Fully-Adaptive Algorithms for Long-Lived Renaming

Alex Brodsky[1], Faith Ellen[2], and Philipp Woelfel[2]

[1] Dept. of Applied Computer Science, University of Winnipeg, Winnipeg, Canada,
a.brodsky@uwinnipeg.ca
[2] Dept. of Computer Science, University of Toronto, Toronto, Canada,
{faith,pwoelfel}@cs.utoronto.ca

Abstract. Long-lived renaming allows processes to repeatedly get distinct names from a small name space and release these names. This paper presents two long-lived renaming algorithms in which the name a process gets is bounded above by the number of processes currently occupying a name or performing one of these operations. The first is asynchronous, uses `LL/SC` objects, and has step complexity that is linear in the number of processes, c, currently getting or releasing a name. The second is synchronous, uses registers and counters, and has step complexity that is polylogarithmic in c. Both tolerate any number of process crashes.

1 Introduction

Renaming is an interesting and widely-studied problem that has many applications in distributed computing. For *one-shot renaming*, each of the n processes in the system can perform `GetName` to get a distinct name from a small name space, $\{1,\ldots,m\}$. For *long-lived renaming*, a process that has gotten a name, x, can also perform `RelName(`x`)`, so that it or another process can get this name later. In this more general version of the renaming problem, each process can alternately perform `GetName` and `RelName` any number of times (starting with `GetName`). After a process performs `GetName`, its name stays the same until after it next performs `RelName`. If a process performs `GetName` again, it may get the same name it got previously, or it may get a different name.

One application of long-lived renaming is the repeated acquisition and release of a limited number of identical resources by processes [12], something that commonly happens in most operating systems. In this case, names correspond to resources that are acquired and released by processes. Each process that wants to acquire a resource performs `GetName` to get a name, which is essentially permission for exclusive use of the resource. When a process no longer needs the resource, it performs `RelName`.

Another application of renaming is to improve the time or space complexity of an algorithm when only few processes participate [5]. For example, the time complexity of an algorithm may depend on the maximum number of processes that could participate, because it iterates over all processes in the system. Then a faster algorithm can be obtained by having each participating process first get a new name from a small name space and iterating over this smaller space. For some algorithms, such as those that implement shared objects, a process may participate for only short, widely spaced periods of time. Better performance can be achieved if each process performs `GetName` whenever it begins participating and performs `RelName` when it has finished participating for a while [21].

In one-shot renaming, the number of processes, k, that are getting or have gotten a name grows as an execution proceeds. In many applications, the final value of k is not known in advance. To avoid using an unnecessarily large name space, the size of the name space should be small initially and grow as k grows.

In long-lived renaming, we say that a process is *participating* from the time it begins a `GetName` operation until it completes its subsequent `RelName` operation. In particular, a process participates forever if it fails before releasing the last name that it got. The number of participating processes can increase and decrease during an execution of long-lived renaming. When the number is small, a process should get a small name, even though other processes may have received much larger names earlier in the execution (and may still be participating).

An $m(k)$*-renaming* algorithm [5] is a renaming algorithm in which a process always gets a name in the range $\{1,\ldots,m(k)\}$, where k is the number of processes that participate while it performs `GetName`. Note that, by the pigeonhole principle, $m(k)$-renaming is impossible unless $m(k) \geq k$. The special case when $m(k) = k$ is called *strong renaming*.

The cost of performing renaming is also an issue. Renaming algorithms in which the time complexities of `GetName` and `RelName` are bounded above by a function of the number of participating processes, k, are called *adaptive*. A renaming algorithm whose time complexity only depends on the number of processes, c, concurrently performing `GetName` or `RelName`, but not on the number of names that are in use, is called *fully-adaptive*. This is even better, because k can be much larger than c.

For some renaming algorithms, each process is assumed to have an identifier, which is a name from a large original name space. Other renaming algorithms also work when the original name space is infinite or when processes are anonymous (i.e. they have no original names).

Related Work. The renaming problem has been studied extensively in asynchronous systems beginning with Attiya, Bar-Noy, Dolev, Peleg, and Reischuk [5], who studied one-shot renaming in asynchronous message passing systems. They proved that strong one-shot renaming is impossible, even if only one process can fail. They also proved that a process cannot decide on a new name until it has received messages (either directly or indirectly) from at least half of the other processes. Thus, in this model, adaptive one-shot renaming is impossible and one-shot renaming is impossible unless more than half of the processes in the system participate. In addition, they give two algorithms for one-shot renaming which assume that more than $n/2$ processes participate. The size of the name space in their algorithms also depends on n.

For synchronous message passing systems, Chaudhuri, Herlihy, and Tuttle [19, 13] gave an algorithm for one-shot strong renaming with $O(\log k)$ rounds in which a process may send a message to any subset of other processes. They also proved a matching lower bound for comparison-based algorithms. Attiya and Djerassi-Shintel [6] studied the complexity of one-shot strong-renaming in semisynchronous message passing systems that are subject to late timing faults. They obtained an algorithm with $O(\log k)$ rounds of broadcast from a synchronous message passing algorithm and proved an $\Omega(\log k)$ lower bound for comparison based algorithms or when the original name space is sufficiently large compared to k.

