Raft assignments

COS 418: *Distributed Systems*
Precept 9

Themis Melissaris and Daniel Suo

Agenda

- General observations
- Assignment 3: an example in designing an implementation
- Assignment 4: expanding on the example
- Assignment 5: avoiding common pitfalls

On incremental assignments

- We hear you: not having solutions to earlier assignments when later assignments depend on them is hard
- Beating the dead horse:
  – This reflects the reality more often than not in software engineering
  – This is also a forcing mechanism to really understand a distributed system and how to make good design choices

#1 reason for struggle: repeating logic

- Some examples
  - Using multiple state variables for one state
  - Handling heartbeat and AppendEntries are different (more relevant for A4)
  - Start new election from Candidate and Follower are different
  - Resetting timers

We don’t want our code to be

- **Rigid**: difficult to change; need to touch many places to make simple changes
- **Fragile**: changes break system in unexpected ways
- **Immobile**: hard to reuse logic / code

These adjectives caused people a lot of pain!
We’ll revisit throughout this precept

Assignment 3

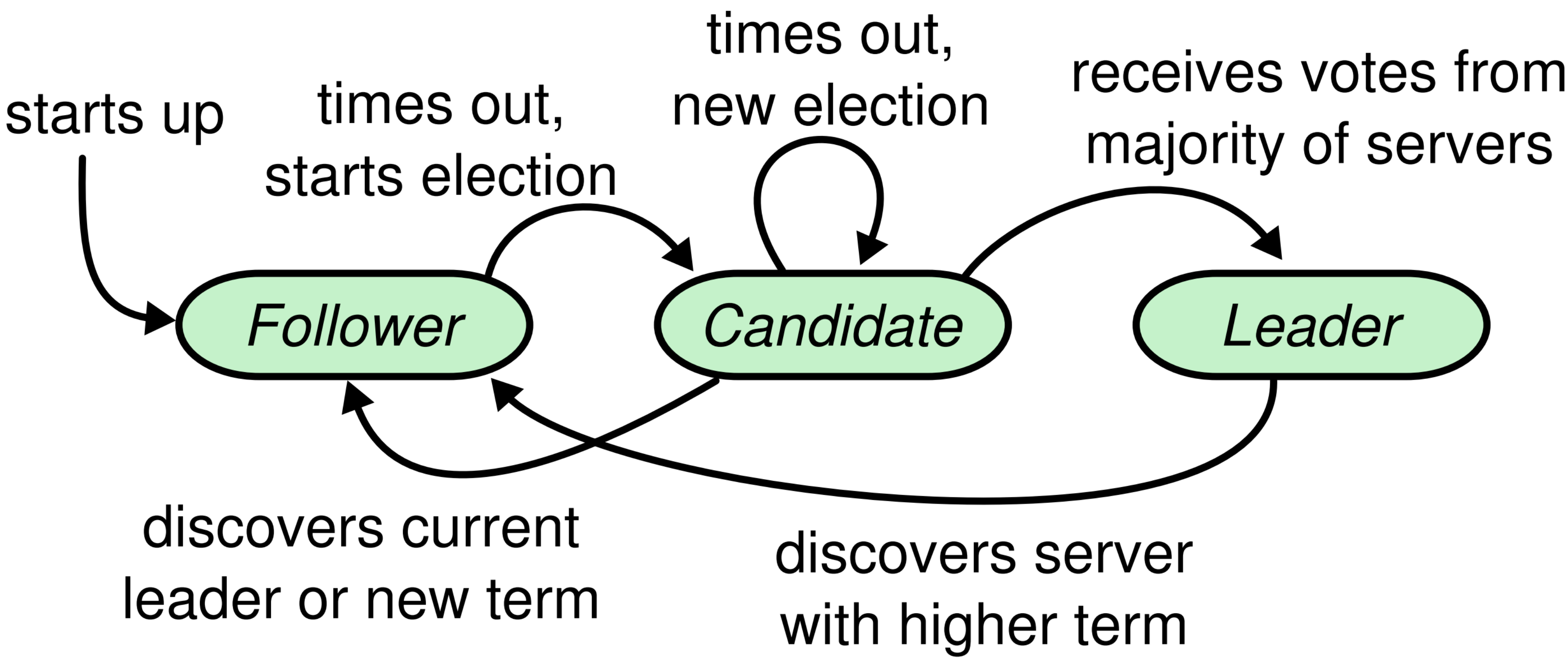
Why are we going over A3 again?

- Should be review
- We spend a lot of time in class describing systems
- You spend a lot of time in assignments implementing systems
- A quick (and simpler) example of how we go from a system description to system implementation
- There are many possible implementations!

Reasoning about state

- Assignment 3 asks us to implement a state machine (i.e., elections)
- Each raft server has a ‘state’ (Follower, Candidate, Leader)
- Raft server can change state according to certain rules

Reasoning about state



Reasoning about state

- What additional state must each server hold?
  - currentTerm
  - votedFor
- We will also assume each server has a Timer object, but there are other implementations to handle timeouts (e.g., timeout loops)

Reasoning about transitions (Follower)

- How can we become a follower?
  - When we first start
  - When we receive an RPC from a server with a higher currentTerm
- Followers can only become candidates (directly, anyway)

Reasoning about transitions (Candidate)

- How can we become a candidate?
  - When we are a follower and haven't received a heartbeat from a leader within the election timeout
  - When we are a candidate and haven't been voted leader or heard from a leader within the election timeout
- Candidates can become any of Follower, Candidate, or Leader

Reasoning about transitions (Leader)

- How can we become a leader?
  - When we are candidate and if we receive votes from majority of servers within election timeout
- Leader can become Follower
  - If we see a server with a higher term
  - Typically happens after we die or there is a partition

When do we change state?

