P3: Tribbler

15-440/640

Timeline & Logistics

- Start early, start early, start early.

Project Release	**Tuesday, November 13, 2018, at 12pm**
Checkpoint Due	**Wednesday, November 21, 2018 at 11:59 pm**
Final Due	**Saturday, December 1, 2018 at 11:59 pm**
Submission limits	**15 Autolab submissions per checkpoint**

- P3 is a group project: please register your teammate before you make any submission.
  - *You should work with a partner who is also in the class*

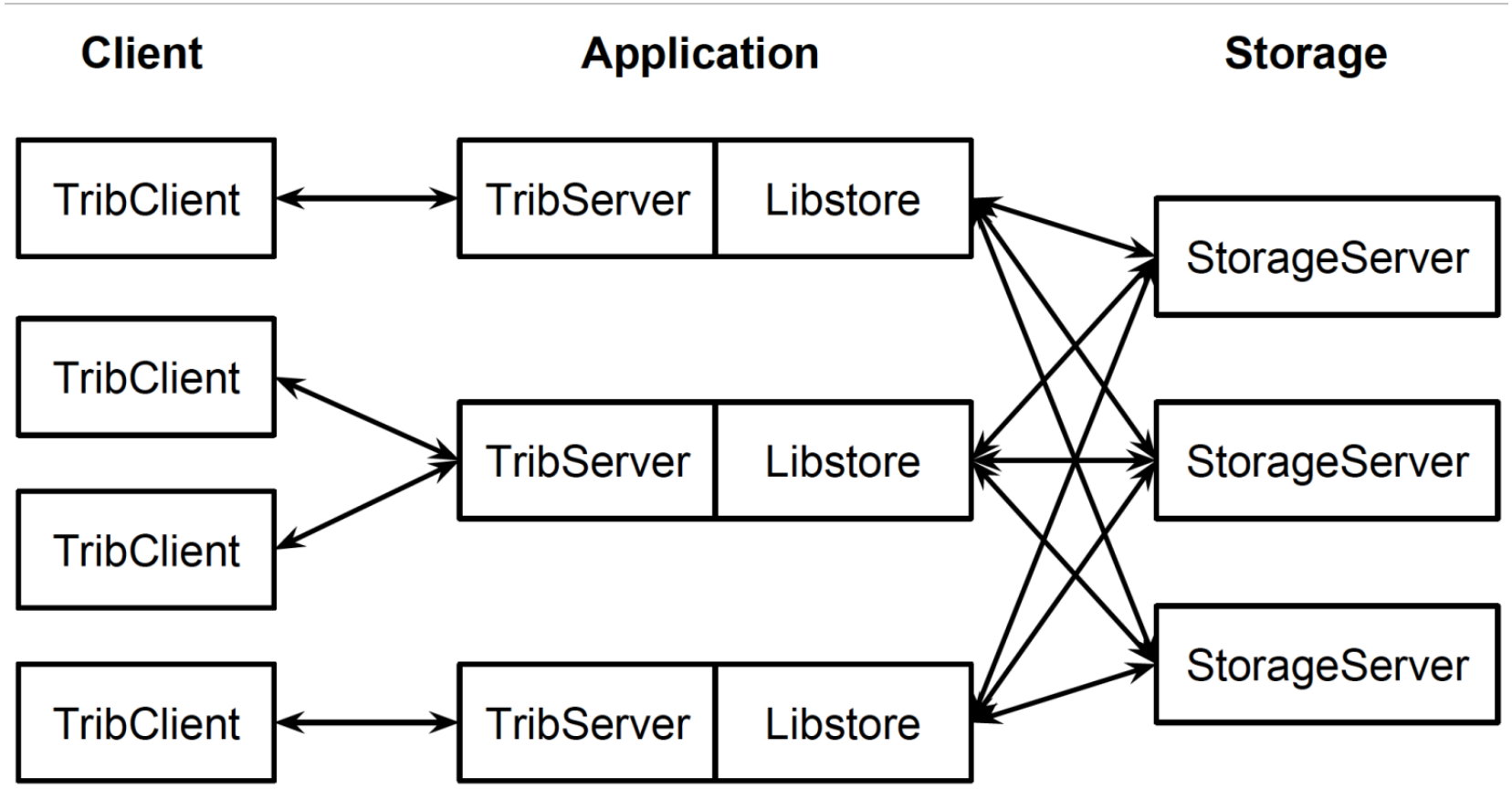
Timeline & Logistics

- With the P3 final, submit a 2-page design document (worth 10% of the grade). Tar it with your source code (refer to writeup/README.md for details)

