HOW TO KEEP A DYNAMIC DISTRIBUTIVE DIRECTED GRAPH ACYCLIC AND YET GRANT ALL REQUESTS OF EDGE ADDITIONS

by

Shimon Even and Yachin Pnueli

Technical Report #604
January 1990
HOW TO KEEP A DYNAMIC DISTRIBUTIVE DIRECTED
GRAPH ACYCLIC AND YET GRANT ALL REQUESTS OF
EDGE ADDITIONS

Shimon Even and Yachin Pnueli
Computer Science Department
Technion, Israel Institute of Technology
Haifa, Israel 32000

ABSTRACT

The problem of cycle prevention in dynamic distributive networks arises in many distributive applications such as distributive databases and operating systems. A typical application is a deadlock prevention algorithm (in which every edge indicates some dependence relation and a cycle indicates a deadlock).

Let $G(V,E)$ be a finite directed graph where every node in $V$ is a distinct processor. The problem is to perform, on-line, a series of given requests to add or delete edges in the graph while keeping it acyclic.

Since, for a given set $E$, some edge additions may be performed only after some other edges are removed from the graph, the problem can be restated as follows: Given such a series of requests, delay each request in such a way that every request is eventually granted while keeping the graph acyclic. We present an algorithm which solves this problem.

Katz and Shmueli [KS] treated the same problem. Our solution differs in that it has the following property. If every request to add an edge $k \rightarrow j$, is such that there is no path from $j$ to $k$ which persists forever, then every request to add an edge is eventually granted. The message complexity of the algorithm is $O(|E| \cdot |V|)$ messages per request. (Requests to delete an edge are performed immediately without sending any messages.)

1. INTRODUCTION

In many distributive applications a dynamic distributive directed graph $G(V,E)$ is maintained in which every node is an independent processor and every edge indicates some dependence or "waiting for" relation. In such a graph the formation of a directed cycle may mean that the distributive system entered some undesired state (such as deadlock). It is therefore of interest to have an algorithm which can add and delete edges in the graph, according to a given set of distributive requests, while keeping the graph acyclic.

January 15, 1990
More formally, assume \( V \) is a fixed set of nodes, where each node represents a processor, and \( E \) is a dynamic set of directed edges. Each node \( k \) can get, from its end-user, requests of the following forms:

- **ADD\_EDGE\( (j) \)** - A request to add the edge \( k \rightarrow j \) to the graph. (We assume there are no requests for self-loops or parallel edges).
- **DELETE\_EDGE\( (j) \)** - A request to delete the edge \( k \rightarrow j \) from the graph. If the edge does not exist then this request is understood to be an instruction to cancel a previous request, ADD\_EDGE\( (j) \), which has not been granted yet. (If there is no such edge and no pending ADD\_EDGE\( (j) \) request then the DELETE\_EDGE request is ignored).

Each node \( k \) can answer an ADD\_EDGE\( (j) \) request with one of the following messages:

- **EDGE\__GRANTED\( (j) \)** - Indicating that the edge \( k \rightarrow j \) has been added to the graph.
- **CYCLE\( (j) \)** - Indicating that the algorithm has discovered that the addition of the edge \( k \rightarrow j \) would have closed a directed cycle in the graph. (Because of the dynamic nature of the graph, it is possible that such a cycle does not exist by the time the algorithm sends this message, or that it has never existed at all; more about this issue later.) This message does not mean that the edge will not be added but merely indicates that the user might have to wait for some changes in the graph before his request is granted.

We say that an algorithm solves the cycle prevention problem in dynamic distributive graphs if it has the following properties:

1. The graph remains acyclic.
2. Every request is eventually granted.

We solve this problem under the following assumption:

**WASM**: For every node \( k \), if an ADD\_EDGE\( (j) \) is presented at node \( k \), then there is no directed path from \( j \) to \( k \) which holds forever.

We can now compare our algorithm to the two algorithms for cycle prevention presented in [KS]:

The first algorithm presented in [KS], the reranking algorithm, seems to have a very good average message complexity. However, under adverse conditions in the network, (even when the following assumption holds: The graph is static and each of the ADD\_EDGE requests can be granted,) the algorithm cannot guarantee that at least one request, out of a set, will eventually be granted.

The second algorithm presented in [KS], the definitive reranking algorithm, guarantees that a request to add an edge \( k \rightarrow j \) will eventually be granted, if the following assumption holds:

**ASM1**: Eventually there is no directed path from \( j \) to \( k \). (i.e. There is a time after which there is no such path - as long as the request has not been granted.)

Our algorithm guarantees a solution of the problem under WASM, which is much weaker than ASM1. However, to ensure that every request is eventually granted, our algorithm has, sometimes, to delay the treatment of a given request, until the configuration of the graph changes (until either some edges are deleted or some requests of "higher
priority" are granted). This causes the undesired property in that if contrary to WASM, a
given path from $j$ to $k$ persists forever, while a request to add the edge $k \rightarrow j$ is pending,
our algorithm may even become deadlocked.

Because our algorithm is complex and the complete formal proofs of its properties
are intricate, we break the presentation of the algorithm into three stages. In each stage
we present an intermediate algorithm which is a "step" in the direction of the final algo­
rithm, and prove those properties which the intermediate algorithm has. We label these
algorithms algorithm1, algorithm2 and algorithm3. The remainder of this document is
organized as follows:
2. The network model.
3. Informal description and motivation.
5. Analysis of algorithm1.
6. What is wrong with algorithm1.
7. Description of algorithm2
9. The final algorithm
10. The complexity of the algorithm.

2. THE NETWORK MODEL

We assume that if the addition of an edge $k \rightarrow j$ is ever requested then a two-way
communication channel between $k$ and $j$ exists. This channel is assumed to be nonin­
terfering (i.e. a message sent from $k$ to $j$ does not interfere with a message sent from $j$ to
$k$, and vice versa) and dependable (i.e. each message arrives intact at its destination after
some unbounded but finite delay). The FIFO assumption is not necessary.

Let $E$ be the set of all potential edges in $G$. Let $TO(v)$ be the number of nodes from
which $v$ is reachable in $(V, E)$ and $FROM(v)$ the number of nodes reachable from $v$ in
$(V, E)$. The requirement on the size of the memory of node $v$ is
$O((1 + TO(v) + FROM(v)))$.

For the sake of simplicity, we assume that a request or message can arrive at some
node $k$ only when it is idle. (If this is not the case, the assumption can easily be enforced
by keeping the requests and messages in a queue and processing them one at a time).

We assume that each node has a distinct ID (identity number). This resolves the dif­
ficulty which arises when two identical nodes receive simultaneous and identical requests
to connect to each other.

3. INFORMAL DESCRIPTION AND MOTIVATION

Our algorithm is based on the ideas of the reranking algorithm of Katz and Shmueli,
from whom we adopt the ideas of a ranking function and a reranking procedure. The idea
is to assign each node in the graph a rank - an element of an ordered set - and to keep the
following property invariant at all nodes:

January 15, 1990
INV: If node $k$ has a directed edge to node $j$ then $\text{rank}(k) < \text{rank}(j)$.

It follows that the graph remains cycle-free.

Clearly, when a DELETE_EDGE($j$) request is presented at $k$, the request can be granted immediately. However, when a request ADD_EDGE($j$) is presented at node $k$, it must first check if $\text{rank}(k) < \text{rank}(j)$. If so, then the edge $k \rightarrow j$ can be safely added. If $\text{rank}(k) \geq \text{rank}(j)$, node $k$ requests node $j$ to increase its rank to some value above the current rank ($k$). This is done by sending a RERANK($k$, min_rank) message to $j$, where \( \text{min\_rank} > \text{rank}(k) \). Clearly, for INV to hold at all times, if $j$ has sons (i.e. nodes which are reachable from $j$ via a single edge), it can increase its own rank to some value $\text{new\_rank}$, greater than or equal to $\text{min\_rank}$, only if it knows that all its sons have a rank higher than this $\text{new\_rank}$. This is done by sending the RERANK($j$, min_rank) message to its sons. This may lead to a propagation of RERANK($x$, min_rank) messages to all descendants of $j$ (i.e. nodes reachable from $j$). When some node $v$ can fulfill such a rerank request it increases its own rank and returns an ANSWER($v$, rank($v$)) message to let its 'father' know its new rank. (If a RERANK($x$, min_rank) message reaches a node $v$, which already has a rank higher than or equal to min_rank, it also responds with an ANSWER($v$, rank($v$)) message.) Eventually this enables $j$ to increase its own rank and send an ANSWER($j$, rank($j$)) message to $k$ (where rank($j$) is the rank of $j$, after the reranking). If now rank($k$) < rank($j$), the original ADD_EDGE($j$) request can be granted. This set of RERANK and ANSWER messages will be called a rerank phase.

In order to make the rerank phase effective we would like every node which is a descendant of $j$ to be reranked no more than once in a single rerank phase. Since we must keep INV true at all times, the descendants of $j$ must be reranked from the bottom up.

