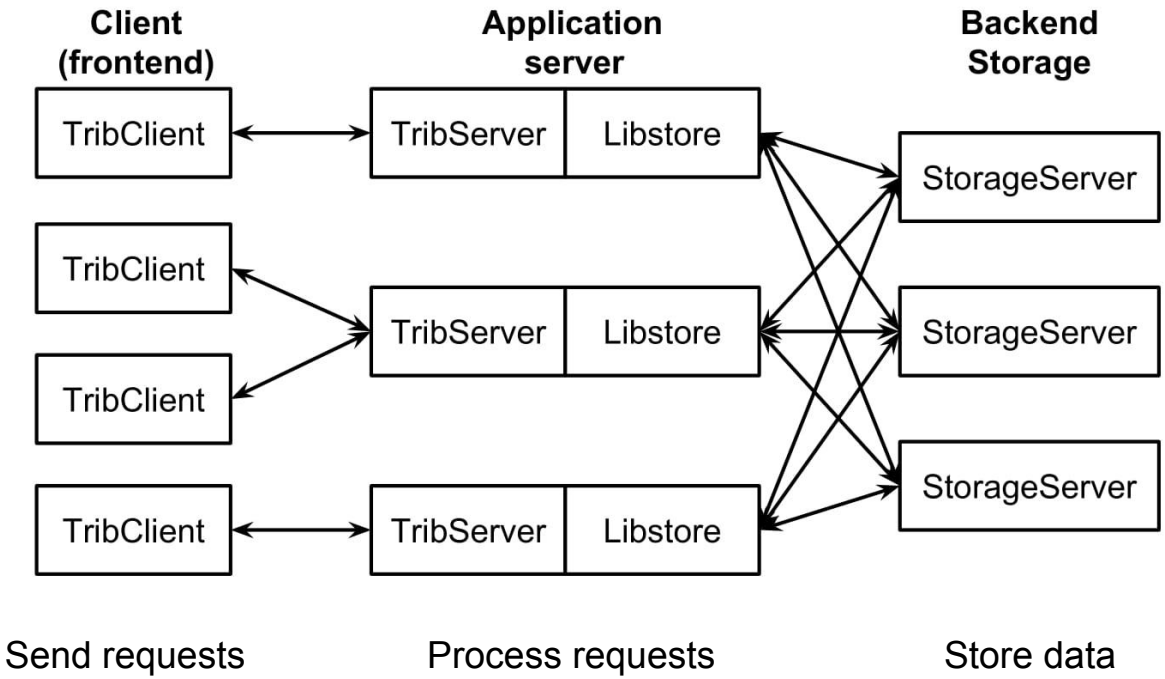
P3: Tribbler

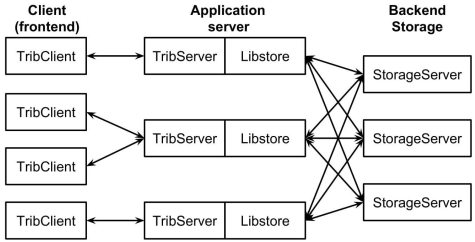
15-440/640 Fall 2021

Start Early for P3!

- Checkpoint is due on **Tuesday Nov 23** (6 days from now)
- Final project due on **Friday Dec 3** (~2.5 weeks from now)
- Contact Han (Head TA) if you don’t have a partner yet

A 3-Tier Architecture for P3

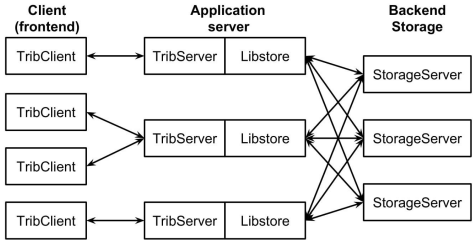




Client Layer (Already implemented!)

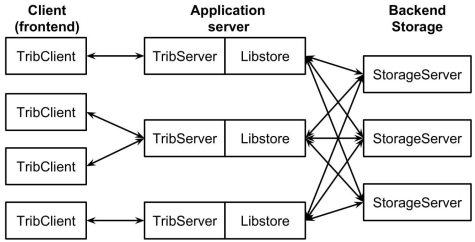
- Calls `TribServer` RPCs to forward requests to application layer
  - `CreateUser`
  - `AddSubscription`
  - `RemoveSubscription`
  - `GetFriends`
  - `PostTribble`
  - `DeleteTribble`
  - `ModifyTribble`
  - `GetTribbles`
  - `GetTribblesBySubscription`
- All RPCs in P3 are real RPCs: they use the `net/rpc` package

Application Layer (You implement this!)