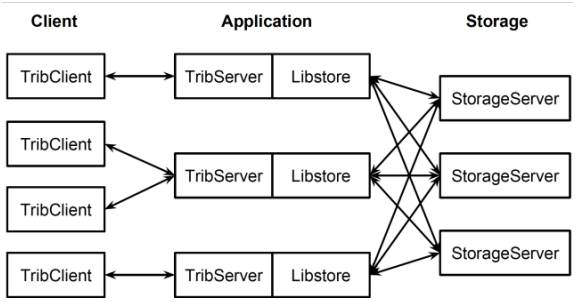
Describe major aspects of your design

- The data storage policy on the Storage Servers
- The synchronization strategy you use
- Your principal data structures and algorithms
- Division of work in your team
- Any other design decisions you would like us to be aware of

P3 Overview

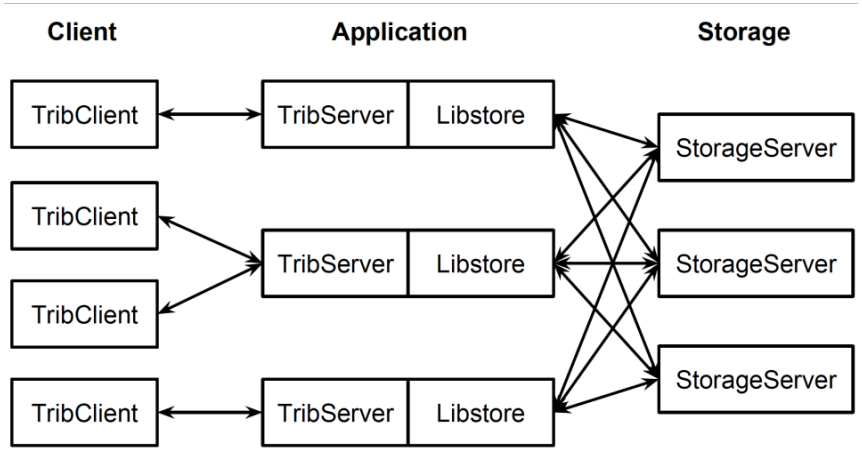

Client
Application
Storage
TribClient
TribClient
TribClient
TribClient
TribServer
Libstore
TribServer
Libstore
TribServer
Libstore
StorageServer
StorageServer
StorageServer

Application Layer



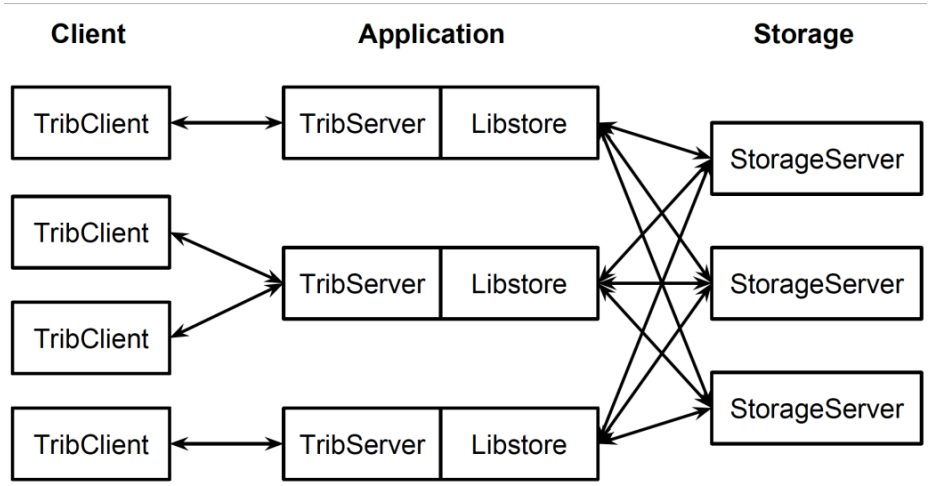
- RPC based Tribbler Server that supports
  - subscribing/unsubscribing to users
  - posting/retrieving/deleting Tribbles, etc
- Two Components
  - TribServer
    - Tribbler clients will interact with the Tribbler servers using Go RPC.
    - All RPC calls reply with a integer status, which is defined in the `rpc/tribrpc` package.
  - LibStore
    - Each Tribble server will create and use an instance of the Libstore library to provide efficient and transparent access to the storage servers

