Fast Long Lived Renaming

Justin Kopinsky
Massachusetts Institute of Technology
jkopin@mit.edu

ABSTRACT

We present a randomized algorithm to solve the renaming problem in distributed computing with step complexity $O(\log \log n)$ per operation against an oblivious adversary. This holds with high probability for any particular operation, which is an improvement over existing $O(\log n)$ algorithms. This result also demonstrates a doubly exponential separation against deterministic algorithms and an exponential separation against strong adversarial algorithms, for which lower bounds of $\Omega(n)$ and $\Omega(\log n)$ respectively are known. We show that our results can be extended to support repeatedly releasing and reacquiring names ('long-lived' renaming), with complexity bounds holding over any sequence of polynomially many executions. We further sketch the same result for infinite executions.

1. INTRODUCTION

The availability of unique identifiers, or 'names', for processors in a distributed setting is an important requirement for many algorithms. For example, if one wishes to assign a block of memory to each node, it is desirable to simply allocate a continuous chunk and allow each node to index into it by name. Unfortunately, while most systems do provide such names, they are often drawn from impractically large namespaces.

This gives rise to the 'renaming' problem, that of assigning unique names in a small namespace to processors drawn from a large or unknown namespace. One may consider *tight* renaming, in which the size of the namespace is exactly equal to the number of participating processors, n. In contrast, *loose* renaming relaxes this constraint to allow $O(n)$ names. A substantial amount of research has already studied renaming complexity in both randomized and deterministic settings, e.g. [5].

Tight renaming is known to be impossible on an asynchronous read/write system in the presence of crash faults[5], and has deterministic step complexity* $\Theta(n)$ per processor even when allowed atomic operations (e.g. test-and-set or compare-and-swap) and no faults are tolerated[3]. The $\Theta(n)$ lower bound also holds for loose renaming with fault tolerance.

It is easy to construct an algorithm to show that these bounds are tight: simply have every processor attempt to atomically† acquire each name in turn, stopping when successful. Obviously, the slowest processor will need to try every name before acquiring one, taking $\Theta(n)$ steps.

Fortunately, randomized algorithms are known to perform better[4, 2]. Relatively simple algorithms can be implemented to solve loose renaming in $O(\log n)$ steps with high probability (see section 3). Generally, processors can select a name at random and attempt to acquire it. With some care, these strategies have been successfully adopted to achieve poly-logarithmic time bounds[2], even for a tight namespace.

Most renaming algorithms have been developed under the assumption that processors only attempt to acquire a single name, and retain that name for the lifetime of the task for which the name was acquired. We refer to this problem as 'one-shot' renaming. However, one might also be interested in allowing a processor to relinquish its name so that a later processor can attain it. This is the idea of 'long lived' renaming[6]. Note that when discussing time bounds for long lived renaming algorithms, we define n to be the maximum number of processors which are ever participating simultaneously.

In this paper, we discuss and analyze a loose, randomized, one shot renaming algorithm given by [1] which completes in $O(\log \log n)$ steps with high probability. Note that this is an improvement over the previously best known $O(\log n)$ algorithms. We then extend the algorithm to long lived renaming and provide a novel analysis establishing that the $O(\log \log n)$ time bound holds even for infinite executions, despite the fact that rare events may cause the algorithm to temporarily enter an unfavorable state.

The remainder of this paper is laid out as follows: Section 2

*In this paper, we refer to step complexity as the number of steps taken by the slowest processor. This is sometimes known as *local* complexity.

†Note that this is not legal on a read/write system and does not violate the aforementioned impossibility result.

describes the model in detail and gives a formal problem statement. Section 3 provides somewhat more detail about known results in the specific setting we consider. Section 4 describes the algorithm of [1] and provides the analysis for the one shot renaming problem. Section 5 gives our novel analysis for the long lived renaming problem. Finally, we give some possible improvements and directions for future work in section 6.

2. BACKGROUND

2.1 Concepts in Distributed Computing

We provide a basic overview of the distributed computing model which we use as the framework for our algorithm. Knowledge of sequential algorithmic techniques is assumed.

2.1.1 Asynchronous Shared Memory Model

We assume that n processors operate independently on a shared memory space. In particular, a given memory address refers to the same location regardless of which processor accesses it, although processors are allowed to maintain local state as well.

The processors operate asynchronously in the sense that its execution can be modelled by any valid sequential *schedule*. A sequential schedule associates with each time step a single processor which is then allowed to perform a single operation. Note that the same processor may be chosen by the schedule many times in a row, or not at all for a period of time. Formally, a schedule is any element of the set of all strings over the alphabet of processor names.

