The Extended BG-Simulation and the Characterization of t-Resiliency

Eli Gafni
Computer Science Department, University of California, Los Angeles.
eli@ucla.edu

ABSTRACT

A distributed task T on n processors is an input/output relation between a collection of processors' inputs and outputs. While all tasks are solvable if no processor may ever crash, the FLP result revealed that the possibility of a failure of just a single processor precludes a solution to the task of consensus. That is consensus is not solvable 1-resiliently. Yet, some nontrivial tasks are wait-free solvable, i.e. $n-1$-resiliently. What tasks are solvable if at most $t < n$ processors may crash? I.e. what tasks are solvable t-resiliently?

The Herlihy-Shavit condition characterizes *wait-free* solvability, i.e., when $t = n-1$. The Borowsky-Gafni (BG) simulation extends this characterization to the t-resilient case for the case "colorless" tasks - tasks like consensus in which one processor can adopt the output of any other processor. It does this by reducing questions about t-resilient solvability, to a question of wait-free solvability. The latter question has been characterized.

In this paper, we amend the BG-simulation to result in the Extended-BG-simulation, an extension that yields a full characterization of t-resilient solvability: A task T on n processors is solvable t-resiliently iff all tasks T' on $t+1$ simulators $s_0, \ldots, s_t$ created as follows are wait-free solvable. Simulator s_i is given an input of processor p_i as well as the input to a set of processors of size $n-(t+1)$ with ids higher than i. Simulator s_i outputs for p_i as well as for a (possibly different) set of processors of size $n-(t+1)$ with ids higher than i. The input/output of the $t+1$ simulators have to be a projection of a single original input/output tuple-pair in T.

We demonstrate the convenience that the characterization provides, in two ways. First, we prove a new equivalence result: We show that n processes can solve t-resiliently weak renaming with $n+(t+1)-2$ names, where $n > 1$ and $0 < t < n$, iff weak-renaming on $t+1$ processors is wait-free solvable with $2t$ names. Second, we reproduce the result that the solvability of n-processors tasks, t-resiliently, for $t > 1$ and $n > 2$, is undecidable, by a simple reduction to the undecidability of the wait-free solvability of 3-processors tasks.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*distributed networks*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*relations between models*

General Terms

Algorithms, Performance, Theory

Keywords

Wait-freedom, t-resilience, solvability, renaming, symmetry breaking, decidability

1. INTRODUCTION

The most celebrated result of distributed computing, the FLP impossibility result [9], is couched in the domain of resiliency: Any number of processors, communicating asynchronously by message-passing, with the possibility of at most one faulty (1-resiliently), cannot asynchronously reach consensus. The result applies to Shared-Memory (SM) too [4].

It may be thought that in SM 2 processors with one faulty cannot reach consensus, since it is "one against one;" yet, 100 processors with the possibility of just one faulty, then with the overwhelming majority of 99 non-faulty processors they could reach consensus. The FLP result proved they cannot.

The FLP proves the result about 100 processors with 1 faulty directly. It has to deal with "fair executions" and other cluttering details. Proving FLP for 2 processors in SM is clean and short [4]. Can the FLP impossibility be reduced to the impossibility of 2 processors solving consensus wait-free in SM?

The BG-Simulation [2, 6] does just that.

It reduces a class of questions about resiliency to questions about wait-free (i.e. $n-1$-resiliency). It can be used to prove FLP in the SM context by proving the impossibility for 2 processors, and then showing that the existence of r/w protocol that can solve consensus for 100 processors 1-resiliently, will imply a wait-free consensus protocol for 2 processors, leading to a contradiction.

The idea of the reduction is to take 2 simulators s_0 and s_1. If there exist a code $code_0$, $code_1$,..., $code_{99}$, for 100 processors that solves consensus 1-resiliently, then w.l.o.g there exists an input and an execution e_0 (in which only 99 processors are induced to participate) that results a decision 0, and execution e_1 (in which all processors are induced to participate) that results a decision value 1 [9]. Simulator $s_i,\ i = 0, 1$ in solo execution emulates execution e_i where they service the participating codes in a round-robin fashion. It is shown that the BG-simulation is such that the failure of a simulator manifests itself as lack of progress of a single code. Thus the emulated execution is 1-resilient. Consequently, at least 99 codes will have an output. Since for consensus all outputs are the same, it can be adopted as a consensus output for s_0 and s_1.

Moreover, the BG-simulation not only reduces 1-resilient consensus to 2-processors wait-free consensus, it also fully characterizes the 1-resilient solvability of the task T of n-processors consensus: T is 1-resiliently solvable iff there exists a wait-free protocol for 2 processors to solve 2-processor consensus. The BG simulation shows that if T is solvable 1-resiliently then 2-processor consensus is solvable wait free. In the other direction: Were 2 processors wait free able to do consensus, then n processors 1-resilently would have been able to do consensus too; all processors wait on any processor among processors p_0 and p_1 to reach consensus and announce it, as at least one among the two processors must be alive.