Let $i$ be some descendant of $j$, such that there is more than one directed path from $j$ to $i$. Let new_rank($i$) be the new rank given to $i$ in the reranking procedure. Let

$$P = j, v_1, v_2, \ldots, v_n, i$$

be a path from $j$ to $i$, different from the path through which $i$ has been reranked. It is then possible that $i$ is reranked before $P$ was traversed (since $i$ does not know the number of paths from $j$ to $i$, it is infeasible for $i$ to wait until all such paths are traversed). In this case we have a group of nodes $v_1, v_2, \ldots, v_n$ on $P$ such that eventually we want

$$\text{min\_rank} \leq \text{new\_rank}(j) < \text{new\_rank}(v_1) < \text{new\_rank}(v_2) < \ldots < \text{new\_rank}(v_n) < \text{new\_rank}(i)$$

while the values of min_rank and new_rank($i$) have already been determined. However, the number of the nodes, $n$, on this path is as yet unknown. We therefore have to use for rank an ordered set, such that for a given min_rank we can always find a new_rank($x$) without bounding the number of values between them. For this purpose we could use the set of pairs (round, height), members of $I \times I$, with the ordering (round$_1$, height$_1$) > (round$_2$, height$_2$) iff round$_1$ > round$_2$ or (round$_1$ = round$_2$) and (height$_1$ < height$_2$). We also want each node to be able to identify rerank requests which were initiated by itself. Therefore we use for rank the set of triplets ((round_number, round_initiator), height), members of $((I \times I) \times I)$, with the ordering

$$((\text{round}_1, \text{initiator}_1), \text{height}_1) > ((\text{round}_2, \text{initiator}_2), \text{height}_2)$$

January 15, 1990
iff

\[ \text{round}_1 > \text{round}_2 \] or

\[ ((\text{round}_1 = \text{round}_2) \text{ and } (\text{initiator}_1 > \text{initiator}_2)) \] or

\[ ((\text{round}_1 = \text{round}_2) \text{ and } (\text{initiator}_1 = \text{initiator}_2) \text{ and } (\text{height}_1 < \text{height}_2)). \]

(When it is not necessary to explicitly state the initiator we refer to the pair \((\text{round}_\text{number}, \text{round}_\text{initiator})\) as \text{round}.)

Remark: This rank is the same as the rank used by Katz and Shmueli with the following modifications:

1. They use for \text{round} a single integer, instead of our \((\text{round}_\text{number}, \text{round}_\text{initiator})\) pair.
2. Our \text{height} is equivalent to their 'priority'.

A single rerank phase as above is not sufficient for the solution of our problem for two reasons:

1. If \(k\) is reachable from \(j\) then any increase in the rank of \(j\) must lead to an increase in the rank of \(k\). After the reranking, \(\text{rank}(k)\) is still greater than \(\text{rank}(j)\), even if the path from \(j\) to \(k\) no longer exists, and \(k\) cannot add \(k \rightarrow j\) without sending some more messages.
2. Since the algorithm is distributive, it might well be that, while \(k\) caused the increase of \(\text{rank}(j)\), some unrelated node in the graph requested an increase in the rank of \(k\), and even if \(k\) is not reachable from \(j\), \(\text{rank}(k)\) may still be greater than \(\text{rank}(j)\) after the reranking.

We solve the first problem by coding into the reranking messages the ID of the node which initiated the rerank phase (as shown above). When a node receives a rerank message initiated by itself, it does not respond. (Observe that in such a case node \(k\) does not receive an \text{ANSWER} to its rerank request as long as it is reachable from \(j\) - which is fine, since \(k\) cannot add an edge to \(j\) as long as it is reachable from \(j\). If node \(k\) communicates with an end-user, we can use this rerank message to let him know that his request would have to wait for some changes in the graph and give him the choice of waiting for these changes or cancelling his request. This is done by sending the \text{CYCLE}() message to the end-user).

We solve the second problem by using more than one rerank phase. A collection of rerank phases which attempt to satisfy the same \text{ADD\_EDGE} request is called a rerank process.

In \text{algorithm} 1, (which can be viewed as a slight modification to the reranking algorithm of Katz and Shmueli) we use a naive method of re-reranking. When a rerank phase terminates \((k\) receives an \text{ANSWER}() message \) and the requested edge cannot be granted, \(k\) waits some arbitrary time and then tries again to rerank \(j\). This results in an algorithm which solves the cycle prevention algorithm under the following two assumptions (which will be formalized in section 4) on every \text{ADD\_EDGE} request:

- **NON-CONNECTIVITY**

There is no lasting connectivity between \(j\) and \(k\).
NON-DISTURBANCE
Eventually \( k \) sends a RERANK message to \( j \), such that \( k \) is not forced to increase its rank (by a rerank message from another node) until after it received the ANSWER message from \( j \).

In algorithm \( _2 \) we add to the algorithm additional "fairness" data-structures which guarantee that for every request the NON-DISTURBANCE assumption eventually holds. Therefore algorithm \( _2 \) needs only the NON-CONNECTIVITY assumption.

The "fairness" data-structures introduce two mechanisms: A capturing mechanism which guarantees that from a given set of requests at least one eventually succeeds; and a priority mechanism which guarantees that each request eventually become the one which succeeds.

We say that rerank phase \( S \) captured node \( k \), if node \( k \) increased its rank as a result of a rerank message of phase \( S \) to a value greater than the value of a pending rerank phase initiated by \( k \). We require each node to remember its captors and we demand that if a rerank phase, initiated by node \( k \), fails to grant a request, because node \( k \) has been captured (i.e. has been forced to increase its rank), \( k \) does not initiate a new rerank phase until it knows that at least one of its captors has terminated successfully (i.e. the request which caused this capturing phase has either been granted or cancelled). In this case we say that \( k \) is released. This mechanism guarantees that, in any finite, network at least one node completes a successful rerank phase.

The priority mechanism operates as follows. A node which has no pending request has priority zero. A node having a pending request has a priority \( p \), which is equal to the number of times that a reranking phase has been initiated in an attempt to satisfy this request. We give each rerank message, in a rerank phase, a priority which is the priority of the initiator of the phase. We demand that a node of priority \( p \) increases its rank as a result of a rerank message of priority \( p' \) only if \( p \leq p' \). When a phase terminates successfully, its initiator lowers its priority to zero. We prove that this mechanism together with the capturing mechanism guarantees that no node needs more than \( |V| \) rerank phases, before one of them terminates successfully and the requested edge is granted.

In algorithm \( _3 \) we add virtual nodes (called ports) to the graph and run on the "new" graph (which is called a port-graph) the same algorithm \( _2 \) above. We show that in the port-graph the NON-CONNECTIVITY assumption is identical to the weaker WASM and that therefore algorithm \( _3 \) solves the cycle prevention problem on the original graph under WASM.

4. FORMAL DESCRIPTION OF ALGORITHM

The algorithm is presented from the viewpoint of node \( k \), all data-structures and procedures are such that the node in which they are kept and performed is node \( k \). All nodes in the graph have identical structures and procedures, except the distinct id’s.
This section is divided into the following subsections:
1. Data types.
2. Messages to and from users.
3. Messages between nodes.
4. Data structures kept in each node.
5. Procedures.

4.1. Data Types
nodename: An id of some node in G.
round: An ordered pair \((\text{round}_\text{number}: \text{integer}, \text{round}_\text{initiator}: \text{nodename})\).
rank: An ordered triple \(((\text{round}_\text{number}: \text{integer}, \text{round}_\text{initiator}: \text{nodename}), \text{height}: \text{integer})\). We also refer to a rank as a pair of \((\text{round}, \text{height})\).
request\_id: An integer.

4.2. Messages To and From Users
(See the introduction for the meaning of these messages).

4.2.1. Requests (messages from user)
REQUEST\_EDGE\_U(j: nodename)
DELETE\_EDGE\_U(j: nodename)

4.2.2. Messages to User
EDGE\_GRANTED\_U(j: nodename)
CYCLE\_U(j: nodename)

4.3. Messages Between nodes
The messages are described here from the point of view of the receiver.
RERANK\_U(j: nodename; (r, i): round; qid: request\_id) - A message from \(j\), requesting the receiver to increase its rank to some value higher than \((r, i), qid\). The request\_id, as recorded at \(i\), is qid; it is used by the receiving node to decide whether this request is still relevant.
ANSWER\_U(j: nodename; rank) - A message from \(j\), indicating that it has a rank rank.
RELEASE\_U(i: nodename; qid: request\_id) - A message indicating that a request at node i, with request\_id = qid, has been terminated (either the edge was granted or the request cancelled).

January 15, 1996
4.4. Data Structures

At each node $k$ the following data-structures are maintained:

SONS - The set of edges from $k$; $i \in SONS$ iff there is an edge $k \rightarrow j$.

MY_REQUEST_QUE - A queue of REQUEST_EDGE requests. When a request cannot be granted immediately, it is placed in the queue. It is removed from the queue only when it is granted or canceled. A request REQUEST_EDGE($j$) is stored in the queue as a nodename $j$ and a request id ($qid$). When a request is added to the queue, it receives a request id higher than that of any request previously entered to the queue. The first node in the queue is called the CURRENT_REQUEST. A rerank phase can be initiated only for the CURRENT_REQUEST. The request id of the CURRENT_REQUEST is called CURRENT_ID.

RANK($k$) - The rank of the node $k$. RANK($k$) is a triple $((R(k), I(k)), H(k))$. It can also be noted as the pair $(RND(k), H(k))$. Initially it is set to $((0, k), 1)$.