Note that, schedules allow us to ignore the issue of concurrent accesses to the same location in memory. If the next operation for two separate processors each access the same memory location, one of them will be scheduled before the other, resolving the conflict. Bear in mind that there is no guarantee *which* will be scheduled first.

It is important to note that the output of an algorithm may depend on the schedule. For example, consider the following algorithm on two processors: processor A writes $x = 0$, and then outputs x. Processor B writes $x = 1$ and then outputs x. If we write the schedule as a string of processor names indicating which processor acts, then for schedule $AABB$, A will output 0 and B will output 1. However, for schedule $ABAB$ both processors will output 1.

In the setting of renaming, one would expect that the name emitted by a given processor depends on the schedule for a given execution of the renaming algorithm, even for deterministic algorithms (or randomized algorithms with a fixed seed). Naturally, this is indeed the case, but the algorithm can still be considered correct, as described in section 2.3.

2.1.2 Compare-and-Swap

As mentioned briefly in section 1, our primary technique for solving renaming will require processors to pick a name, check if that name has already been acquired by someone else, and if not, acquire it. Unfortunately, this is impossible to do with only reads and writes. To illustrate this, consider two processors, A and B, who both randomly select the same, currently vacant name, say name i. Each executes the pseudocode given in figure 1.

```
...
x = read(i)
if (!x) { write(i,1); return i }
...
```

Figure 1: An incorrect algorithm

Then, under the schedule $ABAB$, both processors will read $i = 0$, and both processors will execute the conditional code, causing both to choose name i. This algorithm is incorrect, since it is illegal for multiple processors to be assigned the same name. A full impossibility proof is out of scope of this paper, but this problem is very well studied, see e.g. [8].

There are many mechanisms which solve this problem by allowing slightly more power than just reads and writes. In this paper, we will assume processors have access to a hardware implementation of the `CompareAndSwap` instruction. In this paper, we will typically abbreviate invocations of `CompareAndSwap` to `CAS`. `CAS` takes three arguments: a memory address, a, an expected value, e, and an update value, v. `CAS` implements the pseudocode given in figure 2.

```
CAS(a,e,v):
  if(read(a)=e)
    write(a,v)
    return true
  else
    return false
```

Figure 2: The `CompareAndSwap` operation

The crucial point of the `CAS` operation is that it executes *atomically*. In the language of schedules, this means that one must think of `CAS` as a single operation; i.e. no other processor may act while some processor is in the middle of performing a `CAS`. Using the `CAS` instruction, we can fix the pseudocode shown in figure 1 as demonstrated in figure 3. Note that it is standard practice to read the location in question before attempting a `CAS`, as a read is significantly less expensive than a `CAS`.

```
...
x = read(i)
if (!x)
 test = CAS(i,0,1);
 if (test)
   return i
...
```

Figure 3: A correct algorithm using `CompareAndSwap`

Now, supposing processors A and B try to simultaneously execute the code of figure 3, at most one of them will see a value of `true` returned by the call to `CAS`, and so at most one of them will output name i. Again, see e.g. [8] for a fuller discussion of `CAS` and other methods of circumventing the weakness of read/write systems.

2.1.3 Adversarial Schedules

When analyzing algorithms for distributed systems, one must take care to consider the schedule being executed. In particular, different schedules might result in different running times, even for deterministic algorithms. Bearing this in mind, we must assume that the schedule is determined by an adversary when conducting our analysis. The adversary is assumed to have full knowledge about our algorithm when creating the schedule.

For deterministic algorithms, any fixed schedule results in an execution with a fixed runtime, so we must analyze algorithms assuming that the adversary chooses the worst possible schedule, similarly to sequential algorithms for which the worst case input is often considered.

However, for randomized algorithms, there are varying levels of power which can be given to the adversary. The strongest adversary knows the entire random string used by our algorithm in advance, and can determine the schedule based on that information. The astute reader will notice that this adversary is not particularly interesting, as we might as well choose the random string, r which runs the fastest against its corresponding adversarial schedule, and simply use the deterministic algorithm which simulates the original algorithm on r. In particular, any lower bounds which hold for deterministic algorithms also hold for randomized algorithms against this super-powered adversary.

The strongest adversary which is still interesting (often referred to as simply the *strong* adversary) does not define a schedule in advance. Rather, at each time step, he picks the next processor to act. He sees the results of all coin flips made at that time step before determining the next time step. Renaming against this adversary *is* tractable, but as discussed in section 3, the best known algorithm have step complexity $\Theta(\log n)$, even for loose renaming.