TribServer



- Functions
  - *CreateUser*
    - *tribrpc.OK, tribrpc.Exists*
  - *AddSubscription*
    - *tribrpc.NoSuchUser, tribrpc.NoSuchTargetUser, ...*
  - *RemoveSubscription*
  - *GetFriends*
  - *PostTribble*
  - *DeleteTribble*
    - *tribrpc.NoSuchPost, ...*
  - *GetTribbles*
  - *GetTribblesBySubscription*

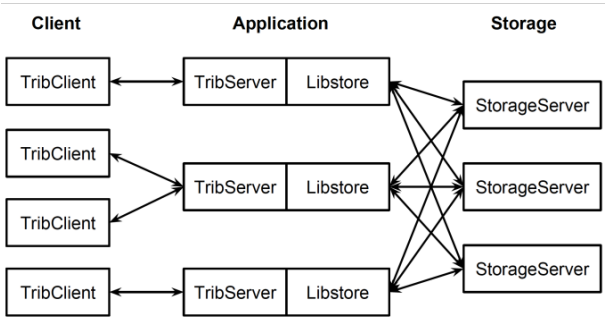
LibStore



Two major functions:

- Request Routing
  - Given a key, the *Libstore* must route the request to the appropriate storage server
  - *Libstore* contacts Master Storage Node using *GetServers* RPC and creating a consistent hashing ring
  - If receive an OK, *Libstore* will begin communicating via RPC
  - *Libstore* should cache any Storage server connections (reuse the connection)

LibStore

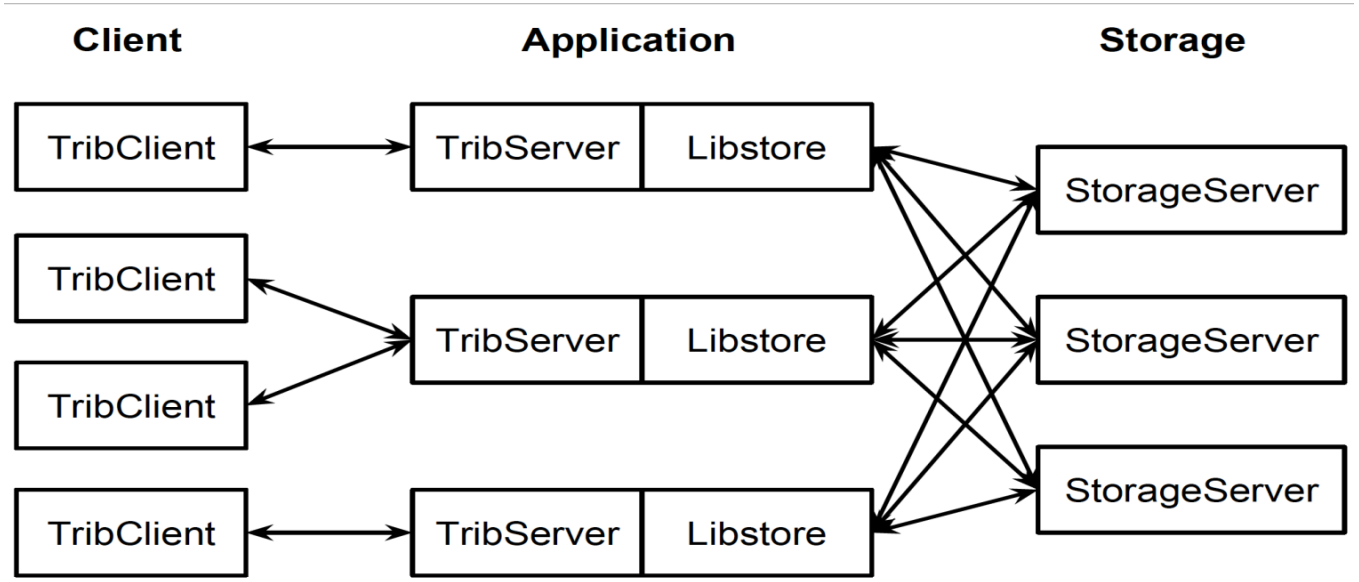


- Lease based Caching
  - `Libstore` must keep a local copy (e.g., in a small hash table) respond to lease expiration/revocation events from the storage servers
  - `Libstore` looks for entries with valid lease in the cache; if absent, forwards the request to the appropriate storage server

Storage Server

Storage Server

- Each server is a key-value store
- Servers have a list of virtual ids and together form a Consistent Hashing Ring.
- Master-Slave architecture



