significant changes: Definitions 77, 79, and 85; Lemmas 81, 84, and 88. These definitions and lemmas are marked with a *.

Given any k-set agreement algorithm $\mathcal{A}$, we now describe how to simulate its counterpart $\mathcal{A}'$ (which, by Observation 67 also solves k-set agreement). Our goal is the following: given an arbitrary history of Algorithm 6, construct a history of $\mathcal{A}'$ that is used by the real processes to solve k-set agreement. We begin with some simple definitions.

Definition 68. *Let I be the set of real processes and J be the set of virtual processes. $|I| = n$, $|J| = m$.*

Throughout this section i and i' will denote real processes in I, and j and j' will denote virtual processes in J.

Throughout this section we will consider an arbitrary history $\mathcal{R}$ of Algorithm 1 by the real processes in I. All mentions of the steps of *real* processes refer to their steps in $\mathcal{R}$. We now define what it means for a real process to simulate a step of a virtual process.

Definition 69. *Consider iteration k of thread j by real process i if it exists. Let S be the value of $state[j]$ at the beginning of this iteration. There are three possible cases:*

- $\text{NEXTOP}(S) = (\text{WRITE}, v)$. *If real process i executes line 12 in iteration k, we say that process i simulates step $(j, (S, k), \text{DONE})$ of $\mathcal{A}'$; we also say that this is a write step in which virtual process j writes (v, k).*
- $\text{NEXTOP}(S) = \text{SNAP}$. *If real process i executes line 20 in iteration k and gets V from safe agreement object $SA[j][k]$, we say that process i simulates step $(j, (S, k), V)$ of $\mathcal{A}'$; we also say that this is a read step in which virtual process j snaps V.*
- $\text{NEXTOP}(S) = (\text{OUTPUT}, o)$. *If real process i executes line 24 in iteration k we say that process i simulates step $(j, (S, k), \text{DONE})$ of $\mathcal{A}'$; we also say that this is an output step in which virtual process j outputs o.*

Definition 70. *Let $\sigma^{i,j}$ be the sequence of steps of $\mathcal{A}'$ that real process i simulates for virtual process j, in the order that i simulates them.*

We will now show that each real process i simulates an infinite sequence of steps for at least $m - \hat{f}$ virtual processes, where $\hat{f}$ is the number of real processes that crash in real history $\mathcal{R}$.

Lemma 71. *Suppose $\hat{f}$ real processes crash in real history $\mathcal{R}$. For each correct real process i there are at least $m - \hat{f}$ virtual processes j such that $\sigma^{i,j}$ is infinite.*

Proof. The only blocking statements in the main loop (lines 9 to 26) are the decide operations on lines 6 and 20. By the Termination property of safe agreement, these lines will block only if some real process invokes PROPOSE on the same safe agreement object as the DECIDE operation, and crashes during this invocation. Since each instance of PROPOSE is guarded by the use of mutual exclusion object *safe_agreement_lock*, a real process may crash while at most one thread is executing one of lines 4 or 18. Therefore, for each crashed real process there is a single safe agreement object whose invocations of DECIDE may block. Since at most $\hat{f}$ real processes may crash and no two threads use the same safe agreement object, at most $\hat{f}$ threads may be blocked.

Let i be any correct real process. Under the assumption that the threads are scheduled in a fair way, the main loop in at least $m - \hat{f}$ threads of i will be executed an infinite number of times. Therefore i will simulate an infinite number of steps for at least $m-\hat{f}$ virtual processes, which, by the definition of $\sigma^{i,j}$, implies that there are at least $m-\hat{f}$ virtual processes j for which $\sigma^{i,j}$ is infinite. $\square$

Lemma 72. *For each real process i and each virtual process j, $\sigma^{i,j}$ is locally consistent with respect to $\mathcal{A}'$.*

Proof. Consider any two consecutive steps $\sigma^{i,j}[k]$ and $\sigma^{i,j}[k+1]$ of $\sigma^{i,j}$. By Definition 69,

- $\sigma^{i,j}[k] = (j, (S, k), \mathit{ret})$ where S is the value of *state*$[j]$ at the start of iteration k of thread j by real process i, and *ret* is the vector of pairs V returned by $CO[j][k]$ (if $\sigma^{i,j}[k]$ is a snapshot step) or DONE (if $\sigma^{i,j}[k]$ is a write or output step); and
- $\sigma^{i,j}[k+1] = (j, (S', k+1), -)$, where S' is the value of *state*$[j]$ at the start of iteration $k+1$ of thread j by real process i.