Thus, if wait-free tasks are characterized by the HS conditions [11], we would like to take any question about t-resilient solvability and reduce it to a question about wait-free solvability. The BG-simulation does this only for a subclass of such questions.

It can bee seen that the correctness of the emulation relies heavily on the fact that the two simulators, one perhaps observing only 99 entry output tuple and the other observing 100, will nevertheless, output the same value, as all simulated processors output the same value. But what about tasks in which this is not the case.

Imagine you wanted to simulate 100 processors, 1-resilient, by 2 simulators wait-free, such that s_0's output includes the output of $code_0$ and s_1's output include the output of $code_1$. If the task is not consensus, then when s_0 observes 99 outputs, these may be outputs for codes $code_1, \ldots, code_{99}$, while $code_0$ is the one on which progress was blocked as a result of a failure of simulator s_1. Thus the BG-simulation cannot accommodate such a requirement.

This drawback is remedied here by the Extended BG-simulation. It allows a simulator to be associated uniquely with certain code and guarantees that when simulator s_0 returns wait free with 99 or 100 outputs, one of these outputs in both cases is the output of $code_0$. This is derived from a simple observation that if s_0 cannot return only because $code_0$ is blocked by s_1, then s_1 has enough outputs to depart. Thus, we can implement a "safe abort" operation that releases $code_0$ from being blocked by s_1, under the condition that s_1 will depart and will not obstruct s_0 anymore.

Extending the BG-simulation by this simple property results in a full characterization of t-resiliency in terms of wait-freedom.

Given a task T on $n > t$ processors we create the task T' on $t + 1$ simulators $s_0, \ldots s_t$, such that T is t-resiliently solvable iff T' is wait-free solvable: Simulator s_i is given the input of processor p_i as well as the input to a set of processors of size $n - (t + 1)$ with ids higher than i. Simulator s_i outputs for p_i as well as for a (possibly different) set of processors of size $n - (t + 1)$ with ids higher than i. The input/output of the $t + 1$ simulators have to be a projection of a single original input/output tuple-pair in T, where all inputs in the tuple appear as an input to at least one simulator.

As an example, consider the task of WeakRenaming $(n + t, n, n + t - 1)$ [12]. This is a task on $n + t$ processors and requires a processor in a participating set of size n to output a unique integer in the range 1 to $n + t - 1$. The question is whether this task is solvable $2t = r$-resiliently. Thus the induced wait-free task is on $r + 1 = 2t + 1 = 2(t + 1) - 1$ simulators. The characterization implies that when $t + 1$ simulators $s_i\ i \in \{0, \ldots, 2t\}$ participate, where the input to s_i is the id of p_i, as well as all the ids of processors in $\{p_{2t+1}, \ldots, p_{n+t-1}\}$, then the problem is wait-free solvable. Notice that the induced size of participating set of processors in T, that is, all the union of ids that appear as inputs to simulators, is n. Thus, each processors s_i will output a unique integer for p_i as well as they will all agree to output the same unique integers for all the processors in $\{p_{2t+1}, \ldots, p_{n+t-1}\}$. We will see that this implies that simulators chose for themselves a unique integer in a range of size $2(t + 1) - 2$.

Why do we bring this "complex" example? Turns out that just because s_i has an output for p_i we can show that the wait-free task T' on $2t + 1$ simulators induced by WeakRenaming $(n + t, n, n + t - 1)$, is equivalent WeakSymmetryBreaking $(2(t + 1) - 1, t + 1)$ (WSB) which is a task on $2(t + 1) - 1$ processors and just asks that for participating set of size $t + 1$ all processor output 0 or 1, but at least one processor outputs 0 and at least one processor outputs 1. This task was recently been shown to be unsolvable for certain values of t and solvable for others [7]. We can immediately infer that the same holds for WeakRenaming $(n + t, n, n + t - 1)$.

The answer to the question about the solvability of WeakRenaming $(n + t, n, n + t - 1)$ [10] is the new tangible ramification of the characterization of the t-resilient solvability, and indeed, the quest to answer this question motivated this paper.

Another by-product of the characterization is a significant simplification of the derivation of the HR result that implies that the question about the solvability of a task on n processors, for any given n, 2-resiliently, is undecidable [15]. Prior to the HR result the GK result proved that the question of solvability of tasks on 3 processors wait-free is undecidable [16, 17]. The HR derivation appealed to first principles. Here we reproduced the HR result by using the new resiliency characterization and showing that solvability question n processors tasks 2-resiliently is undecidable iff 3 processors wait-free is.