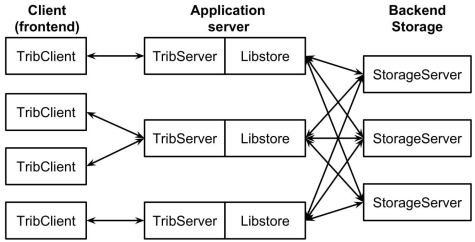
- A `Libstore` struct is embedded in a `TribServer`
  - `TribServer` RPCs registered in provided `NewTribServer` method
- The `TribServer` handles RPCs sent from the `TribClient`
  - Replies with one of five `Status`es (`rpc/tribrpc/proto.go`)
- The `Libstore` provides transparent access to persistent storage
  - `Libstore` must register `LeaseCallback` RPCs
  - `TribServer` calls regular `Libstore` methods, not `StorageServer` RPCs
    - E.g. `Get`, `Put`, `Delete`, `GetList`, `AppendToList`, `RemoveFromList`
  - `Libstore` serves two additional functions: **Request Routing** + **Caching** (see later)

Libstore: Request Routing



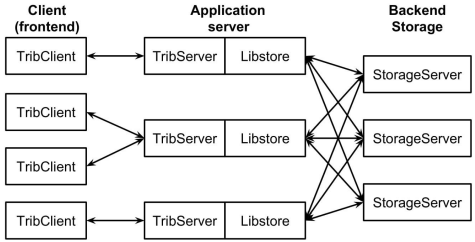
- Route a request to correct storage server based on its key
  - Assume multiple `StorageServer`s can exist, one of which is the Master
  - On initialization, set up Consistent Hashing ring with Master Node
  - Contact all available `StorageServer`s and cache connections
    - You'll need those for request routing later
  - Generate key using `util/keyFormatter.go`, Partition using `StoreHash`
- Call `StorageServer` RPCs defined in `rpc/storagerpc/rpc.go`
  - `GetServers` (see above)
  - `Get`, `GetList`, `Put`, `Delete`, `AppendToList`, `RemoveFromList`

Libstore: Lease-Based Caching



- Keep a local cached copy of data (in some hash table)
- Return data from cache if it holds a valid lease
- Otherwise, contact appropriate `StorageServer`
  - `Get`, `GetList`, `Put`, `Delete`, `AppendToList`, `RemoveFromList`…
- Handle `RevokeLease` RPC calls from `StorageServer`s

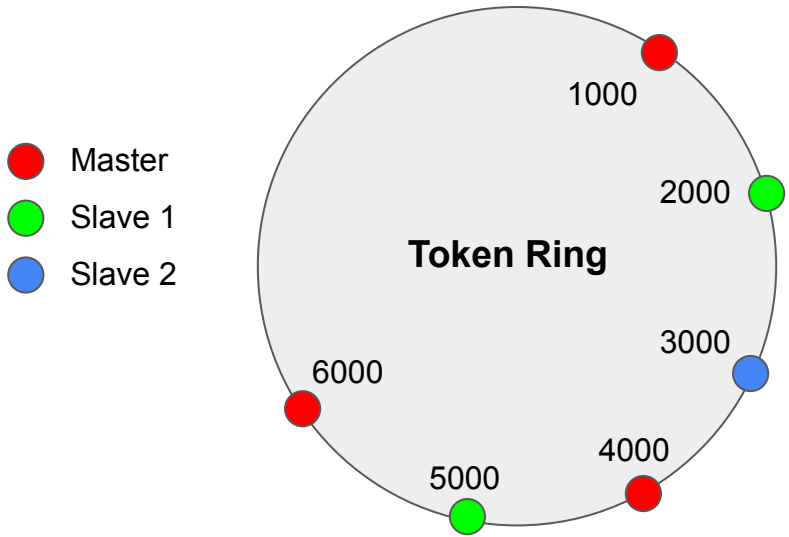
Storage Layer (You implement this!)