Bar-Noy and Dolev [11] showed how to transform the asynchronous message passing algorithms of Attiya, Bar-Noy, Dolev, Peleg, and Reischuk [5], to shared-memory, using only reads and writes. They obtained a one-shot $\frac{k^2+k}{2}$-renaming algorithm that uses $O(n^2)$ steps per operation and a one-shot $(2k-1)$-renaming algorithm that uses $O(n \cdot 4^n)$ steps per operation.

Burns and Peterson [12] proved that long-lived m-renaming is impossible in an asynchronous shared memory system using only reads and writes unless $m \geq 2k-1$. They also gave the first long-lived $(2k-1)$-renaming algorithm in this model, but its time complexity depends on the size of the original name space. Herlihy and Shavit [18] proved the same lower bound on m for one-shot renaming. Herlihy and Rasjbaum [17] extended this result to systems that also provide set-consensus primitives.

In asynchronous shared memory systems using only reads and writes, the fastest adaptive one-shot $(2k-1)$-renaming algorithm has $O(k^2)$ step complexity [3]. There are also an adaptive one-shot $(6k-1)$-renaming algorithm with $O(k \log k)$ step complexity [8] and an adaptive one-shot $O(k^2)$-renaming algorithm with $O(k)$ step complexity [22, 8]. For adaptive long-lived renaming, $O(k^2)$-time suffices for $O(k^2)$-renaming [1, 4, 9, 20], but the fastest $(2k-1)$-renaming algorithm has $O(k^4)$ step complexity [7]. There are no fully-adaptive one-shot or long-lived renaming algorithms in this model.

With a single object that supports `Fetch&Increment`, fully-adaptive one-shot strong renaming is very easy: The object is initialized to 1. To get a name, a process simply performs `Fetch&Increment` and uses the responses as its new name [13].

For long-lived renaming, the only fully-adaptive algorithms use much stronger primitives. Moir and Anderson [22] presented a fully-adaptive long-lived strong renaming algorithm that uses an n-bit object storing the characteristic vector of the set of occupied names. This object supports two operations, `SetFirstZero` and bitwise-AND. `SetFirstZero` sets the first 0 bit in the object to 1 and returns the index of this bit. This index becomes the new name of the process that performed the operation. To release this name, the process sets this bit back to 0 by performing bitwise-AND with an n-bit string that has a single 0 in the corresponding position. A similar algorithm can be obtained using an n-bit `Fetch&Add` object in a synchronous system. These algorithms have constant time complexity. However, they use very large objects.

There are a number of adaptive, long-lived strong renaming algorithms from smaller, standard objects. For example, consider an array of n `Test&Set` objects, each initialized to 0. To get a name, a process performs `Test&Set` on the array elements, in order, until it gets response 0. Its name is the index of the element from which it gets this response. To release the name i, a process performs `Reset` on the i'th array element. This algorithm, presented in [22], performs `GetName` in $O(k)$ time and `RelName` in constant time. A similar algorithm uses a dynamically allocated doubly-linked list implemented using `Compare&Swap`, in which each node has a counter containing the number of larger names that are currently occupied, being acquired or being released [16].

The same idea can be used to obtain an adaptive long-lived strong renaming algorithm in a synchronous shared memory with $\lceil \log_2(U+1) \rceil$-bit registers, where U is the size of the original name space. This is because a `Test&Set` object can be implemented from a $\lceil \log_2(U+1) \rceil$-bit register in a synchronous system in constant time: a process competes for an unset `Test&Set` object by writing its name to a corresponding register.

Adaptive long-lived strong renaming can also be performed in $O(\log k)$ time. Use a sequence of $O(\log n)$ complete binary trees. The first tree has height 1 and the height of each subsequent tree increases by 1. The leaves of these trees correspond to the new names and a process gets a name by acquiring a leaf. Each node in the tree contains a counter that denotes the number of free leaves in the subtree rooted at that node. To acquire a leaf, a process accesses the counters at the root of each tree until it finds one with a free leaf. It then proceeds down the tree to find a free leaf, using the value of the counter at each node to guide its search and decrementing the counters on this path as it descends. To release its name, the process starts at the corresponding leaf and walks up the tree, incrementing the counter in each node it visits, until it reaches the root. In a synchronous system, each counter can be implemented by an $O(\log n)$-bit object that supports `Fetch&Decrement` and `Increment`. In an asynchronous system, a bounded version of `Fetch&Decrement` is needed, which does not change the value of the object when it is 0. (See [22] for details.)

