

TCCS 558: APPLIED DISTRIBUTED COMPUTING

Consensus, Consistency, Replication

Wes J. Lloyd
 Institute of Technology
 University of Washington - Tacoma

OBJECTIVES

- Assignment #2 Questions
- Assignment #3 Questions
- Assignment #1 Feedback
- Feedback from 11/30

- Raft Consensus Algorithm
- Ch. 7 - Consistency and Replication
 - Introduction
 - Data centric consistency models

December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma
L18.2

ASSIGNMENT #1 FEEDBACK

- Java 9 was just released in September
 - No requirement to use Java 9
 - Tested a few programs in Java 8 and Java 9. Observed programs running 2x as slow in Java 9 vs. Java 8
- Developing the entire program as **static** methods is a poor programming practice
 - Static void main is a static method, which provides the program's entry point.
 - Once inside static void main, it is typical to instantiate instances of classes (**called objects**) to provide program functionality
- Have seen many cases of using a separate thread for an entire server (e.g. TCP, UDP), but not for new client requests

December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma
L18.3

A1 FEEDBACK - 2

- Have seen many cases of using a separate thread for an entire server (e.g. TCP, UDP), but not for every new client request
 - **UP SIDE:** Avoiding the use of threads to handle concurrent operations for the key-value store makes all operations sequential
 - Concurrency problems are avoided: *race conditions, deadlock*
 - Requests are simply queued until **the server thread** is available
 - **PROBLEM:** Approach will not scale
 - Consider running the KV-store on a c4.8xlarge AWS EC2 Instance
 - Instance has 36 vCPUs and can process up to 36 requests in parallel
 - Server can process at most 1 sequential request at a time
 - Load average will generally not exceed 1.0
 - Cost of c4.8xlarge is \$1.59, but sequential code only returns 4.4¢
 - **Only a 2.7% return on investment**

December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma
L18.4

A1 FEEDBACK - 3

- Have seen many cases of using a separate thread for an entire server (e.g. TCP, UDP), but not for every new client request
 - **DOWN SIDES**
 - 10 points for using multiple threads
 - 10 points for concurrency
 - Avoiding the problem is not necessarily a solution
 - Does not scale, no ROI
- **KEY GOAL:** *obtain practice developing multithreaded, synchronized code which is scalable, that can leverage the powerful multicore systems of today and tomorrow*

December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma
L18.5

A1 FEEDBACK - 4

- One **KEY** advantage of distributed objects based solutions, RPC, RMI, etc. is the ability to create **RICH** interfaces of methods to avoid painstaking parsing of the socket streams (TCP,UDP)
- For example, a rich interface for the RMI client/server key-value store may include:
 - get(), put(), del(), store(), exit() methods in Java
- **Advantage:** the rich interface allows the client to directly invoke the remote server method, and nicely **hand-off** the required data, and receive either a primitive data type (String, int, etc.) or custom object as a result
- **That being said . . .**

December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma
L18.6

A1 FEEDBACK - 5

- What is not ideal about this RMI interface for the Key-Value store?

```
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

public interface RMHandler extends Remote
{
    String RMRequest(String Msg) throws RemoteException;
}
```

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma L18.7

FEEDBACK FROM 11/30

- For Distributed Mutual Exclusion, Ring Algorithm:
 - What if the skipped successor becomes active when message is passed to other node?
 - If initially skipped, then the node will **NOT** appear in the active node list which is passed around the ring.
 - The newly restored node will not have a vote in the election.
 - The node should be able to participate in future elections.