KNOWN_RANK - An array of ranks such that KNOWN_RANK($j$) is the highest rank of $j$ which $k$ has received in some message from $j$. KNOWN_RANK($j$) is a pair $(krnd(j), round, kh(j); integer)$. If $k$ has not received any such message from a node $j$ then KNOWN_RANK($j$) = $((0, j), 1)$. RANK($k$) can be regarded as a special entry in the KNOWN_RANK array.

WAITING_FOR_AN_ANSWER - A boolean variable, indicating whether node $k$ is waiting for an answer on a rerank phase which it initiated.

WAITING - A data-structure containing the pending RERANK requests (requests received by $k$, but not yet answered). The RERANK requests are stored as tuples of the form: (WS: set of nodename, WT: set of nodename, WRND: round, WQID: request ID). (WRND is also referred to as a pair (WR,WI)).

WS is a set of nodes such that node $v$ is a member of WS in a tuple (WS,WT,WRND,WQID) iff $k$ has received a message RERANK($v$, WRND, WQID). We have to remember all such nodes $v$, so that when the request can be answered we know to which nodes to send ANSWER messages.

WT is a set of nodes such that node $v$ is a member of WT in a tuple (WS,WT,WRND,WQID) iff $k$ has sent it a message RERANK($k$, WRND, WQID). We wish to remember all such nodes $v$ so that when $k$ receives a RELEASE(WI,WQID) message we know to which nodes to send this RELEASE messages.

WAITING_FOR_RELEASE - A data-structure containing the RERANK requests which $k$ answered and for which a RELEASE message has not yet arrived. The RERANK requests are stored in the same way as in WAITING. Note: In this algorithm WAITING_FOR_RELEASE is redundant; however, since we use it in the next version of the algorithm we introduce it here to point out where it is updated.

LATEST - An array of pairs $(round, requestid)$. For each node $v$, LATEST($v$) indicates the highest pair of round and request id of a RERANK request received with $v$ as the initiator. Because for every node only a single request may be pending at a time. Receiving a RERANK request initiated by node $v$ with a given pair $(rnd, qid)$ renders all rerank requests initiated by $v$ with request ids lower than $qid$ or with the same request id but a round lower then $rnd$ obsolete. Also a RERANK message received by $k$ which was initiated by node $v$ can safely be ignored if its (request id, round) pair is smaller then LATEST($v$).
4.5. Procedures

4.5.1. Service Procedures:

begin
Send to node \( j \) all the requests in WAITING which have not been sent to it yet (tuples in WAITING such that \( j \) is not a member of WT of that tuple) and which are not trivial (a request is trivial for node \( j \) if the KNOWN_RANK of \( j \) is already greater the the requested RANK).
end;

begin
Add to \( j \) all the requests in WAITING which are not trivial (a request is trivial for node \( j \) if the KNOWN_RANK of \( j \) is already greater the the requested RANK).
end;

begin
Send EDGE_GRANTED \( (j) \) to user;
SEND_RERANK_TO_SON \( (j) \); {Send to \( j \) all pending rerank messages}
end;

begin
if not WAITING_FOR_AN_ANSWER then
begin
send RERANK \( (k, (R(k)+1, k), CURRENT_ID ) \) to CURRENT_REQUEST;
WAITING_FOR_AN_ANSWER:=true;
end;
end;

begin
if not WAITING_FOR_AN_ANSWER then
begin
send RERANK \( (k, (R(k)+1, k), CURRENT_ID ) \) to CURRENT_REQUEST;
WAITING_FOR_AN_ANSWER:=true;
end;
end;

begin
Send to node \( j \) all the requests in WAITING which have not been sent to it yet (tuples in WAITING such that \( j \) is not a member of WT of that tuple) and which are not trivial (a request is trivial for node \( j \) if the KNOWN_RANK of \( j \) is already greater the the requested RANK).
end;

begin
Add to \( j \) all the requests in WAITING which are not trivial (a request is trivial for node \( j \) if the KNOWN_RANK of \( j \) is already greater the the requested RANK).
end;

begin
Send EDGE_GRANTED \( (j) \) to user;
SEND_RERANK_TO_SON \( (j) \); {Send to \( j \) all pending rerank messages}
end;

begin
if not WAITING_FOR_AN_ANSWER then
begin
send RERANK \( (k, (R(k)+1, k), CURRENT_ID ) \) to CURRENT_REQUEST;
WAITING_FOR_AN_ANSWER:=true;
end;
end;

begin
Send to node \( j \) all the requests in WAITING which have not been sent to it yet (tuples in WAITING such that \( j \) is not a member of WT of that tuple) and which are not trivial (a request is trivial for node \( j \) if the KNOWN_RANK of \( j \) is already greater the the requested RANK).
end;

begin
Add to \( j \) all the requests in WAITING which are not trivial (a request is trivial for node \( j \) if the KNOWN_RANK of \( j \) is already greater the the requested RANK).
end;

begin
Send EDGE_GRANTED \( (j) \) to user;
SEND_RERANK_TO_SON \( (j) \); {Send to \( j \) all pending rerank messages}
end;

begin
if not WAITING_FOR_AN_ANSWER then
begin
send RERANK \( (k, (R(k)+1, k), CURRENT_ID ) \) to CURRENT_REQUEST;
WAITING_FOR_AN_ANSWER:=true;
end;
end;
procedure TRY_TO_INCREASE_THE_RANK;

{ Try to increase the rank of \( k \) to satisfy a pending rerank request. \( RANK(k) \) is increased if all the sons of \( k \) have a KNOWN_RANK larger than the new rank of \( k \). If the rank is increased then an ANSWER message is sent to all nodes waiting for an increase in the value of \( RANK(k) \) of up to the new rank. The corresponding tuples are moved from WAITING to WAITING_FOR_RELEASE. }

begin

(* ) If there is no tuple \((WS,WT,WRND,WQID) \in WAITING\) for which \((WRND < KRND(j) \text{ for all } j \in \text{SONS})\)
then
exist this procedure;
else
begin

Let \( \overline{WRND} \) be the highest requested round of such a tuple;

{ Increase the rank of \( k \) up to \( \overline{WRND} \) and assign it a height which insures \( RANK(k) < RANK(j) \text{ for all } j \in \text{SONS} \) }

\[ RND(k) := \overline{WRND}; \]
\[ H(k) := 1 + \text{(the maximum } \text{KH(j)} \text{ ("height") of any } j \in \text{SONS with } KRND(j) = \overline{WRND}); \]

{ ANSWER all rerank requests which are satisfied. }

send ANSWER\( (k,RANK(k)) \) to all nodes \( s \) such there is a tuple \((WS,WT,WRND,WQID) \in WAITING\)
with \( WRND < RND(k) \) and \( s \in WS \);

(**) for any tuple \( T = (WS,WT,WRND,WQID) \in WAITING\)
such that \( WRND \geq RND(k) \)
begin
add to \( \text{WAITING\_FOR\_RELEASE} \) the tuple \((WT,WI,WQID);\)
remove \( T \) from \( \text{WAITING} \);
end;

end;

===========================================================================

January 15, 1990
procedure TERMINATE_REQUEST(j: nodename);
{ A procedure to terminate a request REQUEST_EDGE(j) }
begin
    if j = CURRENT_REQUEST then
    begin
        WAITING_FOR_AN_ANSWER := false;
        (*)
        send RELEASE(k, CURRENT_ID) to CURRENT_REQUEST;
        remove CURRENT_REQUEST from MY_REQUEST_QUE;
        if MY_REQUEST_QUE ≠ ∅ then
            TRY_TO_INITIATE_RERANK_PHASE; { Move to next request }
    end
    else
        remove j from MY_REQUEST_QUE;
end;

4.5.2. Request Procedures - Performed in response to the corresponding user request

procedure REQUEST_EDGE(j: nodename);
{ A request to add an edge $k \rightarrow j$ to the graph }
begin
    if KNOWN_RANK(j) ≤ RANK(k) then
    begin
        add j to MY_REQUEST_QUE;
        if j is the CURRENT_REQUEST then
            TRY_TO_INITIATE_RERANK_PHASE;
    end
    else
        ADD_AN_EDGE(j);
end;

procedure DELETE_EDGE(j: nodename);
{ A request to delete an edge $k \rightarrow j$ from the graph. }
begin
    if j ∈ SONS(k) then
    begin
        remove j from SONS(k); { Remove the edge from the graph }
        TRY_TO_INCREASE_THE_RANK;
    end
    else
        if j ∈ MY_REQUEST_QUE then
            TERMINATE_REQUEST(j);
end;

---

January 15, 1990

---
4.5.3. Message Procedures - Performed in response to the corresponding messages

procedure RERANK(j:node; r, i:round; qid:request_id);

