GENERATING UNIQUE IDENTIFIERS IN A DISTRIBUTED SYSTEM[1]

Gad J. Dafni

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-05 April 1985

[1]This research was supported by the National Science Foundation under Grant MCS-8122039.

Table of Contents

1. Introduction 0
2. Assumptions 1
3. The Algorithm 2
    3.1. Basic Algorithm 2
    3.2. Error Handling and Recovery 3
    3.3. Formal Description 5
4. Proof of the Fulfillment of the Properties 9
    4.1. Definitions 9
    4.2. Assertions 9
    4.3. Rules of Use 10
    4.4. Serializability of Recovery Processes 10
    4.5. Sketch-Proof of the Assertions 11
        4.5.1. Proof of Assertion 1 11
        4.5.2. Proof of Assertion 2 11
        4.5.3. Proof of Assertion 3 14
5. Summary 14
6. Acknowledgment 14
References 14

List of Figures

Figure 3-1: UID Generation 3
Figure 3-2: The Recovery Process 5
Figure 4-1: Recovery Sequence for Node i 12

Generating Unique Identifiers in a Distributed System*

Gad J. Dafni
Department of Computer Sciences
The University of Texas at Austin

abstract

An algorithm is proposed for generating unique identifiers in a distributed system. The algorithm is robust in the sense that it can recover from node a failure, as long as at least one of the nodes did not fail. It has a modest communication overhead, which is two message passing per each identifier generated, and about $2\log_2 N$ message passing per recovery if node failure rate is low (N being the number of nodes). It is economical in using the identifiers space, in the sense that the size of the identifiers may be relatively small. These properties make it superior to other unique identifiers generation algorithms that run on non-broadcast distributed systems.

1. Introduction

Generating unique identifiers (UIDs) is needed in certain systems, particularly in object-oriented systems [5, 1, 2, 3]. It was strongly argued by Fabry [5] that UID-based addressing avoids most of the problems that appear while addressing shared relocatable objects. His arguments were made in the context of centralized system. The problems are even more acute in Distributed Systems. Thus there is a need for an effective distributed mechanism for generating UIDs.

It is relatively easy to generate UIDs in centralized systems; however, this is not the case in a distributed system that needs to preserve the following properties:

a. Identifiers are unique systemwide, through the system's lifetime.

b. The identifiers space is utilized economically.

c. The system is immune against node failure.

d. The communication overhead necessary to generate dentifiers is limited.

By using the identifiers space economically we mean that the size of the UIDs may be decided on the basis of the average rate of generating UIDs per node, rather then the

*This research was supported by the National Science Foundation under Grant MCS-8122039

the maximal rate per node (see example 2 below). Therefore we may assign a smaller UID size. If the system is to be immune against failures, the mean time between failures (MTBF) of a node should be considered a determining factor for identifiers space size.

Following are two examples of simple algorithms for generating UIDs, both satisfying some of the requirements above but fail to satisfy all of them.

1. A centralized algorithm will satisfy properties a,b and d, but it is very vulnerable to node failure. It may also suffer from contention for UIDs generation.

2. A distributed algorithm, where each node generates identifiers for its own use, by appending the node number to a local sequence, thus generating identifiers which are unique systemwide. This algorithm is neither economical nor immune against failures. It is not economical, because each node is assigned an equal share of the identifiers space, while some may need much more than others. That will force us to make the size of that equal share fit the maximal needed. It is not immune against failures since when a node that does not have a nonvolatile memory recovers from failure, it cannot know what identifiers it has already used (unless an exhaustive search is conducted all over the system).

It may seem that a solution which is both economical and relatively robust must involve a considerable amount of communication overhead (unless we use a broadcast channel). Yet, as will be seen below, this is not the case.

2. Assumptions

In order to develop an effective algorithm for generating UIDs, we need to make the following assumptions:

1. When a node failure occurs it will be detected by the node, which will immediately stop sending any messages until it recovers.

2. There is a timeout mechanism that will be set at a node waiting for an answer, if and only if its counterpart in a dialogue is not active.