The paper is organized as follows: In Section 2, we define resiliency, recall the details of the BG Agreement-Protocol [2, 6], and present its extended version. In Section 3, we prove the extended characterization of t-resiliency. Finally, in Section 4, we then show the above two ramifications of the characterization, and in Section 5, conclude the paper.

2. THE MODEL

We assume a single-writer multiple-reader Atomic Snapshot shared-memory [1] model. Each processor p_i has a dedicated cell C_i, to which it writes exclusively, and can read the whole memory in a snapshot. Processors alternate between writing and reading in a snapshot. W.l.o.g we assume that processors when they write, they write their whole history. To start with, the memory cells are initialized to $\perp$, and the initial state of each processor is its own private input. The last snapshot by a processor is called its *view*.

A *task* T is a binary relation Δ between *input tuples*, vectors of inputs to subsets of processors, called the *participating sets*, and *output tuples*, a commensurate size vectors of output values.

For instance, consider the consensus task on n processors. For each subset of k processors, there are 2^k input vectors, telling a processor whether to start with 0 as an input or 1. If all processors start with 0, then the output vector is all 0's, and if all start with 1 then the output vector is all 1's. Otherwise, for each input vector there are two possible output vectors one of all 0's and the other one of all 1's.

A task is "colorless" if given an input-output tuple (In, Out) $\in \Delta$ where the length of the Out vector is k, and the set of values that appear in Out is σ, then for all vectors $v \in \sigma^k$, $(In, v) \in \Delta$. (For a generalization of colorlessness that captures the outputs of the BG-simulation declaratively, see [6].)

A *protocol* is a partial mapping from views to output values.

An *execution* is an infinite sequence of processors names. Every appearance of a processor in the sequence is called a "step."

With each entry in an execution we associate a set of views, one for each processor. We do this by interpreting the first appearance of a processor name in the sequence as a write action, the second appearance as a snapshot action, the third as a write action, etc. Initially the view of a processor is its own input. A processor which took a step in an execution is called "participating."

A processor p_i is *non-faulty* in an execution e if its name appears infinitely many times in e. It is otherwise *faulty*.

An *environment* is a set of infinite executions. A task T is *solvable in an environment* $\mathcal{N}$ *by a protocol* Π, if for all executions $e \in \mathcal{N}$, every non-faulty processor reaches a view that is mapped by Π to an output value. Considering only the first such view for each processor, the resulting input output can be extended to a pair in Δ: If we then take the vector of the participating set, and fill out the corresponding-size vector with these output values in the appropriate positions, then we say that the task is solvable by Π if the possible empty entries can be filled-out (completed) such that the resulting output values vector and the vector of the participating set are members of Δ.

Given a system of n processors, a *t-resilient* environment is the set of executions $\mathcal{N}_t$ such that for all $e \in \mathcal{N}_t$ at least $n-t$ processors are non-faulty. A task is *t-resiliently solvable* if there is a protocol Π such that T is solvable in $\mathcal{N}_t$ with Π. Environment $\mathcal{N}_{n-1}$ that places no constraints on the infinite executions is called *wait-free.*

It is straightforward to see that if a task T is t-resiliently solvable by a protocol Π, then there is a protocol Π' such that solves T in a environment $\mathcal{N}'_t$ in which a processor stops taking steps as soon as it decides with Π. Intuitively, once a processor p_i decides, it can be simulated taking steps in a lock-step manner together with some not yet decided processor, chosen and registered in the shared-memory by p_i before deciding. By repeating this procedure for each decided processor, we obtain a simulated t-resilient execution of Π in which every non-faulty process decides.

2.1 Special Tasks

In this paper we deal with two specific tasks. The task WeakRenaming(u, p, s) [12] requires that if only p processors participate out of a universe of u processors $p_0, \ldots, p_{u-1}$, then each outputs a unique integer in the range $1, \ldots, s$.

Why do we choose p out of u? To force the algorithm on the p processors to be a "comparison-based algorithm." By Ramsey Theorem [3] if the p processors are chosen from a universe big enough then the algorithm is comparison-based. We know that here exist a wait-free comparison-based algorithm — an algorithm that only compares ids - for n processors to wait-free rename in the range $1, \ldots, 2n-1$ [12]. A simple variation of that algorithm shows that there exists a comparison algorithm that t-resiliently rename n processors in the range $1, \ldots, n+t$.

It was recently shown that WeakRenaming$(2n-1, n, 2n-2)$ is wait-free unsolvable for certain values of n while it is solvable for others [7]. It makes sense that WeakRenaming$(n+t, n, n+t-1)$ will exhibit the same behavior when at most t out of the n may fail, with $t+1$ as the value that determines the solvability. This observation made by Hagit Attiya [10] initiated this research.