{ node j requests that k raise its rank to some value higher than \((r, i), \infty\). If k can comply it will. If k cannot increase its rank because some of its sons have a lower rank it sends RERANK messages to them. }

begin
  if \(i = k\) then
    { k is the initiator of this rerank phase. }
    begin
      { A cycle will be created if the CURRENT_REQUEST edge is added }
      if \(qid = \text{CURRENT_ID}\) and \(r = R(k) + 1\) then
        send CYCLE(CURRENT_REQUEST) to user;
    end
  else
    if RND\(k\) \(\geq (r, i)\) then
      { Node k is already reranked up to the required rank }
      send ANSWER\(k(RANK(k))\) to j
    else
      { If this request was already received (from another node) }
      add j to the WS of the entry corresponding to this request
      \{ *\} if there is a tuple \((WS, WT, WRND, WQID) \in \text{WAITING}\) with \(WRND = (r, j)\) and \(WQID = qid\) then
        add j to WS { A reminder to later send it an ANSWER }
    else
      { If this request is relevent (no request was yet received from the same initiator with a higher request id) }
      begin { create a new tuple in \text{WAITING} }
        \{ **\} add \(((j), \emptyset, (r, i), qid)\) to \text{WAITING};
        if \(KRND(v) \geq (r, i)\) for all \(v \in \text{SONS}\) then
          TRY_TO_INCREASE_THE_RANK
        else
          for every \(v \in \text{SONS}\) such that \(KRND(v) < (r, i)\)
            send RERANK\((k, (r, i), qid)\) to v;
      end;
end;
procedure ANSWER(j :nodename; rnk :rank);

{ Node j answers some RERANK message of k }

begin
  KNOWN_RANK(j) := rnk;
  if j \in MY_REQUEST_QUE then { There is a request to add the edge k \rightarrow j }
    begin
      if RANK(k) < rnk and j = CURRENT_REQUEST then
        begin { The CURRENT_REQUEST terminated successfully }
          ADD_AN_EDGE(j);
          TERMINATE_REQUEST(j);
          end;
      else if j = CURRENT_REQUEST then
        begin { Inconclusive answer }
          WAITING_FOR_AN_ANSWER := false;
          maybe wait some time;
          TRY_TO_INITIATE_RERANK_PHASE;
        end
      else if RANK(k) < rnk then
        begin { A request that is not the first in the queue can be granted }
          ADD_AN_EDGE(j);
          remove j from MY_REQUEST_QUE;
        end
    end
  else ( An ANSWER from a son )
    TRY_TO_INCREASE_THE_RANK;
end;

procedure RELEASE(i :nodename; qid :request_id);

{ The RELEASE message informs node k that the request qid at node i terminated. If the request is still pending this message removes it. }

begin
  send RELEASE(i,qid) to all nodes t such that there is a tuple (WS,WT,(WR,WI),WQID) \in (WAITING or WAITING_FOR_RELEASE) with WI = i and WQID \le qid and t \in WT
    remove all tuples with WI = i and WQID \le qid from WAITING and
    from WAITING_FOR_RELEASE;
end;

Januarv 15, 1990

Technion - Computer Science Department - Technical Report CS0604 - 1990
5. ANALYSIS OF ALGORITHM

Informally, we show that the algorithm has the following properties:

1. The graph remains cycle free (a formal definition of a cycle follows).
2. The algorithm sends CYCLE(j) from node \( k \) only if the addition of the edge \( k \rightarrow j \) would have closed a cycle.
3. Every request to delete an edge is granted immediately.
4. Every request to add an edge from a given node \( k \), which is not canceled by the user, is eventually granted if the following two conditions hold:
   - ASM: For every REQUEST_EDGE(j) presented at node \( k \), \( j \) is not connected forever to \( k \). (A more formal definition follows).
   - ND: Eventually there is some RERANK PHASE initiated by \( k \) such that between the time \( k \) starts the phase and the time \( k \) ends it, \( k \) is not forced to increase its rank (by the messages it receives and the protocol of the algorithm).

Note that an assumption that other nodes do not interfere with the granting of the request of \( k \) implies ND; obviously, if in the entire graph there is a single REQUEST_EDGE message, then ND holds.

We introduce some terminology:

ACTION
The transmission of a message, the reception of a message and the execution of a computation are actions.

EVENT
A collection of actions which includes the reception of a message (from a user or from another node) plus the computations and message transmissions performed in a node in response to this message.

LOCAL TIME
A complete ordering of actions in a given node. Since we assume that messages can only arrive at a node when it is idle, there is a complete ordering between events, so that for any two different events \( A \) and \( B \) which occur in the same node, either \( A \) occurs before \( B \) or \( B \) occurs before \( A \), but not both. Let \( a_1 \) and \( a_2 \) be two actions belonging to events \( A \) and \( B \), respectively. If \( A \neq B \) then \( a_1 \) is before \( a_2 \) (\( a_1 < a_2 \)) if \( A \) occurs before \( B \). If \( A=B \) then \( a_1 \) is before \( a_2 \) if the code of \( a_1 \) is executed before the code of \( a_2 \).

DISTRIBUTED TIME
We define a partial ordering (causality) between actions in the following way:

1) If \( a_1 \) and \( a_2 \) are performed in the same node then \( a_1 \) is before \( a_2 \), \( a_1 < a_2 \), if \( a_1 < a_2 \) (\( a_1 \) is before \( a_2 \) according to the local-time).
2) If \( a_1 \) is the transmission of a message \( M \) and \( a_2 \) is the reception of \( M \) then \( a_1 \) is before \( a_2 \).
3) The transitive closure of (1) and (2).

January 15, 1990

Technion - Computer Science Department - Technical Report CS0604 - 1990
We assume that the resulting relation is a strong partial order. This is similar to the situation in [A1].

We define the relation after as the opposite of before: action $a_1$ is after action $a_2$ ($t_{a_1} > t_{a_2}$) if and only if action $a_2$ is before action $a_1$ ($t_{a_2} < t_{a_1}$).

**DISTRIBUTIVE CYCLE**

Let us define formally the notion of a cycle in a distributive dynamic graph. Assume we add a hypothetical new type of messages FREEZE_CHECK($v$ : nodename). A node $v$ in the idle state can send a FREEZE_CHECK($v$) message to a son, any node receiving a FREEZE_CHECK($v$) message sends it to all its sons, also any node sending a FREEZE_CHECK($v$) message will not add or delete any out edges.

We say that node $k$ is in a CYCLE, if $k$ can send a FREEZE_CHECK($k$) message and eventually receive it.

This definition of a cycle includes the cycles which we intuitively wish to prevent.

We say that a graph is acyclic if it never contains a distributive cycle.

**DISTRIBUTIVE LASTING CONNECTIVITY**

We say that a node $j$ has a Distributive Lasting Connectivity (DLC) to node $k$, after time $t$ ($j$’s local time), if one of the following holds:

1. $j = k$.
2. After time $t$ the edge $j \rightarrow k$ persists forever.
3. There is a set of nodes $V_j$, such that after time $t$, $j$ always has an edge to one of the nodes in $V_j$ and for each node $i \in V_j$ there is a time $t_i < t$ such that, after $t_i$, $i$ has a DLC to $k$.

We use the above definition to formally define

**ASM:** If REQUEST_EDGE($j$) is presented at $k$ then $j$ does not have a DLC to $k$.

### 5.1. PROOF OF CLAIM 1: THE GRAPH REMAINS ACYCLIC

**LEMMA 1.1:** For any idle node $k$, if there is a tuple $(WS,WT,WRND,WQID)$ in WAITING then $WRND > RND(k)$.

**Proof:** Clearly, in procedure RERANK, a tuple $(WS,WT,WRND,WQID)$ is added to WAITING only if $WRND > RND(k)$. Since this is the only place were tuples are entered to WAITING, it is left to be shown that when the rank of $k$ changes the lemma remains true. The rank of $k$ can change only in procedure TRY_TO_INCREASE_THE_RANK and in this procedure when the rank is changed, all tuples in WAITING with a $WRND \leq RND(k)$ are removed from WAITING, before $k$ becomes idle again. □

**LEMMA 1.2:** The rank of any node $k$ cannot decrease.

**Proof:** A node $k$ may only change its rank when performing the TRY_TO_INCREASE_THE_RANK procedure. Let $old\_rank = (old\_rnd, old\_h)$ be the rank of node $k$ before it changed its rank and let $new\_rank = (new\_rnd, new\_h)$ be its rank after the change. There is a tuple in WAITING with $WRND = new\_rnd$, therefore
LEMMA 1.3: Let \( \text{KNOWN\_RANK}(j) \) be the known rank of \( j \) at node \( k \), at local time \( t_2 \). Either \( \text{KNOWN\_RANK}(j) = ((0, j), 1) \) or there is a local time \( t_1 \) at \( j \) such that \( t_1 < t_2 \) and \( \text{RANK}(j) \) at \( t_1 \) equals \( \text{KNOWN\_RANK}(j) \).

Proof: At initialization \( \text{KNOWN\_RANK}(j) = ((0, j), 1) \). The known rank of a node \( j \) can change at node \( k \) only when an \( \text{ANSWER}(j, \text{rk}_j) \) message is received at \( k \). But node \( j \) must have sent this message before it was received at \( k \), and at that time \( \text{RANK}(j) \) equaled \( \text{rk}_j \) at node \( j \).

Let us redefine our invariant

INV: For any node \( k \): If \( j \in \text{SONS}(k) \) then \( \text{RANK}(k) < \text{KNOWN\_RANK}(j) \).

THEOREM 1: The graph is always acyclic.