3. Any two nodes will always have a bidirectional (virtual) FIFO trustable channel between the processes that handle UID generation at each node (id-

managers). Communication between other processes is assumed to go through other virtual channels.

4. At any point in time, except before the system was initiated (i.e. before any node become active), there exists at least one node which is not down.

5. There is a globally known linear ordering of the nodes.

3. The Algorithm

Reexamining our second example we can see that the identifiers space for both the **generation** and the **usage** of the UIDs was divided in equal shares between the nodes. The reason for the uneconomic behavior in that algorithm was the possibly unequal rates of **usage**. The reason for the sensitivity for node failure was the fact that no information was **shared** between nodes, regarding UIDs. These two principles of **distribution of usage** and **sharing of information** are used to cure the above problems. As we shall see, there is no need to distribute the **generation** of UIDs.

3.1. Basic Algorithm

The algorithm divides the identifiers space for the **generation** of UIDs in equal shares between the nodes. The identifiers space is a set of pairs $<n,Seq_n>$, where n is the number of the node that generated the UID and Seq_n is sequence number in that node.

To distribute the **usage** of UIDs, we require each node to request different nodes for UIDs, visiting one node at a time (including itself) in a cycle. This will be handled by a distributed *type manager* [4] for UIDs, called the *id-manager*, that will maintain the following variables:

- *cycle_count* the number of times modulo 2 it cycled through the network.
- *lastnode* a pointer to the last node it visited in its cycle.

The kernel of each of the nodes will include a copy of the *id-manager*. These copies will communicate with each other, in order to regulate the **generation**, **usage** and **error recovery** of UIDs.

A process at node i that needs to acquire a new UID will call the *getid* function of its resident copy of the *id-manager* (Figure 3-1). This function will request the id-

manager of the next node k in its cycle (which may be the same node where the process resides) for a UID, by sending it a request, accompanied by $cycle_count_i$. $cycle_count_i$ is used in connection to the recovery process (see Sec.3.2). If the request is validated (see Sec.3.2), the *id-manager* that got the request will generate the next UID in its own exclusive class and send it back. The *getid* function will then return the UID to the calling process.

This scheme requires the sending and receiving of at most one message per UID generation, in the case of no failures.

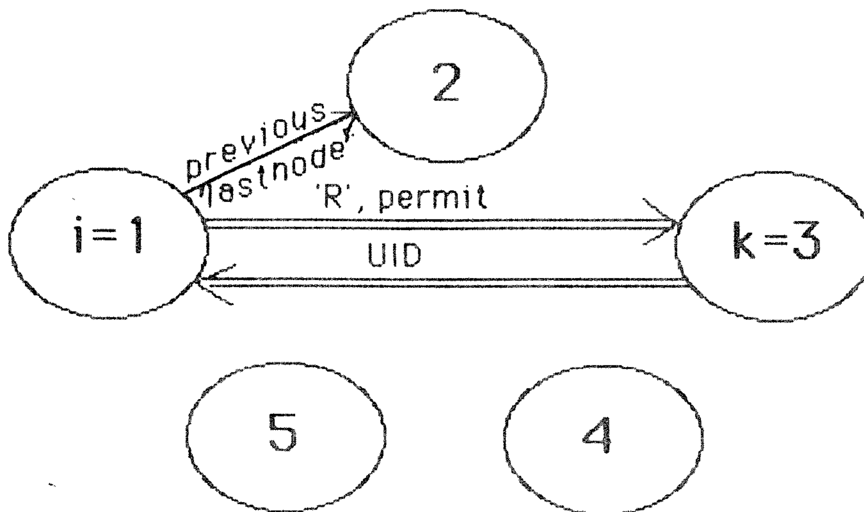

Figure 3-1: UID Generation

3.2. Error Handling and Recovery

As a result of assumptions 2 and 3 of section 2, occurrence of a failure at one node is detected by the other nodes as a timeout. When a time-out is detected after delivering a request, the originating node will ask the next node in its cycle for a UID. It must ultimately find a responsive node, would it be someone else or itself.