Initialization & Setup

- Master Server
  - Listens to incoming connections from other Servers
  - Waits for Slaves to join the Consistent Hashing Ring
  - Replies with *OK* if all slaves have registered; else, replies *NotReady*
- Slave Servers
  - Register with Master Server on startup, using *RegisterServer* RPC
  - Wait for *OK* status and a slice consisting all servers (including itself) in the ring
  - If receives *NotReady*, sleep for 1 sec and call *RegisterServer* again

Initialization & Setup

- Starting Servers with `srunner`

```
./srunner -port=9009 -N=3 -vids=1000,4000,6000 # master
./srunner -port=9010 -master="localhost:9009" -vids=2000,5000 # slave 1
./srunner -port=9011 -master="localhost:9009" -vids=3000 # slave 2
```

- Assume: List of servers is static throughout

Partitioning and Sharding

- Use *`util/keyFormatter.go`* to generate keys
  - FormatUserKey: key for a user id
  - FormatSubListKey: key for user subscriptions
  - FormatTribListKey: key for user tribs
  - FormatPostKey: key for a specific tribble
  - Helper functions separate user ID with colon, eg. , daniel:usrid or yuvraj:post-23ac9138d7
- Partitioning keys using *`StoreHash(key)`*
  - Partitioning must be based on **only** substring before the colon (eg. daniel or yuvraj)
  - eg. yuvraj:radio and yuvraj:head should be handled by the same storage server

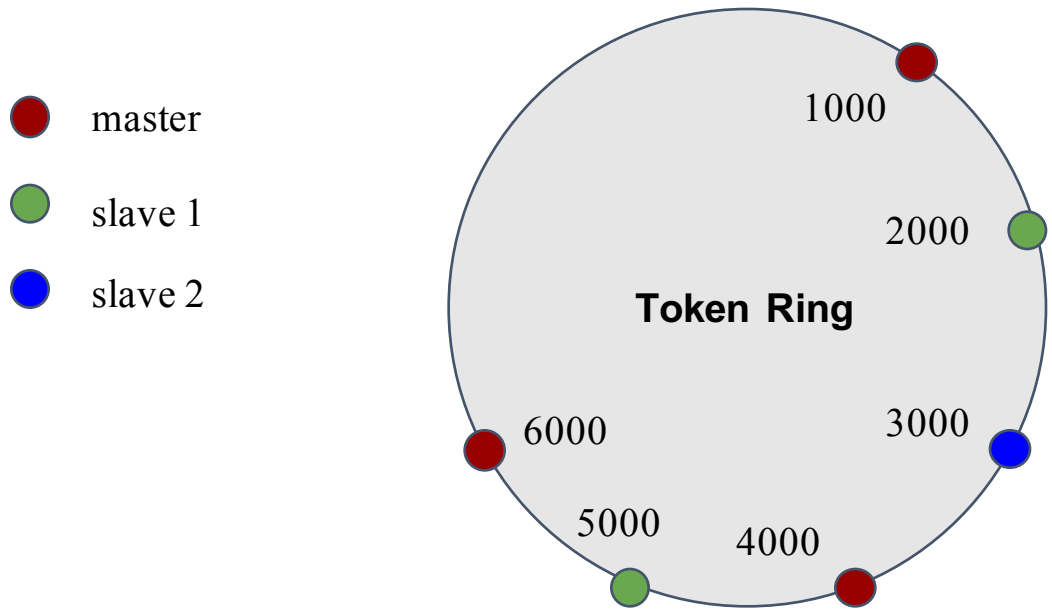
Partitioning & Sharding

- Each Storage Server stores subset of key/value pairs by partitioning using **Consistent Hashing**.
- How to perform Consistent Hashing:
  - Every Node is assigned a list 32 bit integers (virtualIDs) in range [0 to 2^{32}-1]
  - For a given key, node ID = successor of hash(key)
  - Eg. If node#1 is has [10555], node#2 is at [19200], and hash(key) = 13232, then key will be handled by node#2

