ALGORITHMS FOR DISTRIBUTED SPANNING TREE CONSTRUCTION IN DYNAMIC NETWORKS

(Extended version of TR#401)

by

E. Korach and M. Markowitz

Technical Report #464
August 1987
Algorithms for Distributed Spanning Tree Construction in Dynamic Networks

by

Ephraim Korach and Moshe Markowitz

Computer Science Department
Technion - Israel Institute of Technology
Haifa, Israel 32000

ABSTRACT

The main result of this paper, is an algorithm for leader election and spanning tree construction in distributed asynchronous networks, whose links may fail and recover during the execution of the algorithm. This algorithm is obtained by a natural three step extension of Korach-Moran-Zaks algorithm for leader election in a reliable complete network. The algorithms presented at the intermediate stages have each advantages of their own.

In the first step, we present an algorithm that chooses a leader (and constructs a spanning tree) in a general reliable network. The message complexity of this algorithm is $2m + 4n \log k$, where $k$ is the number of processors that were spontaneously awakened. (It is a slight improvement of the known algorithm of Gallager-Humblet-Spira which construct a spanning tree with at most $2m + 5n \log n$ messages, although their algorithm constructs, in fact, a minimum weight spanning tree.) A different approach presented by Korach-Kutten-Moran, yield a $2m + 3n \log k$ messages bound algorithm to this problem.

In the second step we add a dynamic links failure to our model and further extend the algorithm to work under this model. Assuming that the network eventually remains connected, this algorithm elects a leader and constructs a spanning tree. The message complexity of this case is $O(t(m + n \log k))$, where $t$ is the number of links that failed during the execution. This result is better than the $O(t(m + n \log k))$ message complexity that a naive approach to this problem would yield.

Finally we deal with the full dynamic network model, i.e. a network whose links may fail and recover infinitely often during a communication process. The algorithm we present guarantees to maintain a spanning tree and a single leader in every connected component of such a network, provided that the topology changes of the network occur slowly enough with respect to the time of a global communication process.

Some other useful algorithms for different models of unreliable networks are also presented.
1. INTRODUCTION

The goal of this paper is an algorithm for leader election and spanning tree construction in a dynamic asynchronous network. We start with a basic model of a reliable complete network, for which an algorithm exists, and gradually extend the model along with its algorithm, up to a dynamic network.

Our basic model is a complete network of \( n \) processors with distinct identities \( \text{id}(1), \text{id}(2), \ldots, \text{id}(n) \). A processor has communication lines which connect it with all the other processors. Initially, the processors do not know any other processor's identity and are all performing the same algorithm. In the leader election problem an arbitrary subset of processors, awaken spontaneously at arbitrary times and start the execution of the algorithm. In addition to local computation, the algorithm may include operations of sending and receiving messages through the communication lines. A processor that was not a starter remains asleep until it receives a message. A message reaches its destination within a finite time and with no error. After the message exchange terminates, exactly one leader is elected. The algorithm we use for this model is from [KMZ], which requires at most \( 5n \log k + O(n) \) messages, where \( k \) is the number of nodes that were spontaneously awaken \((1 \leq k \leq n)\).

In the next step we extend our complete network model to a general connected network. For this purpose we give an extension to the [KMZ] algorithm which finds a leader with at most \( 2m + 4n \log_2 k \) messages \((m = |E|)\).

The algorithm that is presented here improves the message complexity of the known algorithm for a spanning tree construction, given in [GHS], that requires at most \( 2m + 5n \log_2 n \) messages. However, a different approach to this problem presented in [KKM] yields an \( 2m + 3n \log_2 k \) algorithm. Note that if only \( O(1) \) nodes start the algorithm then the message complexity is reduced to \( O(m) \). However our algorithm constructs only a spanning tree - not a minimum weighted spanning tree.

On the next step we assume that links may fail during the leader election process. When a failure happens, it is observed, within a finite time, by the processors at both ends of the link. The algorithm for a general reliable graph is extended to deal with these kinds of failures. Assuming that the network eventually remains connected, this algorithm elects a leader and construct a spanning tree in it. The message complexity of this algorithm is \( O(tm + n \log_2 k) \), where \( t \) is the number of links that failed during the execution (the number of failures is at most \(|E| \) since there is no recovery and a link may fail only once.) This result is better than the \( O(t(m + n \log_2 k)) \) message complexity that a naive approach to this problem would yield.
Finally we deal with the full model of dynamic network, i.e. a network in which the links may fail and recover infinitely often during a communication process. The algorithm we present guarantees to maintain a spanning tree and a single leader in every connected component of such a network provided that the topology changes of the network occur slowly enough with respect to the time of a global communication process.

The technique used to extend the basic algorithm to a general graph could be generalized also for some other special classes of graph where the message complexity remains $O(n \log k)$.

In particular, this technique is shown to be useful for devising an efficient adaptive algorithm. Consider a network class that originally has some kind of completeness property (e.g. complete graph, complete $k$-partite graph), for which an $O(n \log n)$ algorithm does exist. These complete classes have a threshold condition where up to the threshold point, the network may loose links and still, leader election could be done in $O(n \log n)$. It might be that due to the failure of links (occurring before the algorithm has started), the complete model or even the threshold condition does not hold. Hence, we must assume that the network is a general graph. Applying the algorithm for a general graph from the beginning, solves the problem, but will cost $O(n^2)$, in a complete or an almost complete networks. Checking of the threshold condition and then applying the right algorithm, also requires a great communication effort and will be very inefficient (in fact, it is as hard as leader election). We suggest an adaptive algorithm whose total behavior (the sequence of messages sent during the execution) on a specific case (complete graph, graph which satisfies the threshold condition or general graph) will be as if the special purpose algorithm for this case would have been applied from the beginning.

A different type of extension of the [KMZ] algorithm yields an algorithm for leader election in a complete network which up to $d$, ($d < \sqrt{n/2}$) processors may fail before the algorithm starts. A processor that fails, stops sending messages without the awareness of its neighbors. Our algorithm uses no more than $5n \log_2 k + 2dk$ messages which is better than the result of [KWZ] for this problem ($12 \log_2 k + 8dk$).

2. LEADER ELECTION IN A GENERAL NETWORK

2.1. The Basic Protocol Mechanism

We now present the basic mechanism used by the algorithm for electing a leader, which is essentially the leader election algorithm for a complete network as described in [KMZ]. Most of the relevant definitions, and descriptions are taken from there.
Each node in the network has a state, that is either KING or CITIZEN. Initially every node $i$ is a king (i.e. $\text{state}(i) = \text{KING}$), and - except for one - everyone will eventually become a citizen. A node $i$ with $\text{state}(i) = \text{KING}$ is called king$_i$. The algorithm starts by a WAKE message, received by any non-empty subset of nodes.

During the algorithm, each king is the root of a directed tree which is its kingdom. All the other nodes of this tree are citizens of this kingdom, and each node knows its father and sons. Each node $i$ also stores the identity $\text{king}(i)$ and the phase $\text{p}(i)$ of its king, which are updated during the execution of the algorithm. The status of node $i$ is defined by $\text{status}(i) = (\text{p}(i), \text{king}(i))$. We say that $\text{status}(i) < \text{status}(j)$ if either (a) $\text{p}(i) < \text{p}(j)$ or (b) $\text{p}(i) = \text{p}(j)$ and $\text{king}(i) < \text{king}(j)$. Before the algorithm starts $k(i) = \text{identity}(i)$ and $p(i) = -1$ for each node $i$. The following variables are also used: $\text{father}(i)$ denotes the edge connecting $i$ to its father, $\text{sons}(i)$ denotes the set of edges connecting $i$ to its sons, and $\text{unused}(i)$ denotes the set of untraversed edges of node $i$.

A king tries to increase its kingdom by sending messages towards other kings (possibly through their citizens), asking them to join, together with their kingdoms, its kingdom. A citizen, upon receiving a message originated by a king, delays it, ignores it, or transfers it to (or from) its king. When king$_i$ receives a message asking it to join the kingdom of king$_j$, it does so if $\text{status}(i) < \text{status}(j)$. Otherwise, the message is simply ignored, and the sending king will eventually become a citizen. The process of joining $j$'s kingdom consists of two stages: first king$_i$ sends a message to king$_j$ along the same path which transferred $j$'s message to $i$, telling it that it is willing to join its kingdom; during this stage the directions of the edges in this path are reversed. In the second stage, if $\text{p}(i) < \text{p}(j)$ then king$_j$ announces to its new citizens that it is their new king, or if $\text{p}(i) = \text{p}(j)$ then it first increases its phase by 1 and then sends an appropriate updating message towards all its citizens (new and old).

A node $i$ in the CITIZEN state has a unique father edge leaving it (which may be changed during the algorithm) and zero or more son edges. It may be in one out of two substates: CITIZEN(regular) and CITIZEN(waiting). It enters the waiting substate upon receiving an ASK message with status higher than its own status, and it returns to the regular substate upon getting a status which is higher or equal to the status of that ASK message. While in the waiting substate, $i$ remembers the status $(\text{p}(j), \text{king}(j))$ of the last ASK message it received, and also the edge $e'$ along which it received this message. Also, in this substate it receives messages only from its father, and delays the reception of other messages (e.g., by moving them to the end of a queue), until it enters the regular substate again (if at all). This basic protocol ends when king$_i$ know that all its neighbors are in its king-
dom. At this stage in a complete network, *king* \(i\) would announce as a leader; however, for a general graph some additional mechanism is required.