To be able to help others to recover from node failure, each node will maintain an array of the number of times modulo 2 it was visited by each of the other nodes, called *count*. When node i realizes it is faulty, it will cut communication with other id-managers until it has recovered from the fault. At that time, it will look for its active or

recovering closest successor in the cycle and inform him about its recovering. Then it will look for its closest active predecessor in the cycle, call it j, and ask it for its last sequence Seq_j and its $count_j$ table. After it got these variables, it will set its state variables as follows:

$$Seq_i := \begin{cases} Seq_j & \text{if } i>j \\ Seq_j+N & \text{otherwise} \end{cases} \qquad count_i := \begin{cases} count_j & \text{if } i>j \\ \neg count_j & \text{otherwise} \end{cases} \qquad (1)$$

In order to avoid serving nodes that were already served by node j during the current cycle, node i will ignore requests for UIDs from any node k if $cycle_count_k = count_i[k]$.

After recovering its sequence, node i will recover its *lastnode* pointer to point to the last node it accessed in its cycle, by conducting a binary search on the set $\{count_k[i] \mid k=1..n\}$, getting $count_k[i]$ from node k as needed. When the point CUT_i is found, where $count_k[i]$ changes values between $k=CUT_i$ and the next node, (existence of a single such point is given by lemma-1, page 11), node i will set $lastnode_i := CUT_i$.

Node i will recover its $cycle_count_i$ by setting it to $count_{CUT_i}[i]$.

Note: The values of the state variables of an inactive node i will be defined by those of its active predecessor j, according to Eq.1.

Since the recovery process involve in general communication with several nodes, we need to take measures to make recovery processes serializable. To achieve that, we do the following:

- While recovering, the node will not answer requests from other nodes. Instead, it will send them a *wait* message and queue the requests for later handling. The *wait* message include the time to wait, given in timeout units. This time is the time the sending node itself have to wait. The receiving node adds 1 to it, and send it to the nodes waiting for him.

- A recovering node i will inform its active or recovering successor in the cycle - node j, about its being recovering. If node j is also recovering, it will stop its recovery process (not before it informed its successor about its recovering), and wait for node i to send him the data for recovering its state variables.

The average number of messages sent and received for node recovery, will not exceed

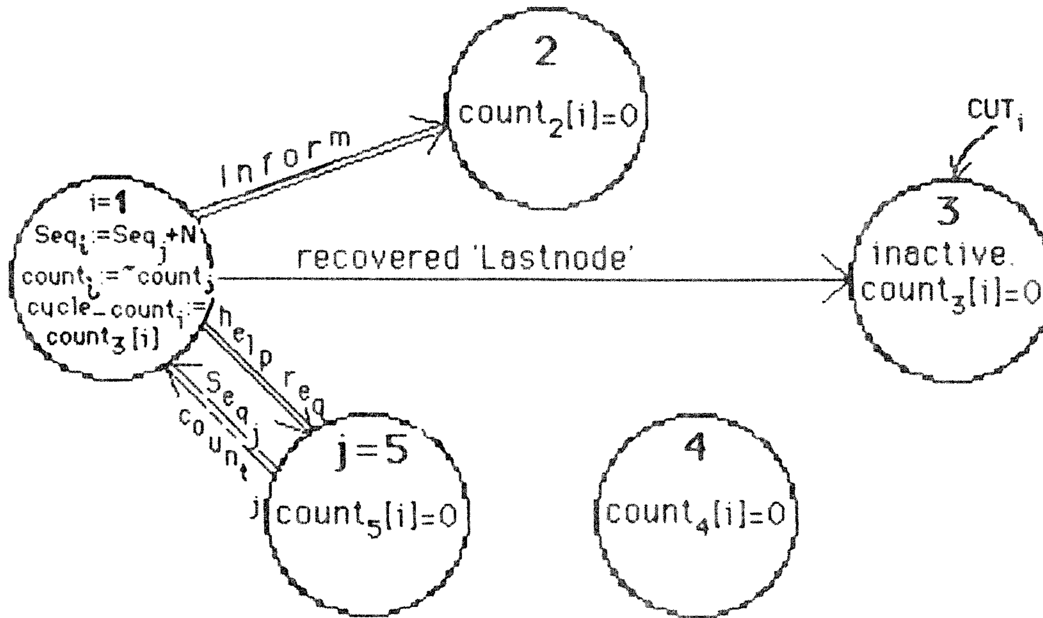

Figure 3-2: The Recovery Process

$(2N\text{-}K)[\log_2(N\text{-}K) + 4]/(N\text{-}K)$

where K is the number of inactive nodes. This approaches $2\log_2 N$ when $K \ll N$.

