# Leaderless Distributed Hierarchy Formation

Jacob Beal

December, 2002

**Abstract**

I present a system for robust leaderless organization of an amorphous network into hierarchical clusters. This system, which assumes that nodes are spatially embedded and can only talk to neighbors within a given radius, scales to networks of arbitrary size and converges rapidly. The amount of data stored at each node is logarithmic in the diameter of the network, and the hierarchical structure produces an addressing scheme such that there is an invertible relation between distance and address for any pair of nodes. The system adapts automatically to stopping failures, network partition, and reorganization.

# 1 Introduction

While trying to get a bunch of simulated amorphous fireflies to flash in unison, I discovered that my life would be made much easier if I could cluster them together into a hierarchy, where fireflies would listen to each other in proportion to how closely related their addresses in the hierarchy were. Solving the problem of hierarchical clustering in an amorphous computing context ended up producing an algorithm valuable in its own right, which I present in this paper.

Hierarchical clustering is an old and well-studied problem, but amorphous computing poses some new challenges which change the problem significantly. Nevertheless, the results in amorphous computing depend on little in that domain,[1] and may be relevant in general to clustering problems — particularly as it applies to ad hoc networking and biological computing. In this paper, I will describe the amorphous hierarchical clustering problem and present as a solution the LeaderlessHierarchy algorithm, along with analysis and experimental results.

The algorithm I have developed, LeaderlessHierarchy, specifies a distributed system which builds and maintains hierarchical clustering in an amorphous computer. Its expected convergence time is $O(d)$, where $d$ is the diameter of the network, and it has a storage and communication density logarithmic in the diameter. Despite these low costs, the LeaderlessHierarchy algorithm is extremely robust, scales to networks of arbitrary size and is able to transparently adapt to disruptions caused by stopping failures

[1] Rather, you could view them as the result of not being able to depend on anything!

and reorganization of the network. Finally, the clusters produced LEADERLESSHIERARCHY have the property that for any two nodes in the network, we can predict their distance from their cluster membership and vice versa.

## 2 Problem Description

First, we need a precise definition of hierarchical clustering. Informally, every node in the network needs a unique address, starting with the most specific field being its unique identifier and rising through progressively less specific identifiers until the most general identifier, which is shared with every node in the network.

More formally, we will say that a network is hierarchically clustered if every node $n$ has an address $A_n$ which is composed of a set of identifiers $A_{n_i}$ such that the equivalence classes of nodes with equal $i$th identifiers form a tree topology. Additional desiderata, though not strictly required, are that the size of the equivalence classes transition smoothly between single nodes and the entire network, and that all the addresses be the same length.[2]

Throughout the rest of the paper, I'll name the identifiers in the addresses interchangeably as either clusters or groups.

The amorphous computing context for this problem is described as follows: consider a piece of paper with tiny processors distributed randomly across its surface. Each processor can broadcast to those other processors less than distance $r$ away. Thus, the network graph has processors for nodes, and an edge between any two processors less than distance $r$ from each other. Congestion in this network is local in scope (from overlapping broadcasts), and thus we care about the *density* of communication in the network, rather than the actual number of messages sent: a small number of messages all in one area may be worse congestion than a very large number of message spread evenly across the network.

Processors may run at slightly different speeds, lose messages, or suffer stopping failures.[3] Furthermore, processors have small memory, relatively little computational power available, and no global information. Finally, depending on the size of our sheet of paper and the density at which processors are applied, the number of processors we are clustering may vary wildly, anywhere from dozens or hundreds to millions or (in the case of a bucket of

[2] This last mostly just makes addresses more intelligible at a glance for humans.

[3] In other words, they can die at random.

biological computing) trillions — in other words, our algorithm must scale well.

Now the hard part: I want this system to automatically adapt to changes in its environment. If I spill coffee on the paper, killing processors in a region, it should adapt its groups appropriately. Similarly, if I cut the paper in half, each half should adapt its groups, then adapt again if I paste it back together differently: I don't want to end up with half a cluster on each of two distance edges. More generally, I want to be able to do clustering on any two dimensional surface, whether it be a piece of paper or a skin of "smart paint" covering a suspension bridge, and I want that clustering to adapt appropriately when something happens to disrupt or reconfigure the surface.

What does it mean to "adapt appropriately"? What I mean is that the clustering our piece of paper converges to shouldn't show evidence of the cutting and pasting we've done to it: the structure should be seamless, just like it would be if the final geometry was the one it started with. It's OK if things are incoherent and transient for a brief period following a change, as long as the system converges again in a short period of time.

To summarize: we need a distributed algorithm that builds hierarchical groups quickly and robustly on the basis of local communication, adapts automatically to changes and reorganizations of the network, and scales to networks of enormous size.

# 3 Algorithm

The LeaderlessHierarchy algorithm takes a bottom-up clustering approach to building the group hierarchy. Starting with the bottom level, where the groups are just individual nodes, group building proceeds according to three stages: first, each group finds its neighbors, then an election algorithm clusters together each group into a clique with other nearby groups, and finally the cliques become supergroups which form the next level of the hierarchy.

The devil, of course, is in the details that allow this to happen in a leaderless distributed manner with only local communication and the ability to adapt to changes and disruptions in the network.

In the subsequent sections, I will first sketch the high-level behavior of the algorithm. Next I will give the backbone of the algorithm, then fill in details for each of the subfunctions. Appendix A contains a complete tabulation of