In this paper, we consider a weaker adversary, often referred to as the *oblivious* adversary. Much like deterministic adversaries, oblivious adversaries must fix a schedule in advance, and are given no information about the random string used by the processors. No non-trivial lower bounds are known for renaming against the oblivious adversary. An upper bound of $O(\log \log n)$ will be established in section 4.2.

2.2 Probability

This section presents two important bounds we will use in our analysis. We do not prove them.

The *union* bound states that, if each of k independent events occurs with probability at least $1-p$, then all k events occur with probability at least $1-kp$. Even if the events are not independent, so long as each event occurs with probability $1-p$ given any result for the other $k-1$ events, the bound still holds.

The *Chernoff* bound has many forms. The form we will use is the following: given k independent events which each occur with probability at most p, let X be the number of events which succeed. Then, for $c \geq 1$,

$$\Pr[|X - kp| \geq ckp] \leq exp(-\Omega(ckp)) \quad (1)$$

Similarly to the union bound, this bound will also hold for dependent events, provided every event occurs with probability at most p given any result for the other $k-1$ events.

2.3 Problem Statement

Formally, the one shot renaming problem asserts that each processor makes at most one call to a `Get` operation. The `Get` operation must terminate in finitely many steps, returning a member of some fixed namespace. Furthermore, no two `Get` operations called by different processors should return the same name. We assume all processors know the total number of participating processors, n.

As described previously, the size of the namespace should be exactly n for *tight* renaming. The algorithms we present in this paper solve *loose* renaming, and thus only require that the size of the namespace be $O(n)$.

We extend the problem to *long lived* renaming by supporting a new operation: `Free`. Processors are allowed to call `Get` multiple times, but only if a call to `Free` is made before each successive `Get` call. Furthermore, two `Get` operations, say called by processors A and B are allowed to return the same name, provided (without loss of generality) that after calling `Get`, processor A initiates a call to `Free` before processor B's call to `Get` terminates[‡]. In the case of long lived renaming, we assume that n represents the maximum number of simultaneous processors which are either busy preforming a `Get` or `Free` operation, or whose last operation was a `Get` operation.

When referring to the complexity of an operation, we consider the number of steps performed by the processor executing that operation. Even if a processor is left idle for a long time by the scheduler, that time does not count against its runtime. In this paper, we assume that `CAS` operations have unit cost.

The results presented in this paper will hold *with high probability*. Whenever this term is used, it is assumed to mean that the event of concern occurs with probability at least $1 - 1/n^c$, for some $c \geq 1$.

3. RELATED WORK

Renaming originally appeared in [5], in which an asynchronous, $O(n)$ step algorithm was presented. Shavit and Herlihy showed a $2n-2$ lower bound on the minimum size of the namespace for deterministic renaming to be possible when only reads and writes are allowed[7]. Note that introducing the compare-and-swap operation avoids this lower bound.

Some work has considered *adaptive* renaming, for which the value of n is not known to the processors, but the size of the namespace must nevertheless depend on the current value of n. A *global* lower bound of $\Omega(n \log(n/c))$ total steps by all processors was given for adaptive randomized renaming

[‡]This restriction is a special case of *linearizability*, see e.g. [8] for more details.

into cn names against a strong adversary in [3]. This implies operations with (local) step complexity $\Omega(\log n)$ for namespaces of size $O(n)$ against a strong adversary. The global bound is tight, as demonstrated by an algorithm given in [2].

It is straightforward and instructive to construct an $O(\log n)$ step algorithm for non-adaptive loose renaming, even against a strong adversary. In particular, construct an array of size cn. At each step, a processor picks a name at random from the array and attempts to obtain it via a compare-and-swap. The attempt succeeds with probability at least $1 - 1/c$, and so the number of attempts is given by a geometric random variable with parameter $1 - 1/c$. It is well known that the maximum of n geometric variables with a constant parameter p is $O(\log n)$ with high probability (in particular, pn processors succeed in one step, and in general $(1-p)^{i-1}pn$ processors succeed in i steps).

This algorithm also works for long lived renaming against even a strong adversary, and previously $O(\log n)$ (with high probability) was the best known upper bound in this setting, for both strong and oblivious adversaries. This is improved on in [1] with a $O(\log \log n)$ step complexity algorithm for loose renaming.

4. ALGORITHM

In this section, we present a data structure to solve the renaming problem without supporting `Free`, renaming n processors using $O(n)$ names. We call this algorithm the `LevelNaming` algorithm. Section 4.1 describes the layout and operations of the structure. We will show in section 4.2 that these operations succeed after at most $O(\log \log n)$ compare-and-swap operations.