3.3. Formal Description

In the following, i will be the local node number, N is the number of nodes.

```
module id_manager
entry start,recover,getid
const
   N = number of nodes;
   i = local node number;
type
   idtype : 0..maxint;
   cycletype : 0..1;
   nodetype : 1..N;
var
   Seq : idtype;
   count : array[1..N] of cycletype;
   cycle_count : cycletype;
   lastnode : nodetype;
```

```
function getid;
[ Find next active node j in the cycle
  and get a UID from it ]
var
   j : integer;
   found : boolean;
   id : idtype;
begin
      P(semaphore);
      j:=lastnode;
      repeat
         j:=j mod N + 1;
         if j=1 then cycle_count:=(count_cycle + 1) mod 2;
         if j=i then
         begin
            if cycle_count<>count_i[i] then              Critical-Section I
            begin
               count[i]:=cycle_count;
               Seq:=Seq+1;
               id:=N*Seq + i;    [ <i,Seq> ]
               found:=true
            end
         end
         else requestid(j,cycle_count,id,found)
      until found;
   getid:=id;
   lastnode:=j;
   V(semaphore)

end;
```

```
procedure answer(j:nodetype; cycle_count_j:cycletype);
[ Answer a request(i,id,cycle_count_j) from node j ]
begin
   if cycle_count_j<>count_i[j] then
   begin
      count_i[j]:=cycle_count_j;                          Critical-Section II
      Seq:=Seq+1;
      id:=N*Seq + i;    [ <i,Seq> ]
      Send id back to node.
   end

end;
```

```
procedure recover;
[ recover from node failure ]
begin
   Initiate semaphore, queue etc.
   Inform closest active or recovering successor on our recovering.
   Ask closest active or recovering predecessor h for help.
   repeat
      get(message);
      case message.type of
         wait:         [ wait-message ]
                       Adjust wait-count.
                       Pass Wait to nodes waiting on us.
         info:         [ information-message about other node's recovery ]
                       Answer the message by telling that we are recovering.
                       Then act as for Wait.
         help:         [ help-message from node h ]
                       Seq=Seq_h (+N if h>i);                Critical-Section III
                       for m:=1 to N do
                          if h<i
                             then count[m]:=count_h[m]
                             else count[m]:=(count_h[m]+1) mod 2
         other:        [ other message-types ]
                       Queue the message.
                       Send Wait-message as an answer.
      end;
   until message.type=help;
   find CUT_i by binary search;  [ see Lemma-1 page 11
   lastnode:=CUT_i                  for definition of CUT_i ]
   if i<lastnode
      then cycle_count:=count_lastnode[i]
      else cycle_count:=(count_lastnode[i]+1) mod 2;
   handle queued messages.
   V(semaphore);

   poll
end;


procedure start;
begin
   If no active node exist then
   begin
      Seq:=0;
      cycle_count:=0;
      lastnode:=N;   (* set node-pointer in cycle *)
      for j:=1 to n do count[j]:=0
   end
   else
      recover
end;
```

```
procedure poll;
begin
   repeat
      if queue is not empty
        then begin
           P(semaphore);
           getqueue(message)
        end
        else begin
           wait for message from any channel;
           P(semaphore);
           get(message)
        end;
      case message.type of
         request:       [ request for UID from node j ]
                        answer(j,cycle_count_j);
         help_req:      [ help request from a recovering node ]
                        send help_data.
         info:          [ information-message about other node's recovery ]
                        tell the other node that we are active.
      end;
      V(semaphore)
  until false (* forever *)
end;


procedure requestid(node:nodetype; cycle : cycletype;
                         var id:idtype; var found:boolean);
begin
   set wait_count to 1;
   send(node,'R',cycle,wait_count); [ R stands for Request ]
   repeat
      wait for any channel;
      get(message);
      case message.type of
         answer:   [ answer to request ]
                   begin
                      found:=true;
                      id:=message.data
                   end;
         request:  [ request for UID from node j ]
                   answer(j,cycle_count_j);
         help_req: [help request from a recovering node ]
                   send help_data;
         info:     [ information-message about other node's recovery ]
                   tell the other node that we are active.
         wait:     [ wait-message ]
                   begin
                      Adjust wait_count.
                      Pass Wait to nodes waiting on us.
                   end;
      end
   until timeout  [ according to wait_count ]
end;
```