Our Results. In this paper, we present two new fault tolerant and fully-adaptive algorithms for long-lived strong renaming. They are the first such algorithms that do not rely on storing a representation of the set of occupied names in a single (very large) object. The first algorithm is asynchronous and uses $\Theta(\log U)$-bit `LL/SC` objects, where U is the size of the original name space. Its step complexity is $O(c)$, where c is the number of processes concurrently performing `GetName` or `RelName`. The second algorithm is synchronous, uses $O(\log n)$-bit counters and registers, and has $O(\log^3 c/\log\log c)$ step complexity. Both algorithms tolerate any number of process crashes.

The key to both algorithms is an interesting sequential data structure that supports `GetName` and `RelName` in constant time. We then apply a universal adaptive construction by Afek, Dauber and Touitou [2] to obtain our fully-adaptive asynchronous renaming algorithm. This is presented in Section 3. In Section 4, we develop our fully-adaptive synchronous algorithm. Directions for future work are discussed in Section 5.

2 Models

We consider models of distributed systems in which n deterministic processes run concurrently and communicate by applying operations to shared objects. Our implementations make use of (multi-writer) registers, counters, and `LL/SC` objects. A register stores an integer value and supports two operations: $\ell \leftarrow$ `read`(R), which reads register R and assigns the value to the local variable ℓ, and `write` $R \leftarrow \ell$, which writes the value of ℓ into register R. A counter, C, supports `read`, `write`, $\ell \leftarrow$ `Fetch&Increment`(C), and $\ell \leftarrow$ `Fetch&Decrement`(C). These last two operations assign the current value of C to ℓ and then increment and decrement C, respectively. An `LL/SC` object, O, supports `write` and $\ell \leftarrow$ `Load-Linked`(O), which reads the value in object O and assigns it to ℓ. It also supports `Store-Conditional` $O \leftarrow \ell$, which stores the value of ℓ to O only if no store to O has occurred since the same process last performed `Load-Linked`(O). In addition, this operation returns a Boolean value indicating whether the store occurred. We assume that all operations supported by these objects occur atomically.

In asynchronous systems, an adversarial scheduler decides the order in which processes apply operations to shared objects. The adversary also decides when processes

begin performing new instances of GetName and RelName. In synchronous systems, the order chosen by the adversary is restricted. Time is divided into rounds. In each round, every process that is performing an instance of GetName or RelName (and has not crashed) applies one operation. The order chosen by the adversary can be different for different rounds. We assume that the adversary only begins new instances of GetName or RelName in a round if no other instances are in progress. We also assume that the adversary does not begin an instance of GetName and an instance of RelName in the same round. These assumptions can be removed by having a flag which processes update frequently to indicate they are still working on the current batch of operations. Other processes wait until a batch is finished before starting a new batch. This is discussed in more detail in the full version of the paper.

In both models, we measure the complexity of an instance of GetName or RelName by the number of operations it applies. A process that crashes simply performs no further operations. Our algorithms tolerate any number of process crashes.

3 Asynchronous Renaming

Afek, Dauber, and Touitou [2] have shown that, using LL/SC objects, fast sequential implementations of an object suffice for obtaining fully-adaptive implementations:

Theorem 1. *If an object has a sequential implementation in which an update takes a constant number of steps, then it also has a fully-adaptive implementation from* LL/SC *objects in an asynchronous system such that an update takes $O(c)$ steps, where c is the number of processes that update the object concurrently.*

This is a special case of their universal construction, which maintains a queue of operations to be performed on the object. To perform an operation, a process records the operation it wants to perform and enqueues its identifier. Then it repeatedly helps the processes at the head of the queue to complete their operations, until its own operation is completed. Processes use Load-Linked and Store-Conditional to agree on the results of each operation and how the value of the object changes.

We consider a renaming object whose value is the subset of free names in $\{1,\ldots,n\}$. Here n is the number of processes in the system and, thus, an upper bound on the number of names that will ever be needed. This object supports two operations, GetName and RelName. If $\mathcal{F}$ is the set of free names, then an instance of GetName removes and returns a name from $\mathcal{F}$ which is less than or equal to the maximum number of participating processes at any point during the execution of the instance. RelName(x) returns x to $\mathcal{F}$. It can only be applied by the last process that received x from a GetName operation.

Next, we describe a data structure for representing $\mathcal{F}$ and constant time sequential algorithms for performing GetName and RelName. From these, we get our first fully-adaptive, long-lived, strong renaming implementation, by applying Theorem 1.

Theorem 2. *In an asynchronous system in which processes communicate using* LL/SC *objects, there is a fully-adaptive implementation of long-lived strong renaming which performs* GetName *and* RelName *in $O(c)$ steps.*

The `LL/SC` objects used by this implementation must be able to store process identifiers. If these identifiers are from an original name space of size U, then the `LL/SC` objects must have at least $\lceil \log_2 U \rceil$ bits.