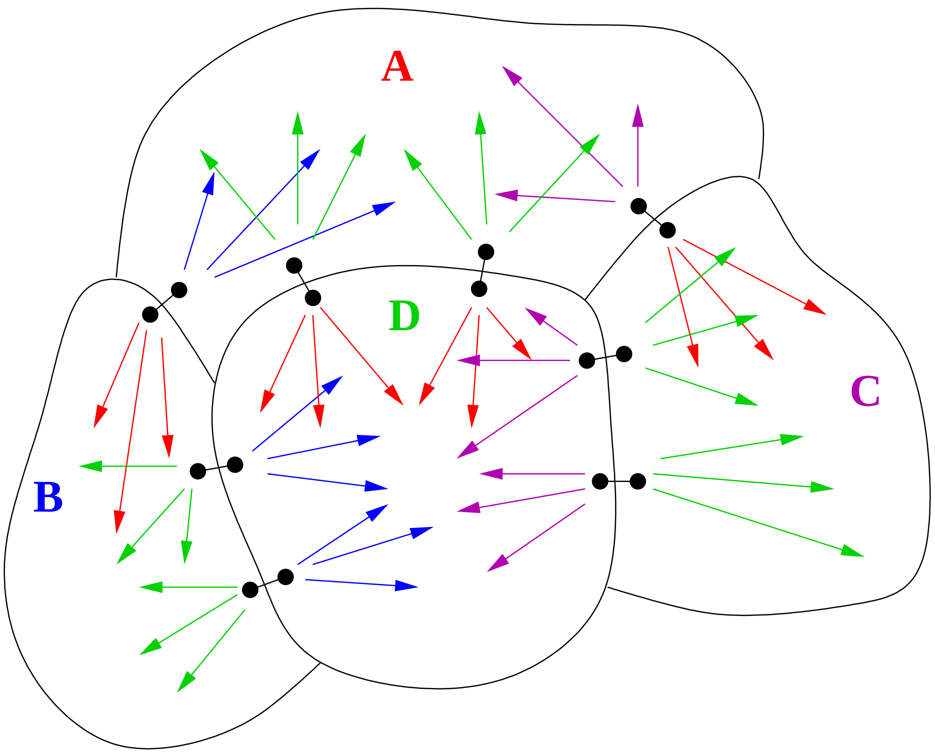



Figure 1: Information about neighbors propagates from nodes on the border with neighboring groups. Since each piece of information is labelled with how far removed it is from its source, and nodes only accept information updates from neighbors with better information, eventually all the nodes in a group agree about the set of neighbors.

the constants and state information used by the algorithm.

## 3.1 Behavior Sketch

There are three phases to construction for each level of the hierarchy: find neighbors, elect "leaders", and promote to the next level. In practice, we run all three phases simultaneously and continuously, but behavior reaches convergence in the order listed, and it's easiest to describe as though they are happening in sequence.

Finding a neighbor is easy for a node at the border of a group, since it is directly connected to the neighboring group. Each such bordering node sends out a gradient telling the rest of the nodes in its group about the neighbor it has discovered. If the distance is short enough, the gradient will pass across group boundaries into other groups — this lets the neighbor relation be connected with physical distance even high in the hierarchy.

Once groups know about their neighbors, then cliques can be created to serve as the basis for the next level of the hierarchy. Cliques are formed by a leader election algorithm: if a group's ID is greater than that of all its non-follower neighbors, then it declares itself a leader, and if it has one or more neighbors with a greater ID that have declared themselves leaders, then it declares itself a follower of the one with the greatest ID. "Leaders" and "Followers" do not have any different behavior, however — the only effect is that the ID of the clique leader will be the ID of the group at the next level.

Once a clique has formed, its members all join a group at the next level of the hierarchy (The ID of the new group is that of the group which was clique leader). The new group then starts looking for its neighbors, and so on, until there is a single group which covers the entire network.

Even after a level has completed this process, it is kept running, to avoid synchronization issues and ensure robustness against failure. Since the system converges rapidly, given a stable network (see Section 4.2 for proof of convergence) there is no need to ever stop running the algorithm — it simply reaches the point where it makes no changes while the network is stable.

## 3.2 Skeletal Algorithm

The algorithm run at each node has three responsibilities. First, compiling information about its neighbors at each level of the hierarchy. Second, using that information to construct the portion of hierarchy in which it is contained. Finally, keeping its physical neighbors informed.

At startup of the LeaderlessHierarchy algorithm, the node constructs the zeroth level of the hierarchy — the leaf which contains only itself. We assume that each node can pick a random numerical ID with enough bits to make ID collision unlikely. After that, it broadcasts a copy of its internal state to its physical neighbors once at regular intervals, and continually runs a set of subroutines that maintain its internal state. ElectLeader and MaintainLevels are responsible for constructing the hierarchy. Compiling information about the neighbors is managed by FindNeighbors, which is called when incoming state messages from neighbors invoke the ReceiveMessage function, and MaintainNeighbors, which is responsible

```
LeaderlessHierarchy()
S.L[0].t ← S.clock ← S.lastSend ← 0   ▷ Initialize zeroth level
S.L[0].I.uid ← S.uid ← Random()
S.L[0].I.status ← "none"
while true   ▷ main loop
do if S.clock − S.lastSend ≥ T_m
      then Send(S)   ▷ Send state to neighbors
           S.lastSend ← S.clock   ▷ Schedule next transmission
   for i ← 0 to |S.L| − 1   ▷ For each level...
   do ElectLeader(S, i)   ▷ Run leader election
      MaintainNeighbors(S, i)   ▷ Prune dead neighbors
      MaintainLevels(S, i)   ▷ Create and destroy levels
```

```
ReceiveMessage(S, N)
  ▷ S is internal state, N is neighbor's state message.
for i ← 0 to Min(|S.L|, |N.L|) − 1
do FindNeighbors(S, N, i)   ▷ Pass along info for each level
```

Figure 2: Pseudocode for top-level algorithm

for pruning out neighbors which no longer exist.

We assume that the node's period $T_m$ between sending messages is set such that some underlying network protocol can ensure delivery of most messages, and the processor generally has enough time to run ReceiveMessage on a message from each of its neighbors once per send cycle.

In addition, the clock of a given node is allowed to run slightly fast or slow, but if the clock runs more than some fixed difference in frequency $\epsilon_f$ from accurate time, then the node is considered to be malfunctioning. The time indicated by a clock, however, is immaterial so long as it advances regularly.

## 3.3 Getting Neighbor Information

Neighbor state information originates at a group's borders and flows across the group in a gradient of increasing "hearsay" values, then across its borders into subsequent groups if those groups are close enough to the source. These gradients flow a minimum distance of $N_0 N_m^i$ hops at level $i$, so even at high levels, neighborhood is based in distance rather than adjacency (a fact that will become important in bounding the depth and extent of the hierarchy).

Discovering and passing information about live neighbors is simple: the border nodes receive updated information every time their cross-border physical neighbors transmit, and from there it flows down the gradient throughout the group. Multiple sources of information are, of course, no problem so long as the neighbor group has converged its own information.

It is much harder to determine that a neighbor no longer exists and should be deleted from records. There are basically two ways that this can happen: network disruption and transient construction effects. Network disruption could be either a permanent effect like a tear in material, or something temporary like a network failure. Transient effects in the hierarchy construction come from a group making a decision, then receiving information from farther away which changes it — for example, a group might hear from a farther neighbor and decide to follow it instead of a closer neighbor.