### 2.2. The Additional Protocol Mechanism for General Graphs

The idea of the additional protocol is moving the king in its kingdom tree in a The DFS moves consist of two basic steps: the forward move and the backward move. These moves are actually two special messages sent by the king after it has communicated with all its neighbors (i.e., each of these either becomes its son or acknowledges being its citizen). In the forward move the king moves itself toward one of its unsearched sons. The backward move is done only after all but one of the king's sons have been searched. In this case it moves itself toward the unsearched son (from which it first came to this node).

While moving, the king reverses the direction of the tree edge it has moved on, by doing so it makes its new place the root of the kingdom tree. The node \(i\) that the king has just left keeps in \(father(i)\) the direction to the current king and in \(DFS_{father}(i)\) it keeps the edge that the king has entered for the first time. The equality \(identity(i) = k(i)\) holds for node \(i\) only if \(i\) is the original place of the king. (Note that there are always two directed kingdom trees, with the same edges. The first that is rooted at the king's origin node on which the DFS search is being done and is called the DFS-tree, and the second that is directed to the current place of the king.)

Once a king has completed its search in a sub tree of its kingdom tree, it will not visit in this sub tree again. This is accomplished by keeping in \(searched(i)\) any edge that was traversed by the king. This set is the set of edges that meet node \(i\) and on which the king has left the node and return from there. This variable does not change when another king takes over the kingdom, so once the king has returned from an edge no king will go through it again. From this we can see that the elected king, in its DFS moves does not necessarily visit all the nodes in the network, since there are subtrees that were searched by other kings.

The king in its new node \(i\), continues to try to increase its kingdom by using the same basic protocol as before. In order to reduce message complexity the king does not send \(ASK\) message through edges that have been removed from \(unused(i)\) by other kings.

The search is completed when the king is back in its origin node and all its sons have been searched. Now the king announces itself as leader by propagating a \(LEADER\) message through the spanning tree.
We assume that links obey FIFO, however a small change in the algorithm would make it suitable for networks with non-FIFO links.

2.3. The Messages used by the Algorithm

1) \textit{ASK (status (i))} : this message is sent by \textit{king}_i through an unused edge in an attempt to increase its kingdom, and might be transferred onwards by citizens. Each \textit{ASK} message has a status, which is the \textit{status (p(i),k(i))} of the king that originated it (at the time it was originated).

2) \textit{ACCEPT (p(i))} : this message is sent by \textit{king}_i in return to an \textit{ASK} message from another king, telling it that it is willing to join its kingdom. (This message also might be transferred onwards by citizens.)

3) \textit{UPDATE (status (i))} : this message is sent by \textit{king}_i (after receiving an \textit{ACCEPT} message from another king) updating its new (and in some cases also its old) citizens of its identity and phase.

4) \textit{YOUR_CITIZEN} : this message is returned by a citizen as an answer to \textit{ASK} message originated by its own king.

5) \textit{MOVE} : this message is sent by a king to one of its sons in order to make a forward DFS move. The sender becomes a citizen and the receiver becomes a king.

6) \textit{RETURN} : this message is sent by a king to its original father in order to make backward DFS move.

7) \textit{LEADER} : this message is sent by the leader to all other nodes, announcing its leadership and terminating the algorithm.

2.4. The Algorithm for a King

We now give the formal description of the algorithms to be performed by node \textit{i} as long as it is a king. As mentioned before, this algorithm based on [KMZ] algorithm and include some additional control mechanism for forward and backward move of the king.

Before the algorithm starts every node \textit{i} has the following initial values. (Local variable \textit{x} at node \textit{i} is denote \textit{x(i)}.)

\begin{align*}
\text{state}(i) &= \text{KING}, \\
\text{status}(i) &= (p(i),k(i)) = (-1,\text{identity}(i)), \\
\text{unused}(i) &= \text{the set of all its adjacent edges}, \\
\text{sons}(i) &= \emptyset, \\
\text{DFS\_father}(i) &= \text{null}, \\
\text{searched}(i) &= \emptyset.
\end{align*}
(K0) Upon becoming a king, perform procedure check.

    Wait for message: upon receiving message m along an edge e, according to m's type, do the following:

(K1)  $m = WAKE$: If $p(i) = -1$ then increase your phase to zero and perform procedure check. (Given at the end of this subsection.)

(K2)  $m = ASK(status(j))$: If status(j) ≤ status(i) then ignore this message. Otherwise, send an ACCEPT($p(i)$) message along e, and change state to CITIZEN(waiting), with father(i) = e.

(K3)  $m = YOUR_CITIZEN$: Delete e from unused(i) and if $e = e'$, perform procedure check.

(K4)  $m = ACCEPT(p(j))$: Do the following:

    (1) Add e to sons(i).
    (2) Delete e from unused(i).
    (3) If $p(j) < p(i)$, send an UPDATE(status(i)) message along e.

    Otherwise (i.e., $p(j) = p(i)$) increase phase by one and send an UPDATE(status(i)) message along each edges in sons(i).

    (4) If $e = e'$ (where $e'$ is the edge from which the last ASK was received) perform procedure check.

Procedure check:

if unused(i) ≠ ∅ then
    choose an edge $e'$ from unused(i) and send an $ASK(p(i), k(i))$ message along $e'$,
else
    if sons(i) - searched(i) - {DFS_father(i)} ≠ ∅ then
        (1) Choose e ∈ sons(i) - searched(i) - {DFS_father(i)}.
        (2) Send MOVE along edge e.
        (3) sons(i) = sons(i) - {e}.
        (4) father(i) = e.
        (5) state(i) = CITIZEN(regular).
    else
        if $k(i) = identity(i)$ then
            send a LEADER message to all your sons and terminate.
        else
            (1) Send RETURN along edge DFS_father(i).
            (2) father(i) = DFS_father(i).
            (3) sons(i) = sons(i) - {DFS_father(i)}.
            (4) state(i) = CITIZEN(regular).

2.5. The Algorithm for a Citizen

Node i reacts to a message m it receives along an edge e according to m's type and its substate, as follows:

regular substate:
(CR1) \[ m = \text{ASK}(\text{status} \,(j)) \text{): If } e \neq \text{father} \,(i) \text{ then do the following:} \]

1) If \( k \,(i) = k \,(j) \text{ and } e \text{ is not a son edge, return YOUR\_CITIZEN along } e \).

2) If \( \text{status} \,(j) > \text{status} \,(i) \text{ and } k \,(i) \neq k \,(j) \), send \( m \) to its father and enter the waiting substate.

(CR2) \[ m = \text{UPDATE}(\text{status} \,(j)) \text{): If } e = \text{father} \,(i) \text{ and } p \,(j) > p \,(i) \text{ then update status to } \text{status} \,(j) \text{ and forward } m \text{ to all your sons.} \]

(CR3) \[ m = \text{ACCEPT}(p \,(j)) \text{): If } e \text{ is in unused} \,(i) \text{, then make } e \text{ your son edge, forward an UPDATE(\text{status} \,(i)) message along } e \text{ and remove } e \text{ from unused} \,(i). \]

(CR4) \[ m = \text{MOVE} \text{: Make } e \text{ your son, set } e \text{ as DFS\_father} \,(i) \text{ and change state to } \text{KING}. \]

(CR5) \[ m = \text{RETURN} \text{: Make } e \text{ your son, put } e \text{ into searched} \,(i) \text{ and change state to } \text{KING}. \]

(CR6) \[ m = \text{YOUR\_CITIZEN} \text{: Delete } e \text{ from unused} \,(i). \]

(CR7) \[ m = \text{LEADER} \text{: Send } m \text{ to all your sons and terminate the algorithm.} \]

waiting substate:

(Recall that: (1) \( (p \,(j),k \,(j)) \) is the status of the last ASK message received by \( i \), (2) \( e' \) is the edge along which this ASK message was received, and (3) in this substate \( i \) receives messages only from its father):

(CW1) \[ m = \text{UPDATE}(p \,(l),k \,(l)) \text{ and } e = \text{father} \,(i) \text{): If } p \,(l) > p \,(i) \text{ then } i \text{ does the following:} \]

1) Updates your status to \( (p \,(l),k \,(l)) \) and forward \( m \) to all your sons.

2) If \( k \,(l) = k \,(j) \text{ and } e' \text{ is not a tree edge, send a YOUR\_CITIZEN message along } e' \text{ and return to the regular substate.} \]

3) If \( (p \,(l),k \,(l)) \geq (p \,(j),k \,(j)) \) return to the regular substate.

(CW2) \[ m = \text{ACCEPT}(p \,(k)) \text{ and } e = \text{father} \,(i) \text{:} \]

1) Make \( e \) (which is your father) your son,

2) make \( e' \) your father

3) Forward \( m \) along \( e' \).

(CW3) \[ m = \text{MOVE} \text{ and } e = \text{father} \,(i) \text{:} \]

1) Make \( e \) your son and set \( e \) as DFS\_father \,(i)

2) Sends ACCEPT \((p \,(i))\) on edge \( e' \) and make \( e' \) your father.
(CW4)  \( m = \text{RETURN} \) and \( e = \text{father}(i) \):

1. Make \( e \) your son, put \( e \) into \text{searched}(i),
2. Sends \( \text{ACCEPT}(p(i)) \) on edge \( e' \) and make \( e' \) your father.

