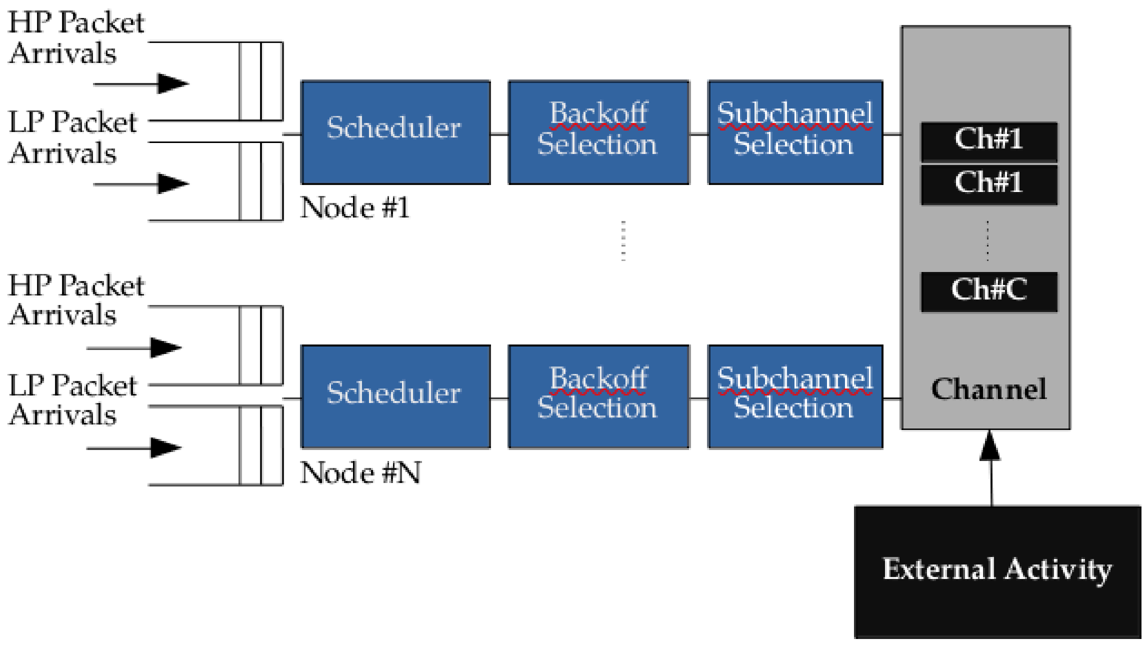
WN sim

Overview 📡

The WN simulator models a wireless network where multiple **nodes** compete to transmit data packets over a set of shared communication **channels**. The simulation runs over discrete **time slots**. It's designed as a baseline system where nodes follow fixed rules, without any intelligent learning agents.



Node Components 💻

Each node in the simulation represents a device that wants to send data. Its key components are:

1. **Packet Queues:** Each node has two separate buffers (queues) to hold outgoing packets:
    - `packet_queue_high`: Stores **High Priority** packets (e.g., voice, video).

    - packet_queue_low: Stores **Low Priority** packets (e.g., file transfer, email). Both queues have a maximum size (Q_LIMIT). If a packet arrives when its corresponding queue is full, it's considered **dropped**.
2. **Backoff Timer (backoff_timer):** This is a counter that implements a simple CSMA/CA-like mechanism.
    - A node **cannot** attempt to transmit if its backoff_timer is greater than 0.
    - The timer is decremented by 1 in each time slot.
    - When the timer reaches 0, the node can potentially transmit.
3. **Statistics Trackers:** Each node keeps track of:
    - stats: Performance metrics like packets sent, received, dropped, and average latency, separated for High and Low priority traffic.
    - tx_stats: Records per channel how many transmission attempts resulted in success or collision.
    - learning_stats: Simulates passive sensing. It records the observed state (Idle, Success, Collision, External Interference) of channels the node "listens" to when it's not transmitting.

External Interference 🌩️

- The simulation models **external noise** or interference independently on each channel.
- The GAMMA array defines the probability of external interference for each channel j. For example, GAMMA[j] = 0.7 means channel j has a 70% chance of being unusable due to external noise in any given time slot.
- This interference is generated randomly in each slot based on the GAMMA probabilities.

Channel Access and Interaction 🚦

Nodes follow a fixed protocol to access the channel:

1. **Packet Arrival:** Nodes receive High and Low priority packets randomly based on their individual arrival rates (P_ARRIVALS_HIGH, P_ARRIVALS_LOW).
2. **Backoff Check:** In each slot, a node first checks its backoff_timer. If it's greater than 0, it decrements the timer and does nothing else.
3. **Transmission Decision (if backoff_timer == 0):**
    - **Strict Priority:** The node checks if its packet_queue_high has packets. If yes, it decides to send a High Priority packet. If not, it checks packet_queue_low. If both are empty, it does nothing.