We must prove that $(S', k+1) = \textsc{trans}'_j\Big((S,k), \mathit{ret}\Big)$.

By Definition 66,

$$\textsc{trans}'_j\Big((S,k),\mathit{ret}\Big) = \begin{cases} \Big(\textsc{trans}_j(S, \textsc{done}), k+1\Big), & \text{if } \mathit{ret} = \textsc{done} \\ \Big(\textsc{trans}_j(S, V.\mathit{val}), k+1\Big), & \text{if } \mathit{ret} = V \end{cases}$$

By lines 13, 21, and 24,

$$S' = \begin{cases} \textsc{trans}_j(S, \textsc{done}), & \text{if } \mathit{ret} = \textsc{done} \\ \textsc{trans}_j(S, V.\mathit{val}), & \text{if } \mathit{ret} = V \end{cases}$$

Therefore, $\textsc{trans}'_j\Big((S,k),\mathit{ret}\Big) = (S', k+1)$, as wanted. $\square$

We will now show that the real processes simulate the *same* sequence for every virtual process.

Lemma 73. *For each pair of real processes i and i', each virtual process j, and each $k \in \mathbb{N}$, if steps $\sigma^{i,j}[k]$ and $\sigma^{i',j}[k]$ exist, then $\sigma^{i,j}[k] = \sigma^{i',j}[k]$.*

Proof. Let i and i' be arbitrary real processes. We prove the lemma by induction on k.

Base Case: $k=1$. If $\sigma^{i,j}[1]$ or $\sigma^{i',j}[1]$ does not exist, we are done; otherwise let $\sigma^{i,j}[1] = (j, (S,1), \mathit{ret})$, and $\sigma^{i',j}[1] = (j, (S',1), \mathit{ret}')$. At the

beginning of the first iteration of the main loop in thread j by any real process, $\mathit{state}[j] = \textsc{init\_state}_j(w)$ (line 7). Since w is the value returned by the $\textsc{decide}$ operation of safe agreement object $\mathit{SA}[j][0]$ (line 6), we can conclude that $S = S'$. If $\textsc{nextop}_j(S) = (\textsc{read}, -)$ then $\mathit{ret} = \mathit{ret}' = V$, since V is the value returned by the $\textsc{decide}$ operation of safe agreement object $\mathit{SA}[j][1]$ (line 20); otherwise $\mathit{ret} = \mathit{ret}' = \textsc{done}$. Therefore $\sigma^{i,j}[1] = \sigma^{i',j}[1]$.
Induction Step: If $\sigma^{i,j}[k+1]$ or $\sigma^{i',j}[k+1]$ does not exist, we are done. Otherwise $\sigma^{i,j}[k]$ and $\sigma^{i',j}[k]$ also exist and, by induction hypothesis we assume that $\sigma^{i,j}[k] = \sigma^{i',j}[k] = (j, (S_k, k), \mathit{ret}_k)$. We will prove that $\sigma^{i,j}[k+1] = \sigma^{i',j}[k+1]$. Let $\sigma^{i,j}[k+1] = (j, (S, k+1), \mathit{ret})$, and $\sigma^{i',j}[k+1] = (j, (S', k+1), \mathit{ret}')$. From the code (lines 13, 21, and 24) we see that for each real process, at the beginning of the $k+1$-th iteration of the main loop in thread j, $\mathit{state}[j] = \textsc{trans}_j(S_k, \mathit{ret}_k)$. Therefore $S = \textsc{trans} S_k, \mathit{ret}_k = S'$. As in the base case, if $\textsc{nextop}_j(S) = (\textsc{read}, -)$ then $\mathit{ret} = \mathit{ret}' = V$, since V is the value returned by the $\textsc{decide}$ operation of safe agreement object $\mathit{SA}[j][k+1]$ (line 20); otherwise $\mathit{ret} = \mathit{ret}' = \textsc{done}$. Therefore $\sigma^{i,j}[k+1] = \sigma^{i',j}[k+1]$. □

Lemma 43 justifies defining the sequence of steps of each individual virtual process independent from the real process performing the simulation.

