IV. Proposed Protocol: HEACT

The proposed Hybrid Energy-Aware Cluster-Tree (HEACT) protocol aims to optimize network lifetime and stability in WSNs through an adaptive, hierarchical routing strategy. It combines periodic clustering with a dual-mode inter-cluster communication mechanism involving an initial direct transmission phase followed by multi-hop routing along an energy-aware tree. HEACT operates in rounds, with network reconfiguration occurring at predefined intervals.

4.1 Overview

HEACT divides the network operation into reconfiguration cycles. Within each cycle, Cluster Heads (CHs) are selected based on residual energy and distance metrics. Non-CH nodes form clusters around these CHs, subject to size limitations. Data transmission follows a two-phase approach depending on the network's operational stage: initially, CHs communicate directly with the Base Station (BS) to ensure stability; subsequently, they transition to using an energy-aware multi-hop tree constructed amongst themselves for improved long-term efficiency. The key components and parameters are detailed below and summarized in Table [Insert Table number for notations, similar to VCC Table 1].

Table X: Mathematical Notations and Definitions for HEACT

Notation	Definition
N	Total number of sensor nodes
Nodes	Set of all sensor nodes {n_1, ..., n_N}
BS	Base Station position coordinates
E_initial	Initial energy of each node
R	Current simulation round number
R_max	Maximum simulation rounds
I_recluster	Reconfiguration interval in rounds (HEACT_RECLUSTER_INTERVAL)
I_direct	Number of initial reconfiguration cycles using direct TX (HEACT_INITIAL_DIRECT_INTERVALS)
reconfig_count	Counter for completed reconfiguration cycles
P_ch	Target average CH percentage (HEACT_P_CH)
E_cand_thresh	Minimum energy for CH candidacy (HEACT_MIN_ENERGY_FOR_CH_CANDIDACY)
E_relay_thresh	Minimum energy for a CH to act as a relay in inter-CH tree (HEACT_MIN_ENERGY_FOR_CH_RELAY)
C_max_size	Maximum number of members per cluster (HEACT_MAX_CLUSTER_SIZE)
D_factor	Distance factor influencing CH selection probability

	(HEACT_DIST_FACTOR_CH_SELECTION)
P_exp	Exponent for energy penalty in inter-CH tree cost calculation (HEACT_TREE_ENERGY_PENALTY_EXPONENT)
n_i	Sensor node i
E_i	Current residual energy of node i
Dist_i_BS	Distance from node i to the BS
CH_Set	Set of nodes currently selected as Cluster Heads
Clusters	Dictionary mapping ch_id to list of member_ids
n_i.parent_ch_id	ID of the parent CH for CH i in the inter-CH tree ("BS" if root)
n_i.children_ch_ids	Set of child CH IDs for CH i in the inter-CH tree
n_i.is_relay_ch	Boolean indicating if CH i can relay for other CHs
n_i.path_cost_sq_ch	Accumulated (potentially weighted) cost from CH i to BS via tree
T(i)	Calculated probability threshold for node i becoming a CH
E_Tx(m, d), E_Rx(m), E_DA(m)	Energy for transmit, receive, aggregate (from Section III)
PACKET_SIZE_DATA, PACKET_SIZE_CTRL	Packet sizes in bits

The core adaptive mechanism of HEACT relies on periodic reconfiguration, detailed in Algorithm 1. This process is triggered at the start of round 1 and subsequently every I_recluster rounds.

- Initialization: The reconfiguration counter (reconfig_count) is incremented. The algorithm determines if the network is still within the initial direct transmission phase (is_initial_phase) based on whether reconfig_count exceeds I_direct. Node roles and cluster assignments from the previous cycle are reset.
- CH Selection: The refined CH selection process (Algorithm 2, select_heact_chs_refined) is invoked to elect CHs for the current cycle based on energy share and distance factor.
- Cluster Formation: Non-CH nodes attempt to join the nearest available (alive and non-full) CH using Algorithm 3 (form_heact_clusters), respecting the C_max_size limit. Energy consumed during the join handshake is accounted for.
- Inter-CH Tree Building: If the network is *beyond* the initial direct phase (NOT is_initial_phase), the energy-penalized tree building process (Algorithm 4, build_inter_cluster_tree_heact_further_revised) is executed among the currently alive CHs. This step establishes the multi-hop paths for the subsequent steady-state phase. A flag (build_tree_this_cycle) indicates whether the tree was successfully constructed.

Algorithm 1. Pseudocode for HEACT Main Loop

Input: Set of nodes Nodes, BS position BS, Max rounds R_max, Reconfiguration interval I_recluster, Initial direct transmission intervals I_direct, CH selection params (P_ch, E_cand_thresh, D_factor), Cluster params (C_max_size), Tree params (E_relay_thresh, P_exp).
Output: Simulation statistics Stats (including FDN, NL, Total Packets, etc.).

```
Initialize Nodes (energy, status, etc.)
round_num ← 0
reconfig_count ← 0
CH_Set ← Ø
Clusters ← Ø
Stats ← Initialize statistics dictionary
build_tree_this_cycle ← False

while round_num < R_max and count(alive nodes) > 0 do
round_num ← round_num + 1
num_alive_before ← count(alive nodes)
round_energy_consumed ← map(node_id → 0.0)
round_packets_to_bs ← 0

// --- Reconfiguration Phase ---
if (round_num - 1) mod I_recluster == 0 or round_num == 1 then
reconfig_count ← reconfig_count + 1
is_initial_phase ← (reconfig_count <= I_direct)
build_tree_this_cycle ← False // Reset flag for this cycle
Setup_Energy ← Call HEACT-SETUP(Nodes, P_ch, E_cand_thresh, D_factor, C_max_size, BS, E_relay_thresh, P_exp, is_initial_phase, CH_Set, Clusters) // Pass CH_Set & Clusters to be updated
Update round_energy_consumed with Setup_Energy
// Determine if tree was built in this setup phase
if not is_initial_phase and {ch ∈ CH_Set such that ch.is_alive()} is not empty then
build_tree_this_cycle ← True
end if
end if

// --- Steady-State Phase ---
use_tree ← (reconfig_count > I_direct) and build_tree_this_cycle
Live_CHs_Steady ← {ch ∈ CH_Set such that ch.is_alive()}
if Live_CHs_Steady is not empty then
Steady_Energy, Steady_Packets ← Call HEACT-STEADY-STATE(Nodes, Live_CHs_Steady, Clusters, use_tree)
Update round_energy_consumed with Steady_Energy
round_packets_to_bs ← Steady_Packets
end if

// --- Update Stats & Node Status ---
Call UPDATE-STATS-AND-STATUS(Nodes, Stats, round_num, num_alive_before, round_packets_to_bs)
if count(alive nodes) == 0 then break end if // Exit if all nodes died
```