- **Set New Backoff:** If the node decides to send a packet (of either priority), it immediately sets a *new* random backoff timer for its *next* attempt. The timer value is chosen uniformly from the range `[0, FIXED_BACKOFF_WINDOW]`.
- **Attempt Transmission (if new backoff is 0):** Only if the *newly chosen* backoff timer is 0 does the node attempt to transmit in the *current* slot.
- **Random Channel Selection:** If transmitting, the node picks one of the `C_CHANNELS` **uniformly at random**.
- **Record Attempt:** The node's ID is added to the list of transmissions for the chosen channel in that slot.

Transmission Outcomes ✅💥

At the end of each slot, the simulation resolves the outcome for each channel based on who attempted to transmit and the external interference:

1. **Idle:** No nodes attempted to transmit on the channel, and there was no external interference. The channel was free.
2. **External Interference Only:** No nodes attempted to transmit, but external interference occurred. The channel was busy due to noise.
3. **Successful Transmission: Exactly one** node transmitted on the channel, AND there was **no** external interference.
   - The node removes the packet from its queue.
   - Latency and throughput statistics are updated.
   - The global `total_successful_tx` counter is incremented.
4. **Collision (External): Exactly one** node transmitted, BUT external interference **also** occurred.
   - The transmission fails. The packet remains in the node's queue.
   - Collision statistics are updated (both for the node and globally).
5. **Collision (Internal): Two or more** nodes attempted to transmit on the same channel in the same slot (regardless of external interference).
   - All transmissions on that channel fail. Packets remain in the nodes' queues.
   - Collision statistics are updated (for each involved node and globally).

This cycle repeats for `T_SLOTS`, allowing the collection of long-term performance statistics for this fixed, non-learning transmission strategy.

Functions Susceptible to Enhancement with ML agents

The core decision logic within the `Node` class during **Step 3: Transmission Decisions** is where MABs can be integrated. The specific fixed/random parts we can replace are:

1. **QoS Selection:** Currently uses **Strict Priority**. An MAB could learn a more dynamic priority policy.
2. **Channel Selection:** Currently **Random**. An MAB can learn to prefer channels with lower interference or less congestion.
3. **Backoff Window Selection:** Currently uses a **Fixed Maximum Window** (`FIXED_BACKOFF_WINDOW`) from which a random backoff is chosen. An MAB can learn the optimal *maximum* window size for different situations.

MAB Integration Strategies

We can approach this in two main ways:

1. Single Agent per Node (Monolithic Approach)

- **Concept:** Each node has *one* complex MAB agent responsible for making *all* decisions simultaneously.
- **Actions:** Each "arm" of this single MAB would represent a complete transmission strategy tuple: `(QoS Level to Service, Channel to Use, Max Backoff Window)`.
  - *Example Arm:* `(HIGH_PRIO, Channel 3, CW_max=7)`
  - The total number of arms would be `Num_QoS_Levels * Num_Channels * Num_Backoff_Options`. With 2 QoS levels, 8 channels, and 5 backoff options, this is already 80 arms. This grows very quickly!
- **Reward:** The reward for pulling an arm (executing the chosen strategy) would be based directly on the transmission outcome:
  - `REWARD_HIGH_PRIO` (e.g., 2.0) for a successful High Priority transmission.
  - `REWARD_LOW_PRIO` (e.g., 1.0) for a successful Low Priority transmission.
  - `REWARD_FAILURE` (e.g., 0.0) for a collision or selecting an empty queue.
- **Pros:** Conceptually simpler – one agent learns the entire policy.
- **Cons:** The **action space becomes very large**, making learning extremely slow and inefficient. It doesn't easily capture the sequential or contextual nature of the decisions.

2. Multiple Agents per Node (Hierarchical/Modular Approach)

- **Concept:** Each node has multiple, simpler MAB agents, each responsible for a specific part of the decision process. The choice made by one agent can provide the **context** for the next agent. This mirrors the structure we developed previously.
- **Agents & Functions Enhanced:**
  - **Agent 4: QoS Agent**
    - *Replaces:* Strict priority logic.