Definition 74. *For each virtual process j, $\sigma^j = \sigma^{i,j}$ where i is the real process such that $\sigma^{i,j}$ is of maximum length.*

Thus, $\sigma^j[k]$ (if it exists) is the k-th step of the sequence of steps for virtual process j. We denote $\sigma^j[k]$ simply as σ^j_k; so $\sigma^j = \sigma^j_1 \sigma^j_2 \sigma^j_3 \ldots$. Note that σ^j may be finite.

Observation 75. *For each virtual process j, σ^j is locally consistent with respect to $\mathcal{A}'$.*

Recall that $\mathcal{R}$ is the history of Algorithm 6 under consideration.

Definition 76. *A step of history $\mathcal{R}$* occurs at time t *if it is the t-th step of $\mathcal{R}$, i.e., it is step $\mathcal{R}[t]$.*

***Definition 77.** *A snapshot operation executed by a real process (line 15)* yields V, *if $V = \textsc{pcmax}(\mathit{MEM})$ at the time the snapshot is executed.*

Note that if a snapshot yields V, then for all virtual processes j', $V[j']$ is of the form (w, k'), and (w, k') must be in $\mathit{MEM}[i][j']$ for some i at the time of the snapshot. If a pair (w, k') appears in $\mathit{MEM}[-][j']$ during real history $\mathcal{R}$, we define the interval of time during which it is "available" from

j'. Intuitively this interval starts when (w, k') is first written to $MEM[-][j']$ and ends when a pair with a higher timestamp is written to $MEM[-][j']$. More precisely,

Definition 78. *Let j' be any virtual process and (w, k') be any pair that appears in $MEM[-][j']$. We say that (w, k')* is available from virtual process j' during interval (t_1, t_2) *where t_1 and t_2 are defined as follows.*

Case 1. $k' = 0$.
$t_1 = 0$. *Let t_2 be the time of the first write to $MEM[-][j']$ by any real process, if such a write exists, and ∞ otherwise.*

Case 2. $k' > 0$.
Let t_1 be the time of the first write of (w, k') to $MEM[-][j']$ by any real process. Let t_2 be the time of the first write of some value $(-, k'')$ to $MEM[-][j']$ by any real process where $k'' > k'$, if such a write exists, and ∞ otherwise.

We now define the availability of an array of pairs V that a snapshot operation yields. Intuitively, it is the intersection of the availability intervals of all the pairs in V (see Figure 4).

***Definition 79.** *Suppose that:*

(a) a snapshot executed by a real process (line 15) yields V, and

(b) for each j, $1 \leq j \leq m$, $V[j]$ is available during some interval of time I_j.

Then we say that V is available during interval $I = I_1 \cap I_2 \cap \ldots I_j \cap \ldots I_m$.

The next observation follows immediately from the code of Algorithm 6.

Observation 80. *For each real process i, and each virtual process j, if $MEM[i][j]$ contains (w, k) and later contains (w', k'), then $k' \geq k$.*