Proof: Lemmas 1.2 and 1.3 together with INV imply that the graph is acyclic. This can be seen from the definition of a distributive cycle: If we send a \( \text{FREEZE\_CHECK}(v) \) message from some node \( v \) in the graph, the message will travel through a set of nodes such that whenever it travels from a node \( u \) to \( w \) by lemma 1.3 \( \text{RANK}(w) \geq \text{KNOWN\_RANK}(w) \) at \( u \) and by INV \( \text{KNOWN\_RANK}(w) > \text{RANK}(u) \). Since by lemma 1.2 the rank cannot decrease at any node, the message will never arrive back at \( v \).

It remains to show that INV always holds. For any node \( k \), INV is voidly true at initialization since there are no edges.

The only ways INV can be violated at \( k \) are:
1. When \( k \) increases its rank.
2. When \( k \) adds a node to SONS.

The only place in the algorithm in which \( \text{RANK}(k) \) is changed is procedure \( \text{TRY\_TO\_INCREASE\_THE\_RANK} \), where clearly the new rank of a node remains less than the \( \text{KNOWN\_RANK} \) of its sons and INV continues to hold. Whenever procedure \( \text{ADD\_AN\_EDGE}(j) \) is called (the only procedure which adds a node \( j \) to \( \text{SONS}(k) \)), \( \text{KNOWN\_RANK}(j) > \text{RANK}(k) \) and again INV continues to hold.

5.2. PROOF OF CLAIM 2: A CYCLE MESSAGE IS SENT ONLY IF A REQUESTED EDGE WOULD HAVE CLOSED A CYCLE

LEMMA 2.1: A RERANK\((k, (r, i), qid)\) message where \( k \neq i \) is sent from \( k \) to \( j \) only after \( k \) received a RERANK\((v, (r, i), qid)\) message from some node \( v \) and only when there is an edge between \( k \) and \( j \).

Proof: A RERANK\((k, (r, i), qid)\) message where \( k \neq i \) is sent from node \( k \) only in procedure \( \text{SEND\_RERANK\_TO\_SON} \). In this procedure such a message is sent only if there is a tuple \( \text{\((WS, WT, (r, i), qid)\)} \) in \( \text{WAITING} \), but we can clearly see in the procedure \( \text{RERANK} \) that such a tuple is entered to waiting only when a message \( \text{RERANK}(v, (r, i), qid) \) is received at \( k \) from some node \( v \). The only procedures which

January 15, 1990
call SEND_RERANK_TO_SON are RERANK and ADD_AN_EDGE. In both cases the procedure is called with parameter \( j \) only if \( j \in \text{SONS} \). \( \Box \)

THEOREM 2: If node \( k \) sends a CYCLE\((j)\) message to a user then there was a time \( t_1 \) at \( k \) such the addition of the edge \( k \rightarrow j \) at \( t_1 \) would have closed a cycle.

Proof: A CYCLE\((j)\) message is sent to a user from node \( k \) only when \( k \) receives a RERANK\((v,(r,k),qid)\) message from some node \( v \). This message was initiated at \( k \) and sent to \( j \) at some time \( t_1 \). By lemma 2.1, \( k \) could only receive it from \( j \) via a series of nodes \( v_1, v_2, \ldots, v_s \) such that when each node sent the RERANK message, it had an edge to the next node in the series. If the edge \( k \rightarrow j \) would have been added to the graph and instead of the RERANK message, \( k \) would have sent a FREEZE_CHECK\((k)\) message, the FREEZE_CHECK message could follow the same route and eventually arrive at \( k \). \( \Box \)

5.3. PROOF OF CLAIM 3: A DELETE_EDGE REQUEST IS IMMEDIATELY GRANTED

Follows immediately from the code of the DELETE_EDGE procedure.

5.4. PROOF OF CLAIM 4: UNDER ASM AND ND EVERY REQUEST_EDGE IS EVENTUALLY GRANTED

To prove this claim we define formally a RERANK PHASE and a RERANK PROCESS.

A RERANK PHASE - Is a set of messages identified by a tuple \( ((r,k),qid) \) where \( k \) is a nodename, called the initiating node, \( r \) is a round number and \( qid \) is the id of the request to add an edge \( k \rightarrow j \) at node \( k \).

The following messages belong to rerank phase \( ((r,k),qid) \):

1. RERANK\((v,(r,k),qid)\) messages (\( v \) can have any value).
2. ANSWER\((v,(r,n,h))\) messages which are sent in response to RERANK messages of the PHASE.

An ANSWER message is said to be a response to a specific RERANK message if \( rnd \geq (r,k) \) and either it was sent in the same event in which the RERANK\((v,(r,k),qid)\) message was received, or it was sent to some node \( s \in \text{SONS} \) in a tuple \( (WS,WT,(r,k),qid) \) in \( \text{WAITING} \). Notice that this tuple will be removed from \( \text{WAITING}(v) \) when the message is sent. Also notice that an ANSWER message can be a response to more than one RERANK message and thus belong to more than one single rerank phase.

When a rerank message of a phase \( ((r,k),qid) \) is sent from node \( k \) it is called the initiating message of the phase. It is easy to see that each phase has a unique initiating message.

When the initiating node \( k \) receives an ANSWER message which belongs to the phase it is called the terminating message of the phase.

A TERMINATING PHASE - identified by a tuple \( (k,qid) \) where \( k \) and \( qid \) are as above - is the set of all RELEASE\((k,qid)\) messages.

January 15, 1990
A RERANK PROCESS - identified by a tuple \((k,qid)\) where \(k\) and \(qid\) are as above - is the union of all rerank phases \(((r,k),qid)\), for any \(r\), and the terminating phase \((k,qid)\).

When a RELEASE\((k,qid)\) message is sent from node \(k\) it is called the terminating message of the process. Note that when a terminating message is sent, the request to add an edge has already been granted or cancelled; however it is not necessarily the last message of the phase to be sent in the network.

Note the following observations about rerank phase and processes:

1. The number of messages which belong to a given phase is \(O(|E|)\). (Follows from the fact that each node sends on each of its edges at most one RERANK message and receives one ANSWER message).
2. The number of messages which belong to a terminating phase is \(O(|E|)\). (Follows from the fact that each node sends on each of its edges at most one RELEASE message).

Proof outline:

1. Every node \(v \neq i\) which receives a RERANK\((k,(r,i),qid)\) message and never answers it has a bounded rank.
2. Every node which receives a RERANK\((k,(r,i),qid)\) message and never answers it has a DLC to \(i\).
3. ASM implies that every rerank phase terminates.
4. The fact that every rerank phase terminates together with ND imply that every request to add an edge from a given node \(k\) which is not canceled by the user is eventually granted.

**Lemma 4.1**: For every node \(v \neq i\) which receives a RERANK\((k,(r,i),qid)\) message and never answers it there is a time after which \(\text{RANK}(v)\) never changes.

Proof: It is easy to see that whenever \(v\) increases its rank, its round is also increased. However since \(v\) never answers the rerank request \(((r,i),qid)\), its round must be bounded by \((r,i)\). Since the number of nodes in the graph is finite, \(v\) can increase its round only a finite number of times before its round will increase above \((r,i)\). After the last of these times, \(\text{RANK}(v)\) will not change.

**Lemma 4.2**: Every node which receives a RERANK\((k,(r,i),qid)\) message and never answers it has a DLC to \(i\).

Proof: Let \(V\) be the set of nodes which receive the rerank message and never answers it. By Lemma 4.1 for each such node \(v \neq i\), eventually there is a time after which its \(\text{RANK}(v)\) remains fixed. Clearly any \(v \in V\) such that \(v \neq i\), eventually, always has a son in \(V\) (not necessarily a fixed one). Let \(w\) be a node of highest rank in \(V\) which does not have a DLC to \(i\). Since any node has a DLC to itself then \(w \neq i\). But then \(w\) must always have an edge to at least one of a set of nodes \(V_w \subseteq V\) each having a rank higher than the rank of \(w\). Each such member of \(V_w\) must therefore have a DLC to \(i\). Hence \(w\) also has a DLC to \(i\).

**Corollary**: ASM implies that every rerank phase eventually terminates.

Proof: If \(k\) initiates a rerank phase by sending a RERANK message to \(j\) then \(k\) must have received a REQUEST_EDGE\((j)\) message previously. But then ASM implies that \(j\)

January 15, 1990
THEOREM 4.3: ASM and ND imply that every persistent request to add an edge is eventually granted.

Proof: The Corollary to Lemma 4.2 and ASM imply that every rerank phase eventually terminates. When a rerank phase terminates either $\text{KNOWN_RANK}(j) > \text{RANK}(j)$ and the request is granted or $\text{KNOWN_RANK}(j) < \text{RANK}(j)$. It follows that $k$ must have received a message which forced it to increase its rank sometimes between the time it has initiated the rerank phase and the time it has received the terminating message of the phase. However ND implies that eventually there is a rerank phase during which $k$ is not forced to increase its rank. When that phase terminates the requested edge will be granted.

6. WHAT IS WRONG WITH ALGORITHM 1?

In the previous sections we showed that algorithm 1 guarantees that every request to add an edge to the graph is eventually granted, under the assumptions ASM and ND. While ASM seems like a natural assumption on the graph, for a cycle prevention algorithm, ND does not seem very natural. If the pattern of requests to add edges to the graph is such that the number of nodes, with pending requests to add edges at any given time, is small, it is possible that in practice ND holds. However, even in such cases there is no guarantee that ND holds. Indeed it is possible to build worst case scenarios in which a given small set of requests remains pending forever, while at least one of the requested edges could be added to the graph without creating a cycle.