4. Proof of the Fulfillment of the Properties

We now prove that our algorithm is correct in the sense that it meets properties (a) to (d) of section 1. This is accomplished by making several assertions that directly imply these properties. The assertions are proved either directly or by showing that they hold before initiation, and each state transition on a state where they hold, generate another state where they hold. In order to accomplish this, we have to prove first the serializability of certain processes that may execute at different cites at the same time. That means that their total effect will be the same as running them in some serial order.

4.1. Definitions

Referring to critical-sections as they appear in section 3.3, we define:

1. A node is called **active** if it is not down and it is outside critical-section III.

2. A node is called **stable** if it is active and not in a critical-section.

3. The values of the state variables of an inactive node i will be defined by those of its active predecessor j, according to equation 1 page 4.

4. The **state** of the system is defined as

$$< \mathrm{Seq}_1, count_1, cycle_count_1, lastnode_1, \ldots$$
$$\ldots, \mathrm{Seq}_N, count_N, cycle_count_N, lastnode_N >$$

4.2. Assertions

1. For any active node i, $Seq_i(t)$ is a monotonically increasing function in time.

2. For any two stable nodes j,k,

   let

   $$A_{jk} = \{ i \mid count_k[i] \neq count_j[i] \}$$

   and

   $$a_{jk} = \begin{cases} |A_{jk}| & \text{if } j \leq k \\ -a_{kj} & \text{otherwise} \end{cases}$$

   then

   $$Seq_k = Seq_j + a_{kj} \tag{2}$$

3. For any two stable nodes j,k

$$j > k \Rightarrow Seq_k \geq Seq_j \geq Seq_k\text{-}N$$

4.3. Rules of Use

We assume that any use of the procedures of the *id-manager* will obey the following rules:

1. The first procedure that will be called at each node is *start*, and it will never be called again.

2. After a node failure, the first procedure to be called is *recover*, and it will not be called in any other case.

There are various mechanisms to enforce such rules [6], which we will not refer to here.

4.4. Serializability of Recovery Processes

Concurrent recovery of different processes may result in inconsistent system state (where the assertions does not hold). Therefore, certain situations are avoided by the interaction between nodes, as described earlier. In the following we prove that under our algorithm, node recoveries are serializable, that is - their effect is always equivalent to some serial schedule of execution.

Let us define a relation $<_t$ between recovering nodes. $j <_t k$ means that at time t, k is waiting for j to fully recover, before it proceeds with its own recovery process (if it does not wait on other nodes). We can see that if $j <_t k$, the same relation will hold between j and k at later time, as long as none of them did not terminate its recovery process. Since there cannot be any active nodes between j and k while k is waiting directly on j to recover, $j <_t k$ implies that there are no active nodes in the cycle between j and k. By Sec.2.4, that implies that at any time except for initiation, there are no cycles in the graph of this relation.

If $j <_t k$ then k's recovery is based on data that came from j after it became active, which means that k might have started recovering after j became active, with the same results.

The above means that $<_t$ for $t > 0$ is the precedence order and, since it has no cycles, the set of recovering processes is serializable for any $t > 0$. At initiation, if several

nodes start at the same time, they will wait on one another and finally get timeout (since no new wait-messages are sent). So they will try the next node, until one or more will find there are no active nodes, in which case they initiate their variables to zero. Those that initiate are serializable, since they do not relay on each other. For the others, the former argument holds.

