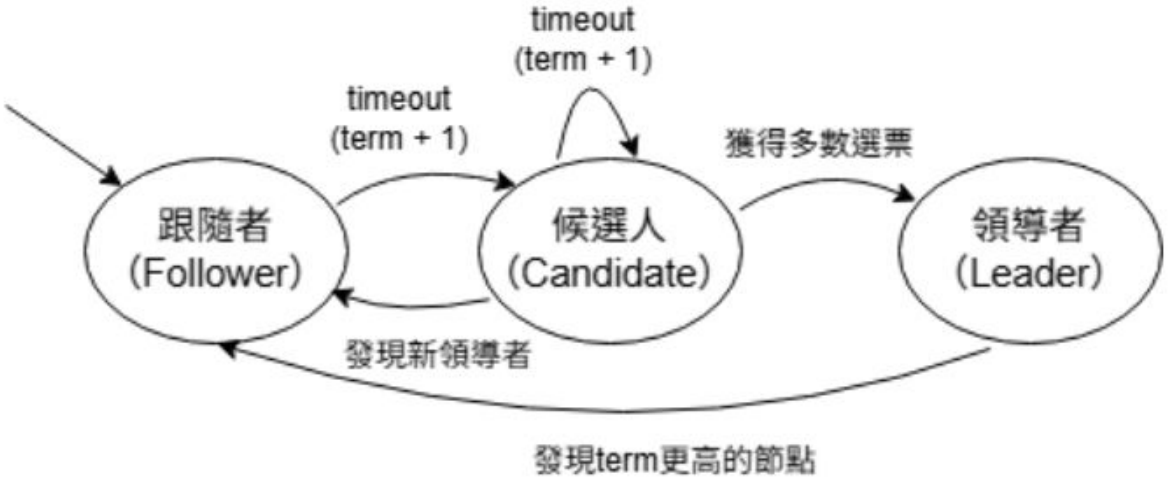
應用於資源受限邊緣裝置之輕量級自適應 Raft 共識機制：基於 Tabular Q-Learning 之優化研究

報告人：周雨澄
指導教授：謝孫源

一、研究背景與動機

1-1 Raft[1] 演算法及其痛點



依賴固定的心跳/選舉逾時區間來維持系統穩定>在高延遲或封包遺失的真實複雜網路中，固定逾時會導致頻繁的誤判 (Misjudgement)，引發選舉風暴，降低系統可用性

1-2 現有解法的限制

靜態配置 /統計方式：Standard Raft[1]、TCP RTO、Dynatune[3]

缺點：滯後性 (Lag)：需累積歷史數據，無法應對突發干擾、可能會因為過於保守的選擇而太慢發現 Leader 故障

基於強化學習的動態調整：LinUCB[2]、DQN-Raft+[4]