40: **end while**
41: Finalize Stats['all_dead'] if needed
42: **Return** Stats

Procedure 1. HEACT-SETUP

Input: Nodes, P_ch, E_cand_thresh, D_factor, C_max_size, BS, E_relay_thresh, P_exp, is_initial_phase.
Input/Output: CH_Set, Clusters (modified by reference or globally).
Output: Energy consumed during setup Setup_Energy.

1: Setup_Energy ← map(node_id → 0.0)
2: *// Reset roles*
3: **for each** node n ∈ Nodes **such that** n.is_alive() **do** n.reset_heact_round_state() **end for**
4:
5: *// 1. Select Cluster Heads*
6: Current_CH_Set, Select_Energy ← Call **Algorithm 2: Select_HEACT_CHs_Refined**(Nodes, P_ch, E_cand_thresh, D_factor)
7: Update Setup_Energy with Select_Energy
8: CH_Set ← Current_CH_Set // Update the main CH set
9:
10: *// 2. Form Clusters*
11: Live_CHs_Selected ← {ch ∈ CH_Set **such that** ch.is_alive()}
12: **if** Live_CHs_Selected is not empty **then**
13: Current_Clusters, Form_Energy ← Call **Algorithm 3: Form_HEACT_Clusters**(Nodes, Live_CHs_Selected, C_max_size)
14: Update Setup_Energy with Form_Energy
15: Clusters ← Current_Clusters // Update the main cluster assignments
16: **else**
17: Clusters ← ∅
18: **end if**
19:
20: *// 3. Build Inter-CH Tree (conditionally)*
21: **if not** is_initial_phase **then**
22: Live_CHs_For_Tree ← {ch ∈ CH_Set **such that** ch.is_alive()}
23: **if** Live_CHs_For_Tree is not empty **then**
24: Call **Algorithm 4: Build_InterCH_Tree_HEACT_Revised**(Live_CHs_For_Tree, BS, E_relay_thresh, P_exp)
25: **end if**
26: **end if**
27: **Return** Setup_Energy

Algorithm 2: Select_HEACT_CHs_Refined

Input: Nodes, P_ch, E_cand_thresh, D_factor.
Output: Set CH_Set, Energy map Select_Energy.

```
CH_Set ← Ø
Select_Energy ← map(node_id → 0.0)
Alive_Nodes ← {n ∈ Nodes such that n.is_alive()}
N_alive ← |Alive_Nodes|
E_total ← Σ n.energy for all n ∈ Alive_Nodes
if E_total <= 0 or N_alive == 0 then return CH_Set, Select_Energy end if
Avg_Dist_BS ← Mean(n.dist_to_bs for n ∈ Alive_Nodes)
Avg_Dist_BS ← max(1.0, Avg_Dist_BS)
for each node n ∈ Alive_Nodes do
if n.energy < E_cand_thresh then continue end if
BaseProb ← P_ch * (N_alive * n.energy / E_total)
DistModifier ← 1.0
RelDist ← n.dist_to_bs / Avg_Dist_BS
MaxRatio ← 2.0
if RelDist >= 1.0 then
scale ← min(1.0, (RelDist - 1.0) / max(1e-6, MaxRatio - 1.0))
DistModifier ← 1.0 + (D_factor - 1.0) * scale
else
scale ← 1.0 - RelDist
DistModifier ← (1.0 / D_factor) + (1.0 - 1.0 / D_factor) * (1.0 - scale)
end if
Threshold ← max(0, min(1.0, BaseProb * DistModifier))
rand_num ← random number in [0, 1)
if rand_num < Threshold then
n.is_ch ← True; n.role ← CH; n.cluster_id ← n.id
CH_Set ← CH_Set ∪ {n}
end if
end for
if CH_Set is empty and Alive_Nodes is not empty then
Eligible_Fallback ← {n ∈ Alive_Nodes such that n.energy >= E_cand_thresh}
if Eligible_Fallback is not empty then
best_fallback ← node in Eligible_Fallback with maximum energy
best_fallback.is_ch ← True; best_fallback.role ← CH; best_fallback.cluster_id ← best_fallback.id
CH_Set ← CH_Set ∪ {best_fallback}
end if
end if
Return CH_Set, Select_Energy
```

Algorithm 3: Form_HEACT_Clusters

Input: Nodes, CH_Set, C_max_size.
Output: Dictionary Clusters, Energy map Form_Energy.

```
Form_Energy ← map(node_id → 0.0)
```