The other task we deal with is WSB(u, p). This task requires that if only p processors participate out of a universe u, then each outputs either 0 or 1 with at least one 0 and one 1 among the outputs. It is known [5] that WSB$(2n-1, n)$ is wait-free equivalent to WeakRenaming$(2n-1, n, 2n-2)$. This paper extends this and shows that WSB$(2(t+1)-1, t+1)$ is equivalent to WeakRenaming$(n+t, n, n+t-1)$. for all $t < n$.

2.2 An Agreement-Protocol (AP)

An agreement-protocol [2, 6] is a protocol that solves consensus (leader election) in an environment in which all processors that take a step, take at least b steps for some $b \geq 3$.

Notice that directly from the abstract definition of an AP we can derive the following property: If all participating processors are past b steps then every participating processor can immediately decide; it can simulate itself continuing solo. Indeed, by applying the valence-based arguments [9], we can derive that no further steps can change the valence of the system state: otherwise, there is an extension of the state in which no decision is possible. Consequently, we can talk about the state of the AP. It is either "decided" or not.

To make things concrete, here's a simple instance of an agreement-protocol:

Suppose that each cell C_i in the SWMR memory is subdivided into two cells $C_{i,1}$ and $C_{i,2}$, both initialized to $\perp$. The algorithm proceeds in two phases. In the first phase p_i writes its name i in $C_{i,1}$. It then takes a snapshot of the names, $S_i := C_{*,1}$, to get a set of names S_i. In the second phase processor p_i posts the snapshot, $C_{i,2} := S_i$. It then waits until its snapshot of $C_{*,2}$ is such that there exists a not-a-$\perp$ cell $C_{j,2}$, such that for all $k \in C_{j,2}$, $C_{k,2} \neq \perp$. When this happens p_i return the smallest name from the smallest set $C_{l,2}$ it sees (No need for tie-breaker as snapshots of the same size are identical).

For the this agreement-protocol $b = 3$. Safety - all processor that return a name return the same name - follows from the fact that as time progresses the smallest set posted at phase 2 can only decrease. Since posted sets are snapshots it follows that two sets of the same size are identical. By way of contradiction assume $S_p > S_q$ and one processor returned a name from S_p and another from S_q. Thus S_q was posted after S_p and by virtues of snapshots $S_q \subset S_p$ and $q \in S_q$. But the halting condition says that there exist some snapshot S_m greater or equal to S_p and all its processors have a already posted their set. Since q belongs to S_m we have a contradiction.

Liveness follows from the fact that if all processors that took a step executed at least 3 steps, then if q appears in any snapshot, then it took the first step, and since it took the 3rd step, a snapshot is now posted at $C_{q,2}$, and consequently the halting condition holds.

To extend as AP to an Extended AP (EAP) we need the the protocol below.

2.3 Commit-Adopt (CA) Protocol

The Commit-Adopt Task [13] is specified as follows:

Processors are divided into groups, the members of each group share the unique group-name.

1. If a processor outputs before anybody in another group started, then it outputs "commit my-group-name,"
2. If a processor outputs "commit j" for some group name j, then all processors output either "commit j," or "adopt j."

The CA task is wait-free solvable in two phases similar to the phases of the agreement protocol:

In the first phase p_i writes its group name to $C_{i,1}$ and then reads all $C_{*,1}$. If in the first phase p_i observes only a single group name it writes "commit my-group-name" in $C_{i,2}$, otherwise it writes "abort." It then reads $C_{*,2}$ if it reads only "commit group-name" it outputs "commit group-name," and if it reads both "commit j" and "abort" it outputs "adopt j."

Obviously, only the group name written first in the first phase has a chance that a processor will write it as a commit argument in the second phase. Also, if a processor commits, it did not read any abort in the second phase, consequently "commit" was written first. Thus all will see a commit posting and will either commit or adopt.

If we use two CA in succession and the adopt value of the first CA is the input of a processor to the second CA, then if a processor commit in the first CA, all will commit either in the first CA or the second CA.

2.4 Extended Agreement Protocol

The Extended Agreement Protocol (EAP) is a sequence of sub-protocols. Each element in the sequence is an Agreement-Protocol or a Commit-Adopt. The sequence starts with an AP protocol and from there on alternates between an AP and a CA. The output of an AP is the input to the succeeding CA, while a value adopted in CA is an input to the succeeding AP.

What are the properties of EAP? To terminate an EAP we will require a processor to execute the EAP sub-protocols in order until it commits an agreement value in some CA. Consequently by the properties of CA, EAP is safe: All processors that terminate an EAP will output the same value. If one processor commits in one CA, the rest that do not commit but adopt will submit the same value to the subsequent AP. Consequently, if the AP is decided it decides that value which will be then submitted to the succeeding CA, which will commit the decision.

The addition, the CA protocol provides for a safe abort of an AP within the EAP. Suppose that a simulator waiting on some AP within an EAP decides to abort the AP and continue. It will write in SM that it aborts that AP (so that other simulators do not wait at this AP any more) and can now safely proceed to the succeeding CA with any value it wants.