At a gradient source, it's easy to decide when the connection with a neighbor has been disrupted: if it hasn't heard from a physical neighbor across the border for more than some constant time $T_k$, then it can assume that the connection has been severed and declare the neighbor dead.

The difficulty in deleting neighbors comes in propagating the information through the rest of the group. In essence, how do we ensure that when we delete a neighbor, it won't just get re-added by another part of the group

```
FindNeighbors(S, N, i)
for each x ∈ N.L[i].N    ▷ Go through nbr's nbr records
do if x.finalDead or x.I.uid = S.L[i].I.uid
      then continue    ▷ Ignore self and nbrs about to be deleted
   if S.L[i].I.uid ≠ N.L[i].I.uid and x.hearsay ≥ N_0 N_m^i
      then continue    ▷ Gradient passes nearby borders only
   if ∃y ∈ S.L[i].N s.t. y.I.uid = x.I.uid
      then if x.hearsay < y.hearsay and ¬x.dead
              then y ← x    ▷ Copy info if it's better
                   y.hearsay ← x.hearsay + 1
                   y.lastHeard ← S.clock
              else if x.hearsay > y.hearsay and x.dead and y.dead
                      then y.lastHeard ← S.clock    ▷ Postpone deletion
      else if ¬x.dead
              then y ← x
                   y.hearsay ← x.hearsay + 1
                   y.lastHeard ← S.clock
                   S.L[i].N ← S.L[i].N ∨ y    ▷ add new info
if S.L[i].I.uid ≠ N.L[i].I.uid    ▷ If nbr is in a different group...
  then x.I ← N.L[i].I    ▷ Copy info from neighbor
       x.lastHeard ← S.clock
       x.hearsay ← 1
       x.dead ← x.finalDead ← false
       if ∃y ∈ S.L[i].N s.t. y.I.uid = N.L[i].I.uid
          then y ← x    ▷ Update existing info...
          else S.L[i].N ← S.L[i].N ∨ x    ▷ ... or add it if it's new
```

Figure 3: Pseudocode for FindNeighbors function

```
MaintainNeighbors(S, i)
for  each nbr in S.L[i].N
do if S.clock − nbr.lastHeard > T_k and ¬nbr.dead
      then nbr.dead ← true    ▷ Mark neighbor as inactive
           nbr.lastHeard ← S.clock
   if S.clock − nbr.lastHeard > 2T_k and nbr.dead
      then nbr.finalDead ← true    ▷ Mark for deletion
           nbr.lastHeard ← S.clock
   if S.clock − nbr.lastHeard > T_k and nbr.finalDead
      then S.L[i].N ← S.L[i].N − nbr    ▷ Delete neighbor record
```

Figure 4: Pseudocode for MaintainNeighbors function

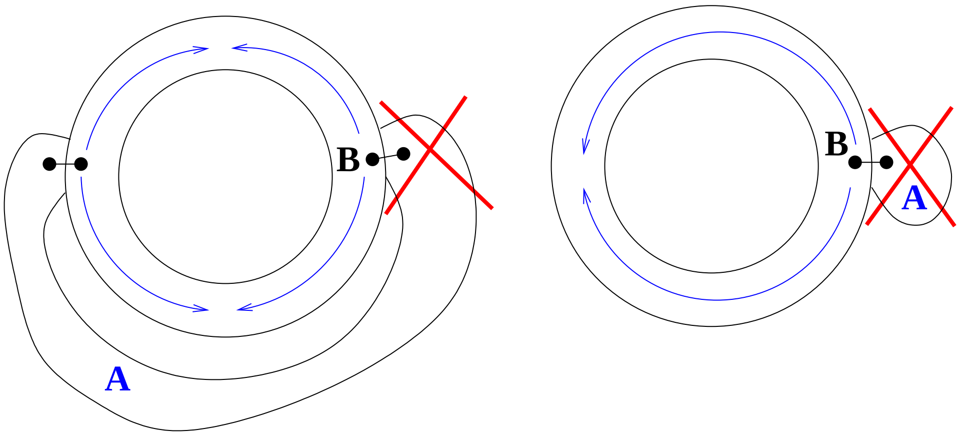



Figure 5: When deciding to delete a neighbor, we must distinguish between losing just one of two distant connections and losing a single connection. We must guard against both deleting a neighbor that still connects and keeping a spurious connection to a neighbor that no longer exists.

that hasn't deleted it yet? Figure 5 shows two situations which exemplify this difficulty. In the first case, a connection to a neighbor has been lost, and the gradient must be re-adjusted to reflect the new distances. In the second case, the gradient must be deleted entirely. However, if we aren't careful, the the second ring can end up with a maximum distance end of the gradient growing into areas which have already been deleted, as though it were the first case, and may even create a standing wave of deletion and re-growth around the loop.

We solve this by having actual deletion occur in a backgradient. The decision that a neighbor is dead propagates forward from the source with a time constant of $T_k$: if the time is greater than $T_k$ since a node last heard a neighbor was alive from a node closer to the source than itself, then it flags the neighbor as dead, but does not yet delete it. Once a portion of a neighbor-gradient is dead, it can no longer propagate itself, though it can be revivified by good news from closer to the gradient source.

The decision to delete a neighbor propagates backward from the extrema of the gradient with a time constant of $2T_k$, deleting everything that has no more distance dead neighbor. This deletion back-gradient is run twice as slowly to ensure that the death-gradient out-paces it in killing live nodes. Thus it is impossible for a standing wave of life-death-deletion-life to develop, since the death wave travels twice as fast as the deletion wave, and must eventually catch up with all live nodes before they can propagate into the void left by the deletion wave.

Thus, by a system of forward gradients propagating information about neighbors and back gradients deleting neighbors which no longer connect, the FindNeighbors and MaintainNeighbors routines are able to maintain a consistent set of neighbor information across groups in the hierarchy.

## 3.4 Leader Election

At each level, a node may be either a leader, a follower, or undecided. The difference between leaders and followers has no computational or organizational consequence: rather, it just determines where a node gets its group ID for the next level up in the hierarchy, and ensures that groups get bigger and farther apart at higher levels.