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma L18.8

RAFT CONSENSUS

L18.9

DESIGN GOALS FOR RAFT

- Complete and practical foundation for building systems
 - Reduce design work for developers
- Safe under all conditions
- Efficient for common operations
- UNDERSTANDABLE**
 - So Raft can be implemented and extended as needed in real world scenarios

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma L18.10

DESIGN GOALS FOR RAFT - 2

- Raft decomposes consensus into sub-problems:
 - Leader election:** leader election algorithms adjustable
 - Log replication:** leader accepts log entries and coordinates replication across cluster enforcing log consensus
 - Safety:** if any state machine applies a log entry, then no other server can apply a different log entry for the same log index
 - Membership changes:** must migrate from old-configuration to new-configuration in a coordinated way

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma L18.11

DESIGN GOALS FOR RAFT - 3

- Simplify the state space
- Reduce the number of states to consider
- Make system more coherent
- Eliminate non-determinism
- LOGS not allowed to have holes
- Limit ways logs can be inconsistent

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma L18.12

RAFT ALGORITHM BASICS

- Begins by electing a **leader**
- **Leader** manages log replication
- **LEADER ACTIVITIES**
 - Accepts log entries from other nodes
 - Replicates them on other servers
 - Tells nodes when safe to apply log entries to their state machines (KV store)
- **Leader** can make decisions without consulting others
- Data flows from **leader** → to nodes
- When **leader** fails, a new **leader** is elected

December 5, 2017

TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma

L18.13

RAFT BASICS - 2

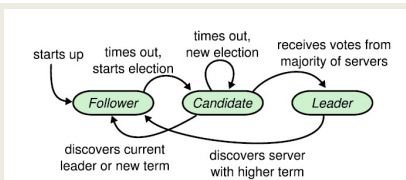
- Server states: **leader**, (*****)**follower**, **candidate**
 - (*****) – initial state of every node is **follower**
- Nodes redirect all requests to the **leader**
- **Candidate** server in a leader election
 - Server with most votes wins election, becomes **leader**
 - Other nodes become **followers**
 - Each **candidate** sponsors its own election, and solicits votes
 - More than one **candidate** can be conducting an election at the same time

December 5, 2017

TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma

L18.14

RAFT ELECTION: NODE STATES



- **Follower**
- **Candidate**
- **Leader**

December 5, 2017

TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma

L18.15

TERMS

- Raft divides time into **TERMS** of arbitrary length
- Terms are numbered with consecutive integers
- Terms start with an election (term # is incremented)
- If election results in a **SPLIT VOTE**, term ends, and a **new term** is started with an election
- There is only (1) **Leader** in any given term
- Terms act as a **logical clock**
- Each server stores current term number
- Terms are exchanged in communication

December 5, 2017

TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma

L18.17

TERMS - 2

- If a larger term # is found, then **all nodes** update term # and defer to the term's **leader**
 - If **candidate** or **leader** finds its term is out of date, will immediately become a **follower** node
- If server receives request with stale term #, then request is rejected

December 5, 2017

TCCS558: Applied Distributed Computing [Fall 2017]
 Institute of Technology, University of Washington - Tacoma