Alive_Members ← {n ∈ Nodes **such that** n.is_alive() **and** NOT n.is_ch}
Live_CH_Map ← {ch.id: ch **for each** ch ∈ CH_Set **such that** ch.is_alive()}
Clusters ← {ch_id: [] **for each** ch_id ∈ Live_CH_Map}
Cluster_Sizes ← {ch_id: 0 **for each** ch_id ∈ Live_CH_Map}
if Live_CH_Map is empty **then return** Clusters, Form_Energy **end if**

Shuffle(Alive_Members)

for each member ∈ Alive_Members **do**
member.cluster_id ← -1; best_ch_id ← -1; min_dist_sq ← ∞
Eligible_CHs ← {ch **for each** ch_id, ch ∈ Live_CH_Map **such that** ch.is_alive() **and** Cluster_Sizes[ch_id] < C_max_size}
if Eligible_CHs is empty **then continue end if**
for each ch ∈ Eligible_CHs **do**
dist_sq ← calculate_distance(member.position, ch.position)^2
if dist_sq < min_dist_sq **then** min_dist_sq ← dist_sq; best_ch_id ← ch.id **end if**
end for
if best_ch_id != -1 **then**
target_ch ← Live_CH_Map[best_ch_id]; dist_to_ch ← sqrt(min_dist_sq)
e_tx_join ← calculate_transmit_energy(PACKET_SIZE_CTRL, dist_to_ch)
if member.energy >= e_tx_join **then**
member.energy ← member.energy - e_tx_join; Form_Energy[member.id] ← Form_Energy.get(member.id, 0) + e_tx_join
e_rx_join ← calculate_receive_energy(PACKET_SIZE_CTRL)
if target_ch.is_alive() **and** target_ch.energy >= e_rx_join **then**
target_ch.energy ← target_ch.energy - e_rx_join; Form_Energy[target_ch.id] ← Form_Energy.get(target_ch.id, 0) + e_rx_join
member.cluster_id ← best_ch_id; Append member.id to Clusters[best_ch_id]; Cluster_Sizes[best_ch_id] ← Cluster_Sizes[best_ch_id] + 1
else if target_ch.is_alive() **then** Form_Energy[target_ch.id] ← Form_Energy.get(target_ch.id, 0) + target_ch.energy; target_ch.energy ← 0; target_ch.status ← "dead"; **end if**
else Form_Energy[member.id] ← Form_Energy.get(member.id, 0) + member.energy; member.energy ← 0; member.status ← "dead"; **end if**
end if
end for
Return Clusters, Form_Energy

Algorithm 4: Build_InterCH_Tree_HEACT_Revised

Input: Live_CHs, BS, E_relay_thresh, P_exp.
Output: Modifies CH node attributes.

if Live_CHs is empty **then return end if**

for each ch ∈ Live_CHs **do** ch.parent_ch_id ← None; ch.children_ch_ids ← ∅; ch.is_relay_ch ← False; ch.path_cost_sq_ch ← ∞ **end for**

Sorted_CHs ← Sort Live_CHs by increasing dist_to_bs
Visited_CH_IDs ← ∅; Root_CHs ← ∅
Potential_Relay_CHs ← {ch ∈ Sorted_CHs **such that** ch.energy >= E_relay_thresh}
primary_root ← First element in Potential_Relay_CHs (if exists), else None
if primary_root is not None **then**
primary_root.parent_ch_id ← "BS"; primary_root.path_cost_sq_ch ← primary_root.dist_to_bs^2
primary_root.is_relay_ch ← True; Add primary_root to Root_CHs; Add primary_root.id to Visited_CH_IDs
else if Live_CHs is not empty **then**
closest_ch ← Sorted_CHs[0]; closest_ch.parent_ch_id ← "BS"; closest_ch.path_cost_sq_ch ← closest_ch.dist_to_bs^2
closest_ch.is_relay_ch ← False; Add closest_ch to Root_CHs; Add closest_ch.id to Visited_CH_IDs
end if
if Root_CHs is empty **then return end if**
Unvisited_CHs ← {ch ∈ Live_CHs **such that** ch.id ∉ Visited_CH_IDs}
Potential_Parents_Map ← {ch.id: ch **for each** ch ∈ Root_CHs **such that** ch.is_relay_ch}
while Unvisited_CHs is not empty **do**
best_next_ch ← None; best_parent_ch_id ← None; min_cost ← ∞
possible_parent_nodes ← list of values in Potential_Parents_Map
if possible_parent_nodes is empty **then break end if**
found_parent_this_iter ← False
for each q_ch ∈ Unvisited_CHs **do**
for each i_ch ∈ possible_parent_nodes **do**
dist_qi ← calculate_distance(q_ch.position, i_ch.position)
energy_ratio ← max(0.01, i_ch.energy / i_ch.initial_energy)
penalty ← (1.0 / energy_ratio) ** P_exp
current_cost ← (dist_qi^2 + i_ch.path_cost_sq_ch) * penalty
if current_cost < min_cost **then**
min_cost ← current_cost; best_next_ch ← q_ch; best_parent_ch_id ← i_ch.id; found_parent_this_iter ← True
end if
end for
end for
if found_parent_this_iter **then**
q ← best_next_ch; i_parent_ch ← Potential_Parents_Map[best_parent_ch_id]
q.parent_ch_id ← i_parent_ch.id; Add q.id to i_parent_ch.children_ch_ids
q.path_cost_sq_ch ← min_cost
q.is_relay_ch ← (q.energy >= E_relay_thresh)
if q.is_relay_ch **then** Potential_Parents_Map[q.id] ← q **end if**
Add q.id to Visited_CH_IDs; Remove q from Unvisited_CHs
else break end if
end while

Algorithm 5: Get_CH_Traversal_Order_HEACT

Input: Set Cluster_Heads.
Output: Ordered list Order of CH IDs.