4.1 Description

Allocate a shared memory array of size $4n$[§], initialized to 0. We will divide this array into $O(\log n)$ *levels*, $L_0, L_1, \ldots$ so that L_0 consists of the first $2n$ entries of A, L_1 consists of the next n entries, and in general, L_i consists of $2n/2^i$ entries. We will denote $|L_i|$ to be the size of L_i, in particular $|L_i| = 2n/2^i$ by construction.

We implement `Get` as follows: for each level L_i in succession, randomly pick an index $l \in L_i$ and execute $\texttt{CAS}(A[l], 0, 1)$. If successful, then stop and return l. Otherwise, repeat this process c times, where $c = O(1)$ is a parameter of the data structure. If all c attempts fail, continue to L_{i+1}. If every level has been traversed without success, then try every index of A in succession (though we will show that this will essentially never happen).

4.2 Analysis

Define S_i to be the number of processors which are currently assigned a name in L_i. We will prove by induction that when the execution is finished, $S_i \leq n/2^{2^i}$ with high probability for all $i \geq 1$. Note that this implies that $S_i \leq n/2^{2^i}$ at *any* point during the execution, since once a processor receives a name, it does not lose it. It follows immediately that no processor reaches level $1 + \log \log n$, and that therefore every processor terminates in $O(\log \log n)$ steps. Because of this fact, we will generally assume $i \leq \log \log n$ throughout this section.

The base case, $i = 1$, is simple. $|L_0| = 2n$, so it is impossible for L_0 to be more than half full at any point during the execution. Therefore, the probability that a processor advances to L_1 is at most $1/2^c \leq 1/8$, provided we choose $c \geq 3$. Since this bound holds for any particular processor regardless of the state of the other processors, we can apply the Chernoff bound to show that the probability that more than $n/4 = 2 \cdot (n/8)$ processors advance to L_1 is at most $e^{-\Omega(n)}$, as desired.

To prove the induction step, we introduce the following lemma, which will prove useful throughout our analyses.

LEMMA 4.1. *Suppose $S_{i-1} \leq \delta n/2^{2^{i-1}}$ at all times for some $\delta = O(1)$. Then for appropriate choice of c, the probability that a processor is assigned a name in L_i is at most $1/2^{2^{i+2}}$.*

The probability that a processor is assigned a name in L_i is obviously upper bounded by the probability that the processor fails to find a name in L_{i-1} given that it gets to L_{i-1} in the first place.

The proof now follows directly from the asymptotics of S_{i-1} and $|L_{i-1}|$. In particular, the probability that a single randomly selected name is already taken is given by

$$\frac{S_{i-1}}{|L_{i-1}|} \leq \frac{\delta n/2^{2^{i-1}}}{2n/2^{i-1}} = \frac{\delta 2^{i-2}}{2^{2^{i-1}}}$$

However, for any choice of r, it holds that $i = o(\frac{1}{r}2^i)$, and therefore that $2^{i-2} = o(2^{2^{i-1}/r}) = o(2^{2^{i-1}})^{1/r}$. Taking $r = 2$, we have $\delta 2^{i-2} = o(\sqrt{2^{2^{i-1}}})$. Now, choosing $c = 16$, we can compute:

$$\Pr[p \text{ collides } c \text{ time}] \leq \left(\frac{\delta 2^{i-2}}{2^{2^{i-1}}}\right)^c \leq \left(\sqrt{\frac{1}{2^{2^{i-1}}}}\right)^{16} = \frac{1}{2^{2^{i+2}}}$$

This holds for all sufficiently large i. Since it fails to hold for only a finite number of i (independent of n), we can choose $c = O(1), c \geq 16$ so that the bound holds for *all* i.

Remark: We can replace the value $1/2^{2^{i+2}}$ with $1/2^{2^{i+k}}$ for any constant k by choosing $c = 4 * 2^k$. We will refer to this fact as the *strong* version of lemma 4.1.

To complete the proof of the main result, we will prove the following lemma:

LEMMA 4.2. *Suppose every processor has been assigned a name in L_i with probability at most $1/2^{2^{i+2}}$, then, with high probability, $S_i \leq n/2^{2^i}$.*

[§]One can acheive constants better than 4, but we use 4 here for simplicity of analysis.

Assume lemma 4.2 is true. By the inductive hypothesis, $S_j \leq n/2^{2^j}$ for every $j < i$. Then the precondition of Lemma 4.1 is satisfied for S_{i-1}, and so every processor is assigned a name in L_i with probability at most $1/2^{2^{i+2}}$. This satisfies the precondition of lemma 4.2, which completes the result.