Data Representation. The subset $\mathcal{F} \subseteq \{1,\ldots,n\}$ of free names is represented by two arrays of n registers, D and L, and two counters, M and F. (In the sequential implementation, which is related to Hagerup and Raman's quasidictionary [15], the counters M and F can be replaced by registers.) The values in M, F, and the entries of D are in $\{0,1,\ldots,n\}$. The entries in L are in $\{1,\ldots,n\}$. The array D is used as a direct access table. We say that name i is *occupied* if $D[i]=0$. If i is occupied and p was the last process that wrote 0 to $D[i]$, then we say that p *occupies* name i. The following invariant will be maintained:

> At any point, if a process p has received name i as its response from a call of `GetName` and, since then, has not called `RelName(`i`)`, then i is occupied by p. (1)

A name that is neither occupied nor free is called *reserved*. Reserved names occur while processes are performing `GetName` and `RelName`, and because of process failures. A name $i > M$ is free if and only if it is unoccupied.

The variable F is the number of free names less than or equal to M and $L[1..F]$ is an unsorted list of the free names less than or equal to M. In particular,

$$1 \leq L[j] \leq M \text{ for all } 1 \leq j \leq F \tag{2}$$

is an invariant of our data structure. We use $\mathcal{L}$ to denote the set of names in $L[1..F]$. Another important invariant is the following:

$$D\big[L[j]\big] = j \text{ for all } 1 \leq j \leq F. \tag{3}$$

This ensures that all free names are unoccupied, there are F different names in $\mathcal{L}$, and, if $i \in \mathcal{L}$, then $D[i]$ is a pointer to a location in L, between 1 and F, that contains the name i. The function `InL`(i) checks whether name i is in $\mathcal{L}$ using a constant number of operations. Specifically, if $i \in \mathcal{L}$, then `InL`(i) returns $D[i]$, the unique index $1 \leq j \leq F$ such that $L[j]=i$. Otherwise, it returns 0.

Note that $i \in \mathcal{F}$ if and only if either $(1 \leq i \leq M$ and $i \in \mathcal{L})$ or $(i > M$ and $D[i] > 0)$. There are two other invariants that are maintained by the data structure:

$$M \leq \text{ the number of participating processes, and} \tag{4}$$

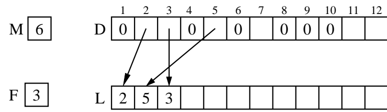
$$\text{the number of occupied names} + \text{ the number of reserved names} \leq M. \tag{5}$$

The data structure is illustrated in Figure 1. Blank entries denote arbitrary values in $\{1,\ldots,n\}$.

Initially, M and F are both 0 and all entries of D are positive, so $\mathcal{F}=\{1,\ldots,n\}$. Furthermore, no processes are participating and there are no occupied or reserved names. Hence, all the invariants are satisfied.

To prove correctness of `GetName` and `RelName`, we must prove that they always maintain the invariants. Furthermore, the name a process receives as its response from a call of `GetName` must be at most the maximum number of participating processes at any point during its execution of this instance of `GetName`.

```
Function GetName
    local: x, j, z
    x ← 1 + Fetch&Increment(M)
    z ← read(D[x])
    if z = 0 then
        j ← Fetch&Decrement(F)
        x ← read(L[j])
    end
    write D[x] ← 0
    return(x)
```




```
Function RelName(x) /* sequential */
    Input: name x
    local: m, f, i, j

    write D[x] ← n
    m ← read(M)
    f ← read(F)
    if x ≤ m then
        /* append x to L[1..F]                */
        f ← f + 1
        write D[x] ← f
        write L[f] ← x
        write F ← f
    end
    j ← InL(m)
    if j > 0 then
        /* remove name m from 𝓛               */
        write F ← f − 1
        i ← read(L[f])
        write D[i] ← j
        write L[j] ← i
    end
    write M ← m − 1
```

Fig. 1. The functions `GetName` and `RelName` and an example of the data structure

GetName. The sequential procedure for getting names is quite straightforward. First, a process that wants to get a name, takes the tentative name $M+1$ and increments M. If this name is unoccupied, the process gets it. If it is occupied, there must be a free name less than or equal to M, since, by Invariant 5, there are at most M names that are occupied or reserved. Thus $\mathcal{L}$ is not empty. In this case, the process decrements F and gets the name that was at the end of the unsorted list. Finally, in both cases, the process occupies the name, x, it got by writing 0 into the corresponding entry, $D[x]$, in the direct access table.

Lemma 1. *No step of* `GetName` *causes any invariant to become invalid. The name returned by a call of* `GetName` *is bounded by the maximum number of participating processes at any point during its execution.*

The step complexity of `GetName` is $O(1)$. A process that fails after incrementing M, but before writing 0 to some element of D, may decrease the number of free names, but does not increase the number of occupied names.

Releasing Names Sequentially. The sequential algorithm for releasing a name consists of two phases. In the first phase, a process changes the status of its name, x, from occupied to free. It begins this phase by writing any positive number, for example n, into $D[x]$. This changes x from being occupied to being unoccupied. If $x > M$, then x is now free. Otherwise, the process has to append x to the end of the list $L[1..F]$. This is

accomplished by writing x into $L[F+1]$, writing $F+1$ into $D[x]$ and then incrementing F. In the second phase, M is decremented. Before doing so, if it is present, the name M must be removed from $\mathcal{L}$. Suppose that it is in location $L[j]$ and that $L[F]=i$. First F is decremented. Then $D[i]$ is updated to point to location j. Finally, i is copied into $L[j]$, overwriting the value M that was stored there. The order in which these operations are performed is important to ensure that the data structure invariants are maintained.