(CW5)  \( m = \text{LEADER} \) and \( e = \text{father}(i) \): Send \( m \) to all your sons and terminate the algorithm.

2.6. Definitions and Axioms

Here we shall give the formal model of the communication network which will be used in our correctness proof. This model is the same as used in [KMZ].

Let \( A \) be a distributed algorithm acting on a graph \( G = (V,E) \). An execution of \( A \) consists of events, each being either sending a message, receiving a message or doing some local computation. With each execution we can associate a sequence \( \text{SEND} = < \text{send}_1, \text{send}_2, \cdots, \text{send}_k > \) that includes all the events of the first type in their order of occurrence (if there are no such events then \( \text{SEND} \) is the empty sequence). In the case that two or more messages are sent at the same time, order them randomly (thus, a number of \( \text{SEND} \) sequences may correspond to the same execution). With each event \( \text{send}_i \) we identify the triple \((v(\text{send}_i), e(\text{send}_i), m_i)\), where \( v(\text{send}_i) \) is the node sending the message \( m_i \) and \( e(\text{send}_i) \) is the edge used by it. We assume that \( \text{send}_1 \) occurred at time \( 0 \), and \( \text{send}_i \) at time \( t_i \geq 0 \).

Let \( \text{SEND}(t) \) be the prefix of length \( t \) of the sequence \( \text{SEND} \), namely \( \text{SEND}(t) = < \text{send}_1, \cdots, \text{send}_t > \) (\( \text{SEND}(0) \) is the empty sequence). If \( t < t' \) then we say that \( \text{SEND}(t') \) is an extension of \( \text{SEND}(t) \), and we denote \( \text{SEND}(t) < \text{SEND}(t') \). \( \text{SEND} \) is called a completion of \( \text{SEND}(t) \). Note that a completion of a sequence is not necessarily unique.

2.7. Basic Properties of the Algorithm

The guide line of the following correctness proof of our algorithm is based on ideas taken from [KMZ].

The basic idea of the algorithm is building a spanning tree of the graph by repeatedly combining trees with one other. These trees will be viewed as directed trees, each considered as a kingdom, with the root being its unique king. An edge \( e \) which is \( \text{father}(i) \) for some node \( i \) is called a tree edge, (and is called a non-tree edge otherwise), and is considered to be directed into \( i \). The only way for a non-tree edge to become a tree edge, or for a tree edge to reverse its direction, is as a result of an \( \text{ACCEPT MOVE} \) or \( \text{RETURN} \) message sent along it: The sender makes this
edge its father (by executing K2 or CW2), and the receiver makes this edge its son (by executing K4, CR3 or CW2).

Part (1) of the following observation is quoted from [KMZ] and is applicable also here.

Observation 1: Let $e$ be an edge connecting $i$ and $j$, then the following holds:

1. A message that reverses the direction of a tree edge (i.e. ACCEPT MOVE and RETURN) will never be sent by $i$ on $e$, if another message of this type was already sent by $j$ on $e$ but has not yet been received by $i$.

2. $e$ is $j$'s son only if it is $i$'s father (the converse is not necessarily true).

The following lemma and its proof is quoted from [KMZ] and is applicable also here.

Lemma 2: An ASK message is originated by a king. The first edge, $e$, that carries an $ASK(p(i),k(i))$ message is a non-tree edge, the other edges are tree edges traversed against their directions.

Proof: An $ASK(p(i),k(i))$ message is originated by a node $i$ with $state=KING$ in the procedure check, and is forwarded from sons to fathers by (CR1).

Let $SEND = <send_1,send_2,\ldots>$ be a sequence corresponding to an execution of the algorithm, as defined in the previous section, and let $send_t = (v,e,ASK(p(i),k(i)))$. Then, since every node that receives an $ASK(status(i))$ message forwards at most one such message (see (CR1)), the edges that carry this message form an (undirected) path. We shall denote this path by $\pi(i,t)$. When no confusion occurs, we simply denote it by $\pi(i)$.

Lemma 3: Let $e$ be an edge connecting $i$ and $j$, such that $e$ is in $sons(i)$ then, $status(i) \neq status(j)$ only if an $UPDATE(status(i))$ message that was sent by $i$ on $e$ has not yet been received by $j$.

Proof: For contradiction, let $t$ be the first time it happens that $status(i) \neq status(j)$, but there is no $UPDATE$ message sent by $i$, yet not been received by $j$. This could happen if at time $t$:

1. A new son is added to $sons(i)$, or
2. The status in either $i$ or $j$ was changed.

In case (1), $sons(i)$ is increased only when one of the messages MOVE, RETURN or ACCEPT was received by $i$. In each of these events, except when an ACCEPT is received from a non-tree edge, the message that was sent by $j$, has only reversed the tree edge direction. That is, $e$, which was $j$'s son is added now to $i$ as a new son. Then, if $status(i) \neq status(j)$ when the message is received, it must have happened before $t$, a contradiction. If $e$ is a non-tree edge and an ACCEPT message was received by $i$, then $send_t = (i,e,UPDATE(status(i))$ (K4 and CR3) in
contradiction to the assumption that no such message was sent by $i$ to $j$.

In case (2), if the change of status was in $i$, it follows from K4, CR2 and CW1 that this change is immediately followed by sending $UPDATE$ to each of $i$'s so in contradiction to the assumption. Otherwise, the change of status must have been in $j$. Since $e$ is in $sons(i)$, it follows from observation 1 that $e = father(j)$, hence, $j$ must be a citizen. As a citizen, $j$ changes its status to $s$ only as a result of an $UPDATE(s)$ message coming from its father, $i$. When this $UPDATE(s)$ was sent, $status(i)$ was equal to $s$, but since we know that $status(i) \neq status(j)$, $status(j)$ must have been changed earlier, a contradiction.

Corollary 4: If $status(i) = s$ when $i$ sends $MOVE$ or $RETURN$ to $j$, then by the time this message is received by $j$, $status(j) = s$.

Proof: Since $MOVE$ or $RETURN$ messages are sent on a son edge, by Lemma 3, $status(i)$ must be equal to $status(j)$, unless an $UPDATE(status(i))$ message was sent earlier from $i$ to $j$. In this model we assume that links obey the FIFO discipline, hence if an $UPDATE(status(i))$ message was sent, it would have been received before the $MOVE/RETURN$, and would cause the statuses to be equal again.

Lemma 6: At any given phase during the algorithm, at most one $ASK$ message is sent by any citizen.

Proof: An $ASK$ message is sent by a citizen $i$ only in (2) of CR1. After sending an $ASK(p(i),k(i))$ message, a citizen enters the waiting substate. While in this substate, it cannot send another $ASK$ message, and it can exit this substate only by receiving $UPDATE(p',k')$, where $p' \geq p(i)$. Since a phase of a node never decreases, $i$ will never get a lower phase, and therefore, by CR1, it will never forward any other $ASK$ message at phase $p(i)$.

Lemma 7:

1. $\pi(i)$ is a simple path except possibly for the last edge $v-w$.

2. The $ASK$ message that was sent on $v-w$ will be ignored.

3. If an $ASK(p(i),k(i))$ message is received by some king $j$ or by a citizen $j$ in the regular substate from its father, then the part of $\pi(i)$ between $i$ and $j$ is simple.

4. Only one node on $\pi(i)$ can response to $ASK(p(i),k(i))$ by sending $ACCEPT$. 
Proof:

(1),(2): From lemma 2 it follows that \( \pi(i) \) is a path. If \( \pi(i) \) happens to be simple, then (1) trivially holds. Else, let's consider at the first time that the \( \text{ASK} \) reaches a node it has visited, say \( w \). If \( w \neq i \) then upon receiving the first \( \text{ASK}(p(i),k(i)) \) \( w \) enters the \text{waiting} substate and won't receive any other \( \text{ASK} \). It can leave this substate only by receiving an \( \text{UPDATE} \) with a bigger or equal status (\text{CW1}), so when the \( \text{ASK}(p(i),k(i)) \) will be received by \( w \) on the second time, it will be ignored (K2 or CR1). If \( w = i \) then when the \( \text{ASK}(p(i),k(i)) \) reaches \( i \), \( i \)'s status is at least \( (p(i),k(i)) \) so the message is ignored.

(3) In both cases the \( \text{ASK} \) message is not forwarded. If node \( j \) is on the end of a simple path \( \pi(i) \) we are done. Else, if \( j \) is on the end of a loop then from (1) it follows that it is the only loop hence the part of \( \pi(i) \) between \( i \) and \( j \) is simple.

(4) For a contradiction assume that there is more than one node on \( \pi(i) \) that has responded to \( \text{ASK}(p(i),k(i)) \). Assume that \( u \) is the node closest to \( i \) that has responded by sending \( \text{ACCEPT}(p(u)) \). There are two cases when an \( \text{ACCEPT} \) message is sent in response to \( \text{ASK} \): directly when \( u \) is a king or indirectly when \( u \) is a citizen in waiting. If \( u \) sends the \( \text{ACCEPT} \) message as a king the \( \text{ASK} \) message couldn't possibly be forwarded so we have a contradiction. If \( u \) is a citizen in the waiting substate and the \( \text{ACCEPT} \) message is a direct response to a \( \text{MOVE} \) message, then the \( \text{ASK} \) message was forwarded by \( u \) to its father \( v \) but since \( v \) received the \( \text{ASK} \) after it had sent the \( \text{MOVE} \) message, the \( \text{ASK} \) came from \( v \)'s father and by CR2 we can see it is ignored, in contradiction to the assumption. \( \square \)

The following lemma and its proof are quoted from [KMZ] and are applicable also here.