Order ← []
Processed_IDs ← ∅
Live_CHs_Map ← {ch.id: ch **for each** ch ∈ Cluster_Heads **such that** ch.is_alive() **and** ch.parent_ch_id is not None}
if Live_CHs_Map is empty **then return** Order **end if**

Parent_Map ← {id: ch.parent_ch_id **for** id, ch ∈ Live_CHs_Map}
Children_Map ← {id: ∅ **for** id ∈ Live_CHs_Map}
for each child_id, parent_id ∈ Parent_Map **do**
if parent_id ∈ Children_Map **then** Add child_id to Children_Map[parent_id] **end if**
end for

Leaves ← {id **for** id ∈ Live_CHs_Map **such that** id ∉ Children_Map **or** Children_Map[id] is empty}
Queue ← Sorted list of Leaves

while Queue is not empty **do**
ch_id ← Remove first element from Queue
if ch_id ∈ Processed_IDs **then continue end if**
Append ch_id to Order
Processed_IDs ← Processed_IDs ∪ {ch_id}
parent_id ← Parent_Map.get(ch_id)
if parent_id **and** parent_id != "BS" **and** parent_id ∈ Live_CHs_Map **then**
all_children_done ← True
for each child_id ∈ Children_Map.get(parent_id, ∅) **do**
if child_id ∉ Processed_IDs **then** all_children_done ← False; **break**; **end if**
end for
if all_children_done **and** parent_id ∉ Processed_IDs **and** parent_id ∉ Queue **then**
Add parent_id to Queue; Sort Queue
end if
end if
end while

Remaining ← Sorted list of {id ∈ Live_CHs_Map **such that** id ∉ Processed_IDs}
Append elements of Remaining to Order
Return Order

Procedure 2. HEACT-STEADY-STATE

Input: Nodes, Live_CHs_Steady, Clusters, Flag use_tree_phase.
Output: Energy consumed Steady_Energy, Packets delivered to BS Packets_To_BS.
Steady_Energy ← map(node_id → 0.0)
Packets_To_BS ← 0
Nodes_Map ← map node id to node object

```
CH_Map ← {ch.id: ch for each ch ∈ Live_CHs_Steady}
CH_Packets_Aggregated ← map(ch_id → 0)

// Phase 1: Member to CH Transmission
for each ch_id, member_ids ∈ Clusters do
ch_node ← CH_Map.get(ch_id)
if ch_node and ch_node.is_alive() then
members_sent_count ← 0
for each member_id ∈ member_ids do
member_node ← Nodes_Map.get(member_id)
if member_node and member_node.is_alive() and member_node.cluster_id == ch_id then
if not ch_node.is_alive() then break end if
dist ← calculate_distance(member_node.position, ch_node.position)
e_tx ← calculate_transmit_energy(PACKET_SIZE_DATA, dist)
if member_node.energy >= e_tx then
member_node.energy ← member_node.energy - e_tx
Steady_Energy[member_id] ← Steady_Energy.get(member_id, 0) + e_tx
e_rx ← calculate_receive_energy(PACKET_SIZE_DATA)
if ch_node.energy >= e_rx then
ch_node.energy ← ch_node.energy - e_rx
Steady_Energy[ch_id] ← Steady_Energy.get(ch_id, 0) + e_rx
members_sent_count ← members_sent_count + 1
else Steady_Energy[ch_id] ← Steady_Energy.get(ch_id, 0) + ch_node.energy; ch_node.energy ← 0; ch_node.status ← "dead"; break end if
else Steady_Energy[member_id] ← Steady_Energy.get(member_id, 0) + member_node.energy; member_node.energy ← 0; member_node.status ← "dead" end if
end if
end for
if ch_node.is_alive() then CH_Packets_Aggregated[ch_id] ← members_sent_count end if
end if
end for

// Phase 2: CH Aggregation
for each ch_id, ch_node ∈ CH_Map do
if ch_node.is_alive() then
num_pkts_from_members ← CH_Packets_Aggregated[ch_id]
if num_pkts_from_members > 0 then
e_da ← calculate_aggregate_energy(PACKET_SIZE_DATA * num_pkts_from_members)
if ch_node.energy >= e_da then ch_node.energy ← ch_node.energy - e_da; Steady_Energy[ch_id] ← Steady_Energy.get(ch_id, 0) + e_da
else Steady_Energy[ch_id] ← Steady_Energy.get(ch_id, 0) + ch_node.energy; ch_node.energy ← 0; ch_node.status ← "dead" end if
end if
if ch_node.is_alive() then CH_Packets_Aggregated[ch_id] ← CH_Packets_Aggregated[ch_id] + 1 end if
end if
end for