Lemma 2. *If a process occupying the name x calls the sequential version of* `RelName`(x)*, then none of its steps causes any invariant to become invalid.*

The step complexity of this sequential version of `RelName` is $O(1)$. A process that fails after overwriting the 0 in the direct access table entry corresponding to its name, but before decrementing M, decreases the number of occupied names, but might not increase the number of free names.

4 Synchronous Renaming

For synchronous renaming, the data representation is the same as in Section 3. The sequential version of `GetName` also works when multiple processes begin performing new instances at the same time. Recall that, in our model, no instances of `GetName` and `RelName` are performed concurrently. The correctness of `GetName` follows, as in the sequential case, from Lemma 1.

Unfortunately, the sequential version of `RelName` does not work if there are multiple processes simultaneously trying to release their names. For example, several processes could write their names to the same location in L. The solution to this problem is to assign distinct ranks to the processes that want to add their names to $\mathcal{L}$. Then the process with rank i can write its name into location $L[F+i]$.

The function `Rank` can be implemented using a counter, to which processes perform `write(1)` and then `Fetch&Increment`. Similarly, `CountAlive`, which counts the number of processes that perform it simultaneously, can be implemented using a counter which is set to 0, incremented, and then read by each of those processes.

If processes crash after they perform `Rank`, but before they write their names into L, there will be garbage interspersed between the names that have been written. One way to handle this is to write 0's in the portion of L that will be written to before processes write their names there. Afterwards, the names that have been written can be compacted, by removing the 0's that remain. To do this and to solve other similar problems encountered when parallelizing `RelName`, we use three auxilliary functions, `WriteAll`, `Count`, and `Compact`. They are based on the synchronous `DoAll` algorithm by Georgiou, Russell, and Shvartsman [14].

`WriteAll` performs the set of instructions, `write` $dest(i) \leftarrow val(i)$, for $1 \le i \le s$, in a fault tolerant way. Here $dest(i)$ denotes a destination register and $val(i)$ denotes the value to be written there. For example, `WriteAll`$(L[i+f] \leftarrow 0,\ 1 \le i \le s)$ writes 0's to the s locations following $L[f]$. Georgiou, Russell, and Shvartsman [14, Theorem 8] give an implementation of `WriteAll` from multi-writer registers which take $O(\log^2 s/\log\log s)$ steps to complete, provided at least $s/2$ of the processes that are

simultaneously executing it do not fail. However, if too many processes fail, the remaining processes could take too long to complete all s tasks.

To ensure that `RelName` remains fully-adaptive, we modify the function `WriteAll`$(dest(i) \leftarrow val(i),\ 1 \leq i \leq s)$ so that it returns whenever fewer than $s/2$ processes remain. This is accomplished by having processes perform "`if CountAlive` $<$ $s/2$ `then return`" after every constant number of steps. If `WriteAll` terminates in this way, there are no guarantees about which, if any, of the s tasks have been performed. In this case, the call returns ∞ and we say that it fails. Otherwise, `WriteAll` returns s.

Note that it does not suffice for processes to wait for a convenient place in the code, such as the end of a phase, to check whether too many processes have crashed. For example, if there is one process that wants to perform `GetName` just after all but one of the s processes performing `RelName` crash, then the number of participating processes, c, is 2. If the process has to wait to start performing `GetName` until the one surviving process completes a phase (whose complexity depends on s), the resulting implementation will not be fully-adaptive.

The function `Count`$(z(i), 1 \leq i \leq s)$ returns the number of values of $i \in \{1, \ldots, s\}$ for which the predicate $z(i)$ is true, provided that at least $s/2$ of the processes that simultaneously begin executing this function, complete it. Otherwise, it returns ∞ and we say that it fails. All processes that complete a particular instance of `Count` return the same value. The implementation of `Count` is very similar to `WriteAll` and it has the same performance [23, Theorem 5.9].

The function `Compact`$(A, z(i), 1 \leq i \leq s)$ compacts the elements $A[i]$ of the array $A[1..s]$ such that the predicate $z(i)$ is true, so that these elements are stored contiguously starting from the beginning of A. If at least $s/2$ of the processes that simultaneously begin executing this function, complete it, then the processes return the number of $i \in \{1, \ldots, s\}$ for which $z(i)$ is true. If not, the call fails, returning the value ∞, and the contents of $A[1..s]$ can be arbitrary. `Compact` has the same performance as `WriteAll`.