缺點：運算成本過高（依賴矩陣及浮點運算）、依賴神經網路運算，推論延遲過高，不適合邊緣裝置運行（如缺乏FPU的MCU）

二、研究方法及目的

引入輕量級的 Q-Learning 演算法，讓 Raft 節點根據當下網路狀況自動學習並動態調整逾時區間。

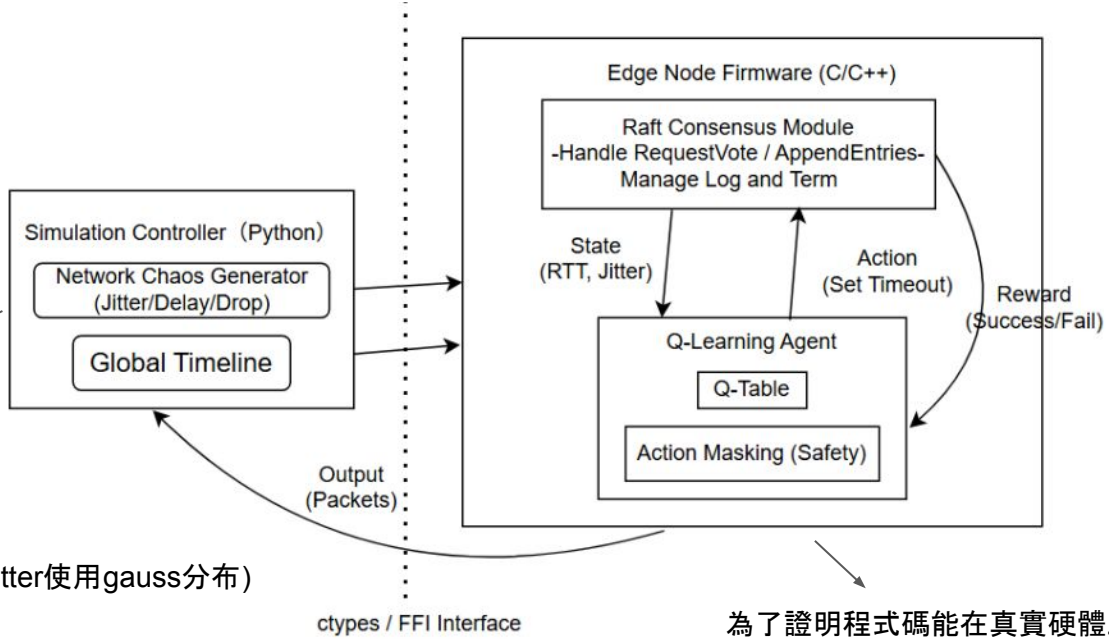
核心優勢：

1. 高可用性：在惡劣網路下仍能減少誤判。
2. 硬體友善：全整數定點數運算，適配低階 MCU。

且為了確保提出的 Q-Learning 機制不僅在理論上可行，且具備移植至真實硬體的工程實用性，本研究採用 Software-in-the-Loop 的混合架構進行驗證及比較

三、系統架構及設計

3-1 系統架構：Software-in-the-Loop (SITL) 驗證環境



透過 FFI/ctypes 結合 Python 開發的模擬器 由 Python 負責生成網路延遲、掉包等場景，驗證 C 核心的強健性

(Latency採用pareto分布模擬，jitter使用gauss分布)

為了證明程式碼能在真實硬體上跑，核心 Raft 與 Q-Learning 均以純 C 語言撰寫

3-2-1 強化學習設計-狀態（State）及行動（Action）

State（根據節點最近 10 次收到封包的資訊）：

延遲 (Latency) 等級：

Low：平均延遲 ≤ 80ms，Med：平均延遲 > 80ms，High：平均延遲 > 150ms

抖動 (Jitter) 等級：

Low：抖動 ≤ 20ms，Med：抖動 > 20ms，High：抖動 > 60ms

State = (Latency 等級 × 3) + Jitter 等級

Action： Arm 0 (150~300ms)/ Arm 1(250~500ms)/ Arm 2(500~1000ms)

3-2-1 強化學習設計-獎勵函數 (Reward Function)

Total Reward = Base Reward - Latency Penalty - Timeout Length Penalty

Base Reward：follower 收到心跳（＋10）/ candidate當選（＋100）

/candidate落選（－40）

Latency Penalty：對當下平均網路延遲按比例扣分，延遲越高，扣越多，間接壓抑它在惡劣環境下的冒險衝動

Timeout Length Penalty（timeout_length_ms * 5）：選的 Arm 越保守（逾時越長），扣的分就越多，避免總是選擇保守的Arm

Q_new = Q_old + α × (Reward − Q_old)

3-3 設計亮點

- 根據連續成功次數動態調整探索率 ε

  連續成功次數　探索率 ε　代表

  - ≤ 5 次　10%　剛失敗過或剛啟動，網路狀況不確定，多探索
  - 6 ~ 20 次　3%　有點穩定了，稍微收斂，少探索
  - > 20 次　1%　極度穩定，幾乎完全信任 Q-Table，幾乎不探索
- follower和candidate的table分開->一個在等heartbeat，一個等選舉封包
- 三層防禦的 Arm 設計：連續失敗 ≥3 次直接強制選最保守的 Arm 2
- 固定點數運算：所有計算使用 int32_t 整數搭配縮放因子 SCALE=1000 模擬小數