A group becomes a leader at a given level if all of its neighbors are either followers of a different leader, or have IDs less than its own. Conversely, a group becomes a follower if it has a non-follower neighbor with an ID greater

```
ElectLeader(S, i)
live ← {x|x ∈ S.L[i].N ∧ ¬x.dead}
if live ≠ NIL and {y|y ∈ live ∧ y.I.status ≠ "follower"∧
   y.I.uid > S.L[i].uid} = NIL
   then S.L[i].I.status ← "leader"
        S.L[i].I.leaderID ← S.L[i].uid   ▷ Declare ourself leader
   else leads ← {y.I.uid|y ∈ live ∧ y.I.status = "leader"}
        if leads ≠ NIL
          then   ▷ Follow the best neighbor
                 S.L[i].I.status ← "follower"
                 S.L[i].I.leaderID ← Max(leads)
          else   ▷ Don't follow anybody
                 S.L[i].I.status ← "none"
                 S.L[i].I.leaderID ← NIL
```

Figure 6: Pseudocode for ElectLeader function

than its own. If a group has no neighbors, on the other hand, it becomes neither a leader, nor a follower. Thus, when the top level group is formed, covering the entire network, hierarchy building stops because that node is neither a leader nor a follower.

Since every node in a group eventually has identical knowledge of neighboring groups, and since the decision is made deterministically, each node can run the ElectLeader function independently, and the entire group will arrive at the same results.

## 3.5 Creating and Destroying Levels

The MaintainLevels routine is responsible for coordinating the actions of different levels of the hierarchy. The three tasks it is responsible for are: creating new and higher levels of hierarchy, destroying excess levels of hierarchy, and propagating information up from lower levels to higher levels.

The information which needs to propagate upwards in the hierarchy are changes in leadership. Since a group's identity at level $i$ is taken from its chosen leader at level $i-1$, then for obvious reasons, when the leader at $i-1$ changes in response to new information, the ID at level $i$ must change as well.

```
MaintainLevels(S, i)
if i > 0 and S.L[i − 1].I.leaderID ≠ S.L[i].I.uid
  then S.L[i].I.uid ← S.L[i − 1].I.leaderID
       S.L[i].I.status = "none"   ▷ Percolate leader-changes upward
if i > 0 and S.L[i − 1].leaderID = NIL
  then S.L ← {S.L[j]|j < i}   ▷ Delete all upper levels
if S.L[i].I.leaderID ≠ NIL and |S.L| = i + 1 and
  S.clock − S.L[i].t > T_m W_0 W_m^i
  then S.L[i + 1].t ← S.clock
       S.L[i + 1].I.uid ← S.L[i].I.leaderID
       S.L[i + 1].I.status ← "none"   ▷ Start the next level
if i > 0 and |S.L| = i + 1
  then live ← {x|x ∈ S.L[i − 1].N ∧ ¬x.dead}
       if live = NIL
         then S.L ← {S.L[j]|j < i}   ▷ Delete top level
```

Figure 7: Pseudocode for MaintainLevels function

If there isn't yet an $i$th level when a leader is selected at level $i-1$, on the other hand, then the node may create one. In practice, this is prohibited from happening any sooner than $T_m W_0 W_m^{i-1}$ after the creation of level $i-1$, to allow time for level $i-1$ to converge and cut down on the creation of transient groups.

Finally, there are two ways in which levels may be deleted (always from the top downwards). First, if the level at the very top has no neighbors, then it will be deleted — this most commonly happens when a transient group existed alongside the whole-network group, causing it to briefly think it had a neighbor and thereby create another level to be the top. Second, if a group entirely loses its neighbors at a lower level (which most often happens in cases of severe damage), then the entirety of the hierarchy above it is deleted and must be reconstructed (since it is presumed to be invalid anyway).

## 3.6 Variations

The amount of network traffic consumed by LeaderlessHierarchy can be reduced by a few simple measures. In the simple implementation, the full state of a processor is transmitted to its neighbors in every cycle. The number of bits sent can be reduced by having the message contain only differences and liveness information, rather than all the rest of the baggage as well. Additionally, the frequency of the messages can be traded off against the responsiveness of the network in adapting to changes; when a processor believes its configuration has converged, it can exponentially back off the frequency of liveness information it sends and requires.

Another tradeoff can be made with identifiers to reduce the compression of addresses by monotonicity of leadership decisions. If the leader group at a given level supplies not its own ID, but a new ID (chosen distributedly from its membership) then the range of the IDs will not shrink at higher levels in the hierarchy, although the amount of information stored at each node must increase accordingly.

Finally, leader election need not take place only between immediate neighbors. Groups with a multi-hop radius can be formed at each level instead, with only slightly more complicated code. See Appendix B for pseudocode.

# 4 Analysis

## 4.1 Addressing/Distance Relation

*Theorem:* Consider any two nodes $n_a$ and $n_b$ in the network. Let $d$ be the minimum number of hops between $n_a$ and $n_b$ in the communications graph, and $l$ be the lowest level of hierarchy in which $n_a$ and $n_b$ belong to neighboring groups. Then $2^{l-1} < d \leq 3^l + 2^{l+2}$ and $\lfloor \log_3 d \rfloor - 2 \leq l \leq \lceil \log_2 d \rceil$.
*Corollary:* There are $O(\log diam)$ levels in the hierarchy of a network which has converged.

**Proof:** The neighbor-detection gradients used by LeaderlessHierarchy flow a minimum of $N_0 N_m^i$ hops at level $i$. Using values $N_0 = 1$ and $N_m = 2$, this becomes a radius of $2^i$. For two nodes distance $d$ apart, they are guaranteed to become neighbors when $d \leq 2^i$. Thus, if we know the minimum level is $l$, we know that $2^{l-1} < d$, because if the distance were less, then $l$ would not be the minimum. Similarly, if we know the distance is $d$, then we know that they must be neighbors no later than $l = \log_2 d$, and since $l$ must be an integer, we take the ceiling and produce the relation $l \leq \lceil \log_2 d \rceil$.