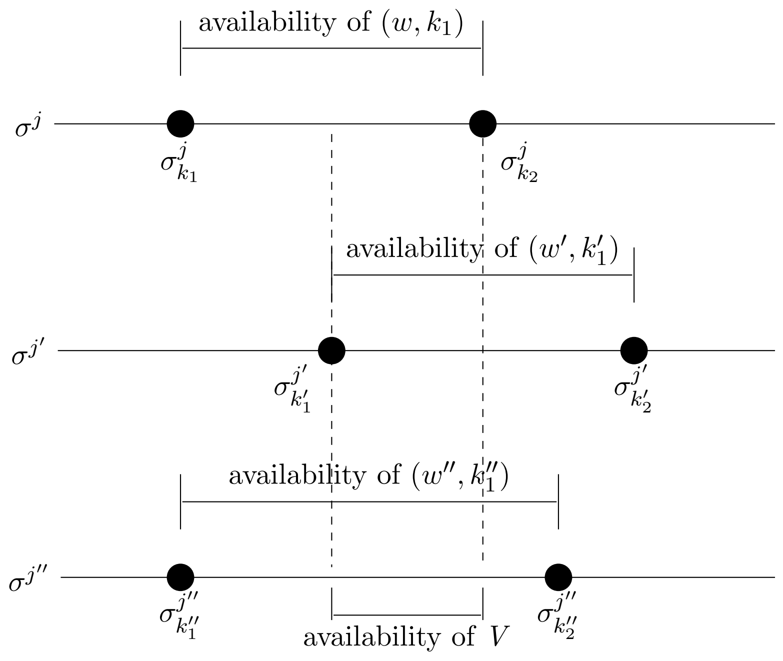
***Lemma 81.** *Suppose that a snapshot yields V. If*

(i) this snapshot occurs at time t, and

(ii) V is available during interval (t_1, t_2),

then t is in (t_1, t_2).

Proof. Suppose, for contradiction, that a snapshot yields V, and(i) and (ii) hold, but $t \notin (t_1, t_2)$. Thus either $t \leq t_1$ or $t_2 \leq t$.



- ◦ $\sigma^j_{k_1}$ is a write step that writes w.
- ◦ $\sigma^j_{k_2}$ is the next write step by j.
- ◦ $\sigma^{j'}_{k'_1}$ is a write step that writes w'.
- ◦ $\sigma^{j'}_{k'_2}$ is the next write step by j'.
- ◦ $\sigma^{j''}_{k''_1}$ is a write step that writes w''.
- ◦ $\sigma^{j''}_{k''_2}$ is the next write step by j''.
- ◦ $V = [(w, k_1), (w', k'_1), (w'', k''_1)]$.

Figure 4: The Availability of V.

CASE 1. $t \leq t_1$. Clearly, $t \geq 1$, and so $t_1 \geq 1$. Thus, from (ii) and Definition 79, there is a j' such that the pair $V[j']$ is first written in $MEM[-][j']$ at time t_1. From (i), the snapshot reads $V[j']$ from $MEM[-][j']$ at time t. So $V[j']$ is read from $MEM[-][j']$ at time t, before this pair is first written at time t_1 — a contradiction.

CASE 2. $t_2 \leq t$. Since t is finite, t_2 is finite. Thus, from (ii) and Definition 79, there is a j', such that $V[j'] = (w, k')$ some real process i wrote a pair $(-, k'')$ in $MEM[i][j']$ at time t_2 where $k'' > k'$. Thus, by Observation 80, the snapshot (which occurs after time t_2) reads some pair $(-, k^*)$ such that $k^* \geq k'' > k'$ from $MEM[i][j']$. Since $k^* > k'$, the snapshot does *not* yield V — a contradiction to (i).

□

We now define the time t_k^j that marks the beginning of the simulation of step σ_k^j (recall that σ_k^j is the k-th step of virtual process j).

Definition 82. *If step σ_k^j exists let t_k^j be the earliest time when any real process executes the first step of iteration k of the main loop of thread j, and let $t_k^j = \infty$ otherwise.*

The next fact follows immediately from the above definition.

Observation 83. *For each pair of steps σ_k^j and $\sigma_{k'}^j$ by the same virtual process j, if $k < k'$ then $t_k^j < t_{k'}^j$.*

***Lemma 84.** *Suppose a step σ_k^j reads V, and let (t_1, t_2) be the interval during which V is available. Then:*

(i) There is a snapshot that yields V.

(ii) This snapshot occurs at a time $t \in (t_k^j, t_{k+1}^j) \cap (t_1, t_2)$.