L18.18

<p>State</p> <p>Persistent state on all servers: (Updated in stable storage before responding to RPCs)</p> <p>currentTerm - latest term served by node (initialized to 0 on first boot; increases monotonically)</p> <p>leaderId - candidateId that received vote in current term (or null if none)</p> <p>log - log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)</p> <p>Vote state on all servers:</p> <p>candidateId - index of highest log entry known to be committed (initialized to 0; increases monotonically)</p> <p>lastApplied - index of highest log entry applied to state machine (initialized to 0; increases monotonically)</p> <p>Vote state on leader:</p> <p>(Uncommitted log entries) for each server: index of the next log entry to send to that server (initialized to leader's log index + 1)</p> <p>matchIndex - the index server index of highest log entry known to be replicated on server (initialized to 0; decreases monotonically)</p> <p>AppendEntries RPC</p> <p>Invoked by leader to replicate log entries (§5.3); allowed as heartbeat (§5.2).</p> <p>Arguments:</p> <p>term - leader's term</p> <p>leaderId - id of leader (can indicate client: index of log entry immediately preceding term start)</p> <p>prevLogIndex - term start of prevLogIndex entry</p> <p>entries - log entries to store (empty for heartbeat; may send more than one for efficiency)</p> <p>leaderCommit - leader's commitIndex</p> <p>Result:</p> <p>term - currentTerm; for leader to update state if more followers returned entry matching prevLogIndex and prevLogTerm</p> <p>Receiver implementation:</p> <ol style="list-style-type: none"> 1. Reply false if term < currentTerm (§5.1) 2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.2) 3. If an existing entry conflicts with a new one (same index log different term), delete the existing entry and all that follow it (§5.3) 4. Append any new entries not already in the log 5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of first new entry) 	<p>RequestVote RPC</p> <p>Invoked by candidates to gather votes (§5.2).</p> <p>Arguments:</p> <p>term - candidate's term</p> <p>candidateId - candidate requesting vote</p> <p>lastLogIndex - index of candidate's last log entry (§5.4)</p> <p>lastLogTerm - term of candidate's last log entry (§5.4)</p> <p>Result:</p> <p>term - currentTerm; the candidate to update itself</p> <p>votesGranted - true means candidate received vote</p> <p>Receiver implementation:</p> <ol style="list-style-type: none"> 1. Reply false if term > currentTerm (§5.1) 2. If voteFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.3, §5.4) <p>Rules for Servers</p> <p>All Servers:</p> <ul style="list-style-type: none"> • If candidateId = lastApplied, increment lastApplied, apply log[lastApplied:] to state machine (§5.3) • If RPC (request or response) contains term T > currentTerm, set currentTerm = T, convert to follower (§5.1) <p>Follower (§5.2):</p> <ul style="list-style-type: none"> • Respond to RPC from candidate and leader • If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate, convert to candidate <p>Candidates (§5.2):</p> <ul style="list-style-type: none"> • On conversion to candidate, start election: • Set term self • Start election timer • Send RequestVote RPCs to all other servers • If votes received from majority of servers, become leader • If AppendEntries RPC received from one leader, convert to follower • If election timeout elapses, start new election <p>Leaders:</p> <ul style="list-style-type: none"> • Upon election, send initial empty AppendEntries RPCs (heartbeats) to each server; repeat during idle periods to prevent election timeout (§5.2) • If committed received from client, append entry to local log; respond after entry applied to state machine (§5.3) • If log index > candidateId for a follower, send AppendEntries RPC with log entries starting at candidateId • If successful, update candidateId and candidateId for follower (§5.3) • If AppendEntries fails because of log inconsistency, decrement candidateId and retry (§5.3) • If term starts at N with that N > commitIndex, a majority of matchIndex[i] > N, and log[i] term == currentTerm, set commitIndex = N (§5.3, §5.4)
--	---

RAFT METHODS

- Implemented as “RPCs”, but can be implemented as TCP stream by marshalling data inputs/outputs
- **RequestVote()**
 - Initiated by **candidates** during an election
- **AppendEntriesToLog()**
 - Sent by **leaders** to **follower** nodes at regular intervals
 - Used as a heartbeat to maintain leadership
 - Provides log updates to nodes
 - Performs consistency checks
- Commands are retried if no response after timeout
- Commands sent in parallel using multiple threads (performance)

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.19

RAFT ELECTIONS

- Every node has a **randomized** ElectionTimeout value
- If a node (**follower**) receives no heartbeat from the **leader** after the timeout, node expects the **leader** has gone offline
- **NEW ELECTION:**
 - (1) The node **begins a new election** as **candidate**, sending RequestVote() to every node in the system
 - **Candidate** immediately votes for itself
 - RequestVote() sent in parallel to all nodes
 - (2) Follower votes for **first candidate** a RequestVote() is received from **only if the candidate's log is at least (or more) up-to-date**
 - *Inspect candidate provided last log index and log term values*
 - (3) If **candidate** obtains a majority of the votes (*determined by calculating majority total from node directory*) **It wins the election!!!**