If no processors invokes an "abort," then EAP coincide with AP. Processors that start the EAP will go through the first AP and wait until it is resolved. Once it is resolved they go with the resolution value to the next CA and since they all start it with the same value they will all commit to that value and the EAP is terminated.

In the Extended BG-simulation an AP in an EAP will be aborted by a simulator s_i which cannot progress in the simulation but cannot halt as it does not have an output for p_i or $n-t-1$ processors with ids higher than i.

3. CHARACTERIZATION OF t-RESILIENCY

In this section, we describe the extended BG-simulation protocols and apply it to completely characterize t-resilient solvability of tasks.

3.1 T is t-resiliently Solvable $\rightarrow$ T′ is Wait-Free Solvable

In this subsection we show how $t+1$ simulators solve T′ wait-free given a *code* with which T is t-resiliently solvable.

The Extended BG-simulation that uses EAP works as follows.

We have $t+1$ simulators where simulator s_i is associated with p_i, $i = 0, \ldots, t$. Each simulator s_i that *started*, i.e., having taken at least one step, has an input value for p_i and the input values for $n-t-1$ processes p_j such that $j > i$. We say that $code_i$ of T has started if some simulator p_i has started with an input for p_i.

As in the original BG protocol [2, 6], simulators, in the round-robin fashion, simulate all codes in $code_0, \ldots, code_{n-1}$ for which any started simulator has an input value (we call these codes also *started*). The simulators asynchronously move the program-counter of the codes, by inductively agreeing on a value for the next read-command in the code. Once a read-command value is agreed on the program counter can be moved to the next read command in the code. The "write" that follows a read command for which there is an agreed value, is deterministically defined by the last read. Thus, when a simulator wants to propose a value for a read command all it needs to do is take a snapshot of the places of the program-counters in order to propose a valid "read" value. The problem is, of course, that different simulators may propose different value for the same simulated read-command, as the program-counters move asynchronously. But as the read-commad is an asynchronous read all teh value they propose are valid and all they need is to agree on one of the values.

They do this by employing an EAP for each read-command. A processors that waits on an AP inside an EAP to terminate proceeds to simulate another read-command in the order. Since we have at the most $t+1$ simulators at most t APs may be unresolved. When an AP is resolved processors proceed to the succeeding CA, commit the resolved value, and by that the program-counter of that code is moved to the next read-command (Figure 1).

In following this procedure, we say that a simulator s_i is

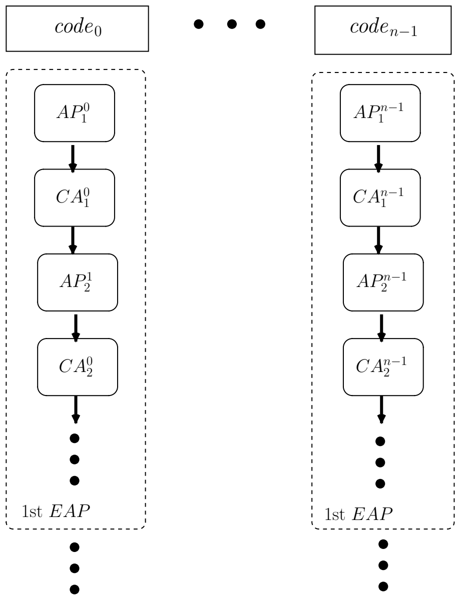

Figure 1: The use of EAP in simulating a protocol on n processors: each read command of every code $code_i$ is simulated using an instance of extended agreement protocol.

complete if s_i witnesses each code to be either terminated, or not started for the lack of input, or having an unresolved EAP for its current read-command (we say that the code is *stalled*). A simulator s_i is *stuck* if it is complete and *cannot depart*: the output of $code_i$ is missing or the number of terminated codes with ids higher i is less than $n - t$.

We say that a simulator s_i *blocks* code $code_j$ if s_i has not completed enough steps in an AP of EAP of $code_j$. Since a simulator only proceeds to the next simulated code when it has completed the AP protocol of the current one, no two codes that any simulator witnesses (using an atomic snapshot) as being stalled can be blocked by the same simulator.

The following lemma is immediate from the definition of a simulator being stuck:

Lemma 1. *When a simulator s_i is complete, all started codes that are witnessed as* not *stalled by s_i have terminated.*

Once a simulator gets stuck, it *aborts* all codes it witnessed stalled. For each stalled code, it marks the stalled AP as "aborted," and proceeds to the next CA proposing any valid value. If a blocking simulators wakes up and finds that the AP has been aborted, it checks if it has enough output values to depart. If so, the simulator departs, otherwise, it also proceeds to the next CA with any valid value.