Proof. Suppose a step σ_k^j reads V. Then the safe agreement object $SA[j][k]$ must return V to at least one real process by some time $t < t_{k+1}^j$. So at least one real process, say process i, invokes $SA[j][k]$.PROPOSE(V) by time $t' < t_{k+1}^j$; this must occur in iteration k of thread j of real process i. Consider the snapshot that process i does in iteration k of thread j. Clearly this snapshot yields V. Moreover, this snapshot must occur after time t_k^j, and before process i invokes $SA[j][k]$.PROPOSE(V), i.e., before time $t' < t_{k+1}^j$. So this snapshot occurs at some time t in (t_k^j, t_{k+1}^j). Since the snapshot yields V, by Lemma 81, t is also in the interval (t_1, t_2) during which V is available. □

To order the virtual steps of the simulated processes, we now assign an "execution time" to each one.

***Definition 85.** *For each step σ^j_k, the* execution time of σ^j_k, *denoted $T(\sigma^j_k)$, is defined as follows:*

CASE 1. *σ^j_k is a step that writes w. Then $T(\sigma^j_k)$ is the time of the first write of (w, k) to $MEM[-][j]$ by any real process.*

CASE 2. *σ^j_k is a step that reads array V. By Lemma 84, there is a time $t \in (t^j_k, t^j_{k+1}) \cap (t_1, t_2)$, where (t_1, t_2) is the interval during which V is available. Let $T(\sigma^j_k) = t$.*

CASE 3. *σ^j_k is an output step. Then $T(\sigma^j_k) = t^j_k$.*

Observation 86. *For each step σ^j_k, $T(\sigma^j_k) \in [t^j_k, t^j_{k+1})$.*

Observation 18 and the above observation immediately implies:

Observation 87. *For each pair of steps σ^j_k and $\sigma^j_{k'}$ by the same virtual process j, if $k < k'$ then $T(\sigma^j_k) < T(\sigma^j_{k'})$.*

***Lemma 88.** *Suppose step σ^j_k in σ^j reads V. Let j' be an arbitrary virtual process and suppose $V[j'] = (w, k')$.*

(i) If (w, k') is the initial value $(-, 0)$ then thehre is no write step $\sigma^{j'}_\ell$ of j' such that $T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$.

(ii) If (w, k') is not the initial value $(-, 0)$ then there is a write step $\sigma^{j'}_{k'}$ of j' that writes w such that $T(\sigma^{j'}_{k'}) < T(\sigma^j_k)$, and there is no write step $\sigma^{j'}_\ell$ of j' such that $T(\sigma^{j'}_{k'}) < T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$.

Proof. Suppose step σ^j_k reads V. Let I be the interval during which V is available. Recall that $I = I_1 \cap I_2 \cap \dots I_{j'} \cap \dots I_m$ where, for each virtual process j', $I_{j'}$ is the interval during which the pair $V[j']$ is available. Let j' be an arbitrary virtual process, and suppose that $V[j'] = (w, k')$ and $I_{j'} = (\tau_1, \tau_2)$. Note that by the definition of T, $T(\sigma^j_k) \in I$. So $T(\sigma^j_k) \in I_{j'} = (\tau_1, \tau_2)$ and $\tau_1 < T(\sigma^j_k) < \tau_2$.

(i) Suppose $(w, k') = (-, 0)$. Since $(-, 0)$ is available from j' during (τ_1, τ_2), $\tau_1 = 0$, and there is no real process that writes in $MEM[-][j']$ before time τ_2 (*). Suppose, for contradiction, that there is a write step $\sigma^{j'}_\ell$ such that

$T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$. Thus $T(\sigma^{j'}_\ell) < \tau_2$. Since $\sigma^{j'}_\ell$ is a write step, by the definition of T, some real process writes $(-, \ell)$ in $\mathit{MEM}[-][j']$ at time $T(\sigma^{j'}_\ell)$. Since $T(\sigma^{j'}_\ell) < \tau_2$, this contradicts (*).