We hereby prove lemma 4.2. We will apply a Chernoff bound to compute the probability that $S_i \geq n/2^{2^i}$. Each processor reaches L_i with probability at most $1/2^{2^{i+2}}$ by assumption. Thus, $S_i \leq n/2^{2^{i+2}}$ in expectation. Define

$$\mu = \frac{n}{2^{2^{i+2}}} = \frac{n}{2^{4 \cdot 2^i}}$$

This allows us to use the Chernoff bound (equation (1)) to compute:

$$\begin{aligned}
\Pr\left[S_i \geq \frac{n}{2^{2^i}}\right] &\leq \Pr\left[|S_i - \mu| \geq \frac{n}{2^{2^i}} - \mu\right] \\
&= \Pr\left[|S_{i+1} - \mu| \geq \mu\left(2^{3 \cdot 2^i} - 1\right)\right] \\
&\leq \Pr\left[|S_{i+1} - \mu| \geq 2^{2^{i+1}}\mu\right] \\
&\leq exp\left(-\Omega\left(2^{2^{i+1}}\mu\right)\right) \\
&= exp\left(-\Omega\left(\frac{n}{2^{2^{i+1}}}\right)\right)
\end{aligned}$$

Note that

$$\log\log n - 1 = \log\log n - \log 2 = \log\frac{1}{2}\log n = \log\log\sqrt{n}$$

Now, for $i \leq \log\log n - 2$, we have $n/2^{2^{i+1}} = \Omega(\sqrt{n})$, and so the bound on S_{i+1} holds with probability $1 - exp(\Omega(\sqrt{n}))$.

One might observe that if even a single S_i fails to achieve the given bound, then our arguments may not hold for all subsequent $S_j, j > i$ either. Fortunately, because the probability of failure is exponentially small, we can directly apply the union bound to show that *every* $S_i \leq 2^{2^i}$ with probability at least $1 - exp(n^{\Omega(1)})$. Thus, the inductive step holds for all $i = 1 \ldots \log\log n - 2$, proving the claim for $i \leq \log\log n - 1$.

We consider $S_{\log\log n}$ separately. Note that we could not hope to prove $S_{\log\log n} \leq 1$ using a Chernoff bound, since only a constant number of processors will reach $L_{\log\log n}$ in expectation. Fortunately, we have already shown that

$$S_{\log\log n - 1} \leq \frac{n}{2^{2\log\log n - 1}} = \sqrt{n}$$

Fix $i = \log\log n$. Lemma 4.1 still applies to L_{i-1} by the inductive result. In particular, the probability that a particular processor fails to find a name in L_{i-1} is at most $1/2^{2^{i+2}} \leq 1/n^4$. As mentioned, we have $S_{i-1} \leq \sqrt{n}$. By the union bound, the probability that even one out of $\sqrt{n}$ trials fails, where each succeeds with probability at least $1 - 1/n^4$, is at most $1/n^3$, so $S_i \leq 1$ with high probability. Obviously, once a processor is the only processor at a level, its `CAS` attempts cannot fail, so $S_j = 0$ w.h.p. for $j > \log\log n$, concluding the proof.

5. LONG LIVED EXECUTIONS

In this section, we consider introducing `Free` to the data structure described above, so that processors can relinquish ownership of a name, and subsequent calls to `Get` can again return that name. This procedure is analogous to a hash table which supports deletes along with inserts.

The implementation of `Free` is natural: when a processor which owns name l calls `Free`, simply set $A[l] = 0$. However, it is not clear that a particularly bad sequence of calls to `Get` and `Free` will not cause the data structure to enter a bad state for which the analysis of section 4.2 does not hold. Section 5.1 resolves this concern for executions over at most polynomially many operations (in n) by showing that the data structure never enters a 'bad' state (to be defined later) with high probability.

Unfortunately, over executions lasting arbitrarily long, high probability results are not enough to guarantee that the structure *never* enters a bad state. In section 5.2 we present sketch a proof which shows that even if the execution enters a bad state, it returns to a 'good' state (i.e. one in which the analysis of section 4.2 does hold) within polynomially many more operations.

Observe that this result guarantees that *a priori* any particular operation succeeds in $O(\log\log n)$ compare-and-swap operations. Without such a result, it is conceivable that once a bad state is reached, the structure spirals out of control, never to return to a good state. In such a scenario, an operation appearing very late in the execution (late enough that a bad state has been inevitably reached, since this happens with non-zero probability) would be unlikely to succeed quickly in any execution.