**Lemma 8:** When a node \( v \) sends an \( \text{UPDATE}(p,k) \) message to node \( u \) along edge \( e \), \( e \) is \( u \)'s father and \( \text{status}(v) = (p,k) \).

**Proof:** An \( \text{UPDATE} \) message is sent only to sons (K4, CR2, CR3 and CW1), and by Observation 1 the receiver must receive it from its father. \( \square \)

The following lemma is very similar to Lemma 5 of [KMZ] but its proof is different.

**Lemma 9:** Suppose that an \( \text{ACCEPT}(p(j)) \) message is sent in response to an \( \text{ASK}(p(i),k(i)) \) message by either king \( j \), or by a citizen \( j \) in the waiting substate upon receiving \( \text{MOVE} \) or \( \text{RETURN} \). Suppose that \( (p(j),k(j)) = (p',k') \). Let \( u \in \pi(i) \), \( u \neq i,j \). Then, after \( u \) forwards this \( \text{ASK} \) message, the following will hold:
(1) u's status cannot exceed \((p', k')\) and cannot change its father before it receives some ACCEPT message, and

(2) The first ACCEPT message \(u\) receives will be an ACCEPT\((p(j))\) which will come from its father.

Proof: (1): For a contradiction, assume that \(u \neq i\) is the node closest to \(j\) in \(n(i)\) whose status becomes \((p'', k'') > (p', k')\) before it has received any ACCEPT message or has changed its father after it forwarded the ASK\((p(i), k(i))\) message. If \(u\) has changed its father it could be by receiving either a MOVE or RETURN message. In response to this message \(u\) will send an ACCEPT. But from (4) in Lemma 7 we have that only one node in \(n(i)\) can originate an ACCEPT in response to ASK\((p(i), k(i))\) which contradicts the assumption that \(u \neq j\).

If \(u\)'s status has become \((p'', k'')\), it could happened only if \(u\) received an UPDATE\((p'', k'')\) message from its father, say \(v\). Since ACCEPT is the only message from fathers to sons, that may change fathers, \(v\) was also \(u\)'s father when \(u\) received and forwarded the ASK\((p(i), k(i))\) message, and hence \(v\) is in \(n(i)\), between \(u\) and \(j\), and hence closer than \(u\) to \(j\). Also by Lemma 8, \(v\)'s status was raised to \((p'', k'')\) before \(u\)'s status was. By the assumption on \(u\), \(v\)'s status became \((p'', k'')\) after it had sent the ACCEPT\((p(j))\) message. This could happened only by receiving an UPDATE message from its father. But while sending the ACCEPT message, \(v\) made \(u\) its father. This means that \(u\)'s status was raised to \((p'', k'')\) before \(v\)'s status - a contradiction.

(2): \(u\) entered the waiting substate upon receiving the ASK\((p(i), k(i))\) message. By (1), \(u\) did not exit the waiting substate unless it received an ACCEPT message. If \(u\) did not receive the ACCEPT\((p(j))\) message first then there exists a node in \(n(i)\), say \(v\); which did not receive this message, but its father, \(w\) did receive it first. Let \(e\) be the node connecting \(v\) and \(w\). Since \(w\) as receiving the ACCEPT\((p(j))\) sends it to its son, \(v\) must receive it from its father. Now the only possibility is that some older ACCEPT message, ACCEPT\((p(k))\), was sent from \(w\) to \(v\) in the opposite direction to some old ASK. ACCEPT\((p(k))\), when sent, made \(e\) the father of \(w\). From Observation 1 it follows that MOVE or RETURN couldn't be received by \(w\) before ACCEPT\((p(k))\) is received by \(v\), so \(e\) is \(w\)'s father when the ASK\((p(i), k(i))\) is sent in contradiction to the assumption that ASK\((p(i), k(i))\) is sent from \(v\) to \(w\).

By Lemma 9, and by CW2, we get the following two corollaries:

Corollary 10: An ACCEPT message that is sent along an edge \(e\) was originated in response to the last ASK sent on \(e\) and is sent in the direction which is opposite to that of the ASK message.

Corollary 11: Suppose an ACCEPT message is originated by \(j\) in response to ASK\((p(i), k(i))\). Then the ACCEPT\((p(j))\) will be sent from \(j\) to \(i\) along \(n(i)\).
Corollary 10 and 11 are quoted from [KMZ] and are applicable also here.

Lemma 12: The number of times ACCEPT messages are sent in any execution of the algorithm is finite.

Proof: This follows from Corollary 11 and the fact that after an ACCEPT message is originated the number of kings is decreased by one.

With each sequence SEND(i) that corresponds to a partial execution of the algorithm, as defined in the previous section, we associate a directed graph F(SEND(i))=F(t) that contains all the edges that carried an ACCEPT message. The direction of each edge is opposite to the direction of the last ACCEPT, MOVE or RETURN message it carried.

Theorem 13

Note: This theorem in general is very similar to Theorem 5 from [KMZ]: Parts (1), (2), (3) and (4) are quoted from there, part (5) is modified and part (6) is new. However, the proof of this theorem is quiet different since the model is weaker (general vs. complete).

The Theorem:

(1) F(t) is a directed forest.

(2) If F(t) ≠ F(t-1), then F(t) is obtained from F(t-1) by reversing or adding one edge, e, that is adjacent to a root of one of the trees in F(t-1).

(3) Assume that send, is an ACCEPT(p(j)) message, originated by node j and sent on a non-tree edge, e, from node u to node v. Then in F(t-1) u and v belong to different trees and status(v) > status(j) ≥ status(u).

(4) Every king is a root and it has the maximal status in its tree.

(5) Let T be a directed tree in F(t) that does not contain a king, and let u be the root of T. Then for some t' < t, send, is one of ACCEPT, MOVE or RETURN that was sent to u, but not yet forwarded by it, and the originator of this message (i.e., the node that sent it by executing (K2) in case of ACCEPT and the sender in case of MOVE or RETURN) has the maximal status in T.

(6) Suppose that an ACCEPT, MOVE or RETURN was sent from v to u over an edge e at time t' ≤ t and has not yet been received by u. Suppose also that e is a father edge on both v and u. Then u is a root in F(t).
Proof: By induction on $I$. For $I = 1$ the theorem is true, as $F(0)$ consists of isolated nodes which are all kings, and $F(1) = F(0)$ since $send_1$ is an $ASK$ message. Assume the theorem holds for $I < t$, and prove it for $I$.

(6) If $t = t'$ then either $v$ was a king at $t - 1$ or it has received $ACCEPT, MOVE$ or $RETURN$ on $t$. If it was a king at $t - 1$ then by using (4) of the induction hypothesis $v$ was a root at $t - 1$. From this and from the fact that $e$ is already a tree edge, it follows that $e$ has only reversed its direction and $u$ has become a root.

If $v$ was not a king at $t - 1$ then it must have received one of $ACCEPT, MOVE$ or $RETURN$ messages at time $t$ on a tree edge. From (6) of the induction hypothesis it follows that an $ACCEPT, MOVE$ or $RETURN$ message was on its way toward $v$ at time $t - 1$ and hence $v$ was a root at $t - 1$ and the message sent at $t$ made $u$ a root.

If $t < t'$ then at time $t - 1$ there was an $ACCEPT, MOVE$ or $RETURN$ message on its way to $u$ so by induction on (6) $u$ was a root. Since $ACCEPT, MOVE$ or $RETURN$ messages are received from the father edge, $u$ is a citizen and hence it cannot change the direction of edges that going out of it before receiving the $ACCEPT, MOVE$ or $RETURN$ that is coming. Thus $u$ remains a root at $t$.

(2) If $F(t) = F(t - 1)$ then (2) trivially holds, so assume that $F(t) \neq F(t - 1)$, which means that $send_t$ is either (a) $ACCEPT(p(j))$ or (b) $MOVE/RETURN$ message sent by a node $u$. Using (1) of the induction hypothesis it remains to prove that $u$ was a root in $F(t - 1)$. In the first case ($m(send_t) = ACCEPT(p(j))$, if $u = j$ - the originator of this message - then the claim holds by the induction hypothesis on (4). If $j$ was a king it holds by using (4). If $j$ originated the $ACCEPT$ as a citizen then it had received a $MOVE$ message and by (6) it follows that it was a root in $F(t - 1)$. Otherwise, by Lemma 9, before sending that message, $u$ received it from its father, $u'$, which, by induction, was then a root. By the definition of $F(t)$ and by (6) of the induction hypothesis, $u'$ became a root at the time $u'$ sent this message. In the second case ($send_t$ is either $MOVE$ or $RETURN$), since $MOVE/RETURN$ is sent from a king, $u$ was a king at $t - 1$. By (4) of the induction hypothesis $u$ is a root in $F(t - 1)$.