(ii) Suppose $(w, k') \neq (-, 0)$. Since $(w, k') \neq (-, 0)$ and (w, k') is available from j' during (τ_1, τ_2), some real process first writes (w, k') into $\mathit{MEM}[-][j']$ at time τ_1. So there is a step $\sigma^{j'}_{k'}$ of j' that writes w and $T(\sigma^{j'}_{k'}) = \tau_1$. Thus, $T(\sigma^{j'}_{k'}) < T(\sigma^j_k)$.

It remains to show that there is no write step $\sigma^{j'}_\ell$ of j' such that $T(\sigma^{j'}_{k'}) < T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$. Suppose, for contradiction, that such a step $\sigma^{j'}_\ell$ exists. Thus, $T(\sigma^{j'}_{k'}) < T(\sigma^{j'}_\ell) \leq T(\sigma^j_k)$. First note that by Observation 87, $\ell > k'$. Furthermore, since $T(\sigma^{j'}_{k'}) = \tau_1$ and $T(\sigma^j_k) < \tau_2$, we have $\tau_1 < T(\sigma^{j'}_\ell) < \tau_2$. So some real process writes a pair $(-, \ell)$ with $\ell > k'$ in $\mathit{MEM}[-][j']$ after time τ_1 and before time τ_2. This contradicts the fact that (w, k') is available from j' during interval (τ_1, τ_2). $\square$

Definition 89. *The sequence of steps (of $\mathcal{A}'$) σ is the sequence consisting of all the steps in the sequences $\sigma^1, \sigma^2, \ldots, \sigma^j, \ldots, \sigma^m$, ordered by increasing step execution times (with process ids resolving any ties).*

Lemma 90. *The sequence of steps σ is a history of $\mathcal{A}'$ that has input $(x_1, x_2, \ldots, x_m)$, where for each j, x_j is the input to some real process in $\mathcal{R}$.*

Proof. By Observation 87, for each virtual process j, the sub-sequence of σ consisting of the steps of virtual process j is $\sigma^j_1\sigma^j_2\sigma^j_3\ldots = \sigma^j$. Thus, from Observation 75, σ is locally consistent.

By Lemma 88, σ is also read consistent.

From Definition 69, lines 5, 7, 8, and Observation 87 it is clear that, for each virtual process j, if $(j, (S, 1), -)$ is the first step of j in σ, then $S = \text{INIT\_STATE}_j(w)$ where w is the input to some real process. Therefore, by Definition 66, $(S, 1) = \text{INIT\_STATE}'_j(w)$. $\square$

Lemma 91. *If d is the output of some real process i in $\mathcal{R}$, then d is the output of some virtual process j in the history σ.*

Proof. Let i be an arbitrary real process that outputs d. Then i simulates an output step, $(j, (S, k), \mathit{ret})$, of some virtual process j that outputs d (see line 23). By Definitions 69 and 74, $\sigma^j_k = (j, (S, k), \mathit{ret})$. By Definition 89, σ^j_k is also a step in σ and therefore d is the output value of virtual process j in σ. $\square$

Lemma 92. *The real processes output at most k different values in $\mathcal{R}$.*

Proof. Let V be the set of values output by real processes, and let V' be the set of values output by virtual processes. By Lemma 91, $V \subseteq V'$. By Lemma 90, and the agreement property of $\mathcal{A}'$, $|V'| \leq k$. Therefore $|V| \leq k$ and the real processes output at most k different values in $\mathcal{R}$. □

Lemma 93. *If some real process i outputs d, then d is the input of some real process i'.*

Proof. Let i be an arbitrary real process that outputs d. By Lemma 91, d is an output of some virtual process j in history σ. By Lemma 90, and the validity property of $\mathcal{A}'$, d is the input to some real process. □

Lemma 94. *If the simulated k-set agreement algorithm is f-resilient for m processes, at most f real processes crash in $\mathcal{R}$, and $f < m$, then all correct real processes output a value. Each real process outputs at most one value.*

Proof. Let i be a correct real process. By Lemma 71 there are at least $m-f$ virtual processes such that $\sigma^{i,j}$ is infinite. Therefore, by Definition 74, and Definition 89, there are at least $m-f$ virtual processes that take infinitely many steps in history σ. Since $\mathcal{A}'$ is f-resilient and, by Lemma 90, σ is a history of $\mathcal{A}'$, by the termination property of $\mathcal{A}'$, all correct virtual processes in σ output a value.