- There is a timeout
  - Follower -> Candidate
  - Candidate -> Candidate
- We receive an RPC
  - Leader -> Follower
  - Candidate -> Follower
- We handle a response to an RPC
  - Leader -> Follower
  - Candidate -> Leader
  - Candidate -> Follower

Our code should reflect!

- Timing out
  - **resetTimer**
    - Create a timer if there isn't one (i.e., when we Make) and start goroutine to call handleTimer whenever there is timeout
    - Set timeout to heartbeat interval if we are leader, to randomized election interval if we are not (note, the same whether we are Follower or Candidate)
  - **handleTimer**
    - If leader, call sendAppendEntries
    - Otherwise, become candidate (note this logic is the same if we are Follower or Candidate)
- Receiving an RPC
  - **RequestVote**: specified in paper (don't need to implement the whole thing)
  - **AppendEntries**: specified in paper (don't need to implement the whole thing)

Our code should reflect!

- Handling RPC responses
  - **handleRequestVoteResponse**: specified in paper (don't need to implement the whole thing)
  - **handleAppendEntriesResponse**: specified in paper (don't need to implement the whole thing)
- Sending RPCs
  - **sendRequestVote**: send RequestVote in separate goroutine to each server; call handleRequestVoteResponse on response
  - **sendAppendEntries**: send AppendEntries in separate goroutine to each server; call handleAppendEntriesResponse on response

Some details (assuming architecture in previous slides)

- Resetting timer
  - When we start
  - Whenever we handle timeout
  - Whenever we change state
- Locking / unlocking (when do we modify state?)
  - When we handle timeout
  - When we receive an RPC
  - When we handle a single RPC response
- Resetting votedFor to null (or -1)
  - When we become follower except in AppendEntries

Assignment 4

High-level overview

- Assume architecture from earlier slides
- Part I
  - Modify all functions involving volatile state or the log (basically everything except Timer stuff)
- Part II
  - Correctly handle persistent state

Part I: sendRequestVote

- May have already done for A3
- Set RPC arguments
  - lastLogIndex: length of the candidate’s log (index of candidate’s last log entry)
  - lastLogTerm: if we have more than one entry, term of the last log entry

Part I: RequestVote

- Need to add check that the candidate’s log is at least as up-to-date as receiver’s log
- See section 5.4.1 in original paper for details

Part I: handleRequestVoteResponse

- If we become leader, initialize nextIndex and matchIndex
  - nextIndex: initialize to the length of the leader’s log (leader last log index + 1)
  - matchIndex: initialize to 0 (why?)

Part I: sendAppendEntries

- Which log index should we send to followers?
- If our last log index is greater than or equal to the nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex

Part I: AppendEntries

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), 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 last new entry)

Part I: handleAppendEntriesResponse

- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- 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).

Part II: persisting state

- Only need to read from persistent storage in Make
- Persist whenever we change currentTerm, votedFor, or log; easy, right?
- This becomes hard if similar logic is sprinkled throughout your code. Besides in Make,
  - log changes in AppendEntries
  - votedFor changes during elections and in AppendEntries receive/handle response
  - currentTerm changes whenever we become Follower
- Not required, but for completeness
  - Should persist before changing in memory; most people did not do this (note that you do need to persist before responding to RPCs!)

Some details

- Locking / unlocking (when do we modify state?)
  - When we handle timeout
  - When we receive an RPC
  - When we handle a single RPC response
  - Start, Kill
- log, matchIndex, nextIndex
  - You should reason about the state of these arrays just as we did for Assignment 3!
  - Many people just started implementing by translating Figure 2 into code; without understanding, debugging will be much harder!

Assignment 5

High-level overview

- Should only depend on public Raft API
- Part I: implement Put(key, value), Append(key, value), Get(key)
  - Must have sequential consistency!
- Part 2: handling failures
  - Deal with duplicate requests

common.go

- Should be relatively quick!
- What additional field(s) do we need to put in PutAppendArgs?

```go
// Put or Append
type PutAppendArgs struct {
	// You'll have to add definitions here.
	Key   string
	Value string
	Op    string // "Put" or "Append"
	// You'll have to add definitions here.
	// Field names must start with capital letters,
	// otherwise RPC will break.
}
```

- What about GetArgs?

client.go

- We just need to properly construct the RPCs to the server
  - Get
  - PutAppend
- These should follow easily once we have the Arg structures from common.go

server.go

- The hard part ☺

Debugging

General tips

- Go debugging isn't great
- If you use print statements, make sure you use unbuffered output (i.e., use stderr)
- Use go's playground: https://play.golang.org/
- Create subsets of the evaluation tests
- Test incrementally:
  - Think about invariants and create appropriate tests

Go slow to go fast