(3) Assume that $send_t$ is an $ACCEPT(p(j))$ message that was originated by $j$ when its status was $(p' k')$. This message is a response to an $ASK(p^* k^*)$ message originated by some node $i$, with $(p^* k^*) > (p' k')$. By Corollary 10, $e$ must belong to $\pi(i)$. By the fact that $e$ is a non-tree edge, $i = v$. By Lemma 9, $status(u)$ was not larger than $(p' k')$ in $F(t - 1)$, hence also in $F(t)$ (recall that $send_t$ is an $ACCEPT$ message and is not associated with a change of status). Since $(p^* k^*) \leq status(v)$ (status never decreases), we have that

$$status(u) \leq (p' k') < (p^* k^*) \leq status(v).$$
By the induction hypothesis on (5), no node in the tree containing \( u \) in \( F(t-1) \) has a status larger than \( (p', k') \), hence \( v \) is not in this tree.

(1) If \( F(t) = F(t-1) \) then (1) clearly holds. Otherwise, if \( F(t) \neq F(t-1) \), by (2) and the definition of \( F(t) \), \( send_t \) is one of \( ACCEPT, MOVE \) or \( RETURN \) sent along an edge \( e \) adjacent to a root. If \( e \) is a tree edge in \( F(t-1) \), then, by (2), its direction was reversed and \( F(t) \) is clearly a directed forest. Otherwise, by (3), \( e \) connects two disjoint trees in \( F(t-1) \), and by (2) the graph obtained by this connection of these two trees is a directed tree.

(4) If a node \( i \) is a king in \( F(t) \), then either (1) \( i \) was a king from \( t = 0 \) or (2) \( i \) became a king by receiving a \( MOVE \) or \( RETURN \) message. In the first case, \( SEND(t) \) does not contain any \( ACCEPT, MOVE \) or \( RETURN \) message sent by \( i \); hence in \( F(t) \) there is no edge entering \( i \), and therefore \( i \) is a root of a tree in \( F(t) \). In the second case, \( MOVE/RETURN \) message was sent on some \( t_0 \) over edge \( e \) to \( i \). By the definition of \( F \), \( MOVE/RETURN \) reverses the direction of \( e \) from entering into \( i \), to exiting out from \( i \). By (1), it follows that \( e \) was the only edge entering \( i \) hence, in \( F(t_0) \) no edge enters \( i \). Since it was assumed that from the moment it received \( MOVE/RETURN \), \( i \) remains a king forever, <send\(_t_0\),...,send\(_t_1\)> does not contain any \( ACCEPT, MOVE \) or \( RETURN \). Hence there is no edge entering \( i \) in \( F(t) \) and therefore \( i \) is a root in \( F(t) \).

Suppose that the maximal status, \( (p, k) \) in a tree \( T \) in \( F(t) \) is greater than the status of the king in \( T \). Let \( u \) be a node closest to the root of \( T \) with status \( (p, k) \). By Lemma 8, \( u \)'s father must have the same (or higher) status, a contradiction.

(5) Let \( \hat{T} \) be the tree that contained \( u \) in \( F(t-1) \) (note that \( \hat{T} \subseteq T \)). Then if \( \hat{T} \) contained a king, that king had a maximal status in \( \hat{T} \) and \( send_t \) is an \( ACCEPT, MOVE \) or \( RETURN \) message sent by this king to \( u \), and by (3) the claim follows.

If \( \hat{T} \) does not contain a king, then by induction, the root of \( \hat{T} \hat{u} \) satisfies the claim. If \( \hat{u} \neq u \), then \( send_t \) is an \( ACCEPT, MOVE \) or \( RETURN \) message sent by \( \hat{u} \) to \( u \), and the claim holds. If \( \hat{u}u \), (i.e. \( send_t \) is not one of \( ACCEPT, MOVE \) or \( RETURN \)) then it suffices to identity of a node in \( T \) equals the maximum status of a node in \( \hat{T} \) if \( V(\hat{T}) = V(T) \) then this is trivial, since the only way for some node with a maximal identity in \( \hat{T} \) to increase its phase is by executing (K4) as a king, and \( \hat{T} \) contains no king. Otherwise, \( send_t \) is an \( ACCEPT \) message sent by a node \( w \) in \( \hat{T} \neq \hat{T} \) to a node \( v \) in \( \hat{T} \). Let \( j \) be the originator of this \( ACCEPT \) message. Then, by the assumption that (5) and (3) hold for \( t-1 \), \( j \) has the maximal status in \( \hat{T} \), and \( status(j) < status(v) \). This implies that the maximum identity of a node in \( \hat{T} \) is smaller than the maximum identity of a node in \( \hat{T} \) and the claim holds.
Let $T$ be a tree in $F(t)$ that contains a node $v$ which is either a king or has sent $MOVE$ or $RETURN$ message that was not yet been received. And let $u$ be the node that satisfies $id(u) = k(v)$ (the node from which the king has started its moves). We shall define $\hat{T}$ to be a directed tree whose its underlying graph is identical to $T$'s underlying graph, $u$ is its root and its edges are directed out of $u$.

Corollary 13:
It is impossible to have two kings, two messages from $ACCEPT, MOVE$ or $RETURN$ or an $ACCEPT, MOVE$ or $RETURN$ message together with a king, at the same tree and on the same time.

Lemma 14: IF a $RETURN$ message is sent over an edge then a $MOVE$ message was sent over it in the opposite direction.

Proof: $RETURN$ is sent on the edge $DFS\_father(i) = e$ (see procedure check), hence $e$ is the edge from which the last $MOVE$ message was received. (CR4 and CW3).

From the above lemma it follows

Corollary 15: Let $u$ be the root of the DFS tree $\hat{T}$ Then a node $v, v \neq u$, becomes a king with $k(v) = id(u)$, for the first time only by receiving of a $MOVE$ message.

Lemma 16: Let $\hat{T}$ be a DFS tree and $u$ its root and $send_i = <m,v,e>, e \in \hat{T}$ then:

1. If $m = MOVE$ then it is sent in the direction of $e$ in $\hat{T}$
2. If $m = RETURN$ then it is sent in opposite to the direction of $e$ in $\hat{T}$

Proof:

(1) For contradiction, let $t$ be the first time that a $MOVE$ is sent from a node, $v$ on its father edge $e$ in $\hat{T}$ Since $v$ is a son in $\hat{T} \ v \neq u$. By Corollary 15 and the fact that $v$ is a king when it sends the $MOVE$, $v$ must have received some $MOVE$ message when $k(v)$ was equal to $id(u)$. From CR4 and CW3 $DFS\_father(v)$ is set to be the edge from which the last $MOVE$ was received. The edge $DFS\_father(v)$ must be a son edge of $v$ in $\hat{T}$ since $MOVE$ is not sent over $DFS\_father(v)$ and $e$ is a father edge of $v$ in $\hat{T}$ Receiving a $MOVE$ that was sent before $t$ by a son of $v$ contradicts the assumption that $send_i$ is the first $MOVE$ that was sent from son to its father in $\hat{T}$

(2) It follows immediately from (1) and Lemma 14.
Lemma 17: Let $\hat{T}$ and $u$ be as in the previous lemma, then any node $v$ in $\hat{T}$ sends at most one MOVE message to each of his sons in $\hat{T}$ while $\text{king}(v) = \text{id}(u)$.

Proof: For contradiction, let $send_i = \langle MOVE, v, e \rangle$ be the first MOVE that is sent for the second time. From (1) of Theorem 13 and (1) of the Lemma 16, $v$ after sending its first MOVE, in order to become a king again, must have received a RETURN message from $e$. Upon receiving RETURN, $e$ is added to $\text{searched}(v)$ and since MOVE is sent to edges not in $\text{searched}(v)$ another MOVE message couldn't possibly be sent on $e$.

From the above lemma it follows that:

Corollary 18: The number of MOVE messages sent in any execution of the algorithm is finite.

Lemma 19: Let $u$ be the root of the DFS tree $\hat{T}$ then any node $v$ in $\hat{T}$ sent at most one RETURN message its father in $\hat{T}$ while $\text{king}(v) = \text{id}(u)$.

Proof: For contradiction, let $send_i = \langle RETURN, v, e \rangle$ be the first RETURN that is sent for the second time. From (1) of Theorem 13 and (2) Lemma 16, $v$ after sending its first RETURN, in order to become a king again, must receive a MOVE message from $e$ again. But by Lemma 16 it is impossible for MOVE to be sent twice to $v$.

From Lemma 19 it we obtain the following:

Corollary 20: The number of RETURN messages sent in any execution of the algorithm is finite.

From Lemma 12, Corollaries 18 and 20 and Theorem 13, we have the following:

Corollary 21: There is a $t_0$ such that

1) For all $t \geq t_0$, $F(t) = F(t_0)$, and

2) Every root in $F(t_0)$ is a king.

Proof: Let $t_0$ be the time that the last message of the kind MOVE, RETURN or ACCEPT sent by the algorithm arrives (such $t_0$ exists by Lemma 12 and Corollaries 18 and 20). (1) follows from the fact that if $F(t) \neq F(t-1)$ then $send_i$ must be one of MOVE, RETURN or ACCEPT. (2) follows from the fact that if there exist a root in $F(t_0)$ that is not a king, then by (5) of Theorem 13 another ACCEPT, MOVE or RETURN message will be sent, a contradiction.
2.8. Correctness of the Algorithm

We prove in this section the correctness of the algorithm. The property which implies this correctness is given in the next theorem.