Since i simulates an infinite number of steps of at least $m-f > 1$ virtual process, there must be a virtual process j such that i simulates and infinite number of steps of j. Therefore, j outputs in σ, and i must simulate a output step of j. When i simulates a output step of j, if i has not already outputted a value, i will output a value (line 23).

Clearly from line 23 each real process outputs at most one value. □

Theorem 95. *Algorithm 6 is an f-resilient k-set agreement algorithm for n processes.*

Proof. Follows by Lemmas 92, 93, and 94. □

6 Concluding Remarks

6.1 A note on resiliency

In our model, a process was defined as crashed if it takes a finite number of steps in an infinite history. This includes a process that takes no steps.

Under this definition, a process that takes no steps at all is classified as faulty. Although this definition is sensible in some contexts (e.g., when we are guaranteed that all processes must participate in solving a task), it is inappropriate in other contexts. For example, it could be that a process takes no steps not because of a crash but because it has no reason to participate in solving a task. For such contexts, an appropriate definition of a faulty process is one that takes a finite but, non-zero, number of steps in an infinite history.

An algorithm that tolerates up to f faulty processes under the more inclusive definition of faulty process (where non-participating processes are classified as faulty), is called *weakly f-resilient*. An algorithm that tolerates the failure of up to f faulty process under the more restrictive definition (where only participating process can be faulty) is called *strongly f-resilient*

If we begin with a strongly resilient algorithm, then there is no need for the BG-simulation. Consider a strongly f-resilient algorithm that solves k-set agreement among m processes. It is possible to use this same algorithm to solve k-set agreement among $n < m$ processes with up to f failures. This is because the "missing" $m - n$ processes will simply be considered non participating by the algorithm.

It is worth noting that the BG-simulation actually transforms a *weakly* f-resilient algorithm into a *strongly* f-resilient algorithm. This is because any non-participating real processes cannot block the simulation of a virtual process; only a real process that crashes during the invocation of a safe agreement PROPOSE operation can block a virtual process.

6.2 Reductions and the Extended BG-Simulation

In the previous sections we focused on the BG-simulation as it was originally used, that is, to solve f-resilient k-set agreement for n processes by simulating a f-resilient solution to k-set agreement among $k + 1$ processes [3]. However, it is possible to use the BG-simulation more generally. Any algorithm that uses only SWMR registers can be simulated by the BG-simulation provided that (a) there is a way to generate inputs for the virtual processes from the inputs of the real processes and (b) there is a way for the real processes to use the outputs of the virtual processes to generate their own outputs.

The nature of the BG-simulation imposes limits on how the inputs to the simulated algorithm can be chosen and how the outputs can be used. When deciding on what value to propose as the input to virtual process j, each real process may choose any value based on their input and the id of the virtual process. This means that each virtual process can have up to n different values proposed as its input. Due to the nature of the safe agreement objects used to decide from among these proposals the actual input to each virtual process is chosen arbitrarily from among the proposals. When deciding on an output, each real process can wait for the output from at most $m - f$ virtual processes to make its decision. Since there is no way to tell which of the m virtual processes simulation may be blocked, real processes must be able to choose an output value based on the outputs of an arbitrary set of $m - f$ virtual processes.

These restrictions are formalized as *fault-tolerant reducibility* in [4]. In essence, the BG-simulation is restricted to working with so called colourless tasks. These are tasks where any process may adopt the input or output of any other process and the task will still be solved. The prototypical example of a colourless task is k-set agreement. Other colourless tasks include consensus and uncoloured simplex agreement[7].