5.1 Polynomial Executions

We will again proceed by induction. In particular, we will induct over successful `CAS` attempts. We claim that each successful `CAS` occurs in L_i with probability at most $1/2^{2^{i+2}}$ for every $i \geq 1$. Note that this result directly implies that each `Get` operation reaches $L_{\log\log n}$ with probability $1/n^4$, and so every operation succeeds in $\log\log n$ steps with high probability.

Clearly, the first `CAS` succeeds in L_0 with probability 1, satisfying the claim. Now, consider a particular `CAS`, say executed by processor A, and suppose every previous successful `CAS` satisfies the claim. Because the adversary must fix the schedule in advance, the probability that a given processor's name is in L_i at any timestep is the same as the probability that that processor's name was in L_i at the moment it completed its `Get` operation. Note that this is *not* the case against a strong adversary, as such an adversary could choose to leave a processor in, for example, L_2 forever once it lands there.

In particular, this observation allows us to invoke the inductive hypothesis to show that the probability that any given processor is in L_i at any point during processor A's execution of `Get` is at most $1/2^{2^{i+2}}$. Then, whenever A makes a `CAS` attempt in any level (say L_i), the precondition of lemma 4.2 is satisfied for every i, and so $S_i \leq 2^{2^i}$. Therefore, the pre-

condition of lemma 4.1 is satisfied, and so A is assigned a name in L_i with probability at most $1/2^{2^{i+2}}$, completing the proof.

Remark: Notice that in the proof of lemma 4.2, we showed $S_i \leq 2^{2^i}$ with probability at least $1 - e^{-\Omega(\sqrt{n})}$ for every $i < \log\log n$, but only probability at least $1 - 1/n$ for $i = \log\log n$. We can improve this probability using the strong version of lemma 4.1 so that any given processor fails to find a name in $L_{\log\log n-1}$ with probability $1/2^{2^{\log\log n+k}} = 1/n^{2^k}$. Then, taking the union bound over the $O(\sqrt{n})$ processors which have made it to $L_{\log\log n-1}$, the probability that no collisions occur in $L_{\log\log n-1}$ is at least $1-1/n^{2^k-1}$.

In particular, for any fixed polynomial $O(n^r)$ we choose, we can pick c so that lemma 4.2 holds with probability at least $1 - 1/n^r$. This improved result allows us to take a union bound over an entire polynomial length execution to show that, with high probability, *no* `Get` operation takes more than $\log\log n$ steps.

5.2 Infinite Executions (sketch)

Given an infinite execution, eventually, any state which can be reached with any non-zero probability will eventually be acheived. However, we claim that, starting from any initial assignment of names to processors, after polynomially many further operations using the `LevelNaming` algorithm, the bounds $S_j \leq 3n/2^{2^j}$ will hold with high probability. We first present the following lemma:

LEMMA 5.1. *Consider the `LevelNaming` algorithm and any initial assignment of names to processors. Consider the following sequence of operations: select a processor uniformly at random from those which have been assigned a name. That processor calls `Free` and then `Get` sequentially (i.e. no other processors perform operations during this time). After repeating this procedure polynomially many times, the bounds $S_j \leq 3n/2^{2^i}$ will hold with high probability.*

One can argue that the oblivious adversary cannot do any better than choosing processors at random to `Free`, and therefore lemma 5.1 implies that the `LevelNaming` algorithm succeeds in $O(\log\log n)$ steps with high probability for every operation in an infinite execution. Although we will not prove the above remark in full detail, we will provide a complete proof of lemma 5.1.

Remark: Even the oblivious adversary can break the `LevelNaming` algorithm from some initial states *if* the adversary knows the initial condition. For example, suppose initially L_i is completely full for every $i \geq 4$. These levels have total size $n/2$, leaving $n/2$ processors in L_0, L_1, L_2, L_3. If the adversary only allows processors from this latter set of $n/2$ processors to act, then with (small) constant probability, `Get` operations will reach L_4, and be forced to traverse all $\log n$ levels, eventually 'falling off' and traversing the namespace linearly (as described in section 4.1). Fortunately, in an actual execution, the oblivious adversary can not identify which processors are in which level with any significant accuracy, invalidating this approach.