**Theorem 22:** In any execution of the algorithm, eventually there is exactly one node with state = KING.

**Proof:** For contradiction, consider an execution for which the theorem is false. Let \( t_0 \) be as in Corollary 21. Then the following must hold:

(**): \( F(t_0) \) contains \( s > 1 \) nodes that remain kings forever. These nodes are the roots of disjoint trees \( T_1, \ldots, T_s \) such that \( \bigcup_{i=1}^s V(T_i) = V \).

Let \( \hat{T} \) be a DFS tree. Denote \( king_i \) the root of \( T_i \) and \( king\_origin_i \) the root of \( \hat{T}_i \) \((king\_origin_i = k(king_i) \) holds for every \( T_i \)). The proof proceeds by a few lemmas.

**Lemma 23:** For \( i = 1, \ldots, s \), each \( king_i \) has its phase eventually equal to some constant \( phase_i \) forever.

**Proof:** It suffices to show that the number of phase-increases by kings is bounded. A king may increase its phase in (K1) or in (K4), after receiving a WAKE or an ACCEPT message, respectively. The number of WAKE messages is clearly bounded by \( n \). By Lemma 12, the number of distinct ACCEPT messages is also bounded. \(\square\)

Let \( s_i \) denote the final status of \( king_i \) (such a status exists by the above lemma) and suppose that \( s_i < s_j \) for \( i < j \).

**Lemma 24:** Under the assumption (**), eventually every node in the tree \( T_i \) will have its status equal to \( s_i \).

**Proof:** For contradiction, assume that some node \( j \) in the tree \( T_i \) has a different status \((p(j), k(j))\) forever. Assume that \( j \) is the node closest to the root having this property, let \( x \) be \( j \)'s father and \( e \) be the edge connecting \( x \) and \( j \). Then by (4) of Theorem 13, \( status(x) = s_i > (p(j), k(j)) \). When \( x \) got its status \( s_i \), \( e \) was either in \( sons(x) \) or it was not.

(a) If \( e \) was in \( sons(x) \), then \( x \) as a response to the raise of its status to \( s_i \) would have sent an UPDATE \( s_i \) message to \( \hat{j} \) (CR2 and K4), a contradiction.

(b) Otherwise, \( e \) is not in \( sons(x) \) at the time that the status was raised, so if \( e \) is \( x \)'s father edge, \( x \) raised its status to \( s_i \) by UPDATE \( s_i \) message that was sent by \( \hat{j} \) - a contradiction. But if \( e \) is not a tree edge, since by (3) of Theorem 13, it follows that from this time and on \( status(\hat{\hat{j}}) < status(x) \), holds forever, \( e \) can become a tree edge only by an ACCEPT message sent by \( \hat{j} \). At the moment \( x \) receives this ACCEPT, \( x \) will send UPDATE \( s_i \) to \( \hat{j} \), a
Lemma 25: Eventually, there is no citizen in the waiting substate.

Proof: For contradiction, let \( u \) be a citizen in \( T_j \) which is in the waiting substate forever, but \( u \)'s father (in \( T_j \)), say \( v \), is not. Assume that \( u \) entered this substate by receiving an \( \text{ASK}(p,i) \) message, which it forwarded to a node \( v' \), that was \( u \)'s father at that time. After receiving this \( \text{ASK} \) message, \( u \) did not receive any \( \text{UPDATE}(\text{status}(l)) \) message with \( \text{status}(l) \geq (p,i) \), as otherwise, by (3) of CW1, it would have returned to the regular substate.

Moreover, after receiving this \( \text{ASK} \) message, \( u \) also did not receive any \( \text{ACCEPT} \), \( \text{MOVE} \) or \( \text{RETURN} \) message: by Corollary 10, the only \( \text{ACCEPT} \) message it could receive was in response to that \( \text{ASK}(p,i) \) message, and any \( \text{MOVE} \) or \( \text{RETURN} \) would have cause it to send \( \text{ACCEPT} \) in response to \( \text{ASK}(p,i) \), in any case. So if such an \( \text{ACCEPT} \) message was sent, it would re-traverse the path \( n(i) \) and reach \( i \), the originator of the \( \text{ASK} \) message. It is easy to see that upon receiving this message, \( i \) would have initiated \( \text{UPDATE} \) messages with status greater than or equal to \( (p,i) \) that eventually reaches \( u \) and changes its state to regular. Therefore we may assume that \( u \), after sending \( \text{ASK}(p,i) \), has not received any \( \text{ACCEPT} \), \( \text{MOVE} \) or \( \text{RETURN} \) message. hence \( v' \) must be equal to \( v \). Moreover by lemma 3 we may assume that \( \text{status}(v) = \text{status}(u) \). It is easy to see that \( v \) cannot be a king since it would react to that \( \text{ASK} \). Hence \( v \) is in \( \text{CITIZEN}(\text{waiting}) \). But this is also not possible since \( v \) couldn't ignore that \( \text{ASK} \) either forward it or be in status greater then or equal to \( (p,i) \), a contradiction.

It follows from the above that \( v' = v \), and hence \( v' \) received this \( \text{ASK} \) message from \( u \). \( v' \) did not forward this \( \text{ASK} \) message, in spite of the fact that it is in the regular substate. This could happened only if \( \text{status}(v') \geq (p,i) \). But eventually \( \text{status}(v') = \text{status}(u) \), and hence \( \text{status}(u) \) is also greater than \( (p,i) \), which means that it had to leave the waiting substate, a contradiction.

Lemma 26: Suppose \( \text{king}_s \) eventually sends an \( \text{ASK} \) message to a node \( u \) which is in \( T_j \) for \( j < s \). Then \( u \) will eventually be in \( T_s \).

Proof: By Lemma 25, \( u \) will eventually exit the waiting substate. Since \( s_s \) is the maximum status in the network, this can happen only by having \( u \) receive an \( \text{UPDATE}(s_s) \) message (see (1) of CW1), which implies the lemma.

Lemma 27: Let \( \hat{T} \) be a DFS tree. If \( v \) sends \( \text{RETURN} \) on \( e \), then every node \( i \) in the subtree of \( \hat{T} \) rooted at \( v \) has also sent \( \text{RETURN} \).

Proof: For contradiction assume that some node in the subtree of \( v \) in \( \hat{T} \) hasn't sent \( \text{RETURN} \). Let \( u \) be the node
closest to \( v \) having this property and let \( w \) be its father in \( \hat{T} \) and \( e' \) the edge connecting \( u \) and \( w \). Since \( e' \) can be added to \( \text{searched}(w) \) only by receiving a \( \text{RETURN} \) message from it by \( w \), \( e' \) is not in \( \text{searched}(w) \) (see CR5 and CW4). \( w \) sends \( \text{RETURN} \) only if \( \text{sons}(w) - \text{searched}(w) - \text{DFS\_father}(w) = \emptyset \) (see procedure check), Since \( e' \) is in \( \text{sons}(w) \) and is not a \( \text{DFS\_father}(w) \) edge, it follows from Lemma 14 that \( w \) won't send \( \text{RETURN} \), a contradiction.

From the above lemma and by observation of procedure check follows:

**Corollary 28:** If \( v \) has sent a \( \text{RETURN} \) message then \( \text{unused}(i) = \emptyset \) in any node \( i \) in the subtree of \( v \) in \( \hat{T} \).

**Lemma 29:** Under the assumption (**), eventually \( \text{king}_v = \text{king\_origin}_v \).

**Proof:** For contradiction, let \( v = \text{king}_v \neq \text{king\_origin}_v \), the node which remains king forever. Since \( v \neq k(v) \), the only way for \( v \) to remain a king forever is by sending an unreplied \( \text{ASK} \) message. Consider this last \( \text{ASK} \) message sent by \( \text{king}_v \) and let \( u \) be the node that received this message. By Lemma 23 \( u \) will be eventually in \( T_q \). If upon receiving this message \( \text{status}(u) = \text{status}(\text{king}_v) \) then it will immediately send a \( \text{YOUR\_CITIZEN} \) message to \( \text{king}_v \) (CR1), a contradiction. Otherwise, if \( u \) is a citizen, by Lemma 25, \( u \) must be eventually in the regular substate, and it will forward that \( \text{ASK} \) message to its father and will enter the \( \text{waiting} \) substate (CR1). \( u \) can exit the \( \text{waiting} \) substate only by receiving an \( \text{UPDATE}(i) \) message (CW1). By Lemma 25, \( u \) will eventually receive this message, and will then send a \( \text{YOUR\_CITIZEN} \) message to \( \text{king}_v \), a contradiction. If \( u \) is a king it will respond with \( \text{ACCEPT} \), a contradiction.

**Lemma 30:** Eventually:

1. \( \text{unused}(\text{king}_v) = \emptyset \)
2. \( \text{searched}(\text{king}_v) = \text{sons}(\text{king}_v) \)

holds forever.