The intuition here is that if a live simulator is stuck, then there must be at least one blocking simulator that has enough outputs to depart: the blocking simulators cannot obstruct each other from departing. Formally:

Lemma 2. *If a simulator s_i is complete, witnessing k codes being stalled, then at most k started simulators cannot depart.*

Proof. We proceed by induction on k. The case $k = 0$ follows directly from Lemma 1: if no started code is stalled, then all started codes have terminated. But each started simulator s_j has started $code_j$ and $n - t - 1$ codes with ids higher than j. Thus, each started simulator can depart.

Now suppose that the claim holds for all $k' \leq k$ and consider the case of $k + 1$ stalled codes, $code_{i_1}$, ..., $code_{i_{k+1}}$, ordered with increasing ids. Let S be the set of started simulators that are blocked by codes $code_{i_1}$, ..., $code_{i_k}$, i.e., the codes that would stay blocked if we assume that $code_{i_{k+1}}$ has terminated. By induction hypothesis, $|S| \leq k$.

Now let s_i be the highest started simulator *not in* S that is blocked by codes $code_{i_1}$, ..., $code_{i_{k+1}}$. (If no such s_i exists, then there are less than $k+1$ blocked codes in total and the claim trivially holds.)

By definition, each started simulator not in S with id higher than i can depart. Now consider a started simulator $s_j \notin S$ with $j < i$. Since $s_i \notin S$, there are at least $n - t - 1$ terminated codes with ids higher than j. Since $s_j \notin S$, by Lemma 1, $code_j$ has also terminated. Thus, s_j can depart.

Hence, in the worst case, s_i is the only simulator not in S that cannot depart, and we have at most $k + 1$ simulators in total that cannot depart because of $k + 1$ stalled codes. □

By Lemma 2, if a simulator s_i is stuck, then at least one blocking simulator can depart. Indeed, since s_i is not blocking any code, by the pigeon-hole principle, at least one blocking simulator can depart.

Thus, aborting the stalled codes in case a simulator gets stuck does not affect liveness. Either the blocking simulator is faulty, and will not obstruct the simulated codes anymore, or it will eventually realize that it has enough outputs and depart. In both cases, the live and not yet terminated simulators got rid of at least one simulator. Inductively, eventually one live simulator will participate in an unobstructed simulation until all the codes it needs to depart terminate.

This finishes the description of the Extended BG-simulation.

To solve T' simulators use the code for T through the Extended BG-simulation, as T is t-resiliently solvable.

Thus:

Theorem 3. *If T is t-resiliently solvable, then T' is wait-free solvable.*

Note that the number of distinct APs and CAs that can be used in an instance of EAP is bounded by t: the maximal number of simulators that can block a given code.

3.2 T′ is Wait-Free Solvable → T is t-resiliently Solvable

We have $n > t$ processors in T. We have a *code* for $t + 1$ simulators to solve T′. When processors in T wake up they write their input and then wait until they see at one processor acting as a simulator, terminate. Initially, only the first $t + 1$ processors, $p_0, \ldots, p_t$, act as *simulators* of T'. To act as a simulator the processor has to observe at least $n + (t - 1)$ inputs of other processors with index higher than it. A code $code_i$ ($i = 0, \ldots, t$) of T' is considered *started in* T' only if there are inputs for p_i and at least $n - (t + 1)$ processors of ids higher than itself. The started code $code_i$ posts this set of $n - t$ inputs as its input to T'. (It is easy to see that this rule satisfies the input condition of T'.) We say that a simulator p_i is *awake* if it is associated with a started in T' code. Notice that the availability of input to p_i in T does not necessarily mean that $code_i$ is started in T'.

Since we work t-resiliently, the pigeon-hole principle implies:

Lemma 4. *For every processor p_i that has an input to T, there is at least one non-faulty awake simulator that has the input of p_i to T in its input to T'.*

Just consider the simulator which is awake and has the highest index. Notice, that to start with, if k simulators are awake, then for any k processors, one can have a one-to-one correspondence between processors and simulators such that if simulator s_i corresponds to processor p_j then p_j's input is in the input of s_i.

This is easy to see. If all k processors' input appear in all simulators then we are done. If not, consider the simulator with the highest index. All processor with index greater equal to it appear as an input, when we move backward on awake simulators if some processor does not appear as input the the simulator going backward, and there can be only one, we associate this processor with this simulator.

We are using this property later implicitly. Consequently it also holds implicitly for output sets as they are the same size as the input sets.

Initially, the awake simulators use Extended BG-simulation of $t+1$ codes solving T'. Note that simulators may have different ideas about the input for any given $code_i$. This is because different simulators may have found different sets of $n-t$ inputs for p_i and $n-t-1$ processors of ids higher than i. Thus, to make sure that the codes are simulated consistently, the first AP for each simulated $code_i$ is used to agree on the input of $code_i$ to T'.