```

47: *// Phase 3: CH Transmission*
48: **if not** use_tree_phase **then**
49: *// Direct Transmission*
50: **for each** ch_id, ch_node ∈ CH_Map **do**
51: **if** ch_node.is_alive() **then**
52: total_pkts ← CH_Packets_Aggregated[ch_id]; **if** total_pkts == 0 **then continue end if**
53: bits ← PACKET_SIZE_DATA * total_pkts; dist ← ch_node.dist_to_bs; e_tx ← calculate_transmit_energy(bits, dist)
54: **if** ch_node.energy >= e_tx **then** ch_node.energy ← ch_node.energy - e_tx; Steady_Energy[ch_id] ← Steady_Energy.get(ch_id, 0) + e_tx; Packets_To_BS ← Packets_To_BS + total_pkts
55: **else** Steady_Energy[ch_id] ← Steady_Energy.get(ch_id, 0) + ch_node.energy; ch_node.energy ← 0; ch_node.status ← "dead" **end if**
56: **end if**
57: **end for**
58: **else**
59: *// Tree Transmission*
60: Order ← **Get_CH_Traversal_Order_HEACT**(Live_CHs_Steady)
61: **for each** ch_id ∈ Order **do**
62: ch_node ← CH_Map.get(ch_id)
63: **if** ch_node **and** ch_node.is_alive() **then**
64: parent_id ← ch_node.parent_ch_id; **if** parent_id is None **then continue end if**
65: total_pkts ← CH_Packets_Aggregated[ch_id]; **if** total_pkts == 0 **then continue end if**
66: bits ← PACKET_SIZE_DATA * total_pkts; parent_alive ← False; distance ← 0; parent_node ← None
67: **if** parent_id == "BS" **then** distance ← ch_node.dist_to_bs; parent_alive ← True
68: **else** parent_node ← CH_Map.get(parent_id); **if** parent_node **and** parent_node.is_alive() **then** distance ← calculate_distance(ch_node.position, parent_node.position); parent_alive ← True **end if**
69: **if** parent_alive **then**
70: e_tx ← calculate_transmit_energy(bits, distance)
71: **if** ch_node.energy >= e_tx **then**
72: ch_node.energy ← ch_node.energy - e_tx; Steady_Energy[ch_id] ← Steady_Energy.get(ch_id, 0) + e_tx
73: **if** parent_id == "BS" **then** Packets_To_BS ← Packets_To_BS + total_pkts
74: **else if** parent_node **then**
75: e_rx ← calculate_receive_energy(bits)
76: **if** parent_node.energy >= e_rx **then**
77: parent_node.energy ← parent_node.energy - e_rx; Steady_Energy[parent_id] ← Steady_Energy.get(parent_id, 0) + e_rx
78: CH_Packets_Aggregated[parent_id] ← CH_Packets_Aggregated.get(parent_id, 0) + total_pkts
79: **else** Steady_Energy[parent_id] ← Steady_Energy.get(parent_id, 0) + parent_node.energy; parent_node.energy ← 0; parent_node.status ← "dead" **end if**
80: **end if**
81: **else** Steady_Energy[ch_id] ← Steady_Energy.get(ch_id, 0) + ch_node.energy; ch_node.energy ← 0; ch_node.status ← "dead" **end if**
82: **end if**
83: **end if**
84: **end for**

end if
Return Steady_Energy, Packets_To_BS

Procedure 3. UPDATE-STATS-AND-STATUS

Input: Nodes, Stats dictionary, round_num, num_alive_before, round_packets_to_bs.
Output: Updates Stats dictionary. Modifies node status.

// Final death check for the round
for each node n ∈ Nodes **do**
if n.energy <= 0 **and** n.status == "alive" **then**
n.status ← "dead"; n.energy ← 0
end if
end for

// Record statistics
num_alive_now ← count({n ∈ Nodes **such that** n.is_alive()})
Append num_alive_now to Stats['alive_nodes']
Append round_packets_to_bs to Stats['packets_to_bs']
total_energy ← sum(n.energy for n ∈ Nodes if n.is_alive())
Append total_energy to Stats['total_energy']

// Update FDN and NLD
if num_alive_now < NUM_NODES **and** Stats['first_dead'] == -1 **then**
Stats['first_dead'] ← round_num
end if
if num_alive_now == 0 **and** Stats['all_dead'] == -1 **then**
Stats['all_dead'] ← round_num
end if

V. Illustration of HEACT Operation

To elucidate the operational flow of the proposed HEACT protocol, let us consider a small example network scenario.

Parameter	Value	Description
N	10	Number of sensor nodes
BS	(50, 150)	Base Station coordinates
INITIAL_ENERGY	0.5 J	Initial energy per node
I_recluster	3 rounds	Reclustering interval
I_direct	1 cycle	Direct transmission interval
P_ch	0.2	Probability of a node becoming a Cluster Head
E_cand_thresh	0.075 J	Minimum energy required to become a Cluster Head candidate

E_relay_thresh	0.15 J	Minimum energy required to act as a relay Cluster Head
C_max_size	7	Maximum allowable cluster size
D_factor	1.1	Distance scaling factor for cluster joining decisions
P_exp	1.5	Power exponent for cost calculation in tree construction

The node positions are as used in the previous dry run: N0(10,10), N1(30,80), N2(50,120), N3(80,20), N4(60,90), N5(90,110), N6(20,140), N7(40,30), N8(70,130), N9(50,50).

Table XI: Specifications of Example Nodes (Initial State)

Node ID	Position (x, y)	Initial Energy (J)	Dist to BS (m)
N0	(10, 10)	0.5	~145.6
N1	(30, 80)	0.5	~80.6
N2	(50, 120)	0.5	30.0
N3	(80, 20)	0.5	~133.4
N4	(60, 90)	0.5	~60.8
N5	(90, 110)	0.5	~56.6
N6	(20, 140)	0.5	~31.6
N7	(40, 30)	0.5	~120.4
N8	(70, 130)	0.5	~28.3
N9	(50, 50)	0.5	100.0