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.20

ELECTIONS - 2

- **Election outcomes**
 - A - **Candidate** wins
 - B - Another server establishes leadership
 - C - There is no winner
- Servers vote for only one **candidate**
- Only (1) winner per election
- Only (1) **leader** per term
 - “Election safety property”
- New **leader** sends empty heartbeat to nodes to establish leadership

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.21

ELECTIONS - 3

- While a **candidate** waits for votes, it may receive an AppendEntries() call from another **leader**
 - If the **leader's** term \geq **candidate's** term then the **candidate** concedes the election and returns to **Follower** state
- If multiple elections, then no one **candidate** may receive a majority vote. One election times out **first** based on a randomized-election-timeout value
 - Random timeout values help spread out the **candidates** to prevent endless looping
- **KEY IDEA:** by using random timeouts, when no majority vote occurs, a random node times out first and starts a new election before anyone else by incrementing the term #, and sending RequestVote()

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.22

ELECTIONS - 4

- Randomized timeout values should be reset every time
- Paper suggests a min timeout of 150ms, and max of 300ms
- Timeout should be “an order of magnitude” greater (10x) than the node-to-node communication latency
 - *I'm presently using 500 - 1000ms*
- Can experiment with different values

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.23

ELECTIONS - 5

- RAFT enforces leader logs to be up-to-date during an election
- Nodes **ONLY** vote for a candidate ***If*** :
 - **Candidate** local term and log number \geq **follower**
 - Candidate's log ***must be*** at least as up-to-date as the majority of follower's log
- **MORE up-to-date log is defined as log with:**
 - Higher term # in last log entry
 - --- OR ---
 - When term of last log entries match, log with more entries
 - *E.g. longer log*

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.24

TYPICAL ELECTION SEQUENCE

- Term 1: normal election
- Term 2: normal election
- Term 3: SPLIT VOTE, no **leader** emerges, election times out
- Term 4: normal election

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.25

RAFT SAFETY

Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

- Raft guarantees that each of these properties is always true

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.26

LOG REPLICATION

- **Leader** receives commands forwarded from **followers**
- **Ways logs can diverge**
 - (a) **Follower** may be missing entries present on **leader**
 - (b) **Follower** may have extra entries not present on the **leader**
 - (c) Both A and B
- Because raft uses a “coordinator” node to achieve consensus the number of possible ways logs can diverge is limited
- Raft **leaders FORCE followers** logs to match its own
- Conflicting entries in follower logs are overwritten

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.27

LOG REPLICATION - 2

- **FOR THE WHOLE SYSTEM THERE IS JUST ONE MONOTONICALLY INCREASING LOG INDEX**
 - Akin to Lamport’s Clocks
- **Possible follower states at start of new term**
 - (a) Missing entries
 - (b) Extra uncommitted entries
 - (c) Both

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.28

RAFT - LOG REPLICATION ALGORITHM

- **Leader:**
 1. Receives command(s)
 2. Appends commands to local log (concurrent hash table)
 3. Sends AppendEntries() to **followers**
- **Leader** tracks index of its highest committed log entry
- Provides this index to **followers** in AppendEntries() RPC
- **Leader commit to state machine:**
 - (1) When log entries replicated at a majority of the **followers**, **leader** commits to its state machine (KV-store)

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.29

LOG REPLICATION ALGORITHM - 2

- **Synchronizing follower logs**
 - (2) If **follower** rejects AppendEntries() then **leader** decrements its “follower-nextIndex” by one, and **retries** AppendEntries().
 - “follower-nextIndex” tracks which logs entries are sent to the follower for each AppendEntries() RPC call
 - Loop continues until **leader walks back** its “follower-nextIndex” until it **matches** what is committed at the **follower**
 - **Follower** has a **commitIndex**
 - Tracks 1st phase of a “two-phase” commit
 - **Follower** has a **lastApplied** index
 - Tracks 2nd phase of “two-phase” commit
 - Once **leader** matches follower-nextIndex, the **follower** accepts the AppendEntries() RPC, and writes data to its log
 - **Conflicting log entries are overwritten**