Consider two nodes $k$ and $j$ in a graph $G$ such that the communication between $k$ and $j$ is not necessarily FIFO. Suppose there are requests to add an edge from $k$ to $j$ and from $j$ to $k$. Clearly, although both requests cannot be granted, each one of them can be granted. Let $((0,k),0) > (0,j),0$ be the initial respective ranks of the two nodes. The worst case protocol is as follows:

$k$ receives an ADD_EDGE request. Since $\text{KNOWN_RANK}(j) < \text{RANK}(k)$, $k$ sends a RERANK$(k,(1,k),qid_k)$ message to $j$.

$j$ receives the message, increases its RANK to $(1,k),1)$ and sends ANSWER$(j,(1,k),1))$ to $k$.

Now $j$ receives an ADD_EDGE request. Since $\text{KNOWN_RANK}(j) = (0,k),0 < \text{RANK}(j) = (1,k),1)$, $j$ sends a RERANK$(j,(2,j),qid_j)$ message to $k$.

Before $k$ receives the ANSWER message from $j$, it receives the RERANK message, increases its RANK to $(2,j),1)$ and sends ANSWER$(k,(2,j),1))$ to $j$.

Now $k$ receives the ANSWER$(j,(1,k),1))$ message, but it is useless since $\text{KNOWN_RANK}(j) = (1,k),1 < \text{RANK}(k) = (2,j),1)$. Therefore $k$ sends a RERANK$(k,(2,k),qid_k)$ message to $j$.

Clearly, if the RERANK messages continue to arrive before the ANSWER messages the above protocol can continue indefinitely.

If we assume a communication model in which all channels are FIFO, the above scenario will no longer loop indefinitely. However it is possible to create a similar scenario which will. Consider a new graph with nodes $k_1, k_2, j_1$ and $j_2$. Suppose the

January 15, 1990
The graph already contains the edges $j_1 \rightarrow k_2$ and $j_2 \rightarrow k_1$. We request to add edges from $k_2$ to $j_2$ and from $k_1$ to $j_1$. The worst case protocol is identical to the previous protocol except that now we force the RERANK messages to arrive before the ANSWER messages by manipulating the delays on each communication channel, so that by the time an ANSWER message travels from $k_2$ to $j_1$ to $k_1$ the interfering RERANK message traveling from $k_2$ to $j_2$ to $k_1$ has already arrived.

Assuming that the communication channels are synchronous helps this specific scenario, but does not really solve the problem, because we can still slow down the ANSWER messages without slowing down the RERANK messages, at nodes $j_1$ and $j_2$ by adding to each a path of two or more nodes.

We therefore develop algorithm 2 which guarantees that every request to add an edge is eventually granted without the assumption ND. (even if FIFO does not hold).

7. DESCRIPTION OF ALGORITHM 2

7.1. INFORMAL DESCRIPTION OF ALGORITHM 2

The second algorithm is similar to the first. The major difference is that we add to each node a new priority parameter which is used to ensure that other nodes do not interfere with its rerank phases "too many times". The priority of a node is determined as follows: A node which does not have any pending requests to add edges has a priority of 0. A node increases its priority by one every time it initiates a rerank phase. When a request to add an edge is granted or cancelled the priority of the node from which this edge emanates drops back to 0. Now the rerank messages include the priority of the initiating node. When a node receives a rerank message it compares its priority to the priority of the initiating node. If its priority is higher it will not increase its rank to comply with this request until its own priority drops (when its CURRENT_REQUEST to add an edge is either granted or cancelled). If the priority of the initiator is equal or higher than the priority of the receiving node, the receiving node increases its rank (if possible). When a node $v$ increases its rank, it is possible that this increase interferes with a pending rerank request initiated by $v$. When this happens $v$ becomes captured by the rerank process which "caused" this increase in the rank of $v$. A node which is captured behaves exactly the same as a node which is not captured, except that it does not initiate new rerank phases. It stops being captured only when a RELEASE message arrives, indicating that one of its capturing rerank processes has terminated.

We show that the priority parameter has the property that the number of nodes with a priority higher than or equal to $n$ is at most $(|V| - n + 1)$. Therefore a node cannot be interrupted more than $|V| - 1$ times and thus every request to add an edge is eventually granted if ASM holds.

7.2. FORMAL DESCRIPTION OF ALGORITHM 2

7.2.1. Data Types

Same as for algorithm 1 except for the following:

priority: A nonnegative integer.
7.2.2. Messages To and From Users
Same as for algorithm 1.

7.2.3. Messages Between Nodes
Same as for algorithm 1 except for the following:
RERANK(j; nodename; (r, i): round; p:priority; qid:request_id) - Here the p is the priority of the initiating node at the time of initiation.

7.2.4. Data Structures
Same as for algorithm 1 except for the following:
P(k) - The priority of node k.
CAPTURED - A boolean variable, indicating whether k is captured or not.
CAPTURING_TUPLES - A set of tuples (i:nodename,qid:request_id) which identify the rerank processes which captured node k. If k is captured then any RELEASE(i,qid) message will release it, if there is a tuple in CAPTURING_TUPLES with the same i and a request_id less than or equal to qid.
WAITING - To each tuple in this data-structure we add a field WP which indicates the priority of the rerank request corresponding to that tuple.

7.2.5. Procedures
Same as for algorithm 1 except for the following changes:
Replace the contents of procedure TRY_TO_INITIATE_RERANK_PHASE with the following:
begin
if (not CAPTURED) and (not WAITING_FOR_AN_ANSWER) then
begin
P(k):=P(k)+1;
send RERANK (k, (R(k)+1, k), P(k), CURRENT_ID)
to CURRENT_REQUEST;
WAITING_FOR_AN_ANSWER:=true;
end;
end;
In procedure TRY_TO_INCREASE_THE_RANK replace line [*] by the following:
If there is no tuple (WS,WT,WRND,WP,WQID) ∈ WAITING for which
(WRND ≤ KRND(j), for all j ∈ SONS) AND (WP ≥ P(k))
In procedure TRY_TO_INCREASE_THE_RANK after line [**] add the following:
if (the increase in RANK(k) in this procedure interferes with
the last rerank phase initiated by k which
has not yet terminated) then
begin
CAPTURED:=true;
for every tuple (WS,WT,(WR,WT),WP,WQID) ∈ WAITING
such that WP ≥ P(k) and

January 15, 1990
RND(\(k\)) ≥ WRND > the round of the last rerank phase initiated by \(k\).
add (\(W_I, W_QID\)) to CAPTURING_TUPLES;
end;

In procedure TERMINATE_REQUEST after line \{'\} add the following:
P(\(k\))=0;

In procedure RERANK replace line \{'\} with the following:
if there is a tuple (\(WS, WT, WRND, WP, W_QID\)) ∈ WAITING

In procedure RERANK replace line \{'\'} with the following:
add ((\(j, \emptyset, (r, i), p, qid\))) to WAITING;

Replace the contents of procedure RELEASE with the following:
begin
  send RELEASE(i, qid) to all nodes \(t\) such that there is a tuple
  (\(WS, WT, (WR, WT), WP, W_QID\)) ∈ (WAITING or WAITING_FOR_RELEASE)
  with \(WI = i\) and \(W_QID ≤ qid\) and \(t ∈ WT\)
remove all tuples with \(WI = i\) and \(W_QID ≤ qid\) from WAITING and
from WAITING_FOR_RELEASE;
if there is a tuple (\(i, \overline{qid}\)) ∈ CAPTURING_TUPLES
such that \(qid ≤ \overline{qid}\) then
begin
  CAPTURED:=false;
  CAPTURING_TUPLES := \(\emptyset\);
  if MY_REQUEST_QUE \(\neq\) \(\emptyset\) then
  TRY_TO_INITIATE_RERANK_PHASE;
end;
end;

8. ANALYSIS OF ALGORITHM 2
The changes made in moving from algorithm 1 to algorithm 2 do not affect the following properties of the algorithm:
1. The graph remains cycle free (a cycle as defined above).
2. The algorithm sends CYCLE(\(j\)) from node \(k\) only if the addition of the edge \(k → j\)
would have closed a cycle.
3. Every request to delete an edge is granted immediately.
We prove an additional fourth property of algorithm 2:
4. ASM implies that every request to add an edge from a given node \(k\), which is not
canceled by the user, is eventually granted.
Proof outline:
1. CLM: From the point of view of any node \(k\), the number of nodes with a priority
greater than or equal to some value \(p\) is at most \(|V|+1-p\).
2. If every request to add an edge from a node, of priority greater than or equal to \( p \), is eventually granted, then every rerank phase initiated by a node of priority \( p-1 \) eventually terminates.

3. A request to add an edge from a node with priority \( |v| \) is eventually granted and so, by induction, every request is eventually granted.