By Lemma 4, at least one awake simulator is non-faulty, and, thus, at least one $code_i$ terminates with values for p_i and $n-t-1$ processors p_j, $j > i$. The output condition for T$'$ implies that, when k terminate in T', at least $n-(t+1)+k$ processors have outputs in T. Each simulator s_i of these k simulators departs with the output of its own code.

When a live non-simulator observes at least $n-t$ outputs of T, it joins the Extended BG-simulation of the $t+1$ (awake) codes in T$'$. A processor that sees an output for itself departs. Since we can make a one-to-one correspondence between k simulators and k processors, if k processors do not have an output then the number of simulators that are awake and did not terminate is at least k, which corresponds to at least k codes in T$'$ which can be simulated.

Here we use the flexibility of Extended BG-simulation that allows arbitrarily many simulators to simulate arbitrarily many codes. What matters here is that, since the number of simulators does not exceed the number of simulated codes, the simulation makes progress, as long as there is at least one live simulator.

As in Section 3.1, a simulator gets stuck when all started codes either have terminated or are stalled, and the simulator still does not have its output for T.

By Lemma 2, when a simulator gets stuck, i.e., all started codes either have terminated or are stalled, and the simulator still does not have its output for T, at least one blocking simulator can depart. As the simulation continues, eventually every simulator departs and the task T is solved.

Thus:

Theorem 5. *If T' is wait-free solvable, then T is t-resiliently solvable.*

4. REDUCTIONS

In this section, we present two applications of the above equivalence result: a characterization of t-resilient weak renaming and decidability of t-resilient tasks.

4.1 Characterization of t-Resilient Weak Renaming

We want to show that if WSB$(2(t+1)-1, t+1)$ is impossible to solve wait-free, then WeakRenaming $(n+t, n, n+t-1)$ is impossible to solve $2t$-resiliently (the other direction is known [5] and therefore after the reduction the problems are proved equivalent).

We will prove that if WeakRenaming$(n+t, n, n+t-1)$ is $2t$ resiliently solvable then WSB$(2(t+1)-1, t+1)$ is solvable. For this we will use our characterization of resiliency. It says that WeakRenaming is equivalent to a task WeakRenaming$'$ on $2t+1$ simulators, out of which we wake up enough simulators such that the induced participating set in the WeakRenaming task is of size n. We will show that if we wake up exactly any $t+1$ simulators with a total of inputs for n processors, then when solving WeakRenaming$'$ wait-free, then each simulator can also output 0 and 1 and such that at least one simulator outputs 0 and at least one simulator outputs 1. This therefore will solve WSB$(2(t+1)-1, t+1)$.

The input to simulator s_i, $(i = 0, \ldots, 2(t+1)-2)$ is the id of p_i and the ids of processors $p_{2(t+1)-1}, \ldots, p_{n+t-1}$. Simulator s_i submits this input to WeakRenaming$'$. It can be easily seen that, for up to $t+1$ started simulators, these inputs satisfy the inputs condition for weak renaming for up to n started processors.

When simulator s_i terminates WeakRenaming$'$ it has an output for at least $n+t-(2t+1)+1 = n-t$ processors, where among them is also processor p_i, for which it return the integer I_i. If it has an output I_j for any $p_j, 0 \leq j \leq 2(t+1)-2$, $j \neq i$, then if $I_i < I_j$ it outputs 0 ("less") and otherwise it outputs 1 ("more").

If s_i observes no output for any other simulator but itself, then it has exactly $n-(t+1)+1 = n-t$ outputs. An output for each of processors $p_{2(t+1)-1}, \ldots, p_{n+t-1}$ and an output for itself. Thus, out of the integers in 1 to $n+t-1$ it is missing $t+(t-1)$ integers. Observe that either the number of missing integers that are less than I_i is less than t, or the number of missing integers that are greater than I_i is less than t. In the former case it returns 0 (less) and in the latter case 1 (more).

We claim that if all $t+1$ simulators return then at least one simulator will output 0 and at least one simulator will output 1.

If each simulator returned output values for at least two processors that correspond to simulators, then the simulator s_j whose I_j is the smallest must return 0, and the simulator with maximum I_j must return 1. The only difficulty is when one of these extreme two simulators, the one that returns the smallest I_i or the one that returns the largest, observed no other simulator. But in this case when the rest of t simulators return and say s_i returned 0, then the number of missing integers less than I_i is less then t. Consequently, by pigeon-hole principle one of simulators has to return an integer larger than I_i and therefore will output 1. Conversely the other case.

4.2 2-resilient Tasks are Undecidable

Some tasks are r/w wait-free solvable and some are not [9]. A task on n processors has a precise finite encoding specification. Why not build a decision program whose input is a task encoding and whose output is "yes" or "no" according to whether the encoded task is r/w wait-free solvable or not? Indeed this question was raised in [14] where they show that 2 processors tasks is a reachability problem and consequently, decidable.