**Proof:**

1. By using the same arguments as in Lemma 29 we have that no \( \text{ASK} \) message of \( \text{king}_v \) remains unreplied forever. From K3, K4, CR3, CR6, we get that after an \( \text{ASK} \) message was replied, the king deletes an edge from \( \text{unused}(i) \) and initiates another \( \text{ASK} \), since \( \text{king}_v \) remains king forever and eventually every \( \text{ASK} \) is replied, this process must end when \( \text{unused}(\text{king}_v) \) becomes empty.
(2) From (1) and from the fact that procedure check is executed immediately after a node becomes a king or after the last \textit{ASK} message sent by this king is replied, the else part of the main if statement must be executed when \textit{unused}(i) becomes empty. Since after \( t_0 \) no \textit{MOVE} messages is sent, sons\((i) - \text{searched}\((i) - \{\text{DFS}\_\text{father}\((i)\})\) must then be empty. From Lemma 29 (\textit{king}, is \textit{king}\_\text{origin},) and Lemma 16 (\textit{king}\_\text{origin}, for any \( T_i \) never receives \textit{MOVE}) it follows that \text{DFS}\_\text{father}\((\textit{king},)\) was never changed thereby it is \textit{null} and since \textit{null} is not an edge \textit{sons}(\textit{king},)=\text{searched}(\textit{king},) holds.

**Lemma 31:** \textit{unused}(i)=\(\phi\) for every node \( i \) in \( T_s \).

**Proof:** From (2) of Lemma 30 and from CW2 we derived that each son of \textit{king}, has sent a \textit{RETURN} message. By this and by corollary 11.5 we get that \textit{unused}(i)=\(\phi\) for every node \( i \neq \textit{king},.\) From (1) of the Lemma 30 we get that \textit{unused}(\textit{king},)=\(\phi\).

Initially \textit{unused}(i) contains all the edges adjacent to node \( i \) and it decreases only upon receiving \textit{ACCEPT} and \textit{YOUR\_\text{CITIZEN}} as a king or sending \textit{YOUR\_\text{CITIZEN}}. In both cases, both ends of the edge removed from \textit{unused}(i) are in the same kingdom. By this and by lemma 30 we get a contradiction to the assumption that \( s > 1 \).

This complete the proof of the theorem.

**Corollary 32:** The unique remaining king, eventually announces its leadership to all other nodes, and then the algorithm terminates.

**Proof:** It can be shown (in a similar way) that Lemmas 29, 30 and 31 hold also for \( s = 1 \). From these lemmas it follows that the king eventually is in its origin node after exhausting all its unused edges and having received \textit{RETURN} from all of its sons, which means that all the nodes in its kingdom have also exhausted their unused edges. After the above happens, the "ELSE" part of the secondary "IF" statement in procedure check must be executed. Since \( k(i)=id(i)\) holds, \( i \) will send the \textit{LEADER} message to all its sons which will then broadcast this message in the tree to all other nodes (CR4 and CW3). Thus this message will reach each node, which will then terminate the algorithm.

### 2.9. Message Complexity

In this section we will give a complexity analysis of the algorithm.

**Theorem 33:** If \( k \) nodes start the algorithm then the number of messages used by the algorithm is bounded by

\[
21E + 4n \log_k + O(n) .
\]
Lemma 34: If $k$ nodes start the algorithm and node $i$ is announced as leader then when the algorithm terminates $p(i) \leq \lfloor \log_2 k \rfloor$ holds.

Lemma 35: Exactly one RETURN message is sent on any edge of the final spanning tree.

Proof of the lemmas is omitted.

Proof of the theorem: An upper bound for number of messages of each kind:

1. **LEADER**: exactly $n - 1$ messages.

2. **ASK**: Consider first ASK messages send by kings. Such an ASK message can be either replied (by ACCEPT or by YOUR_CITIZEN) or remain unreplied. Since a single edge can carry no more the one replied ASK, we have at most $|E|$ ASK messages of this kind. A king that sends an unreplied ASK wont be sending any more messages and eventually it will stop being a king, hence, no more then $k - 1$ unreplied ASK messages are sent. So we have a total bound of $|E| + k - 1$ ASK messages sent by a king. Since a citizen transfers at most one ASK message per phase, the total number of these is bounded by $n \log_2 k$.

3. **ACCEPT**: the total number of such messages sent by a king is $k - 1$. At a given phase a citizen send at most one ACCEPT, hence the total number of ACCEPT messages is bound by $n \log_2 k$.

4. **YOUR_CITIZEN**: since each edge of the graph can be rejected as a reply to an ASK message sent by a king, the total number of such messages is bounded by $|E|$.

5. **UPDATE**: Each citizen receives at most one such message per phase, hence the total number is bounded by $n \log_2 k$.

6. **MOVE**: Each citizen receives at most one such message per phase, hence the total number is bounded by $n \log_2 k$.

7. **RETURN**: By the Lemma 35 exactly $n - 1$ such messages are sent during the execution of the algorithm.

Therefore, the total count of messages sent by the algorithm is no more than $2|E| + 4n \log_2 k + O(n)$.

3. GENERALIZATION

In the previous section we showed how a king, by searching its kingdom spanning tree, can cover all the graph. There are however classes of graphs where it is sufficient for the king to search only over a subset of nodes,
in order to guarantee communication with all the rest.

In particular, in the cases where the size of subset needed to be searched is a constant, we can easily devise a leader election algorithm that requires $O(n \log k)$ messages. The modifications needed to be made in the original algorithm for general network are basically in the condition for a forward move (search condition) and in the set of edges from which the choice is made. In the algorithm for a general graph it was:

$$\text{if there are unsearched sons then choose one of those.}$$

In (3.1)-(3.4) we will see a few examples that have been discussed in [KMZ] and in [KKM]:

3.1. Complete Networks: This is a degenerated case were the size of the king’s search is one. The search condition here is false. And since no MOVE or RETURN messages are sent, we have exactly the algorithm of [KMZ].

3.2. Complete k-Partite Graphs: In a $k$-partite graph the nodes are partitioned into $k$ disjoint subsets. Each node in any subset $U$ has links to all the other nodes but to those in $U$. In order for a king to dominate the graph it needs to visit in two connected nodes. The searched condition here is: if $k(i) = \text{identity}(i)$ and $|\text{searched}| = 0$ then choose one your sons.

3.3. Graphs where any node with any $k$ neighbors sees the others: The search here is of size $k+1$. The search condition is:

$$\text{if } k(i) = \text{identity}(i) \text{ and } |\text{searched}| < k \text{ then choose from unused adjacent edges.}$$

Here we see that the search does not necessarily follow the original edges of the spanning tree. In king’s forward and backward moving, to a node which is not its son, the used link is added to the spanning tree and the previous father of this son is disconnected, and the directed tree shape is preserved.

3.4. Graphs of Radius 1: (Where at least one node is connected to all other nodes.) The king gets the degree of its neighbors, (could be transferred by YOUR_CITIZEN and ACCEPT messages, as an additional parameter) and moves to the one with the maximum degree, unless he has the maximum degree.

3.5. An Adaptive Algorithm for Unreliable Complete Network

Consider a complete network, where some of the links may become faulty before the algorithm starts. Any such failure is noticed by the processors at both ends of the failed link. The threshold point that leader election could still be done with $O(n \log n)$ instead of $O(|E| + n \log n)$ message complexity is that at least one node in the
network remains connected to all the others (graph of radius 1). Since we don't know a priori weather one node is fully connected, the algorithm for graph of radius 1 cannot to be applied here directly. However, since \( n \) is assumed to be known (a processor knows its degree before any link becomes disconnected), instead of applying the general graph algorithm (which would require \( O(n^2) \) messages), we will devise the following king search:

\[
\text{if } k(i) = \text{identity}(i) \text{ and } \text{degree}(i) < n-1 \text{ then}
\]

\[
\text{if you have a neighbor } j \text{ with } \text{degree}(j) = n-1 \text{ then choose } j \text{'s edge to move forward}
\]

\[
\text{else use the search condition as in the general algorithm.}
\]

If the network happens to be complete, each king detects that it has \( n-1 \) connected links and does not make any DFS move. (The behavior of the algorithm will be as in (3.1).) If it is not the case but at least one node remains fully connected, some of the kings will make a single DFS forward move towards nodes that are connected to all the others. (The behavior will be as in (3.4).) If every node in the network has lost at least one edge, the king will perform the DFS search and the behavior will be the same as the general graph algorithm.

The advantage of this approach is that the algorithm adapts itself to the actual case within its regular execution. The work done until the expected topology is found not to be true, would have been done anyhow.

The same idea can be applied to some other classes of graphs which originally satisfy some completeness property and where leader election requires \( O(n \log k) \) messages. For example, consider a \( k \)-partite graph originally complete. If at least two nodes in two different partitions remain fully connected, leader election can still be done in \( O(n \log k) \) messages.

4. COMPLETE NETWORK WITH A SILENT FAILURE

4.1. The Model

Consider a complete network of processors were at most \( d \) processors may fail (\( d < \frac{n}{2} \)) before the algorithm starts. A processor failure causes it to stop sending messages, and since it is an asynchronous network, neighbors of the failed processor can never notice the failure.

Work that was done on this model in [KWZ] achieves an \( 12n \log k + 8kd \) messages bound. The algorithm presented here uses no more then \( 5n \log k + 2kd \) messages.
4.2. The Algorithm

Again, the basic mechanism is as in [KMZ] and the modifications are as follows:

1. The king, instead of sending one ASK message at a time, sends \( d+1 \) messages simultaneously on \( d+1 \) unused edges. Any ASK message that has been acknowledged, causes another ASK to be sent on a new unused edge.