About the proof of CLM:

Intuitively suppose we order the nodes of the graph according to their priorities such that a node with greater priority precedes a node with lesser priority (i.e., is placed in a higher position in the order). In this case we find that CLM is analogous to saying that if one plots the order of the nodes against their priority no point appears above the line \( \text{order} + \text{priority} = |v| + 1 \). This follows if before a node increases its priority by one it always has to move up at least one place in the order axis. Observe that before a node \( v \) can increase its priority by one, another node \( w \) with a priority greater than or equal to \( P(v) \) reduces its priority to 0 (or 1) and thus \( v \) moves one place up the order axis. The fact that there has to be such a node can be seen from the code: Before any node increases its priority it must first be CAPTURED and then RELEASED. Before it is RELEASED the priority of the node which has captured it (by initiating the rerank phase which has captured it) must drop to 0 (or 1) as a result of the successful termination or cancellation of some ADD_EDGE request. Therefore if CLM held before the priority of \( k \) dropped to 0 it should also hold after all the nodes captured by \( k \) are RELEASED and increase their priorities by one.

Unfortunately we are unable to formalize this global time point of view. In the following paragraphs we present a distributive time formal proof of the claim which is based on the viewpoint of an observer located at some generic node \( k \). We claim that if such an observer were to build the above diagram, he could formalize the intuitive proof and convince himself that "from his point of view" CLM holds. As will be seen, this "local" claim is sufficient to prove the fourth (fairness) property of algorithm 2.

We add two data-structures to our algorithm and formalize the rules of their manipulation. Using these additional data-structures we prove our claim. Since it will be evident that these data-structures do not interfere with the execution of the algorithm in any way, our claim remains true even if these data-structures are absent. The additional data-structures are called: LIST_OF_UPDATES and KNOWN_PRIORITIES.

**LIST_OF_UPDATES** - A list of tuples of the form \((i : \text{nodename}, \text{rnd:round}, p:\text{priority}, qid: \text{requescid})\). Initially LIST_OF_UPDATES is empty. When a node \( k \) changes its priority to some value \( p \), it enters into its LIST_OF_UPDATES a tuple \((k, \text{rnd}, p, qid)\) where \( qid \) is the value of CURRENT_REQ at the time and \( \text{rnd} \) is as follows:

1. If \( p \neq 0 \) then \( \text{rnd} \) is the round which \( k \) sends in the RERANK message which has \( p \) and \( qid \) as its priority and request_id, respectively.
2. If \( p = 0 \) then \( \text{rnd} \) is the round which \( k \) sent in the last RERANK message having \( qid \) as its request_id.

When a node \( k \) sends any one of the messages of the algorithm it appends to the message the whole LIST_OF_UPDATES at \( k \). When \( k \) receives a message from another node it updates its LIST_OF_UPDATES to be the union of its previous list of updates and the set of tuples appended to the message.
We define two relations between tuples:

- **locally precede**: If two tuples are created consecutively at the same node then the one created first locally precedes the other.

- **strictly precede**: A tuple $T_w = (w, rnd_w, 0, qid_w)$ strictly precedes any tuple $T_v = (v, rnd_v, p, qid_v)$ if $(w, qid_w)$ is the process which captured and released node $v$. (When the RELEASE($w$, $qid_w$) message was received at $v$, $v$ initiated its $p$'th phase in rerank process $qid_w$).

Observe the following properties of these relations:

1. Whenever $T_i$ locally precedes $T_j$ or $T_i$ strictly precedes $T_j$ then $T_i$ was created in distributive time before $T_j$.

2. If there is a sequence of tuples $T_1, T_2, \ldots, T_n$ such that for each $i$, $T_i$ either locally precedes or strictly precedes the tuple $T_{i+1}$ then if $T_n$ is present in the LIST_OF_UPDATES at some node, then all the tuples in the sequence must also be present in the LIST_OF_UPDATES at that node.

3. Each tuple is locally preceded by at most one tuple.

4. A tuple with priority zero or one is never strictly preceded by another tuple (since a node reaches these priorities without being released).

5. A tuple with priority greater than one is always strictly preceded by exactly one tuple. The strictly preceding tuple must have a zero priority.

We view the LIST_OF_UPDATES as a directed graph where each tuple is a vertex and each arc corresponds to either a locally precede relation or to a strictly precede relation.

A claw, defined by a tuple $T = (v, rnd, 0, qid)$, consists of $T$, its outgoing strictly precede arcs and their endpoints.

**LEMMA 5.1:** If there is a directed path, of local precede arcs, from a vertex $T_a = (v_a, rnd_a, p_a, qid_a)$, in the claw defined by $T_1 = (v_1, rnd_1, 0, qid_1)$, to a vertex $T_b$ in the claw defined by $T_2 = (v_2, rnd_2, 0, qid_2)$, then $rnd_1 < rnd_2$.

**Proof:** Let $T_a, T_{x_1}, T_{x_2}, \ldots, T_{x_n}, T_b$ be the path of local precede arcs.

Clearly if a tuple $T_j$ locally precedes a tuple $T_j$ then $rnd_j \leq rnd_j$, (the equal case may happen when the second tuple has a priority of zero or one). Therefore, $rnd_a \leq rnd_{x_1} \leq rnd_{x_2} \leq \cdots \leq rnd_{x_n} \leq rnd_b$. If $T_b = T_2$ then $rnd_{x_n} \leq rnd_2$.

Also, if a tuple $T_j$ is strictly preceded by a tuple $T_i$ and locally preceded by a tuple $T_k$ then $rnd_k < rnd_i < rnd_j$. Therefore, $rnd_1 < rnd_2$. If $T_b \neq T_2$ then $rnd_{x_n} < rnd_2 < rnd_b$.

Together, these claims imply that $rnd_1 < rnd_a \leq rnd_{x_n} \leq rnd_2$. □

**Corollary:** There are no local precede arcs between vertices of the same claw.

We order the tuples according to the following algorithm:

1. Shrink each claw into a single vertex. Each arc, which enters or exits a vertex in the claw, now enters or exits the new vertex representing the claw.

2. Order the vertices of the (new) graph according to the remaining arcs of the graph (all of which are local precede arcs), such that if there is an arc from $T_i$ to $T_j$ then $T_i$ is before $T_j$ in the order.

January 15, 1990
3. Re-expand each claw and place its vertices in the order as follows: With respect to vertices outside the claw, all the tuples of the claw occupy the same position in the order which the claw vertex has occupied before the expansion. The tuples of the claw are ordered among themselves as follows: The tuple which defines the claw is placed first and the remaining tuples appear in descending order of priority.

Since the arcs of the graph imply distributive time precedence, the original graph is acyclic. Lemma 5.1 and its corollary imply that after shrinking each claw to a single vertex, the graph remains acyclic. Therefore the tuples can always be ordered according to the above algorithm.

KNOWN_PRIORITIES is a local diagram of order vs. priority (as in the intuitive proof approach). For each node of G there is an entry in KNOWN_PRIORITIES which indicates its priority and its order. Whenever the LIST_OF_UPDATES is changed at some node \( k \), our hypothetical observer, located at \( k \), recalculates the KNOWN_PRIORITIES as follows: First he initializes the diagram by setting the priorities of all nodes to zero and ordering them in some arbitrary order. Then he considers the tuples in LIST_OF_UPDATES one by one according to the order produced by the algorithm above, and updates the KNOWN_PRIORITIES as follows:

Consider the next tuple \( (v, \text{rnd}, p, qid) \).

- If \( p > 0 \) then
  1. Switch the order between node \( v \) and the node of highest order among those of priority \( p - 1 \).
  2. Increase the priority of \( v \) to \( p \).
- Else \( (p = 0) \)
  1. Decrease the priority of \( v \) to 0.
  2. Move \( v \) down to the last place in the order.

THEOREM 5.2 At any given node \( k \), under any complete order of the tuples in LIST_OF_UPDATES produced by the ordering algorithm, for each node in the KNOWN_PRIORITIES diagram \( \text{order} + \text{priority} \leq |V| + 1 \), (even while KNOWN_PRIORITIES is being updated).

Proof: Initially, before any tuple is considered, the theorem holds. Let \( T_v = (v, \text{rnd}_v, p_v, qid_v) \) be some tuple in LIST_OF_UPDATES at some node \( k \), such that before \( T_v \) is considered the theorem holds. Clearly, if \( p_v = 0 \) or \( p_v = 1 \) then the theorem continues to hold after \( T_v \) is considered. Let \( p_v \) be some priority greater than 1. In this case there must be a tuple \( T_w = (w, \text{rnd}_w, 0, qid_w) \) which strictly precedes \( T_v \). Clearly, this tuple is present in LIST_OF_UPDATES at \( k \) and is considered before \( T_v \).

Before \( T_w \) is considered the priority of node \( w \) is greater than or equal to \( p_v - 1 \). Since the theorem holds before \( T_w \) is considered, then, at that time, for every node of priority \( p - 1 \) order \( \leq |V| - p_v + 2 \). Therefore, after \( T_w \) is considered for every such node \( p - 1 \) order \( \leq |V| - p_v + 1 \).

From the ordering algorithm, it is easy to see that all the tuples between \( T_w \) and \( T_v \) have a priority greater than or equal to \( p_v - 1 \) and are strictly preceded by \( T_w \). Therefore,
after they are all considered, for every node of priority $p_v - 1$ the order remains less than or equal to $|V| - p_{v+1}$; and so, when $T_v$ is considered, and the priority of $v$ is increased to $p_v$, the theorem continues to hold. □

COROLLARY: A node of priority $|V|$ is never captured.

LEMMA 5.3 If every request to add an edge, emanating from a node of priority greater than or equal to some value $p$, is eventually granted, then every rerank phase, initiated by a node of priority $p - 1$, eventually terminates.