- Each server is either Master or Slave
  - Each has its own `uint32 VirtualID`s for Consistent Hashing
  - Register RPC handlers in `NewStorageServer` (consult `TribServer`/ `TribClient`)
- Master server coordinates slave servers on initialization
  - Handle `RegisterServer` RPC's from slave servers
  - Replies with OK and list of servers if all Slaves have registered
  - Report status when Libstore calls `GetServers`
- Slave Servers register with Master via `RegisterServer`
  - Wait for OK reply from Master
  - Else, sleep for 1 second and retry
- Master server knows how many servers to expect
  - Assume this number is static

Consistent Hashing

- Generate keys using `util/keyFormatter.go`, partition with `StoreHash`
  - E.g. Generate new `UserKey` with `FormatUserKey`, Get `uint32` after hashing
- Form token ring with the `VirtualID`s of each `StorageServer`
- Match hashed key to `StoragerServer` with “successor” `VirtualID`
- Examples
  - Key(1100) → Slave 1 (2000)
  - Key(2000) → Slave 1 (2000)
  - Key(3500) → Master (4000)
  - Key(6001) → Master (1000)



More on Leases (Libstore)

- Frequent READs are faster with caching
- `LeaseMode`s: `Always`, `Never`, `Normal` - see `libstore_api.go` and `NewLibStore` specification
- What is `Normal`?
  - Ask for lease when you receive `QueryCacheThresh` queries within `QueryCacheSeconds`
- Cache entry for `LeaseGuardSeconds`
  - Delete from cache once the lease expires
  - Delete from cache when lease is revoked by `StorageServer` (next slide)
- Forward WRITEs and DELETEs directly to `StorageServer`

More on Leases (StorageServer)

- Grant lease on a READ request if
  - `WantLease` (in `GetArgs`) `== true`
  - The lease is not currently being revoked
- Revoke all existing leases on WRITE/DELETE
  - Stop granting new leases
  - Call `RevokeLease` (part of `Libstore` API) on leaseholders and block until
    - Every leaseholder has responded, OR
    - `LeaseSeconds` and `LeaseGuardSeconds` have elapsed
  - Do not grant new leases, do not allow concurrent updates

Atomicity and Consistency

- Each update should be atomic (all or nothing)
  - Operations should block until they have either succeed or failed
- When an update returns successfully, future reads should reflect that update
- Don’t worry about “cross-key consistency”

1. `TribClient2`: `PostTribble("a", "first post!")`. Returns successfully.
2. `TribClient1`: Calls `GetTribblesBySubscription` (subscribed to `"a"`, `"b"`).
3. `TribClient2`: `PostTribble("a", "a was here")`. Returns successfully.
4. `TribClient3`: `PostTribble("b", "b is sleeping")`. Returns successfully.
5. `TribClient1`: Returns from `GetTribblesBySubscription`.

The return value for `GetTribblesBySubscription` in step 5 could be any of:

- `["a":"first post!"]`
- `["a":"first post!"], ["a":"a was here"]`
- `["a":"first post!"], ["b":"b is sleeping"]`
- `["a":"first post!"], ["a":"a was here"], ["b":"b is sleeping"]`

Checkpoint: Hints and Advice

- Support only a single `StorageServer`
  - No Request Routing/ Consistent Hashing
  - Every request goes straight to the Master Server
- Don’t worry about lease-based caching
  - Test suite sets `LeaseMode = Never`
- Use `keyFormatter.go` for simplicity
  - Store a user’s Subscriber List and Tribble List under respective keys
  - Store users and tribbles individually (not as lists)
- Note that P3 is more modular than P1
  - Each part correct → Should be correct overall
  - Test cases are also more modular

Final Submission: Hints and Advice

- Now you have to worry about routing and hashing
  - Write utility functions, don't repeat yourself
  - "`sort`"/ "`time`" packages may be helpful
  - Number of RPC calls/ bytes transfers will be checked
  - Use `LeaseMode = Always` for debugging
  - Wait as little as possible when revoking leases (else `stresstest` can time out…)
- Think about maintaining consistency without hindering performance
  - How to handle lease conflicts?
  - How to revoke multiple leases at once?
  - More granular locking on users/ tribbles/ other shared data?

Questions