A. Reconfiguration Cycle 1 (Round 1)

1. **Trigger:** (1-1) % 3 == 0. Reconfiguration occurs. reconfig_count becomes 1.
2. **Phase Check:** reconfig_count (1) <= I_direct (1). This is the **Initial Direct Phase**. The inter-CH tree will *not* be built.
3. **CH Selection (Algorithm 2):** All nodes have 0.5 J, exceeding E_cand_thresh. Probabilities T(i) are calculated based on energy share (equal initially) and the distance factor. Nodes N0, N3, N7, N9 (farther) will have slightly higher chances than average (e.g., >20%), while N2, N6, N8 (closer) will have slightly lower chances (e.g., <20%). Let's assume, based on random draw against these probabilities, **N3** and **N6** are elected as CHs.
4. **Cluster Formation (Algorithm 3):**
    - Alive CHs: N3 (80, 20), N6 (20, 140). C_max_size = 7.
    - Members {N0, N1, N2, N4, N5, N7, N8, N9} join nearest CH:
        - N0(10,10) → N3 (closer than N6)
        - N1(30,80) → N6

        - N2(50,120) → N6
        - N4(60,90) → N6
        - N5(90,110) → N6 (closer than N3)
        - N7(40,30) → N3
        - N8(70,130) → N6
        - N9(50,50) → N3
    - Control packets are exchanged, energy deducted. Assume successful.
    - **Table XII: Cluster Formation - Round 1**

Cluster Head	Members	Size
N3 (80, 20)	{N0, N7, N9}	3
N6 (20, 140)	{N1, N2, N4, N5, N8}	5

5. **Tree Building:** Skipped (Initial Phase).

B. Steady State (Rounds 1, 2, 3)

- **Data Flow:** In each round:
    - Nodes {N0, N7, N9} transmit data to CH N3.
    - Nodes {N1, N2, N4, N5, N8} transmit data to CH N6.
    - CH N3 aggregates 3 member packets + its own, transmits directly to BS (distance ~133m, high cost - likely multipath).
    - CH N6 aggregates 5 member packets + its own, transmits directly to BS (distance ~32m, low cost - likely free space).
- **Energy Depletion:** CH N3 consumes significantly more energy per round due to its long transmission distance compared to CH N6. Member energy decreases based on distance to their CH.

C. Reconfiguration Cycle 2 (Round 4)

1. **Trigger:** (4-1) % 3 == 0. Reconfiguration occurs. reconfig_count becomes 2.
2. **Phase Check:** reconfig_count (2) > I_direct (1). This is now the **Tree Phase**. The inter-CH tree *will* be built.
3. **CH Selection (Algorithm 2):** All nodes' roles are reset. Probabilities T(i) are recalculated. N3 likely has much lower energy than N6 and other nodes. Assume nodes **N1** (Mid-dist, high energy) and **N8** (Close, high energy) are now selected as CHs.
4. **Cluster Formation (Algorithm 3):**
    - Alive CHs: N1 (30, 80), N8 (70, 130). C_max_size = 7.

- Members {N0, N2, N3, N4, N5, N6, N7, N9} (assuming N3 survived) join nearest CH.
- N0→N1, N2→N8, N3→N1, N4→N1, N5→N8, N6→N8, N7→N1, N9→N1 (example assignments).
- **Table XIII: Cluster Formation - Round 4**

Cluster Head	Members	Size
N1 (30, 80)	{N0, N3, N4, N7, N9}	5
N8 (70, 130)	{N2, N5, N6}	3

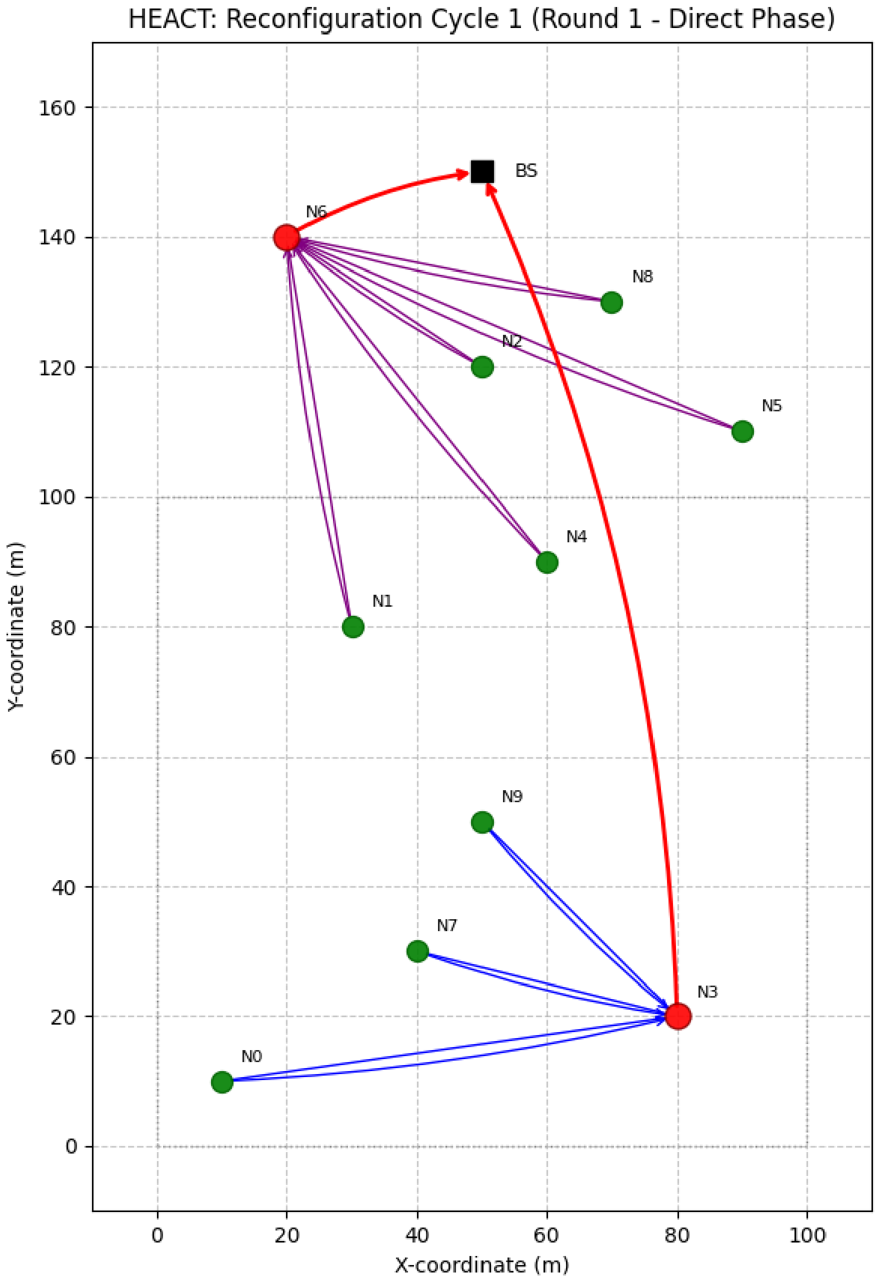
5. **Tree Building (Algorithm 4):**
   - Live_CHs = {N1, N8}. Assume both have energy > E_relay_thresh (0.15 J).
   - Sort by distance to BS: N8 (~28.3m) is closer than N1 (~80.6m).
   - Root Selection: N8 is the closest eligible relay. N8.parent_ch_id = "BS", N8.path_cost_sq_ch = N8.dist_to_bs^2 ≈ 800, N8.is_relay_ch = True. Visited = {N8}. Root = {N8}.
   - Unvisited = {N1}. Potential_Parents = {N8}.
   - Consider adding N1 via parent N8:
     - dist(N1, N8) = calculate_distance(30,80, 70,130) = sqrt((-40)^2 + (-50)^2) = sqrt(1600+2500) = sqrt(4100) ≈ 64.0m.
     - Assume N8.energy ≈ 0.48 J. EnergyRatio(N8) = 0.48/0.5 = 0.96.
     - Penalty(N8) = (1/0.96)**1.5 ≈ 1.06.
     - Cost(N1, N8) = (dist(N1,N8)^2 + N8.path_cost_sq_ch) * Penalty(N8)
     - Cost ≈ (64.0^2 + 800) * 1.06 ≈ (4096 + 800) * 1.06 ≈ 4896 * 1.06 ≈ 5190.
   - N1 joins under N8. N1.parent_ch_id = N8, N8.children_ch_ids = {N1}, N1.path_cost_sq_ch = 5190. N1.is_relay_ch = True.
   - **Final Tree:** BS ← N8 ← N1