Proof: Let $rph$ be a rerank phase, initiated at some node $k$, in attempt to add an edge $k \rightarrow j$, such that $k$ has a priority $p - 1$ when $rph$ is initiated. Let $v$ be some node which receives a rerank message belonging to $rph$. If $v$ does not answer by some given time, then either its priority is greater than $p - 1$ or it has a son which has not yet answered $rph$.

By the assumption of the lemma, if the priority of $v$ is greater than $p - 1$ then the CURRENT_REQUEST of $v$ is eventually granted. Therefore, whenever it has a priority greater than $p - 1$, it eventually drops its priority to 0 or 1 (depending on whether it has other requests in the request queue). Clearly, if $v$, always eventually increases its priority above $p - 1$ again, then eventually it answers $rph$ (for a node to increase its priority above $p - 1$ it has to be captured $p - 1$ times, each time it is captured its round increases, until it is eventually higher than the round of $rph$ and it answers). Therefore, if $v$ never answers $rph$ then, after some time $t$, $v$ never increases its priority above $p - 1$ and so must always have a son which has not yet answered $rph$.

In this case the situation is similar to that of Lemma 4.2 and so, by the same arguments as in the proofs of Lemma 4.2 and its corollary, $rph$ eventually terminates. □

THEOREM 5.4: ASM implies that every ADD_EDGE request, not canceled by the user, is eventually granted.

Proof: By induction on $p$, the priority of the node $k$ which has received the ADD_EDGE request.

Basis: The CURRENT_REQUEST of a node of priority $p = |V|$ is eventually granted.

Proof of basis: Clearly, every node of priority $|V|$ initiates a rerank phase with this priority. For such a node, algorithm 2 is identical to algorithm 1 with a guarantee that ND holds; since by the corollary to theorem 5.2 a node of priority $|V|$ cannot be captured. By Theorem 4.3, the request of such a node is eventually granted.

Inductive hypothesis: The CURRENT_REQUEST of a node of priority $p = m$ is eventually granted.

Inductive step: The inductive hypothesis implies that the CURRENT_REQUEST of a node of priority $p = m - 1$ is eventually granted.

Proof of the inductive step:

The inductive hypothesis and Lemma 5.3 imply that a rerank phase $rph$, initiated by a node $k$ of priority $m - 1$, eventually terminates. When $rph$ terminates there are three possibilities:

1. $k$ is not captured. In this case the requested edge is granted.
2. $k$ is captured but is eventually released. In this case the priority of $k$ is increased to $m$ and by the inductive hypothesis the requested edge is eventually granted.

January 15, 1990
3. \( k \) is captured and never released.

It remains to show that the third possibility cannot occur.

Let \( C \) be the set of all nodes of priority \( m-1 \) which remain captured forever. Let \( v \) be the node in \( C \) which sent the highest round in its last rerank message. Clearly, \( v \) cannot be captured by a rerank process which ever reaches a priority greater than \( m-1 \), since, by the inductive hypothesis, such a process eventually terminates and \( v \) is released. However, \( v \) cannot be captured by a rerank process of priority less than \( m-1 \), and so must be captured by a rerank process which remains forever of priority \( m-1 \). Following the arguments of the previous paragraph, \( v \) must be captured by a rerank process initiated by a node which is also a member of \( C \). However this is impossible since any node in \( C \), other than \( v \), sent all of its rerank messages with a round less than the round sent by \( v \) and so could not capture it. Therefore \( C \) must be empty.

9. THE FINAL ALGORITHM

In the previous section we showed that \( \text{algorithm}_2 \) can guarantee that every request to add an edge to the graph is eventually granted, under the assumption ASM. Although this assumption is quite natural for requests in a cycle prevention algorithm, let us consider a weaker assumption

\( \text{WASM} \): for every node \( k \), if a request \( \text{edge}(j) \) is presented at \( k \), then there is no directed path from \( j \) to \( k \) which persists forever.

Clearly, a request to add an edge, such that a path from its target to its source persists forever, cannot be granted, if the graph is to remain acyclic. We wish to construct a final algorithm, \( \text{algorithm}_3 \), which keeps the graph acyclic and in which every request to add an edge is eventually granted, under the assumption \( \text{WASM} \).

We define a port-graph \( G_\mathcal{P} = (V_\mathcal{P},E_\mathcal{P}) \) (induced by a graph \( G = (V,E) \)) as follows:

\[
\begin{align*}
V_\mathcal{P} &= V \cup \mathcal{P}, \\
S_1 &= \{(v,(v,w)) | (v,w) \in \mathcal{P}\}, \\
S_2 &= \{(v,w) \rightarrow w | (v,w) \in \mathcal{P}\}, \\
S_\mathcal{P} &= (S_1 \cup S_2)
\end{align*}
\]

We define a dynamic port-graph to be a dynamic graph such that at some initial time \( t \) it is a port-graph and such that although this time only edges from the set \( S_2 \) are added to or deleted from the graph. (After every change in the graph it remains a port-graph).

Theorem 6.1: If \( G_\mathcal{P} \) is an acyclic dynamic port-graph in which \( \text{WASM} \) holds, then ASM holds in \( G_\mathcal{P} \) as well.

Proof: We prove by contradiction. We assume that ASM does not hold on \( G_\mathcal{P} \). Therefore, there is a request to add an edge \( k \rightarrow j \) such that there is a DLC from \( j \) to \( k \). Since the graph is a dynamic port-graph, \( j \) always has out going edges to every node \( u_p \in \mathcal{P} \). Since \( j \) has a DLC to \( k \), there must be at least one node \( u_p \) which also has a DLC to \( k \). Since this node can only have a single edge (to node \( v \in V \)), then this edge must always exist and \( v \) must have a DLC to \( k \). Since the graph is acyclic \( v \neq j \). If \( v=k \) then we have a contradiction to \( \text{WASM} \), since the path \( j \rightarrow u_p \rightarrow k \) always exists after there has been a
request to add the edge \( k \rightarrow j \). If \( v \neq k \) then we follow the same argument as above recursively. Since the number of nodes in \( V \) is finite we eventually reach a contradiction. \( \square \)

Using the above claim we now develop \textit{algorithm 3}. This algorithm is identical to \textit{algorithm 2} except that it is executed on the port-graph induced by original graph. In each node \( k \) of the original graph we maintain data-structures and execute the algorithm as if the node was a \textit{claw} which includes: the actual-node \( k \in V \) which defines the claw, a port \( u_{kv} \in P \) for every \( v \neq k \in V \) (such that the edge \( k \rightarrow v \) might be requested) and appropriate edges (from \( k \) to each node \( u_{kv} \)). Each request to add or delete an edge \( k \rightarrow j \) in the original graph is treated as a request to add or delete the edge \( u_{kj} \rightarrow j \). (Each message sent between nodes which reside in the same "real" node are directly sent and received through the memory of the "real" node, other messages are sent through the communication lines of the graph \( G \)).

Clearly, the induced graph is a dynamic port-graph. It is easy to see that if \textit{WASM} holds on the original graph it also holds on the induced port-graph. By Theorem 6.1 this implies that \textit{ASM} also holds on the port-graph. Therefore, every request to add or delete an edge in the port-graph is eventually granted. However, every request, to add or delete an edge on the original graph, directly corresponds to a request to add or delete an edge in the port-graph. Moreover, after each such request is granted the graph remains a port-graph. Since every request on the port-graph is eventually granted, so does every request on the original graph.

Therefore, \textit{algorithm 3} keeps the graph acyclic and guarantees that every request to add (or delete) an edge is eventually granted under the weak assumption \textit{WASM}. In this respect \textit{algorithm 3} is optimal.

10. A WORD ABOUT COMPLEXITY

It is easy to show that every rerank phase in algorithms 1 and 2 requires at most \( O(\#E) \) messages. Since the number of rerank phases per request in \textit{algorithm 2} is at most \( |V| \), the message complexity of granting a request in \textit{algorithm 2} is \( O(|E| |V|) \). It is possible to show that the message complexity of \textit{algorithm 3} is also at most \( O(|E| |V|) \), where \( V \) and \( E \) are the nodes and edges of the real graph (in this complexity we only count messages between real nodes). This follows from the fact that although the port-graph contains \( O(|V|^{1.5}) \) nodes, only \( |V| \) of them (the actual-nodes) communicate with "end-users" and since these nodes treat the ADD\_EDGE requests one at a time (i.e. send the next ADD\_EDGE request to the appropriate port only after the previous ADD\_EDGE request has been granted), the graph always contains only \( |V| \) nodes which can initiate rerank phases (or capture other nodes). Note that the restriction that ADD\_EDGE requests are treated one at a time can be removed. In this case whenever an "actual-node" receives an ADD\_EDGE request, it immediately sends it to the appropriate port. In the worst case, the message complexity of the unrestricted algorithm increases to \( O(|E| |V|^{1.5}) \) messages per request. However, under a light load of requests, the average time complexity per request (assuming all messages arrive within a bounded time) might actually improve in the unrestricted algorithm.

In all three algorithms, the memory requirement of a node \( v \) remains \( O(|V| \cdot \text{TO}(v) \cdot \text{FROM}(v)) \), (see section 1).

January 16, 1990
11. References
