RAFT Implementation Report

Bộ môn Công nghệ Tri thức
Môn học: Blockchain và ứng dụng

Họ và tên: Lê Thanh Minh Trí
MSSV: 22127422

---

1. GIỚI THIỆU VỀ RAFT

1.1 Tổng quan thuật toán RAFT

Định nghĩa và Mục đích:

RAFT (Reliable, Replicated, Redundant, And Fault-Tolerant) là thuật toán đồng thuận (consensus algorithm) được thiết kế để quản lý replicated log trong hệ thống phân tán. Mục đích chính của RAFT là:

1. **Đạt đồng thuận phân tán**: Đảm bảo nhiều máy chủ đồng ý về một trạng thái chung, ngay cả khi một số máy bị lỗi
2. **Dễ hiểu (Understandability)**: Được thiết kế để dễ hiểu hơn Paxos, phân tách thuật toán thành các module độc lập

Bài toán RAFT giải quyết:
Trong hệ thống phân tán, làm thế nào để nhiều máy chủ:

- Đồng ý về chuỗi các lệnh (commands) cần thực hiện
- Duy trì tính nhất quán khi có node bị lỗi
- Tự động phục hồi khi node quay lại hoạt động
- Đảm bảo không có dữ liệu bị mất sau khi đã commit

1.2 Cách hoạt động của RAFT

RAFT hoạt động dựa trên nguyên lý **replicated state machine**: tất cả server thực thi cùng một chuỗi lệnh theo đúng thứ tự sẽ có cùng trạng thái cuối cùng.

A. Kiến trúc tổng quan:

```
Client → Leader → Followers
         ↓
    Replicated Log (entry 1, entry 2, ...)
         ↓
    State Machine (key-value store)
```

B. Ba trạng thái của node:

1. **Follower (Thành viên)**:

- Trạng thái khởi đầu và phổ biến nhất
- Chỉ nhận và phản hồi RPC từ leader và candidate
- Không chủ động gửi request
- Nếu không nhận heartbeat trong election timeout → chuyển sang candidate

2. **Candidate (Ứng viên)**:

- Trạng thái tạm thời khi node muốn trở thành leader
- Tăng term number, vote cho chính mình
- Gửi RequestVote RPC đến tất cả node khác
- Nếu nhận đủ phiếu (majority) → trở thành leader
- Nếu phát hiện leader khác hoặc term cao hơn → quay lại follower
- Nếu timeout → bắt đầu election mới

3. **Leader (Lãnh đạo)**:

- Xử lý tất cả client requests
- Gửi heartbeat định kỳ (empty AppendEntries) đến followers
- Replicate log entries đến followers
- Quyết định khi nào một entry được commit
- Chỉ có 1 leader trong một term

C. Cơ chế Term (Nhiệm kỳ):

- Term là số nguyên tăng dần, đóng vai trò như logical clock
- Mỗi term có tối đa 1 leader
- Term được chia làm 2 giai đoạn:
  1. **Election**: Bầu cử leader
  2. **Normal Operation**: Leader hoạt động bình thường
- Node luôn cập nhật lên term cao hơn khi phát hiện
- Dùng để phát hiện stale information

1.3 Các cơ chế chính của RAFT

A. Leader Election (Bầu cử Leader):

Quy trình bầu cử:

1. Node chờ random timeout (150-300ms) mà không nhận heartbeat
2. Chuyển sang candidate: tăng term, vote cho bản thân
3. Gửi RequestVote RPC song song đến tất cả peers với:
   - Term hiện tại
   - Candidate ID
   - Last log index và term
4. Các node khác xem xét có vote hay không:
   - Chỉ vote nếu chưa vote cho ai trong term này
   - Chỉ vote nếu log của candidate "up-to-date" hơn hoặc bằng
5. Nếu nhận được majority votes → trở thành leader
6. Leader gửi heartbeat ngay lập tức để ngăn election mới

B. Log Replication (Sao chép Log):

Quy trình replication:

1. Client gửi command đến leader
2. Leader append entry mới vào local log với:
    - Term hiện tại
    - Command
    - Index trong log
3. Leader gửi AppendEntries RPC đến tất cả followers với:
    - Các entry mới cần append
    - `prev_log_index`: Index của entry ngay trước entry mới
    - `prev_log_term`: Term của prev_log_index
    - `leader_commit`: Commit index của leader
4. Follower kiểm tra consistency:
    - Log có entry tại prev_log_index với term = prev_log_term?
    - Nếu có → append entries mới, trả về success
    - Nếu không → trả về false (log inconsistent)
5. Leader retry với index nhỏ hơn cho đến khi tìm được điểm nhất quán
6. Khi majority followers ACK → leader commit entry
7. Leader thông báo followers commit thông qua AppendEntries tiếp theo
8. Followers apply committed entries vào state machine