December 5, 2017 TCCS558: Applied Distributed Computing [Fall 2017] Institute of Technology, University of Washington - Tacoma L18.30

LOG REPLICATION ALGORITHM - 3

- Leader based consensus algorithms require the leader to “eventually store” all committed log entries
- Raft handles follower node failure by retrying communication indefinitely
 - If crashed server restarts, the log will be resurrected, and the follower’s state machine will be restored (kv-store)

December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
Institute of Technology, University of Washington - Tacoma
L18.31

COMMITTING LOG ENTRIES

- Each node keeps a **commitIndex** and **lastApplied** index variable
- **PHASE I**
 - Leader: when log message replicated at a majority of follower logs (not state machines) ***- described next slide*
 - Leader increments its commitIndex
 - Followers set commitIndex to **Min (leader-commitIndex , index of last new log entry)**
- **PHASE II**
 - For any node (follower, leader):
 - If commitIndex > lastApplied
 - Increment lastApplied by 1
 - commit log[lastApplied] to **state machine** (kv-store)

December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
Institute of Technology, University of Washington - Tacoma
L18.32

UPDATING COMMIT-INDEX OF LEADER

• If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm, set commitIndex = N (§5.3, §5.4).

- **How leader determines when to update its commitIndex**
- Use a **majority consensus** of what has been committed at follower logs
- **Leader** maintains follower state arrays:
 - **nextIndex[]**: index of next log entry to send to follower
 - **matchIndex[]**: index of highest log entry known to be replicated (to log) at follower
- Find N, such that N > commitIndex_{leader}
- **and** a majority of matchIndex[i] ≥ N (from followers)
- **and** log_{entry}_{leader}[N].term == currentTerm_{leader}
- **then** set commitIndex_{leader} = N

December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
Institute of Technology, University of Washington - Tacoma
L18.33

RAFT CLUSTER MEMBERSHIP – A3

- Cluster discovery performed at startup
- Use any method:
 - Static file, UDP discovery (kv-store), TCP discovery (kv-store)
- One membership is discovered, it can remain static/fixed
- Nodes can go offline, come back online
- One a common configuration is propagated across the system, it can not be changed without restarting
- RAFT specifies a configuration change protocol where the system does a “hand-off” between an old and new configuration (section 6 of the paper)

December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
Institute of Technology, University of Washington - Tacoma
L18.34

A3 RAFT SIMPLIFICATIONS

- RequestVote() can be single threaded
 - AppendEntries() probably should have one thread per **follower**
- TCP client catch exceptions:
 - IOException – newSocket()
 - IOException – getOutputStream()
 - IOException – getInputStream()
 - **Leader** should catch exceptions, and retry requests indefinitely
 - Use socket method .setSoTimeout() to set a socket timeout in MS
- Node directory should generate and track nodeIDs
 - E.g. 1, 2, 3, 4, ... n
- Node directory should retrieve a node by ID, or IP/PORT


December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
Institute of Technology, University of Washington - Tacoma
L18.35

A3 RAFT SIMPLIFICATIONS - 2

- **Leader** election: if using a single thread for **election candidate** should retry RequestVote() up to 10 times for a **follower** then give-up and move to next **follower**
- Instead of pushing data to **followers** when put() or del() is received by **leader**, can wait until next scheduled heartbeat to **follower**


December 5, 2017
TCCS558: Applied Distributed Computing [Fall 2017]
Institute of Technology, University of Washington - Tacoma
L18.36

QUESTIONS



December 5, 2017
TCSS558: Applied Distributed Computing [Fall 2017]
Institute of Technology, University of Washington - Tacoma
L18.37

EXTRA SLIDES



38