Using the serializability of recoveries we can choose a logical clock (event counter) in which recoveries occur one at a time. We will use this clock in our proof, though it is not part of the algorithm.

4.5. Sketch-Proof of the Assertions

Following is the sketch of the proofs of the assertions of section 4.2.

4.5.1. Proof of Assertion 1

It is easy to see that Seq_i is never decremented on state transitions other than recoveries of node i.

Suppose node i recovers at time t. We can prove by induction on time (i.e. number of node recoveries) the existence of two sequences $\{n_j\}_{j=0}^{k}$ and $\{t_j\}_{j=0}^{k}$ such that n_0 and i were active at time t_0, $n_k=i$, and for j=0..k-1, node n_j helped node n_{j+1} to recover. Figure 4-1 illustrate those sequences. Remembering the recovery procedure and assuming (the induction hypothesis) that assertion 3 held at earlier time, we get

$$Seq_i(t) \geq Seq_i(t_0)$$

4.5.2. Proof of Assertion 2

In order to prove assertion 2 we need to first prove the following lemma.

Lemma-1: For any active node m, there is a node k (not necessarily active - see definition 4.1.3), such that for any active node i

(a) $i \leq k \Leftrightarrow count_i[m] = count_k[m]$

and (b) $m < lastnode_m \Leftrightarrow cycle_count_m = count_{lastnode_m}[m]$

We will call such k "CUT_m".

Proof: The lemma holds trivially immediately after the first node m has initiated.

Assume the lemma holds at state S while a transition T to state S1 occurs.

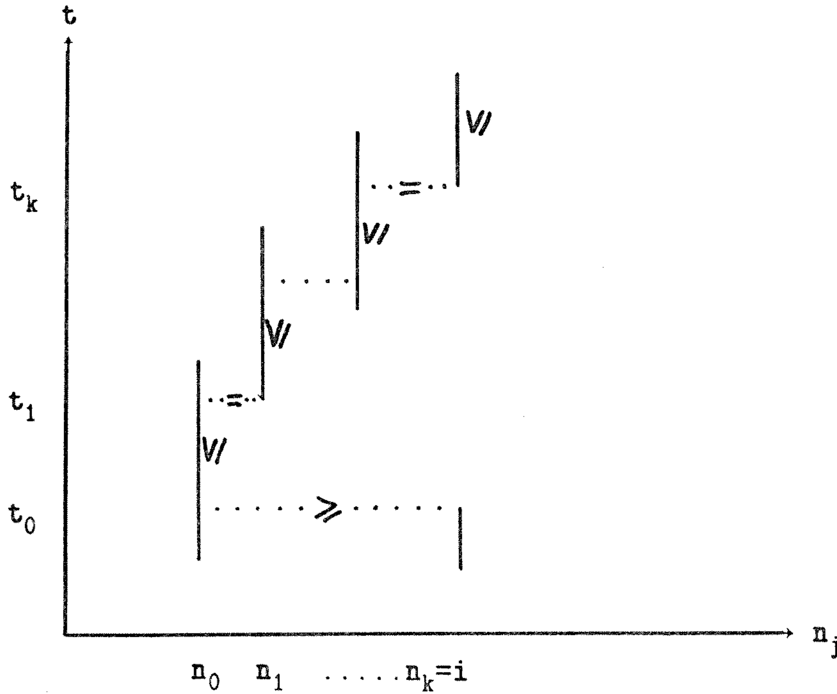

Figure 4-1: Recovery Sequence for Node i

First, assume the transition T was a result of node m requesting a UID.

Assume $m < lastnode_m \leq CUT_m$.
Assuming the lemma holds at state S, we have

$lastnode_m \leq j \leq CUT_m \Rightarrow cycle_count_m = count_j[m]$

therefore m's request for UID will be rejected by all nodes between *lastnode*$_m$ and CUT_m, including, but will be accepted by the next active node to CUT_m, which will become the new CUT_m and the new *lastnode*$_m$ as well. That proves part (a) of the lemma holds at state S1 under the assumption that $m < lastnode_m \leq CUT_m$.