        - *Actions (Arms):* Choose `HIGH_PRIO` (Arm 0) or `LOW_PRIO` (Arm 1).
        - *Reward:* Final transmission outcome (2.0, 1.0, or 0.0).
    - **Agent 3: Group Agent** (Optional, but recommended for larger C)
        - *Replaces:* Part of random channel selection (narrows down the search).
        - *Context:* The QoS level chosen by Agent 4.
        - *Actions (Arms):* Choose a predefined channel group (e.g., `[0,1]`, `[0,1,2,3]`, etc.).
        - *Reward:* Final transmission outcome.
    - **Agent 2: Channel Agent**
        - *Replaces:* Random channel selection (or selection within a group).
        - *Context:* The QoS level (and potentially the group chosen by Agent 3).
        - *Actions (Arms):* Choose a specific channel (either from all C channels or just within the chosen group).
        - *Reward:* Final transmission outcome.
    - **Agent 1: Backoff Agent**
        - *Replaces:* Fixed backoff window logic.
        - *Context:* The specific channel chosen by Agent 2 (and potentially QoS).
        - *Actions (Arms):* Choose a maximum backoff window value (e.g., Arm 0 for `CW_max=0`, Arm 1 for `CW_max=1`, etc.).
        - *Reward:* Final transmission outcome.
- **Pros: Much smaller action spaces** for each agent, leading to significantly faster and more efficient learning. Allows agents to learn specialized policies for different contexts (e.g., different backoff strategies for clean vs. noisy channels, different channel preferences for HP vs. LP traffic). More modular and scalable.
- **Cons:** Implementation is more complex. Requires careful handling of contexts between agents. Credit assignment is indirect (all agents in the chain get the same final reward, even if only one agent made the "bad" choice).

Recommendation ✨

The **Multiple Agents (Hierarchical/Modular) approach** is strongly recommended. It aligns better with the natural structure of the problem and is much more computationally efficient for learning compared to the monolithic single-agent approach, especially as the number of channels or backoff options increases. It allows the system to learn sophisticated, context-dependent policies.

Other things

What are Contexts?

Think of context as situational information an agent has *before* it makes a decision. This information helps the agent choose a more appropriate action for the current circumstances. 状況

- Analogy: You look outside (context: raining 🌧️) before deciding whether to take an umbrella (action).
- In your simulator:
  - The QoS level chosen by Agent 4 is the context for Agent 3 (Group Agent).
  - The channel group chosen by Agent 3 is the context for Agent 2 (Channel Agent).
  - The specific channel chosen by Agent 2 is the context for Agent 1 (Backoff Agent).
  - (Potentially) The sensed state of a channel (Idle/Busy/Collision from learning_stats) could also be used as context *before* choosing an action, though we haven't implemented that yet.

Context allows an agent to move beyond a simple "one-size-fits-all" policy and learn specialized strategies for different situations.[1]

How Agents Share Info via Contexts

Agents in a multi-agent system like yours typically share information indirectly through the environment, which then becomes context for others:[2]

1. Environmental State: One agent's action (e.g., transmitting) changes the state of the environment (e.g., causes a collision on a channel).
2. Observation as Context: Another agent observes this new environmental state (e.g., senses the collision or high channel usage via its learning_stats). This observation *becomes* the context for its next decision (e.g., choosing a larger backoff or a different channel).
3. Directed Output: As implemented in your 4-level system, the output (action choice) of a higher-level agent directly serves as the input context for the lower-level agent. This is a very structured way of passing specific information down the decision chain.

While less common in purely decentralized systems like MABs (but possible in more complex Reinforcement Learning), agents could also share information more directly:

- Shared Observations: Agents might access a common, shared variable representing part of the system state (e.g., a broadcast message indicating overall network congestion).
- Explicit Communication: Agents could send messages to each other (though this adds significant complexity).[3]

Directionality and Contexts

Directional structures are a natural and powerful way to use contexts in multi-agent learning. Your 4-level system is a perfect example. 🏛️

- How it Works: The directionality breaks down a complex decision (like QoS + Group + Channel + Backoff) into a sequence of simpler, context-dependent decisions. Each level provides the necessary context for the level below it.
- Benefits:
  - Reduces Complexity: Each agent faces a much smaller set of actions (a smaller "action space"), making learning vastly faster and more manageable.
  - Enables Specialization: Agents learn policies tailored to specific contexts (e.g., Agent 3 learns different group preferences depending on whether it's handling High or Low priority traffic).[4]
  - Structured Exploration: Exploration happens more intelligently within the relevant context provided by the level above.

Tips for Using Contexts Effectively ✨

1. Choose Relevant Context: Only provide context information that is actually useful for predicting the outcome or reward. Irrelevant context (e.g., the current time slot number if the environment doesn't change over time) just adds noise and slows learning.
2. Ensure Observability: The agent must be able to reliably perceive the context *before* it needs to act. Imperfect sensing can lead to suboptimal decisions.
3. Manage State Space Size: Be mindful of how many possible contexts you create. Too many fine-grained contexts (e.g., using queue length 0-10 as 11 distinct contexts) can lead to the "curse of dimensionality," where the agent rarely encounters the same exact situation twice and struggles to learn. Grouping similar states (e.g., Queue=Empty, Low, Medium, Full) can help. Hierarchy is also a key tool here.
4. Consider Context Staleness: In dynamic environments (especially multi-agent ones where others are also learning), the context can change quickly.
5. If an agent learns too slowly, its policy might always be based on outdated context. Using appropriate learning rates (like the constant LEARNING_RATE for non-stationarity) is important