四、實驗設計與成果展示

4-1 實驗場景設計

對照組：傳統 Random 隨機策略 vs TCP-like vs 本專案 Q-Learning。

測試場景：

Stable→Crash

Stable→Crash→Recovery

Leader Death + Bad Network

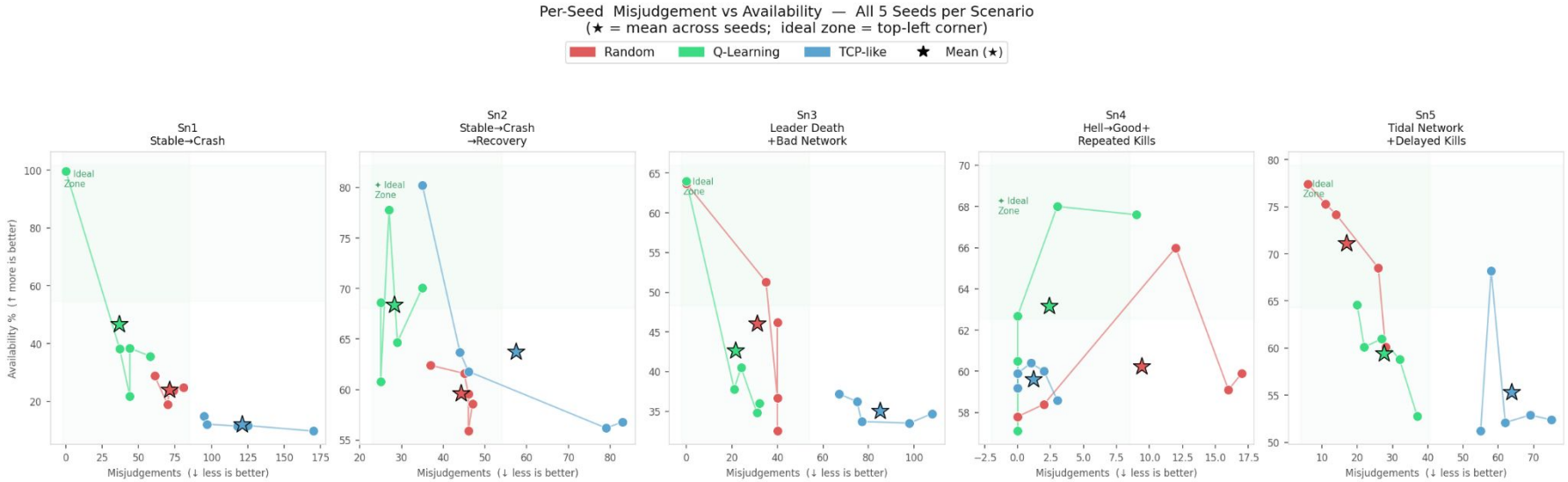
Hell→Good + Repeated Kills (極端波動)

Tidal Network (潮汐網路：週期性波動)

每個場景皆跑60s，且各場景皆跑五組亂數種子確保實驗可重現性及隨機性

實驗指標：Misjudgement（誤判率）/ Availability（系統可用時間）

4-2 整體結果統計



在崩潰與高延遲場景 (Sn1~Sn4)，Q-Learning 的星星 (平均值) 明顯比其他策略更靠攏左上角的理想區域 (高可用、低誤判)

Scenario	Metric	Random	Q-Learn	TCP
Sn 1	Misjudgement	0	5	0
	Availability	1	4	0
Sn 2	Misjudgement	0	5	1
	Availability	0	3	2
Sn 3	Misjudgement	1	4	0
	Availability	2	3	0
Sn 4	Misjudgement	1	3	4
	Availability	1	4	0
Sn 5	Misjudgement	4	1	0
	Availability	4	0	1

Raft Policy Comparison: Random vs Q-Learning vs TCP
Metrics: Misjudgement / Availability%
Note: '*' marks the best performer in that category.