Lemma 3. *If $val(i)$, $dest(i)$, and $z(i)$ can be computed by a process in a constant number of steps, then there are implementations of* `Compact`$(A, z(i), 1 \leq i \leq s)$*,* `Count`$(z(i), 1 \leq i \leq s)$*, and* `WriteAll`$(dest(i) \leftarrow val(i),\ 1 \leq i \leq s)$ *that have $O(\log^2 s / \log\log s)$ step complexity and return within a constant number of steps when fewer than $s/2$ of the processes which simultaneously began executing the same instance remain.*

4.1 Releasing Names.

The procedure for releasing names consists of two phases. In the first phase, a process changes the status of its name x. If $x > M$, then it can simply write n (or any other value larger than 0) into $D[x]$ and the name becomes free right away. Otherwise, the process has to insert the name into list $\mathcal{L}$. This is achieved by a call to the procedure `InsertInL`(x). If several processes call `InsertInL`, but some of them fail, then the number of names that are inserted into $\mathcal{L}$ will be at least the number of these processes that complete their calls. Each name inserted into $\mathcal{L}$ will be a name that was occupied by one of the calling processes. However, it is not necessarily the case that the inserted names include all those that were occupied by the processes completing their calls.

```
Function RelName(x)
/* synchronous */
  Input: name x
  local: m, success

  m ← read(M)
  if x ≤ m then
      InsertInL(x)
  else
      write D[x] ← n
  end
  RemoveLargeNames
```

```
Function InsertInL(x)
  Input: name x
  local: f, i, s
  register: R

  f ← read(F)
  repeat
      s ← CountAlive
      WriteAll(L[f+i] ← 0, 1 ≤ i ≤ s)
      i ← Rank
      write L[f+i] ← x
      a ← Compact(L[f+1..f+s], L[f+i] > 0,
                  1 ≤ i ≤ s)
      WriteAll(D[L[f+i]] ← f+i, 1 ≤ i ≤ a)
  until CountAlive ≥ a/2
  write F ← f+a
```

In the second phase, the processes fix the invariants that involve M. For the upper bound on M in Invariant (4), we have to reduce M by at least s, if s processes complete their call to RelName (and thus stop participating). To avoid violating the lower bound on M in Invariant (5), we cannot reduce M by more than ℓ, if ℓ processes successfully complete the first phase. Moreover, before we can reduce M from m to m', we have to remove all names in $\{m'+1,\ldots,m\}$ from the list $\mathcal{L}$ to maintain Invariant (2). This procedure of removing the desired names from $\mathcal{L}$ and reducing M is performed by the procedure RemoveLargeNames. The implementation guarantees that if ℓ processes call RemoveLargeNames during their call to RelName and s of them finish, then M is reduced by at least s and at most ℓ.

Lemma 4. *Suppose a set of processes, each occupying a name with value at most M, simultaneously call* InsertInL. *If ℓ of these processes complete that call, then at least ℓ names are added to $\mathcal{L}$. The invariants remain true throughout the execution of the call. At any point during the execution, if ℓ' of these processes have not failed, then the call terminates within $O(\log^3 \ell' / \log\log \ell')$ steps.*

Lemma 5. *Suppose a set of ℓ processes simultaneously call* RemoveLargeNames. *If s of these processes complete their call, then $m-\ell \leq m' \leq m-s$, where m and m' are the values of M immediately before and after the call to* RemoveLargeNames. *The invariants remain true throughout the execution of the call. At any point during the execution, if ℓ' of these processes have not failed, then the call terminates within $O(\log^3 \ell' / \log\log \ell')$ steps.*

The correctness of RelName follows from these lemmas. In the remainder of this section, we show how to implement InsertInL and RemoveLargeNames.

Inserting into $\mathcal{L}$. In Procedure InsertInL, processes first write their names to distinct locations following the end of the list $L[1..F]$ and update the entries in the direct access

table D to point to these names. Multiple attempts may be needed, due to process failures. Once they succeed, the processes increment register F, which moves these names into $\mathcal{L}$.

In each attempt, a process first uses `CountAlive` to find the number of surviving processes, s, that want to free their names. Then, using `WriteAll`, array elements $L[F+1], \ldots, L[F+s]$ are initialized to 0. Next, each of these processes writes its name into $L[F+i]$, where i is a unique rank between 1 and s assigned to it by `Rank`. Since some of these processes may have failed prior to writing their names, a, the number of names that were written, may be less than s. `Compact` is performed to compact $L[F+1..F+s]$, so that these a names are stored in $L[F+1..F+a]$. Finally, another `WriteAll` is performed, setting $D\big[L[F+i]\big] = F+i$, for $i = 1, \ldots, a$, to ensure that Invariant (3) will hold after F is increased to $F+a$. If fewer than $a/2$ processes complete the attempt, another attempt is made with s decreased by at least a factor of 2.

If at least $a/2$ processes complete the attempt, then neither of the calls to `WriteAll` nor the call to `Compact` failed. In this case, F is incremented by a, moving the a names that were appended to L into $\mathcal{L}$. Note that a is the number of processes that write their names into $L[F+1..F+s]$ during the last attempt and, hence, it is an upper bound on the number of processes that complete the call.