D. Steady State (Round 4 onwards, until next reconfig)

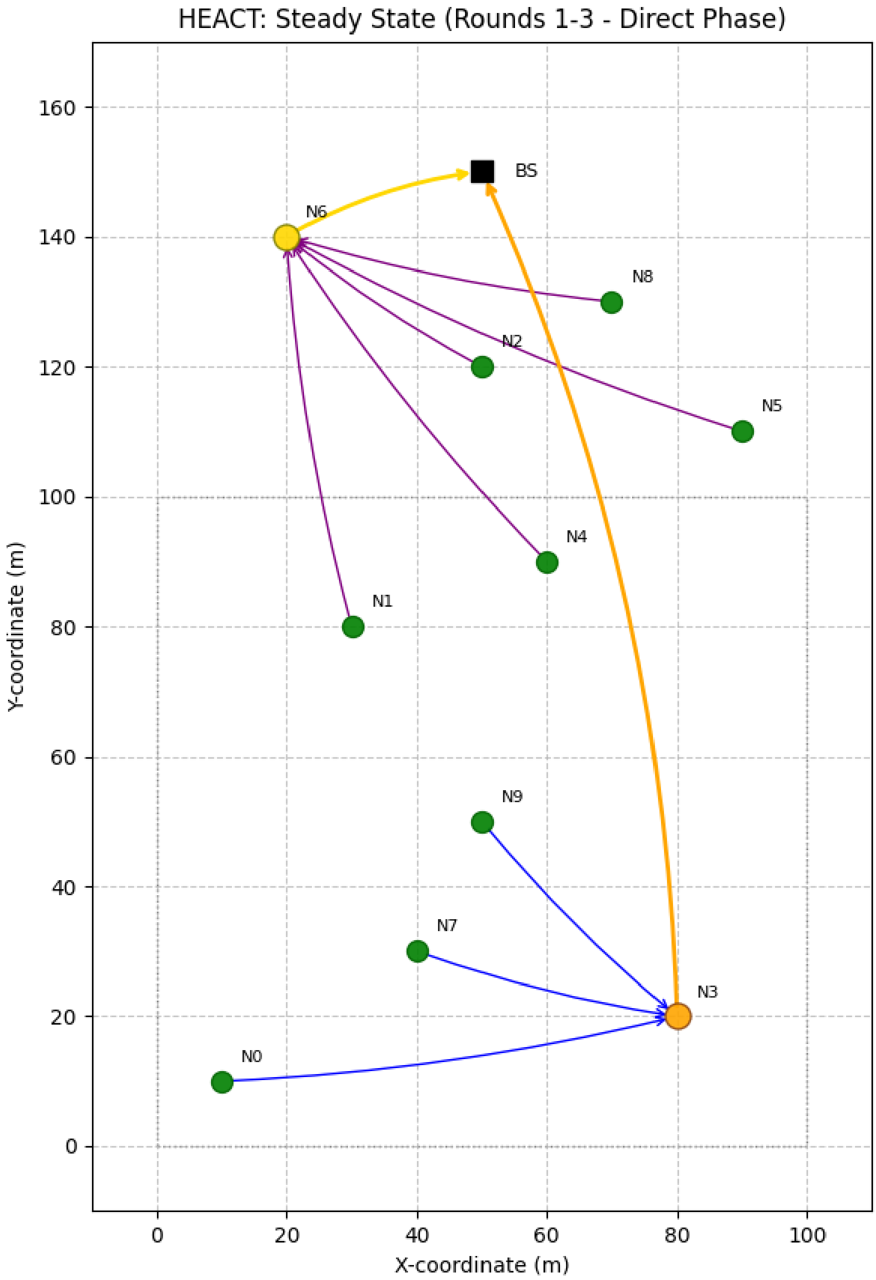
- **Data Flow:**
  - Members {N0, N3, N4, N7, N9} transmit to CH N1.
  - Members {N2, N5, N6} transmit to CH N8.
  - CH N1 aggregates (5 member + 1 self = 6 packets), transmits to parent CH N8.
  - CH N8 receives from N1 (6 packets), aggregates with its own cluster's data (3 member + 1 self = 4 packets). Total = 10 packets.