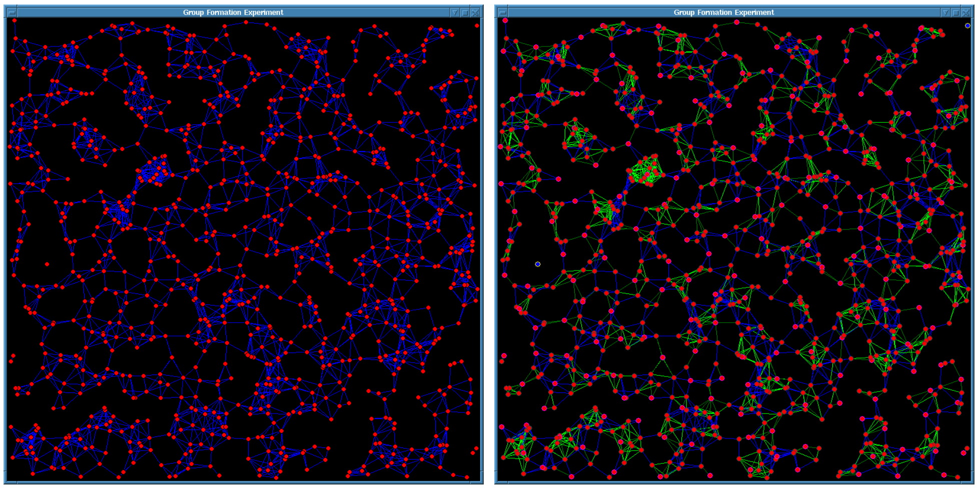
The complementary bounds can be derived by considering the maximum diameter of a group at level $i$, since the two nodes cannot become neighbors until a level at which the boundaries of the groups they belong to come within $2^i$ of each other. Using single-hop groups (or equivalently, $R_G = 1$), this maximal configuration contains a leader and two diametrically opposite followers, each at maximum neighborhood distance from the leader. This gives the recurrence: $D_i = 3D_{i-1} + 2 * (2^{i-1})$. At level zero, the diameter of a node is 1 hop (we consider the partition to occur halfway between nodes) and only immediate neighbors are considered, giving a base case of $D_1 = 3$. Solving the recurrence, we obtain $D_i = 3^i + 2^i$. Thus, if we know the minimum level is $l$, then we know that the distance $d$ can be no greater than the diameter of two nodes, plus the distance between them: $d \leq 2 * (3^l + 2^l) + 2^{l+1} = 3^l + 2^{l+2}$. It is worth noting that the $3^l$ term dominates this equation for $l > 4$. Similarly, the inverse relation will generally be dominated by the $3^l$ term, so we may usefully overestimate the relation as $d \leq 3^l + 4(2^l) \leq 2 * 3^{l+1} \leq 3^{l+2}$, which gives us the slightly loose bound $l \geq \lfloor \log_3 d \rfloor - 2$.

As a corollary to the minimum distance relation, we can bound the maximum depth of hierarchy necessary to cover the entire network (once the system has converged). For a network with diameter $diam$, we have that for $l \leq \lceil \log diam \rceil$, every pair of nodes belong to either the same group or neighboring groups. Thus, the network is fully connected and there can be only one leader, and as a consequence, the next level in the hierarchy will consist of a single group covering the entire network. Therefor, there are $O(\log diam)$ layers in the hierarchy.

## 4.2 Convergence Time

In this section, I will show that, in the absence of error, the system always converges to a unique hierarchy in $O(d)$ time.

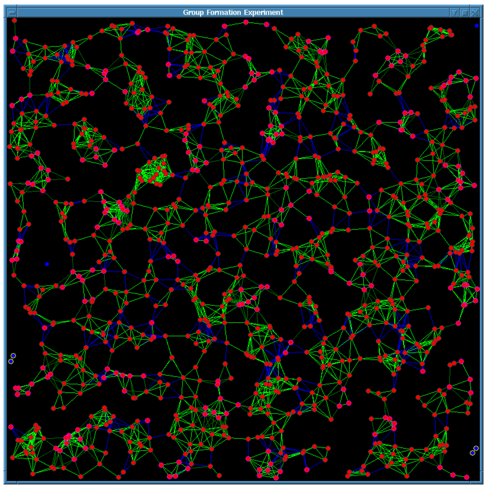
First, we need to show that there is a unique hierarchy to which the network converges at all. Assume that the $i$th level of the hierarchy converges to a unique set of groups and neighbors. Leader election is deterministic given perfect information about group IDs and neighbors, which every node is eventually provided with. Therefor, the groups at the level $i + 1$ are completely determined by the groups at level $i$. Neighbors at level $i + 1$ are determined by the spatial embedding of the nodes in groups at level $i + 1$, which is also fixed. Therefor, each level is uniquely determined by the previous level, and given the base case of level 0 which is fixed to a unique set of groups (individual nodes with fixed IDs) and neighbors (edges in the



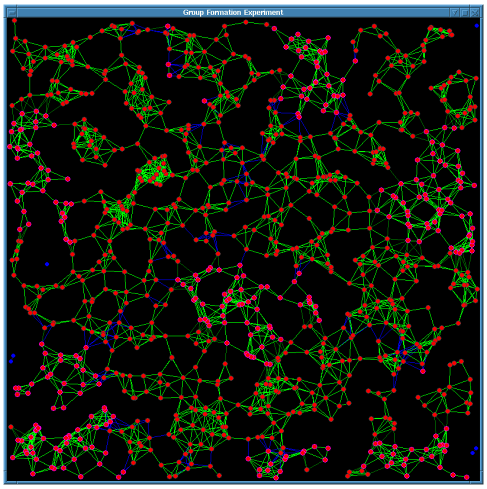

(a) Level 0

(b) Level 1

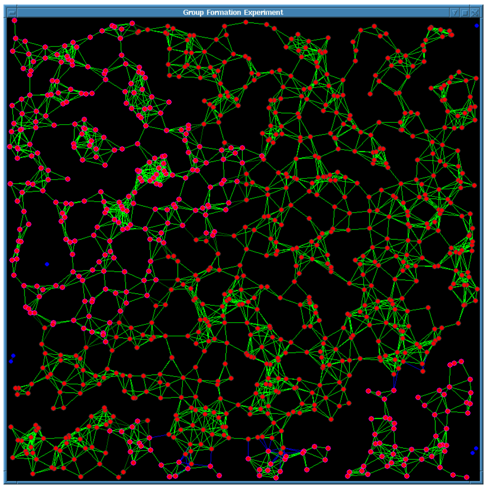
Figure 8: Groups produced by a run of 1000 processors with communication radius 0.05. Pink nodes are leaders, red are followers. Links between nodes are blue for nodes in different groups and green for nodes in the same group. Predictions, based on an expected diameter of 35 hops, are 5 levels of hierarchy and convergence time $70T_m$. In the experiment, the network has converged to 5 levels of hierarchy after a duration of $77T_m$ (though the final level was not added until $130T_m$ due to the exponential delay built into construction of new levels). This figure shows the bottom two levels; Figure 9 shows the upper four.