Seed	Sn	R-Mis/Av%	Q-Mis/Av%	T-Mis/Av%
123	1	70/18.9%	0*/99.7%*	118/11.4%
123	2	46/55.9%	25*/68.6%*	44/63.7%
123	3	40/46.2%	0*/64.0%*	108/34.7%
123	4	12/66.0%	9/67.6%*	1*/60.4%
123	5	6*/77.4%*	32/58.8%	75/52.4%
1234	1	61/28.8%*	44*/21.8%	125/11.6%
1234	2	46/59.6%	29*/64.7%*	79/56.2%
1234	3	40/36.7%	24*/40.5%*	67/37.2%
1234	4	2/58.4%	0*/60.5%*	0*/59.2%
1234	5	11*/75.3%*	37/52.8%	55/51.2%
5555	1	74/23.7%	58*/35.6%*	95/14.9%
5555	2	47/58.6%	25*/60.8%	46/61.8%*
5555	3	0*/63.7%*	21/37.8%	77/33.7%
5555	4	17/59.9%	3/68.0%*	2*/60.0%
5555	5	14*/74.2%*	22/60.1%	69/52.9%
7777	1	81/24.8%	37*/38.2%*	97/12.1%
7777	2	37/62.4%	27*/77.8%*	83/56.8%
7777	3	40/32.5%	32*/36.0%*	98/33.5%
7777	4	16/59.1%*	0*/57.1%	3/58.6%
7777	5	28/60.1%	20*/64.6%	58/68.2%*
9999	1	71/23.7%	44*/38.4%*	170/9.7%
9999	2	45/61.6%	35*/70.1%	35*/80.2%*
9999	3	35/51.3%*	31*/34.8%	75/36.2%
9999	4	0*/57.8%	0*/62.7%*	0*/59.9%
9999	5	26*/68.5%*	27/61.0%	62/52.1%

4-3 邊界案例分析 - 為什麼在潮汐網路 (Sn5) 表現不佳？

分析：潮汐網路 (Tidal Network) 具有高頻且週期性的劇烈變化。因為 Q-Learning 依賴累積的經驗 (Q-Table)，當環境變化太快時，先前的經驗反而變成阻礙（產生振盪），導致反應不及無腦的 Random 策略。

五、結論

5-1 總結

- 極低開銷的邊緣 AI：成功在無 FPU (浮點運算單元) 的 C 語言核心中實全定點數 (Fixed-point)Q-Learning 演算法，運算與記憶體開銷極低，完美適配資源受限的低階微控制器 (MCU)。
- 系統可用性大幅提升：透過動態逾時機制，模型能自適應網路品質。在 80% (4/5) 的惡劣網路場景中，系統穩定度與可用率皆勝過傳統 Random 與 TCP-like 演算法，大幅減少了選舉風暴 (Misjudgement)
- 靈活且強健的 SITL 測試：運用 Python 與 C 語言的跨語言介面 (FFI/ctypes)，成功建立 Software-in-the-Loop 模擬架構，證明了在投入實體硬體前，先於複雜網路模擬中驗證韌體的實用性。

5-2 未來展望

- 演算法優化：目前的 Reward 機制與 Q-Table 容易受歷史經驗綁架。未來可加入 Forgetting Factor或Sliding Window 機制，讓系統在面對潮汐網路等劇烈變動環境時，能加速淘汰過期經驗，提升反應速度。
- 真實硬體與容器化部署驗證
  - 將系統封裝至 Docker 容器中，利用 Docker Network 模擬並限制網路拓樸，進行分散式網路驗證
  - 將本專案的 C Firmware 實際燒錄至多台開發板，建立真實邊緣節點叢集

六、參考資料

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extendedversion)," in 2014 USENIX Annual Technical Conference (USENIX ATC 14), Philadelphia, PA,2014, pp. 305–319.

[2] Q. Wang, "BALLAST: Bandit-Assisted Learning for Latency-Aware Stable Timeouts in Raft,"arXiv preprint arXiv:2512.21165, 2025.

[3] K. Shiozaki and J. Nakamura, "Dynatune: Dynamic Tuning of Raft Election Parameters Using Network Measurement," in 2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2025.

[4] P. Wang et al., "DQN-Raft+: A Deep RL-Optimized Lightweight Consensus Algorithm," Informatica, vol. 49, no. 33, pp. 127–148, 2025.