  - CH N8 transmits the final 10 aggregated packets directly to BS.

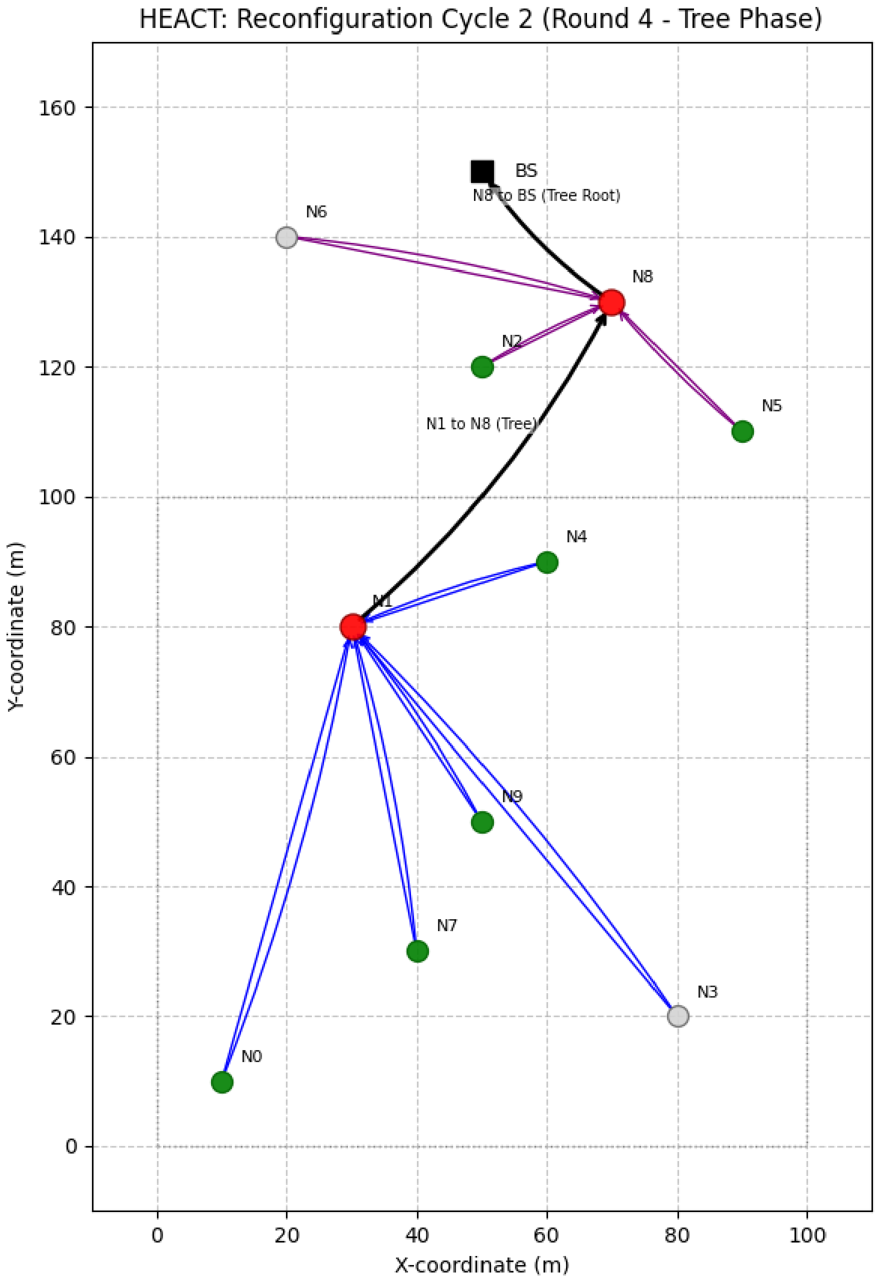
- **Energy Depletion:** N1 consumes energy for receiving 5 packets, aggregation, and transmitting 6 packets to N8 (distance ~64m). N8 consumes energy for receiving 3 member packets, receiving 6 packets from N1, aggregation, and transmitting 10 packets to BS (distance ~28m). N8 bears the higher load as the root CH.



Reconfiguration Cycle 1 (Round 1): Direct Phase. CHs: N3, N6.
Members join nearest CH. CHs transmit aggregated data directly to BS.



Steady State (Rounds 1-3): Direct Phase. CHs N3 & N6 continue direct BS transmission.
N3 (orange) depletes energy faster due to longer transmission distance than N6 (gold).



Reconfiguration Cycle 2 (Round 4): Tree Phase. New CHs: N1, N8.
Former CHs N3, N6 become members. Inter-CH Tree: N1 → N8 → BS.