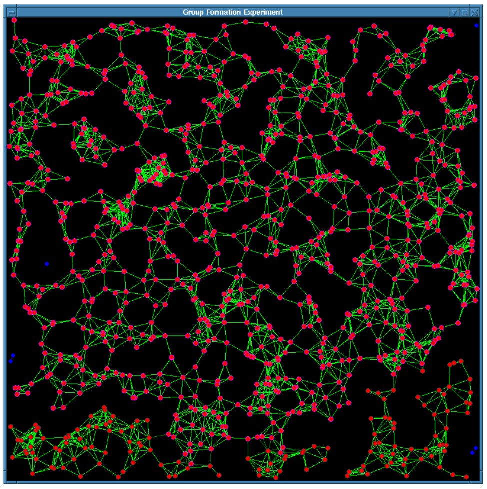

(a) Level 2




(b) Level 3




(c) Level 4




(d) Level 5

Figure 9: Groups produced by a run of 1000 processors, as in Figure 8, showing the upper four levels of the hierarchy produced.

network), we see by induction that there is a unique hierarchy induced by the placement of nodes and the IDs assigned to them.

*Theorem:* Time of convergence is order $O(diam)$.

**Proof:** The worst case time is actually much worse. In the worst case, nodes are arranged in a double spiral, with one spiral containing very low IDs, and the other spiral containing monotonically descending IDs greater than the low ID spiral. The low ID spiral can immediately declare as followers, but the members of the high ID spiral cannot determine whether they are a leader or follower until every node with a greater ID has done so, in sequence. Thus the worst case is $O(n)$ for a $n$ node network, but the expected behavior is much better.

The expected convergence time for any given node is based on how many hops $h$ information must flow for it to decide if it is a leader or a follower. In the simplest case, the processor has a higher ID than any of its $k$ neighbors, and therefor can decide with a single hop of information. However this is has a probability of only $\frac{1}{k+1}$. The rest of the time, the processors fate will be determined by the highest ID neighbor that is not a follower. If we assume that the highest neighbor will determine the fate of a node, then we may overestimate $h$ as the expected length of a monotonic chain of length $l$ starting at the highest neighbor.

$$P(l) = \int_{v_0=0}^{1} \int_{v_1=v_0}^{1} \cdot\cdot \int_{v_l=v_{l-1}}^{1} 1 dv_l \cdot\cdot dv_1 dv_0 = \frac{(1-v_0)^l}{l!}|_{v_0=0}^{1} = \frac{1}{l!}$$

$$E(l) = \sum_{l=0}^{\infty} P(l) = \sum_{l=0}^{\infty} \frac{1}{l!} = e$$

If the chain is of an even length (which we can overestimate as occurring half the time) then the highest neighbor will be a follower rather than a leader, and we must determine our value via the next highest neighbor instead. Thus, the probability that $h$ is determined by the $n$th highest neighbor is overestimated at $P(n) = 0.5^n$. Much of its chain will likely have passed information already, but even if we overestimate by assuming they are calculated sequentially, the sequence of $n$th neighbors still converges rapidly:

$$E(h) \leq \sum_n P(n) n E(l) = e \sum_n \frac{n}{2^n} = 2e$$

Since the expected number of hops for convergence is constant, and the length of a hop grows exponentially with each level, convergence time will be dominated by the final level, where the leader group needs only one hop of length $O(diam)$ to cover the entire network.

## 4.3 Communication Density and Data Storage

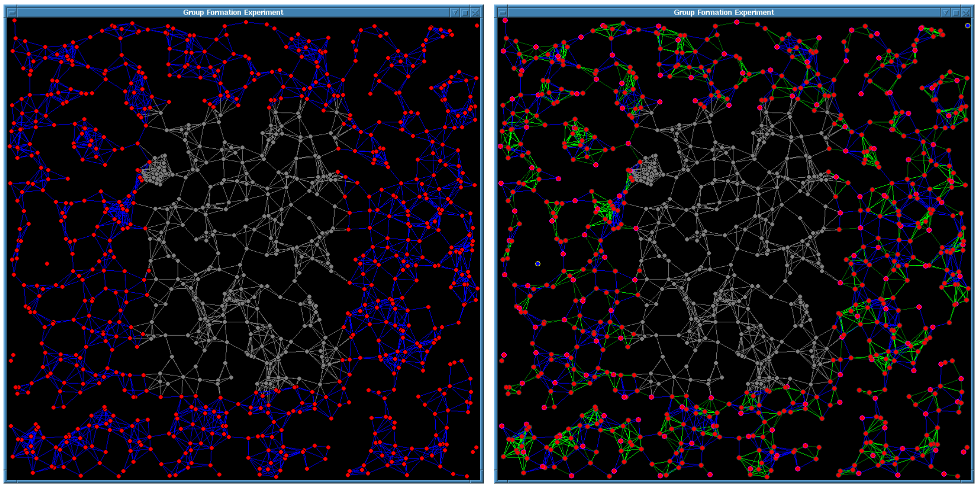
As shown above, the hierarchy contains $O(\log d)$ levels. Each level of the hierarchy needs to store a constant amount of information about each neighboring group at that level. Since neighbors information is broadcast in a radius based on an exponentially increasing distance, the number of neighbors expected at a given level is a constant determined by spatial packing of non-overlapping neighbor broadcasts in the previous level. Thus, each node keeps a state of expected constant size per level and the total storage is $O(\log d)$.

Communication between processors consists of regular state broadcasts from each processor, with a period of $T_m$. The broadcasts contain the entire state, which is of size $O(\log d)$, and there is one per processor. Thus, for a processor density of $\rho$, we produce a communication density of $O(\frac{\rho}{T_m}\log d)$.

## 4.4 Adaptation Rate

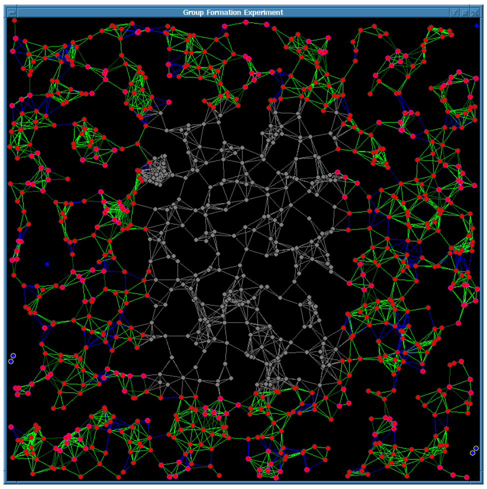
The radius affected by the death of a single processor depends on how important that processor's ID was in configuring the network. While no single processor plays a key computational role in the system, the hierarchy induced from the network depends critically on the IDs of the leader groups at each level. Thus, in the worst case, if the processor with the highest ID in the network dies, this effect will trickle up the entire hierarchy until the top-level group has been renamed with the ID of the second highest ID in the network, costing $O(d)$ time of convergence.