By Lemma 3, each attempt that starts with s processes takes $O(\log^2 s/\log\log s)$ steps. Since s decreases by at least a factor of 2 each subsequent attempt, there are at most $\log s$ attempts, for a total of $O(\log^3 s/\log\log s)$ steps, from this point on. If, at any point during an attempt, more than three quarters of the processes that started the attempt have failed, then within a constant number of steps, the attempt will fail.

Removing Large Names from $\mathcal{L}$. Procedure `RemoveLargeNames` is called simultaneously by every process performing `RelName`, after they no longer occupy their names. Thus, processes with names greater than M must wait before they begin `RemoveLargeNames`, until the processes that are performing `InsertInL` are finished.

First, consider an execution in which s processes call `RemoveLargeNames` and none of them crash. In this case, M will be reduced to $M-s$ after all the *large* names, that is, those which are greater than $M-s$, are removed from $\mathcal{L}$. Names $M-s$ or less will be called *small*.

After processes store the value of F in their local variable f, they determine the number, a, of large names in $L[1..f]$ using `Count`. Next, they copy the last a names in $L[1..f]$ to a temporary array H_1 using `WriteAll`. Then, they remove the large names from H_1 using `Compact`. At this point, $H_1[1..b]$ consists of all the small names in the last a locations of $L[1..f]$. When F is decreased to $f-a$, these small names are no longer in $\mathcal{L}$ and become reserved, instead of free.

The number of large names that remain in $\mathcal{L}$ is exactly equal to b, the number of small names in H_1. These large names are all between $M-s+1$ and M, so they and their locations in $L[1..f-a]$ can be found from $D[M-s+1..M]$ using `WriteAll` and `Compact`. They are stored in the temporary arrays $H_2[1..b]$ and $G[1..b]$, respectively.

The final steps are to overwrite each of the b large names in $L[1..f-a]$ with a different name in $H_1[1..b]$ and then decrement M by s. But, to ensure that Invariant (3) is not violated, the direct access table entries, $D[j]$, of the names $j \in H_1[1..b]$ should first point to the b different locations in $L[1..f-a]$ that contain large names. This is

done using `WriteAll`. Then each name $j \in H_1[1..b]$ can be written to location $L[D[j]]$, also using `WriteAll`.

```
Function RemoveLargeNames
  local: f, a, b, e, e', i, j, m
  register: G[1...n], T[1...n], H1[1...n], H2[1...n]

  m ← read(M)
  f ← read(F)
  repeat
      s ← CountAlive
      a ← Count(InL(m−s+1) > 0, 1 ≤ i ≤ s)
      WriteAll(H1[i] ← L[f−a+i], 1 ≤ i ≤ a)
      b ← Compact(H1, H1[i] ≤ m−s, 1 ≤ i ≤ a)
  until b ≠ ∞
  write F ← f−a
  repeat
      repeat
          s ← CountAlive
          WriteAll(G[i] ← D[m−s+i], 1 ≤ i ≤ s)
          b ← Compact(G, InL(m−s+i) > 0, 1 ≤ i ≤ s)
          WriteAll(H2[i] ← L[G[i]], 1 ≤ i ≤ b)
          e ← WriteAll(D[H1[i]] ← G[i], 1 ≤ i ≤ b)
      until e ≠ ∞
      e ← WriteAll(L[G[i]] ← H1[i], 1 ≤ i ≤ b)
      if e = ∞ then
          repeat
              s ← CountAlive
              WriteAll(T[i] ← H1[i], 1 ≤ i ≤ s)
              h ← Compact(T, InL(G[i]) = 0, 1 ≤ i ≤ s)
              WriteAll(T[h+i] ← H2[i], 1 ≤ i ≤ s)
              Compact(T[h+1..h+s], InL(G[i]) = 0, 1 ≤ i ≤ s)
              e' ← Compact(T, T[i] ≤ m−s, 1 ≤ i ≤ s)
          until e' ≠ ∞
          repeat
              s ← CountAlive
              b ← WriteAll(H1[i] ← T[i], 1 ≤ i ≤ min{e', s})
          until b ≠ ∞
      end
  until e ≠ ∞
  write M ← m−s
```

Difficulties arise when processes crash. One problem is that this changes s and, hence, some large names become small names. If too many processes fail when the small names in $L[f-a+1..f]$ are copied into $H_1[1..b]$, the phase is simply repeated with s decreased by at least a factor of 2. The same is true for the second phase, in which large names in $L[1..f-a]$ are copied into H_2, their locations are copied into G, and the direct entry table entries for elements in H_1 are made to point to these locations.

However, a failure of the call to `WriteAll`, in which small names in H_1 overwrite large names in $L[1..f-a]$ means that an unknown subset of these writes have occurred.

In this case, H_1 has to be fixed up before starting again from the beginning of the second phase, with s decreased by at least a factor of 2.