1.4 Trả lời câu hỏi 1: Làm thế nào RAFT đảm bảo tính nhất quán khi chuyển đổi leader?

Cơ chế đảm bảo consistency khi leader change:

1. Leader Completeness Property:

- Leader mới được bầu PHẢI có tất cả committed entries
- Đạt được qua voting mechanism:
    - Candidate gửi last log index và term
    - Voter từ chối nếu log của mình "up-to-date" hơn
    - Chỉ candidate có log đầy đủ nhất mới đủ votes

2. Không Commit Entries từ Previous Terms trực tiếp:

- Leader mới không đánh dấu entries từ term cũ là committed
- Chỉ commit entries từ term hiện tại
- Khi commit entry term hiện tại → tự động commit tất cả entries trước đó
- Ngăn chặn tình huống: entry "có vẻ committed" nhưng bị ghi đè

3. Log Reconciliation:

- Leader mới tìm điểm cuối cùng mà log của nó và follower giống nhau
- Ghi đè phần không nhất quán ở follower
- Quy trình:

```
Leader decrements nextIndex cho follower
→ Retry AppendEntries với prev_log_index nhỏ hơn
→ Cho đến khi tìm được matching point
→ Ghi đè log của follower từ điểm đó
```

4. Term Numbers làm Logical Clock:

- Term cao hơn luôn thắng
- Node với term cũ tự động step down
- Ngăn chặn "split-brain": 2 leader cùng lúc

Ví dụ cụ thể:

```
Trước khi leader fail:
Leader: [1,1] [2,2] [3,3] [4,3] ← committed đến index 3
F1:     [1,1] [2,2] [3,3] [4,3]
F2:     [1,1] [2,2] [3,3]
F3:     [1,1] [2,2]

Leader fail → Election term 5

Chỉ F1 có thể thắng vì:
- F1 có log dài nhất và term mới nhất
- F2, F3 thiếu entries → không up-to-date → bị reject

Leader mới (F1) đảm bảo:
- Có tất cả committed entries [1,1] [2,2] [3,3]
- Sync [4,3] đến F2, F3 (có thể chưa committed)
```

1.5 Trả lời câu hỏi 2: Hạn chế của RAFT khi xử lý các hành vi độc hại

RAFT chỉ chịu được Crash Faults, KHÔNG chịu Byzantine Faults

A. Mô hình lỗi RAFT giả định:

1. **Crash-stop model**:
    - Node hoặc hoạt động đúng hoặc ngừng hoàn toàn
    - Không có hành vi ngẫu nhiên hoặc độc hại
    - Network có thể mất gói, chậm, nhưng không sửa đổi nội dung
2. **Giả định Trust**:
    - Tất cả nodes follow protocol
    - Không có node cố tình gửi thông tin sai
    - Không có attacker trong hệ thống

B. Một số hành vi độc hại RAFT KHÔNG thể xử lý:
Malicious Leader, Data Corruption, False Voting, Sybil Attack, Network-level Attacks

C. Tại sao RAFT không chống Byzantine?

1. **Thiết kế đơn giản hóa**: RAFT chọn simplicity > security
2. **Trust assumption**: Phù hợp với internal systems (datacenter)
3. **Performance**: Byzantine protocols (pBFT) chậm hơn nhiều (3-10x)
4. **Use case khác nhau**:
    - RAFT: Distributed databases, coordination services
    - pBFT: Blockchain, untrusted environments

D. Giải pháp cho Byzantine Faults:
Nếu cần chống Byzantine attacks, sử dụng:

- **pBFT** (Practical Byzantine Fault Tolerance)
- **HotStuff**, **Tendermint**: Modern BFT protocols
- Sử dụng cryptographic signatures
- Multiple voting rounds để verify

Tóm tắt:

- RAFT: 2f+1 nodes, chịu f crash faults, fast, simple
- pBFT: 3f+1 nodes, chịu f Byzantine faults, slower, complex

---

2. SO SÁNH RAFT VÀ pBFT

2.1 Bảng so sánh tổng quát

Tiêu chí	RAFT	pBFT
Mục đích	Crash Fault Tolerance	Byzantine Fault Tolerance
Số node tối thiểu	3	4 (3f+1)
Chịu lỗi	f trong 2f+1 nodes	f trong 3f+1 nodes
Độ phức tạp	O(n) messages	$O(n^2)$ messages
Hiệu năng	Cao hơn	Thấp hơn (do nhiều rounds)
Bảo mật	Giả định nodes trung thực	Chịu được Byzantine attacks
Ứng dụng	Distributed databases	Blockchain, untrusted systems

2.2 Chi tiết so sánh

Mô hình lỗi:

- RAFT: Crash-stop model (node chỉ dừng hoặc hoạt động đúng)
- pBFT: Byzantine model (node có thể hành động độc hại)

Số lượng node:

- RAFT: 2f+1 nodes để chịu f lỗi (VD: 5 nodes chịu 2 lỗi)

- pBFT: 3f+1 nodes để chịu f lỗi (VD: 7 nodes chịu 2 lỗi)

Quy trình consensus:

- RAFT: Leader → AppendEntries → Majority ACK → Commit
- pBFT: Pre-prepare → Prepare (2f+1) → Commit (2f+1) → Execute

Xử lý lỗi:

- RAFT: Re-election khi leader fail, sync log khi rejoin
- pBFT: View change, phát hiện và loại trừ Byzantine nodes

Hiệu năng:

- RAFT: ~1000s transactions/sec (etcd benchmark)
- pBFT: ~100s transactions/sec (do complexity cao hơn)

2.3 Khi nào dùng gì?

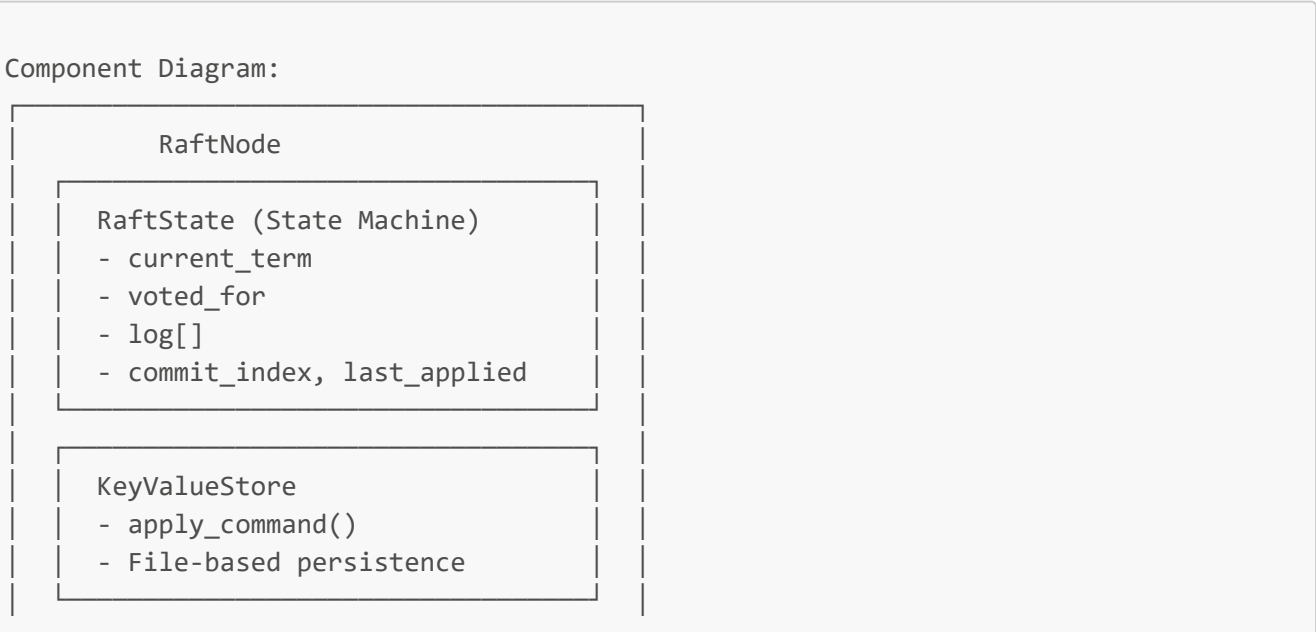
Chọn RAFT khi:
☑ Môi trường đáng tin cậy
☑ Chỉ cần chịu crash faults
☑ Cần hiệu năng cao
☑ Ví dụ: Internal microservices, distributed databases

Chọn pBFT khi:
☑ Môi trường không tin cậy
☑ Có thể có Byzantine nodes
☑ Blockchain applications
☑ Ví dụ: Cryptocurrency, public ledger

---

3. MÔ TẢ CHƯƠNG TRÌNH ĐÃ CÀI ĐẶT (3-4 pages)

3.1 Kiến trúc tổng quan



```
│  ┌──────────────────────────────────┐  │
│  │  gRPC Service                    │  │
│  │  - RequestVote RPC               │  │
│  │  - AppendEntries RPC             │  │
│  │  - SubmitCommand RPC             │  │
│  └──────────────────────────────────┘  │
└────────────────────────────────────────┘
```

3.2 Các thành phần chính

A. raft_state.py - RAFT State Machine

Quản lý trạng thái của node:

- Persistent state: current_term, voted_for, log[]
- Volatile state: commit_index, last_applied
- Leader state: next_index[], match_index[]

Chức năng quan trọng:

```
- become_follower(): Chuyển sang follower
- become_candidate(): Start election
- become_leader(): Khởi tạo leader state
- append_entries(): Thêm log entries
- update_commit_index(): Advance commit
```

B. kvstore.py - Key-Value Storage

Lưu trữ dữ liệu đã commit:

- File-based: `data/node_X_db.json`
- Thread-safe với RLock
- Hỗ trợ: SET, GET, DELETE

C. node.py - RAFT Node Implementation

Core logic:

1. **Election Timer Thread**: Phát hiện timeout, start election
2. **Heartbeat Thread**: Leader gửi heartbeat định kỳ
3. **Apply Thread**: Áp dụng committed entries vào state machine

RPC Handlers:

- `handle_request_vote()`: Xử lý yêu cầu vote
- `handle_append_entries()`: Nhận log entries từ leader
- `handle_submit_command()`: Client gửi command

3.3 Workflow

Leader Election:

```
1. Follower timeout → become_candidate()
2. Increment term, vote for self
3. Send RequestVote RPCs to all peers
4. If majority votes → become_leader()
5. Else if higher term seen → become_follower()
6. Else retry election
```

Log Replication:

```
1. Client → SubmitCommand to leader
2. Leader appends to local log
3. Leader sends AppendEntries to followers
4. Followers append and ACK
5. Leader waits for majority ACKs
6. Leader commits (advance commit_index)
7. Leader notifies followers to commit
8. Apply to state machine (kvstore)
```

Failure Recovery:

```
Leader failure:
1. Followers timeout (no heartbeat)
2. New election starts
3. New leader elected
4. Sync logs from new leader

Follower failure:
1. Leader continues with majority
2. When follower rejoins, leader syncs log
3. Follower catches up automatically
```

3.4 Hướng dẫn chạy chương trình

Bước 1: Cài đặt

```
pip install -r requirements.txt
python scripts/generate_proto.py
```

Bước 2: Chạy cluster

```
python scripts/start_cluster.py
```

Bước 3: Gửi commands

```
python scripts/client.py
raft> SET key1 value1
raft> GET key1
```

3.5 Thay đổi tham số

Điều chỉnh số lượng nodes:
Sửa file `scripts/start_cluster.py`:

```
NODES = {
    "node1": {"host": "localhost", "port": 5001},
    "node2": {"host": "localhost", "port": 5002},
    "node3": {"host": "localhost", "port": 5003},
    # Thêm nodes...
}
```

Điều chỉnh timeout:
Sửa file `scripts/run_node.py`:

```
parser.add_argument('--election-timeout-min', type=int, default=150)
parser.add_argument('--election-timeout-max', type=int, default=300)
parser.add_argument('--heartbeat-interval', type=int, default=50)
```

Tắt/bật node:

- Tắt: Ctrl+C trên terminal của node đó
- Bật: Chạy lại `python scripts/run_node.py ...`

Kiểm tra dữ liệu:

```
# Xem file JSON trong data/
cat data/node_1_db.json
cat data/node_1_state.json
```

---

4. TỰ ĐÁNH GIÁ

4.1 Ưu điểm của chương trình

☑ Đầy đủ chức năng:

- Implement đầy đủ core RAFT: election, replication, fault tolerance
- Có persistence (lưu state và data)
- Test suite comprehensive

☑ **Dễ sử dụng:**

- Scripts tự động start cluster
- Client interactive mode
- Có các test scripts

☑ **Code rõ ràng:**

- Inline comments đầy đủ
- Module hóa (state, storage, node riêng biệt)
- Thread-safe

4.2 Nhược điểm

❌ **Chưa tối ưu:**

- Không có log compaction (log tăng vĩnh viễn)
- Không có snapshot mechanism

❌ **Testing:**

- Phải can thiệp thủ công cho một số tests
- Chưa có edge cases (concurrent elections, log conflicts)

4.3 Cải thiện

1. **Log Compaction:**
    - Implement snapshotting
    - Compact log khi đạt threshold
    - InstallSnapshot RPC
2. **Performance:**
    - Batch log entries
    - Pipeline AppendEntries RPCs
    - Read optimization (lease-based reads)

Tài liệu tham khảo

1. Ongaro, D., & Ousterhout, J. (2014). In Search of an Understandable Consensus Algorithm. USENIX ATC.
2. Castro, M., & Liskov, B. (1999). Practical Byzantine Fault Tolerance. OSDI.
3. Lamport, L. (1998). The Part-Time Parliament. ACM TOCS.
4. RAFT Visualization: https://raft.github.io/
5. gRPC Documentation: https://grpc.io/