The expected time, however, is much better, because at any level of the network, most nodes are not leaders. Given the tightest packing of neighborhoods, the probability of node $p$'s individual ID being used for a group at level $l$ can be bounded as follows: at level $l-1$, every node within distance $2^{l-2}$ hops must be a member of a neighboring group at level $l-1$. Thus, if node $p$'s ID is used in its group at level $l$, all of the neighboring groups at level $l-1$ must be followers at that level, and therefor no node within $2^{l-2}$ hops can have an ID used at level $l$. Assuming a node density $\rho$, we

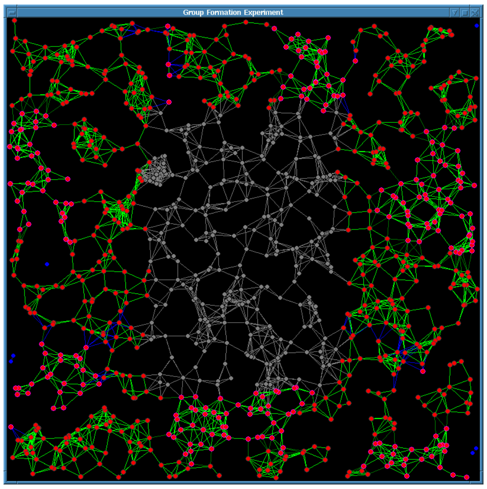



(a) Level 0 (b) Level 1

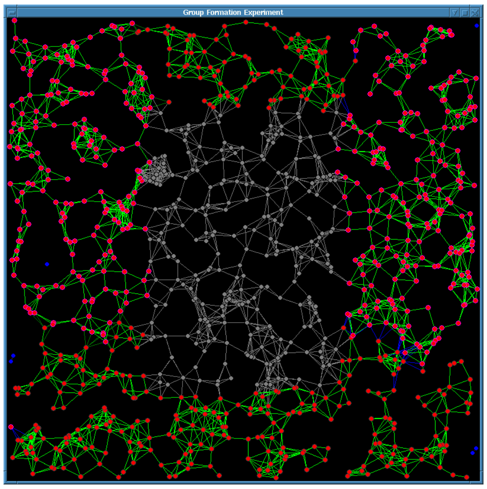
Figure 10: Adaptation to damage in a run of 1000 processors with communication radius 0.05. Pink nodes are leaders, red are followers, and grey are dead. Links between nodes are blue for nodes in different groups and green for nodes in the same group. The system reconverges after $122T_m$ following the disruption. This figure shows the bottom two levels; Figure 11 shows the upper four.



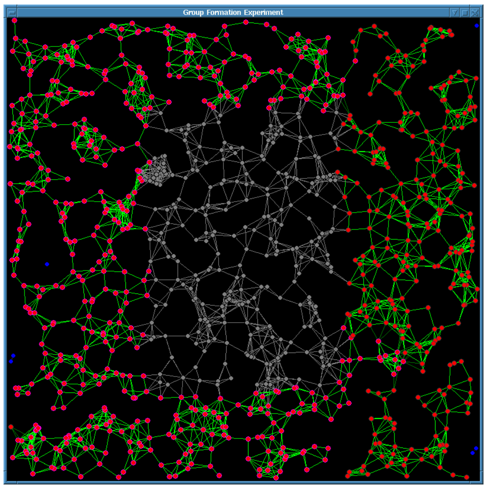

(a) Level 2




(b) Level 3




(c) Level 4




(d) Level 5

Figure 11: Adaption to damage in a run of 1000 processors, as in Figure 10, showing the upper four levels of the hierarchy produced.

can count the nodes in the area and find that the probability of a node's ID being used at level $l$ is bounded at $P_I(l) \leq \frac{1}{\rho\pi(2^{l-2})^2}$.

Now we need to find the expected area of disruption for an ID used at level $l$ (which will dominate all lower levels). We can take the expected number of hops to be the expect length of descending chains from that ID, symmetric to the length of ascending chains used in determining the convergence time, which is $O(1)$. The distance per hop, scales exponentially as $O(3^l)$ (See Section 4.1).

Putting this all together, we have an expected distance of disruption of $E_D = \sum_{l=0}^{\infty} O(3^l) * O(1) * \frac{1}{\rho\pi(2^{l-2})^2} = \sum_{l=0}^{\infty} O(3^l/2^{l^2}) = \sum_{l=0}^{\infty} O((3/4)^l) = O(1)$

So the expected radius $r$ around a dead processor which must reconverge is $O(1)$, and since time of convergence is on the order of the diameter which must converge, the expected time of reconvergence is also $O(1)$.

# 5 Contributions

I have posed the problem of hierarchical group formation in the context of amorphous computation and found that it can be solved via the LeaderlessHierarchy algorithm. This algorithm converges rapidly with low communication and storage costs, scales to networks of arbitrary size, and adapts automatically to stopping failures and reorganization of the network. In addition, the clustering produces has the useful property that cluster address and distance are related for any pair of nodes.

# 6 Acknowledgements

Particular thanks to Joshua Grochow for infecting me with the firefly synchronization problem and spurring me to keep on with this work.

# References

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight, R. Nagpal, E. Rauch, G. Sussman and R. Weiss. Amorphous Computing. AI Memo 1665, August 1999.

[2] Daniel Coore. Establishing a Coordinate System on an Amorphous Computer. MIT Student Workshop on High Performance Computing, 1998.

[3] Daniel Coore, Radhika Nagpal and Ron Weiss. Paradigms for structure in an amorphous computer. MIT AI Memo 1614.

[4] Jeremy Elson and Deborah Estrin. Time Synchronization for Wireless Sensor Networks. Proceedings of the 2001 International Parallel and Distributed Processing Symposium (IPDPS), Workshop on Parallel and Distributed Computing Issues in Wireless and Mobile Computing, San Francisco, California, USA. April 2001.

[5] Deborah Estrin, Mark Handley, Ahmed Helmy, Polly Huang, David Thaler, A Dynamic Bootstrap Mechanism for Rendezvous-based Multicast Routing, Technical Report USC CS TR97-644, University of Southern California, 1997

[6] L. Kleinrock and J. Silvester. Optimum transmission radii for packet radio networks or why six is a magic number. *Proc Natl. Telecomm. Conf.* pp 4.3.1-4.3.5, 1978

[7] Michael Luby. A Simple Parallel Algorithm for the Maximal Independent Set Problem SIAM J. on Computing, November 1986, Vol. 15, No. 4, pp. 1036-1053.