In [16, 17], it was shown that the family of tasks on 3 processors is undecidable.

Later, in [15] the authors worry that perhaps 3 processors tasks are undecidable, but if you take 100 processors and only 2 may fail, then the question of whether the task is solvable is decidable. They prove that it is not but appealed to first principle rather than by reduction to 3 processors tasks.

Here, using the characterization of t-resilient solvability we now show how a program to decide 100-processor 2-resilient tasks can be used to decide 3-processor wait-free tasks.

Let T^3 be a task on 3 processors p_0, p_1, p_2. Build from it a task T^{2-res} on n processors $p_0, \ldots, p_{n-1}$ as follows. The projection of Δ of T^{2-res}, on p_0, p_1, p_2, (i.e., we take an input-output tuple in T^{2-res} and just look at what input-output relation it implies vis p_0, p_1, p_2), is T^3. Conversely, any input-output tuple which may be syntactically in T^{2-res} is there, if its projection on p_0, p_1, p_2 is in T^3, and moreover the rest of the symbols in the output appear as an output of one of the processors p_0, p_1, p_2 in the tuple. In other words, any processor from $p_4, \ldots, p_{n-1}$ is allowed to adopt any output it sees returned by any processor among p_0, p_1, p_2.

We want to show that T^3 is wait-free solvable if and only if T^{2-res} is 2-resiliently solvable.

If T^3 is solvable then T^{2-res} is: Processors $p_4, \ldots, p_{n-1}$ wait on processors p_0, p_1, p_2 to execute T^3 until at least an output is available. They adopt this output and drop out. The rest of the processors in p_0, p_1, p_2 finish executing T^3 to get an output.

If T^{2-res} is solvable then T^3 is: Three simulators s_0, s_1, s_2 run n-processor Extended BG-simulation of T^{2-res}. Each simulator s_i returns among other values a value for p_i, and thus the simulator adopt it as its output in T^3.

5. CONCLUSION

We amended the BG-simulation to get the Extended BG-simulation. The Extended BG-simulation allows each simulator to be associated with a unique processor and return values as in the BG-simulation but with the additional property that a simulator returns an output value for the processor it is associated with.

We show that the Extended BG-simulation fully captures t-resiliency for distributed tasks. With the Extended BG-simulation we can reduce questions about t-resilient solvability to questions about wait-free solvability. The latter is characterized by the HS conditions [11].

Thus this paper "closes" an area with no open questions remaining.

Acknowledgment I am greatly in debt to Hagit Attiya who back in the summer of 2005 made me aware of the seemingly inapplicability of the BG-simulation to the t-resilient Weak-Renaming problem.

Extensive comments, questions, and editorial help by Petr Kuznetsov improved the readability of the paper considerably.

6. REFERENCES

[1] Afek Y., H. Attiya, Dolev D., Gafni E., Merrit M. and Shavit N., Atomic Snapshots of Shared Memory. *Proc. 9th ACM Symposium on Principles of Distributed Computing (PODC'90)*, ACM Press, pp. 1–13, 1990.

[2] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on the Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.

[3] BERGE, C., Graphs and Hypergraphs., North-Holland, Amsterdam, 1973

[4] M Loui, H Abu-Amara, Memory Requirements for Agreement Among Unreliable Asynchronous Processes, Advances in Computing Research, JAI (1987).

[5] Gafni E., Rajsbaum S. and Herlihy M., Subconsensus Tasks: Renaming Is Weaker Than Set Agreement., DISC 2006: 329-338

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

[7] Armando Castañeda and Sergio Rajsbaum, New Combinatorial Topology Upper and Lower Bounds for Renaming. *to appear in PODC 2008.*

[8] Hagit Attiya , Amotz Bar-Noy and Danny Dolev, Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, v.42 n.1, p.124-142, Jan. 1995.

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

[10] Attiya H., Private Communication, August, 2005.

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

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

[13] Eli Gafni, Round-by-Round Fault Detectors: Unifying Synchrony and Asynchrony (Extended Abstract). PODC 1998: 143-152

[14] Ofer Biran, Shlomo Moran and Shmuel Zaks, A combinatorial characterization of the distributed tasks that are solvable in the presence of one faulty processor. *In Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing (PODC'08)*, ACM Press, 1988.

[15] Maurice Herlihy, Sergio Rajsbaum: The Decidability of Distributed Decision Tasks (Extended Abstract). STOC 1997: 589-598

[16] Eli Gafni, Elias Koutsoupias: 3-Processor Tasks Are Undecidable (Abstract). PODC 1995: 271

[17] Eli Gafni, Elias Koutsoupias: Three-Processor Tasks Are Undecidable. SIAM J. Comput. 28(3): 970-983 (1999)