To prove the lemma, define S_i as described in section 4.2 and write $\alpha_i = \lfloor n/2^{2^i} \rfloor$. Recall that we gave α_i as an upper bound on S_i in previous sections, and our analysis of section 5.1 will hold as long as $S_i \leq \delta\alpha_i$ holds for every i and some constant δ (see lemma 4.1). We then define a potential function $\Phi = \sum_{i=1}^{\log n} \Phi_i$ with:

$$\Phi_i = \begin{cases} 0 & : \quad S_i < 2\alpha_i \\ \sum_{j=S_j-2\alpha_i}^{\alpha_i} \frac{1}{2^j} & : \quad 2\alpha_i \leq S_i \leq 3\alpha_i \\ \sum_{j=0}^{\alpha_i} \frac{1}{2^j} + S_i - 3\alpha_i & : \quad S_i > 3\alpha_i \end{cases}$$

An easy way to think about this function is as follows, each of the first $2\alpha_i$ processors with a name in L_i contribute 0 to Φ_i. Then, each of the next α_i processors contribute $1/2^{\alpha_i}, 1/2^{\alpha_i-1}, \ldots, 1/2$ to Φ_i respectively. All further processors contribute exactly 1 to Φ_i.

We make two important observations about Φ. First, $\Phi \leq n$, since, as described above, no processor ever contributes more than 1 to any Φ_i. Secondly, $\Phi = 0$ whenever $S_i \leq 2\alpha_i$ for every i, and if, for *any* j, $S_j \geq 3\alpha_i + 1$, then $\Phi > 1$. We will show that if there exists an i with $S_i > 3\alpha_i$, then after randomly selecting a processor and allowing that processor to call `Free` and then `Get`, $\mathbb{E}[\Delta\Phi] \leq -1/3n$. Thus after at most $n\Phi \leq O(n^2)$ steps in expectation, no i with $S_i > 3\alpha_i$ can exist (otherwise $\Phi < 1$ would hold, a contradiction). Applying a standard weighted random walk argument gives the same result with high probability after $O(n^3)$ steps.

We now compute $\mathbb{E}[\Delta\Phi]$. By linearity of expectation, $\mathbb{E}[\Delta\Phi] = \sum \mathbb{E}[\Delta\Phi_i]$. Let i_0 be the smallest value of i such that $S_i \geq 3\alpha_i$. i_0 exists by assumption (otherwise there is nothing to prove) and $i_0 \leq \log n$ holds, since there are only $\log n$ levels. First, after the `Free`,

$$\mathbb{E}[\Delta\Phi_{i_0}] \leq -S_{i_0}/n \leq -3\alpha_i/n = -3/2^{2^i}$$

We can apply lemma 4.1 to all $i \leq i_0$, so the probability that the `Get` operation reaches level L_{i_0} or higher from L_{i_0-1} is at most $1/2^{2^{i_0+2}} \leq 1/2^{2^{i_0}}$. Furthermore, regardless of which level $L_{i_0}, L_{i_0+1}, \ldots$ the name returned by `Get` falls into, Φ increases by at most 1. Thus,

$$\sum_{i=i_0}^{\log n} \mathbb{E}[\Delta\Phi_i] \leq -3/2^{2^{i_0}} + 1/2^{2^{i_0}} \leq -2/2^{2^{i_0}} \leq -1/n \quad (2)$$

Note that this last inequality assumes $i_0 \leq \log\log n$. If this is not the case, then $S_i \leq 2n/2^{2^i}$ for all $i \leq \log\log n$, so no processor reaches level L_{i_0} with high probability (in particular with probability $1 - 1/n^r$ for some large constant r), by the analysis of section 5.1. However, L_{i_0} cannot be empty by construction of i_0, so there is still a probability at least $1/n$ of selecting a processor from L_{i_0} (or higher) to `Free` (thereby decreasing Φ by 1), so $\mathbb{E}[\Delta\Phi_i] \leq -1/n$ still holds.

For $i < i_0$, $S_i < 3n/2^{2^i}$ holds by construction of i_0. We consider three cases. If $S_i < 2\alpha_i - 1$, then $\mathbb{E}[\Delta\Phi_i] = 0$,

since neither adding or removing a processor from level L_i changes Φ_i at all.

Suppose $2\alpha_i - 1 < S_i < 3\alpha_i$. Write $k = S_i - 2\alpha_i$. The probability of removing a processor from L_i via the `Free` is $S_i/n \geq 2/2^{2^i}$, which would decrease Φ_i by $1/2^k$. By lemma 4.1, the probability of `Get` returning a name in L_i is at most $1/2^{2^i}$, which would increase Φ_i by $1/2^{k-1}$. Thus:

$$\mathbb{E}[\Delta\Phi_i] \leq \frac{2}{2^{2^i}}\left(-\frac{1}{2^k}\right) + \frac{1}{2^{2^i}}\left(\frac{1}{2^{k-1}}\right) = 0$$

Finally, if $S_i = 2\alpha - 1$, then removing a processor from L_i does not change Φ_i at all, but adding one increases Φ_i by $1/2^{\alpha_i}$. Lemma 4.1 still applies, so the new processor is assigned a name in L_i with probability at most $1/2^{2^i}$. In this case,

$$\mathbb{E}[\Delta\Phi_i] \leq \frac{1}{2^{2^i}}\frac{1}{2^{n/2^{2^i}}} = \frac{1}{2^{2^i + n/2^{2^i}}}$$