[8] Lynch, Nancy. Asynchronous Systems Model. In *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc, San Francisco, California, 1996. Chapter 8, pages 199-234.

[9] Radhika Nagpal. Organizing a Global Coordinate System from Local Information on an Amorphous Computer. MIT AI Memo 1999

[10] Radhika Nagpal and Daniel Coore. An Algorithm for Group Formation in an Amorphous Computer. Intl Conf on Parallel and Distributed Computing Systems (PDCS'98), 1998

[11] David G. Thaler and Chinya V. Ravishankar. Distributed Center-Location Algorithms. *IEEE J. Select. Areas in Commun.*, 15(3):291-303, April 1997.

| Name | Value | Description |
|---|---|---|
| $T_m$ | | message period |
| $T_k$ | $3T_m$ | time to kill |
| $W_0$ | $2T_m$ | time to wait for level zero |
| $W_m$ | 2 | multiplier for wait time at successive levels |
| $N_0$ | 1 | distance for neighbors at level zero |
| $N_m$ | 2 | multipliers for distance at successive levels |
| $R_G$ | 1 | group radius for multihop group variant |

Table 1: Constants used by the LeaderlessHierarchy algorithm.

[12] David G. Thaler and Chinya V. Ravishankar. Distributed top-down hierarchy construction. In Proc. of the IEEE INFOCOM, 1998.

[13] Paul F. Tsuchiya. Landmark routing: Architecture, algorithms and issues. Technical Report MTR-87W00174, MITRE Corporation, September 1987.

[14] Ya Xu, John Heidemann and Deborah Estrin. Adaptive Energy-Conserving Routing for Multihop Ad Hoc Networks. Research Report 527, USC/Information Sciences Institute, October 2000.

# A Constants and State

This appendix contains a summary of all the state information and constants used by the LeaderlessHierarchy algorithm.

## A.1 Constants

$T_m$, the fundamental time constant of the algorithm, is left for specification based on the characteristics of the physical network. $T_k$ is used for determining when a connection is considered lost. $N_0$ and $N_m$ determine the hopcount search radius for neighbors, and default to only immediate neighbors at the physical level, followed by a doubling of hopcount at subsequent levels. $R_G$ is used by the multi-hop variant of the algorithm to determine the maximum radius of a new clique. $W_0$ and $W_m$ are estimates derives from $N_0$, $N_m$, and $R_G$ for how long a wait is expected before a reasonable set of neighbors could

| Name | Description |
|---|---|
| $S$ | node state |
| $clock$ | node's internal clock |
| $lastSend$ | time of last message send |
| $uid$ | node ID |
| $L$ | list of level records |
| $t$ | start time for level |
| $I$ | node info record |
| $uid$ | group ID |
| $status$ | election state |
| $leaderID$ | next level group ID |
| $N$ | set of neighbor records |
| $I$ | node info record, as above |
| $hearsay$ | hopcount to information source |
| $lastHeard$ | last refresh on information |
| $dead$ | flag to indicate lost neighbor |
| $finalDead$ | flag to dispose lost neighbor |

Table 2: Node state for LEADERLESSHIERARCHY algorithm.

be found. They also act as a damper on transitory information and excessive level creation.

## A.2 State Information

The state $S$ of a node is composed of its unique ID number ($S.uid$), a continuously running clock ($S.clock$), a record of when it last sent a message ($S.lastSend$) and state information for each level of the hierarchy ($S.L$).

The state information $S.L$ is a list of records, one for each level in ascending order (so $S.L[0]$ is the bottom level), and each record contains level state information ($L.I$), level neighbor information ($L.N$), and the time at which the level's record was created ($L.t$).

A level state information record $L.I$ consists of three pieces of data: the ID of the group the node belongs to at this level ($I.uid$), the status of the group in the clique election process ($I.status$), and the clique member ID which it will take for its next level group ($I.leaderID$).

The level neighbor information $L.N$ is a set of neighbor information records, each of which holds five pieces of data: a state information record for the neighbor ($N.I$, which is formatted the same as a local state information record), how many hops away information about the neighbor originated ($N.hearsay$), a time at which the neighbor information was last refreshed ($N.lastHeard$), and two status flags ($N.dead$ and $N.finalDead$).

# B Multihop Groups Pseudocode

The following pseudocode can be substituted for the ELECTLEADER function to allow groups of radius $R_G$ neighborhoods to be built instead of groups of radius 1.

```
ELECTLEADER(S, i)
if LEADER?(S, i)
   then S.L[i].I.status ← "leader"
        S.L[i].I.leaderID ← S.L[i].uid
        S.L[i].I.leaderDistance ← 0
   else (id, dist) ← FOLLOWER?(S, i)
        if id ≠ NIL
           then S.L[i].I.status ← "follower"
                S.L[i].I.leaderID ← id
                S.L[i].I.leaderDistance ← dist
           else S.L[i].I.status ← "none"
                S.L[i].I.leaderID ← NIL
                S.L[i].I.leaderDistance ← NIL
```

**Pseudocode for ELECTLEADER variation with multihop groups**

```
LEADER?(S, i)
live ← false
for each nbr in S.L[i].N
   do if nbr.dead
         then return false
      live ← true
      if nbr.I.status = "follower"
         then if nbr.I.leaderID > S.L[i].I.uid and nbr.I.leaderDistance < R_G
                 then return false
         else if nbr.I.uid > S.L[i].I.uid
                 then return false
```

```
return live
```

**Pseudocode for multihop LEADER? subroutine**

```
FOLLOWER?(S, i)
best ← NIL
for each nbr in S.L[i].N
    do if nbr.dead
          then continue
       if nbr.I.status = "leader"
          then if (best = NIL or best.leaderDistance > 0 or
                   nbr.I.leaderID ≥ best.leaderID)
                   then best ← nbr.I
       if nbr.I.status = "follower"
          then if (nbr.I.leaderDistance < R_G and
                   (best = NIL or nbr.I.leaderDistance < best.leaderDistance or
                   (nbr.I.leaderDistance = best.leaderDistance and
                   nbr.I.leaderID ≥ best.leaderID)))
                   then best ← nbr.I
if best = NIL
   then return (NIL, NIL)
   else return (best.leaderID, best.leaderDistance + 1)
```

**Pseudocode for multihop FOLLOWER? subroutine**