Only the small name $H_1[j]$ can overwrite the large name $H_2[j]$ in L. Therefore, exactly one of $H_1[j]$ and $H_2[j]$ is in $L[1..f-a]$. So, to fix up H_1, the first s elements of each of H_1 and H_2 are copied to a temporary array T, using `WriteAll`. Then names in $L[1..f-a]$ and large names are removed from T using `Compact`. Because processes can fail during the construction of T, all this has to be repeated, with s decreased by at least a factor of 2, until none of the calls to `WriteAll` and `Compact` fail.

Finally, a sufficiently long prefix of T is copied back into H_1. This is also repeated, with s decreased by at least a factor of 2, until it does not fail.

Theorem 3. *In a synchronous system in which processes communicate using counters and registers, there is a fully-adaptive implementation of long-lived strong renaming which performs* `GetName` *and* `RelName` *in* $O(\log^3 c/\log\log c)$ *steps.*

5 Conclusions

In this paper, we described the first fault-tolerant fully-adaptive implementations of long-lived strong renaming that do not use base objects with $\Omega(n)$ bits. One algorithm is asynchronous and uses `LL/SC` objects in addition to registers. Because processes help one another to get new names, the original names of processes are used for identification purposes. The other algorithm is synchronous, but uses counters and registers. Moreover, its step complexity is substantially smaller: polylogarithmic instead of linear in c. This algorithm never uses the original names of processes, so it also works for systems of anonymous processes.

Fully adaptive one-shot renaming is very easy to implement using a counter. Hence, it was natural to use a shared memory system with counters when trying to get a fully-adaptive long-lived renaming implementation. Two specific open questions arise from this work: First, can more efficient or simpler fully-adaptive long-lived renaming algorithms be obtained using other strong memory primitives, for example `Compare&Swap`? Second, are there fully-adaptive renaming algorithms or more efficient adaptive renaming algorithms using only registers?

Acknowledgements

We are grateful to Hagit Attiya for helpful discussion. This research was supported by NSERC, the Scalable Synchronization Group at Sun Microsystems, and the German Research Foundation (grant WO1232/1-1).

References

1. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–104, May 1999.

2. Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 538–547, May 1995.
3. Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 105–112, 1999.
4. Y. Afek, G. Stupp, and D. Touitou. Long lived adaptive splitter and applications. *Distributed Computing*, 15:67–86, 2002.
5. H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *Journal of the ACM*, 37(3):524–548, 1990.
6. H. Attiya and T. Djerassi-Shintel. Time bounds for decision problems in the presence of timing uncertainty and failures. *Lecture Notes in Computer Science*, 725:204–214, 1993.
7. H. Attiya and A. Fouren. Polynomial and adaptive long-lived $(2k-1)$-renaming. *Lecture Notes in Computer Science*, 1914:149–159, 2000.
8. H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM Journal on Computing*, 31(2):642–664, 2001.
9. H. Attiya and A. Fouren. Algorithms adaptive to point contention. *Journal of the ACM*, 50(4):444–468, 2003.
10. H. Attiya and J. Welch. *Distributed Computing Fundamentals, Simulations, and Advanced Topics (2nd Ed.).* Wiley, 2004.
11. A. Bar-Noy and D. Dolev. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proceedings of the 8th Annual ACM Symposium on Principles of distributed computing*, pages 307–318, Aug. 1989.
12. J. Burns and G. Peterson. The ambiguity of choosing. In *Proceedings of the 8th Annual Symposium on Principles of Distributed Computing*, pages 145–158, Aug. 1989.
13. S. Chaudhuri, M. Herlihy, and M. Tuttle. Wait-free implementations in message-passing systems. *Theoretical Computer Science*, 220(1):211–245, 1999.
14. C. Georgiou, A. Russell, and A. A. Shvartsman. The complexity of synchronous iterative do-all with crashes. *Distributed Computing*, 17:47–63, 2004.
15. T. Hagerup and R. Raman. An efficient quasidictionary. In *Proceedings of SWAT*, volume 2368 of *Lecture Notes in Computer Science*, pages 1–18, 2002.
16. M. Herlihy, V. Luchangco, and M. Moir. Space- and time-adaptive nonblocking algorithms. *Electr. Notes Theor. Comput. Sci*, 78, 2003.
17. M. Herlihy and S. Rajsbaum. Algebraic spans. *Math. Struct. in Comp. Science*, 10:549–573, 2000.
18. M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, Nov. 1999.
19. M. Herlihy and M. Tuttle. Lower bounds for wait-free computation in message-passing systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 347–362, Aug. 1990.
20. M. Inoue, S. Umetani, T. Masuzawa, and H. Fujiwara. Adaptive long-lived $O(k^2)$-renaming with $O(k^2)$ steps. *Lecture Notes in Computer Science*, 2180:123–133, 2001.
21. M. Moir. Fast, long-lived renaming improved and simplified. *Science of Computer Programming*, 30(3):287–308, Mar. 1998.
22. M. Moir and J. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1–39, Oct. 1995.
23. A. A. Shvartsman. Achieving optimal CRCW PRAM fault-tolerance. *Information Processing Letters*, 39(2):59–66, 1991.