One can easily verify that, for discrete i, this value is maximized when $i = \log\log n$, in which case $\Delta\Phi_{\log\log n} \leq 1/2n$. For $i \geq \log\log n + 1$, the 2^i term dominates, and $\Delta\Phi_i \leq 1/n^2$. For $i \leq \log\log n - 1$, the $n/2^{2^i}$ term dominates, and $\Delta\Phi_i \leq 1/2^{\sqrt{n}}$. Recalling that $i_0 \leq \log n$ since there are only $\log n$ levels, we get:

$$\sum_{i=1}^{i_0-1}\mathbb{E}[\Delta\Phi_i] \leq \frac{\log\log n}{2^{\sqrt{n}}} + \frac{1}{2n} + \frac{\log n}{n^2} = \frac{1}{2n} + o\left(\frac{1}{n}\right) \quad (3)$$

Combining equations (2) and (3) gives

$$\begin{aligned}\sum_{i=1}^{\log n}\mathbb{E}[\Delta\Phi_i] &= \sum_{i=1}^{i_0-1}\mathbb{E}[\Delta\Phi_i] + \sum_{i=i_0}^{\log n}\mathbb{E}[\Delta\Phi_i] \\ &\leq -\frac{1}{n} + \frac{1}{2n} + o\left(\frac{1}{n}\right) \\ &\leq -\frac{1}{3n}\end{aligned}$$

as desired, completing the proof.

6. CONCLUSIONS

We have described an algorithm for randomized loose, long lived renaming against an oblivious adversary for which every operation succeeds in at most $O(\log\log n)$ steps with high probability. This result holds even for infinite executions and demonstrates a separation from algorithms against a strong adversary, for which $\Omega(\log n)$ lower bounds hold.

6.1 Possible Improvements

With some care, it should be possible to acheive the result of lemma 4.1 for a namespace of size $(1+\epsilon)n$ for any $\epsilon > 0$ (although the value of c required will grow with $1/\epsilon$). With lemma 4.1 holding, the rest of the analysis of sections 4.2 and 5 will follow as well. This result would provide a scheme describing a $O(f(1/\epsilon)\log\log n)$ algorithm for renaming into any namespace which is arbitrarily close to tight.

If one is satisfied with amortized complexity, it should also be possible to completely reassign names every so often (approximately once per polynomially many operations) to effectively guarantee that 'bad' states do not persist for long, even without relying on the results of section 5. This may perform better in practice, and would likely provide $O(\log\log n)$ amortized bounds even against a strong adversary.

Finally, it is worth noting that our algorithm can be utilized as a hash table. One requires $c\log n$ independent hash functions (although practically only $c\log\log n$ are necessary). Inserts and searches try each hash in turn analogously to the `LevelRenaming` algorithm until an empty slot (for inserts) or the desired key (for searches) is found. Assuming the hash functions (or alternatively, the keys) are chosen at random, our $O(\log\log n)$ bound should hold even for infinite executions against an adversary who must choose the sequence of operations in advance.

6.2 Future Work

There is still a complexity gap between the $O(\log\log n)$ upper bound for loose renaming given in this paper, and the trivial $\Omega(1)$ lower bound. We hope future research may resolve this question, either with a faster algorithm, or contrarily with an improved lower bound.

7. ACKNOWLEDGMENTS

This work was conducted jointly with Dan Alistarh, and the author gratefully acknowledges his contribution to these results and his helpful feedback on this paper. The author would also like to thank Nir Shavit for his supervision and insight throughout this research.

8. REFERENCES

[1] Dan Alistarh. The level renaming algorithm. Personal communication, 2012.

[2] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Morteza Zadimoghaddam. Optimal-time adaptive strong renaming, with applications to counting. In *PODC*, pages 239–248, 2011.

[3] Dan Alistarh, James Aspnes, Seth Gilbert, and Rachid Guerraoui. The complexity of renaming. In *FOCS*, pages 718–727, 2011.

[4] Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Corentin Travers. Brief announcement: New bounds for partially synchronous set agreement. In *DISC*, pages 404–405, 2010.

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

[6] Alex Brodsky, Faith Ellen, and Philipp Woelfel. Fully-adaptive algorithms for long-lived renaming. In *DISC*, pages 413–427, 2006.

[7] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *JOURNAL*

OF THE ACM, 46:858–923, 1996.

[8] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.