Similar argument will prove the same for all cyclic permutations of $<m$, *lastnode*$_m$, $CUT_m>$. Noncyclic permutations are impossible, because CUT_m may depart from *lastnode*$_m$ only as a result of some node recoveries. In order to make a noncyclic permutation, CUT_m have to go past m, which mean that m have to fail and recover. But while recovering, m recovers *lastnode*$_m$ to be equal to CUT_m, which prevent noncyclic permutation.

We can easily see that part (b) of the lemma always hold right after m gets a new UID. Therefore the lemma holds at state S1.

Second, assume the state transition T is a result of node m's recovery. The lemma holds at state S1 because of the way m recovers $count_m$, $lastnode_m$ and $cycle_count_m$.

This ends the proof of lemma-1.

Based on lemma-1, it is easy to see that

$A_{kj}(t)=\{i \mid k \leq CUT_i(t) < j\}$

Assertion 2 holds trivially immediately after the first node has initiated.

Let us assume that assertion 2 (that is - equation (2)) held at certain state of the system, while a state transition that effect some components of equation (2) occur.

Changing Seq_j as a result of a request from node i to node j, is always accompanied by switching $count_j[i]$, which moves CUT_i past j and decrements a_{kj}. At the same time Seq_j is incremented, so equation (2) still holds.

If Seq_j is changed as a result of recovery of node j by the help of node m, it will set

$Seq_j=Seq_m$ (+N if m>j)

and for any i

$count_j[i]=count_m[i] \Leftrightarrow m<j$

Since $count_m[k]$ for $k \neq j$ were not changed by j's recovery (relaying on serializability), we have for any node k (k may be j or m as well)

$\{count_j[i]=count_k[i]$

$\Leftrightarrow count_m[i]=count_k[i]\}$

$\Leftrightarrow m<j$

Therefore

$a_{kj}=a_{km}$ (-N if m>j)

So

$Seq_k=Seq_m+a_{km}=Seq_j+a_{kj}$

and the validity of equation (2) is preserved.

Change in $count_j[i]$ may occur in the event of a request from node i to node j, or in the event of the recovery of node j, which are the same events that were handled above.

Since Seq_k and Seq_j play symmetric roles in equation (2), the above hold for changes in Seq_k as well.

That concludes the proof of assertion 2.

4.5.3. Proof of Assertion 3

Since $j>k$ implies $N \geq a_{kj} \geq 0$ (by definition), assertion 3 follows directly from assertion 2.

5. Summary

The proposed algorithm for generating unique identifiers has the advantages that it is simple, economic, relatively robust and has low communication overhead.

One disadvantage of the algorithm is that the total number of nodes N should be globally known. To overcome this we may choose N to be some maximal number of nodes that the system may have at any time. By that, adding new nodes is made easy (unless the maximum is exceeded), because the algorithm will handle missing nodes the same way it handle failing nodes. This will cause more communication overhead, so it may not be adequate for certain systems. In such systems we may add a reconfiguration protocol for increasing N by some amount when needed.

6. Acknowledgment

I wish to express my gratitude to Prof. J.C. Browne and Prof. A. Silberschatz for their help and contribution to the development of this algorithm.

References

1. J.C. Browne and T. Smith. An Object-Oriented, Capability-Based Architecture. to be published in Lectures Notes in Computer Science.

2. J.C. Browne et al. Detailed Design of ZEUS. To be published

3. J.C. Browne et al. A Reliable High Integrity Object-Oriented System -- ZEUS. To be published

4. J.C. Browne. An Object-Oriented Formulation for Distributed Systems: Integrity Management. To be published

5. R.S. Fabry. "Capability -Based Addressing." *CACM 17*, 7 (July 1974), 403-411.

6. R.B. Kieburtz and A. Silberschatz. "Access-Right Expressions." *ACM Transactions on Programming Languages and Systems 5*, 1 (Jan. 1983), 78-96.