The *extended BG-simulation* [6] is a variant of the BG-simulation designed to expand the applicability of the approach beyond the class of colourless tasks. The extended BG-simulation guarantees that the input to *virtual* process i is proposed by *real* process i and that *real* process i receives the output from *virtual* process i. In [6] it is used to prove some results about renaming [2], a task that is not colourless. In renaming, processes begin with unique names from a large name space and must choose a unique name from a smaller name space.

6.3 Future Work

The BG-simulation is not the only simulation that allows a number of processes to simulate a different number of processes while preserving the number of failures. In [5] a simulation is introduced that provides similar functionality to that of the BG-simulation, except that it uses test-and-set objects and allows the simulated algorithm to use any kind of objects, not only SWMR registers (or objects that can be implemented using them). It is simpler to understand the BG-simulation. To simulate m processes the real processes use m “critical sections” guarded by test-and-set objects. Each real process attempts to access the critical sections in a round robin fashion. When a real process enters critical section j it simulates a single step of process j. To simulate a step of virtual process j a real process accesses any shared memory object on behalf of virtual process j and updates the state of j in a shared register. If a real process crashes while in critical section j it blocks the simulation of virtual process j.

An algorithm that simulates another algorithm by using critical sections in this way solves the following problem, which we call the "multiple critical section" for n process, f of which may crash: given $m > n$ critical sections, processes must execute at least $m - f$ of them infinitely many times, with at most one process executing any single critical section at a time.

If it is possible to solve the multiple critical sections problem using registers then the simulation of [5] may be used in place of the BG-simulation.

This problem is solvable using registers when $m > 2f$ using the "safe" implementation of test-and-set derived from registers presented in [10]. In this implementation of test-and-set if a process crashes in the critical section, at most one other process may be blocked. A single process is required to wait in "the doorway" of a locked critical section to see if the crashed process "wakes up" and the critical section becomes available again. When another process attempts to access a critical section that already has a waiting process it can move on to try to execute the next critical sections in the round robin. This means that if f processes crash in critical sections another f processes may be blocked. If there is a total of $2f + 1$ processes then there will always be an unblocked process that can continue to execute the remaining unblocked critical sections.

The multiple critical section problem cannot be solved using registers when $n = 2$ and $f = 1$. This is because there is a shared memory object *dor* defined in [10] which can be used (along with registers) to solve 1-resilient consensus among three processes but not among two processes. If the multiple critical section problem was solvable when $n = 2$ and $f = 1$ then two processes could solve one resilient consensus using the *dor* object by simulating three processes.

The multiple critical section problem remains open for other values of n and f.

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic Snapshots of Shared Memory. *J. ACM*, 40(4):873–890, 1993.

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

[3] Elizabeth Borowsky and Eli Gafni. Generalized FLP Impossibility Result for t-Resilient Asynchronous Computations. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 91–100, 1993.

[4] Elizabeth Borowsky, Eli Gafni, Nancy Lynch, and Sergio Rajsbaum. The BG Distributed Simulation Algorithm. *Distrib. Comput.*, 14:127–146, 2001.

[5] Tushar Deepak Chandra, Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. Generalized Irreducibility of Consensus and the Equivalence of t-Resilient and Wait-Free Implementations of Consensus. *SIAM J. Comput.*, 34(2):333–357, 2004.

[6] Eli Gafni. The Extended BG-simulation and the Characterization of t-Resiliency. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 85–92, 2009.

[7] Maurice Herlihy and Sergio Rajsbaum. The Decidability of Distributed Decision Tasks. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 589–598, 1997.

[8] Damien Imbs and Michel Raynal. Visiting Gafni's Reduction Land: from the BG Simulation to the Extended BG Simulation. Rapport de recherche PI 1931, 2009.

[9] Marcos Kawazoe Aguilera and Sam Toueg. Borowsky-Gafni Simulation in Simple Pseudo Code. 2011. Personal Communication.

[10] Wai-Kau Lo and Vassos Hadzilacos. On the Power of Shared Object Types to Implement One-resilient Consensus. *Distrib. Comput.*, 13(4):219–238, 2000.