2. A king, increasing its phase sends ASK with its new status to any neighbor from which a reply to an old ASK was not received.

3. The termination condition is if the number of neighbors that answered back is \( \lceil n/2 \rceil \).

4.3. Message Complexity

Theorem 36: If \( k \) nodes start the algorithm, then the number of messages used by the algorithm is bounded by \( 5n \log_k k + kd + O(n) \).

Proof: An upper bound for the number of messages of each kind:

1. \( \text{LEADER} \): exactly \( n - 1 \) messages.

2. \( \text{ASK} \): First let's consider ASK messages sent by a king. At a phase \( p \) the number of ASK messages that are sent by a king and finally answered, is no more than the size of the king's kingdom. And since a node can't have the same phase again, we get no more than \( n \) replied ASK message, for each phase and \( n \log_k k \) totally. A king, during its lifetime, can have at most \( d+1 \) unreplied ASK messages. So we count at most \( k(d+1) \) such messages. Additional ASK that is sent on one edge, can happen after a phase increasing. The number of phases increasing is bounded by \( k \) (since each increasing is a result of a dead king). Hence the number of resumed ASK messages is no more than \( d \times k \). A citizen transfers at most one ASK message per phase, hence the total number of these is bounded by \( n \log_k k \).

3. \( \text{ACCEPT} \): the total number of such a messages sent by a king is \( k - 1 \). At a given phase a citizen send at most one ACCEPT. So the total number of messages is bounded by \( n \log_k k \).

4. \( \text{YOUR \_CITIZEN} \): No more than the replied ASK messages. Hence, \( n \log_k k \).

5. \( \text{UPDATE} \): A citizen receives at most one such message per phase, hence the total number is bounded by \( n \log_k k \).
To conclude, the total number of messages is no more than $5n \log k + 2kd + O(n)$

5. LEADER ELECTING IN PRESENCE OF DYNAMIC LINK FAILURE

The model under investigation is the general network discussed before with the following exceptions:

1. A link failure may occur at any moment during a communication process.
2. Once a link has failed it cannot transfer any other messages.
3. A message sent along a communication link is lost only if this link failed before the message arrives.
4. The processors at both ends of the communication link notice the failure within a finite time.
5. The network graph remains connected.

Since a link may fail only once, the number of failures is finite. A trivial leader election algorithm for this model, could use any leader election algorithm for a reliable network and when a node notices a failure, it broadcasts this fact to its kingdom and all nodes restart the algorithm. This solution requires $O(t[m+n \log k])$ messages, where $t$ is the number of link failures.

The algorithm we present here makes a use of the kingdom spanning tree that was built up to the moment of failure and hence requires only $O(t[m+n \log k])$ messages.

5.1. The Algorithm

Informal description: The algorithm is based on the algorithm for a general graph presented before. We assume that a node notices a failure in one of its edges by receiving a $FAIL$ message from it. The basic idea is as follows:

If the failed edge is not a tree edge, it is ignored. Otherwise, the two parts of the kingdom become two independent kingdoms. The two kings will resume the search in their kingdoms in order to establish an alternative connection (possibly through some other kingdoms). The node that senses the failure in the part of the kingdom that has the king, routes the message $REMOTE\_FAIL$ to its king (unless it is a king itself). The king in response makes itself the origin of the new kingdom ($k(i) = id(i)$), increases its phase and broadcasts a $RESUME(status(i))$ message in its kingdom.

The node at the other end of the failed link (in the part that remained without a king), makes itself a new king and also broadcasts $RESUME(status(i))$ to its kingdom. The two kings and the nodes that received $RESUME$ change all their used edges that are not tree edges into unused edges. This is done because such an edge whose ends
were originally in the same kingdom is now a candidate for reconnecting the two parts.

A formal description:

The algorithm is based on the one presented in section 2 with the following additions:

1. A message, coming from a non-tree edge causes the edge to be deleted from unused(i).
   If i is a king and the message received from the same edge on which an ASK message has been sent but has not been replied, then execute procedure check:

2. A citizen i, upon receiving a FAIL message from its father edge, does the following:
   1) Make itself a king (k(i)=id(i) ; state=KING).
   2) Increases its phase by one.
   3) Broadcasts to all the nodes in its new kingdom a special update message: RESUME (k(i),p(i)).
   4) DFS_father(i)=null
   5) searched(i)=∅
   6) unused(i)={e: e ∈ E(i)}-sons(i)

3. A citizen i, upon receiving a FAIL message from its son, initiates a REMOTE_FAIL message routed to its king and deletes this son edge from sons(i).

4. A citizen i, upon receiving REMOTE_FAIL message from its son, checks if WaitForResume(i) is false and if so, forwards the message to its father and sets WaitForResume(i) to true.

5. A king i; in response to a REMOTE_FAIL message:
   1) Increases its phase by one.
   2) Initiates a RESUME (k(i),p(i)) broadcast on its tree.
   3) DFS_father(i)=null
   4) searched(i)=∅
   5) unused(i)={e: e ∈ E(i)}-sons(i)

6. A king i, in response to a FAIL message from its son:
   1) Delete the son edge from sons(i).
   2) Increases its phase by one.
   3) Initiates a RESUME (k(i),p(i)) broadcast on its tree.
4) $\text{DFS\_father}(i) = \text{null}$

5) $\text{searched}(i) = \emptyset$

6) $\text{unused}(i) = \{e : e \in E(i)\} - \text{sons}(i)$

(7) A citizen $i$ upon receiving a $\text{RESUME}(\text{status}(j))$ message:

1) Updates its status to $\text{status}(j)$,

2) Sends $\text{RESUME}(\text{status}(j))$ to its sons,

3) Sets $\text{WaitForResume}$ to $\text{false}$,

4) $\text{DFS\_father}(i) = \text{null}$,

5) $\text{searched}(i) = \emptyset$,

6) $\text{unused}(i) = \{e : e \in E(i)\} - \text{sons}(i)$

7) If $i$ is in the waiting substate it executes:

7.a) If $\text{status}(i) = \text{status}(j)$ and $e'$ is not a tree edge, it sends a $\text{YOUR\_CITIZEN}$ message along $e'$ and returns to the regular substate.

7.b) If $(\text{p}(i), \text{p}(j)) \geq (\text{p}(j), \text{p}(j))$ it returns to the regular substate.

(8) A node upon becoming a king, if $\text{WaitForResume}$ is $\text{true}$, it executes (4).

(9) CR1 (1) is replaced: $m = \text{ASK}(\text{status}(j))$: If $e \neq \text{father}(i)$ then $i$ does the following:

1) If $\text{status}(i) = \text{status}(j)$ and $e$ is not a son edge, it returns $\text{YOUR\_CITIZEN}$ along $e$.

2) If $\text{status}(j) > \text{status}(i)$ and $\text{k}(i) = \text{k}(j)$, it sends $m$ to its father and enters the waiting substate.

5.2. Ideas For Correctness Proof

The correctness proof is based upon the following properties of the algorithm:

1. A failure that occurred in a non-tree edge is disregarded.

2. A failure of a tree edge breaks the kingdom into two subtrees having a single king.

3. Every node in the two kingdoms will eventually receive a $\text{RESUME}$ message which cause it to disregard all the used edges that a $\text{YOUR\_CITIZEN}$ message was sent on.
6. LEADER ELECTING WITH DYNAMIC LINKS FAILURE AND RECOVERY

Here we assume that links may be added to the network at any time during a communication process. The link that is being added could be either a new one that connects nodes that were not connected before, or a recovery of an old link that has failed and the processor cannot make the difference between the two cases.

The algorithm suggested here constructs a spanning tree in the case where that the failures of tree links are not too frequent. Assume the spanning tree has been constructed. Then we possess a spanning tree all the time, possibly, except some temporary disconnections caused by failure of tree edges. If such a failure occurs the algorithm reconstruct the disconnected paths. Another important property of this algorithm is that any node that was awakened during the algorithm always belongs to some kingdom. And any kingdom has a single kink which its direction is known to every citizen in the kingdom.

6.1. The Algorithm

As a node notices a new link it adds this link to \( unused(i) \) and if the king has left this node for good, \( searched(i) = sons(i) \) it initiates a \textit{RECOVERY} message routed to the direction of the king.

When a \textit{RECOVERY} message is received by node \( i \), if all sons are in \( searched(i) \), it does the following:

\( (i) \) Forward the \textit{RECOVERY} to the king.

\( (ii) \) It deletes the edge from which the \textit{RECOVERY} message was received from \( searched(i) \).

This process opens a path of unsearched edges which guarantees that the king will eventually visit the node with the new link and will communicate through it (send \textit{ASK}).

6.2. Remarks

- One of the main achievements of this algorithm is that it is capable of constructing and maintaining a spanning tree, in a faulty edges environment where a tree edge may fail even after the leader is elected. A link failure causes a new king to be born. The two kings now try to connect the broken tree, using other edges.
- Note that if \( tm = O(n \log k) \) then message complexity does not increase relative to the reliable algorithm.
- If the graph becomes disconnected during the execution of the algorithm and has two or more connected components, then in each of these that will have an elected king.
REFERENCES