Partitioning & Sharding

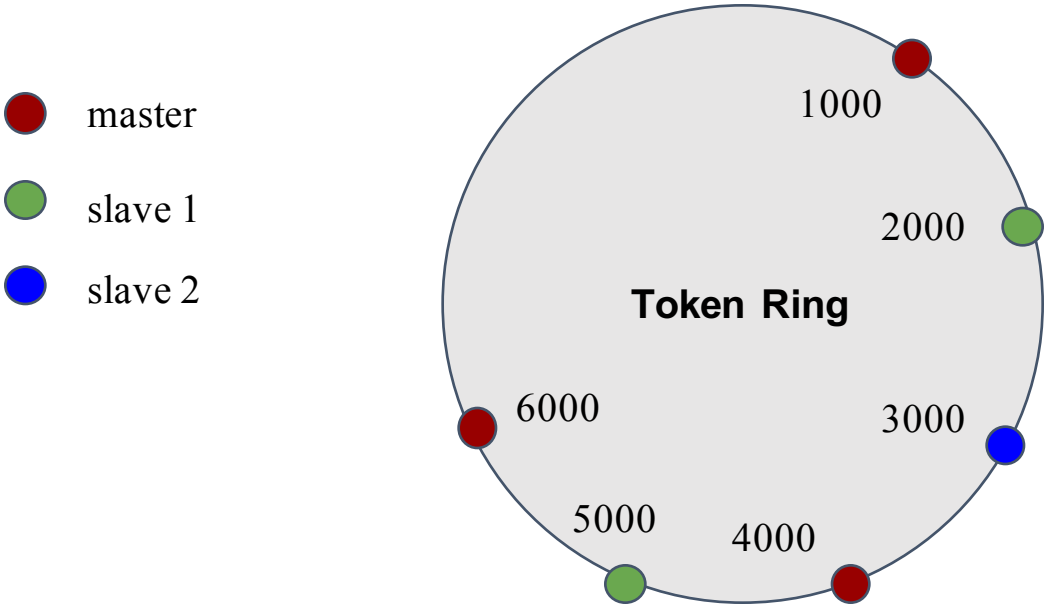
- Storage servers on a global token ring

```
./srunner -port=9009 -N=3 -vids=1000,4000,6000 # master
./srunner -port=9010 -master="localhost:9009" -vids=2000,5000 # slave 1
./srunner -port=9011 -master="localhost:9009" -vids=3000 # slave 2
```



Partitioning & Sharding

- successorID: ***after or equal*** to the key's hash value.
- Examples
  - Key(1100) → slave 1 (2000)
  - Key(2000) → slave 1 (2000)
  - Key(3500) → master (4000)
  - Key(6001) → master (1000)



Libstore & TribServer

Handling Front-end APIs, Leasing, Atomicity

Libstore & TribServer

- Libstore does:
  - Request routing to appropriate backend storage server
  - Handle leases on keys
- TribServer does:
  - Handle Trib Client APIs
  - Translate Client APIs to a set of Storage APIs and Pass to Libstore
- Libstore & TribServer shares the same HTTP Handler (for details, refer to 5.2 on write up)

Lease Based Caching

- Frequent READs are faster with caching
  - Eg: users with huge number of subscriptions (Get and GetList can be faster)
- Three Lease modes
  - Always, Never, Normal
  - Refer to Libstore/TribServer code for setting up lease mode

Lease Based Caching

- Read Queries
  - Look in cache for a valid lease and return if present
    - If not present, *LibStore* can get a lease from a *StorageServer* [*GetArgs.WantLease* is set]
  - Under *Normal* leasing
    - If *QueryCacheThresh* queries in *QueryCacheSeconds*, then ask for lease
  - Storage server provides lease for *LeaseSeconds* period (*LeaseGuardSeconds* - for clock drift) and keeps track of its leases

Lease Based Caching

- Write Queries: Directly forwarded to Storage Server
  - New writes or writes on unleased key can be handled without blocking on storage server to revoke leases
  - For updates, should block until all `RevokeLease` calls reply OK before performing the update
- Delete Request
  - Forward to Storage Server
- When Storage Server wants to revoke lease
  - Delete from cache

Leasing on Storage Server

- Read Request
  - Grant lease if *WantLease* is true and if there no revoke lease operations occurring concurrently for that key
  - Grant for *LeaseSeconds* + *LeaseGuardSeconds* amount of time
- Write/Delete Request
  - Call *RevokeLease* to all libstores and block the update until finish notification
- Handle concurrency for leases

Atomicity and Consistency

- Each update should be atomic
  - All or none
  - Application layer only returns if the update succeeded or failed
- Consistency should be maintained across updates
  - If any previous update returned success then the future reads should reflect that update
  - Cross-key consistency need not be ensured

Atomicity & 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:

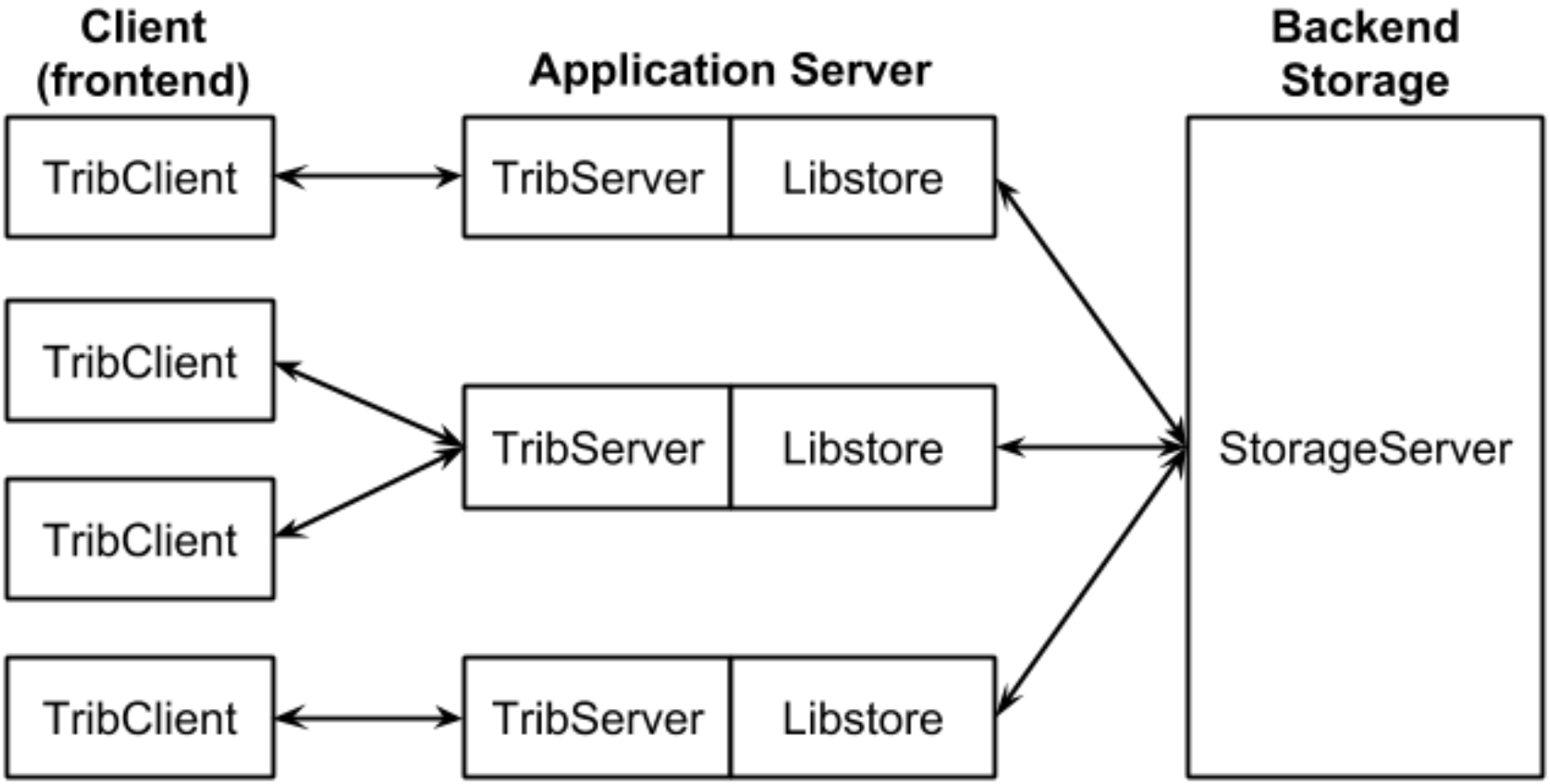
- ["a":"first post!"]
- ["a":"first post!"], ["a":"a was here"]
- ["a":"first post!"], ["b":"b is sleeping"]
- ["a":"first post!"], ["a":"a was here"], ["b":"b is sleeping"]

Checkpoint

Due: November 21 11:59PM

Late submission due: November 23 11:59PM

Checkpoint


Client
(frontend)
Application Server
Backend
Storage
TribClient
TribClient
TribClient
TribClient
TribServer
Libstore
TribServer
Libstore
TribServer
Libstore
StorageServer

Checkpoint (Contd)

Only a single Storage Server

What does this simplify?

- No Request Routing
- No Consistent hashing
- No Lease-Based Caching

All requests directly go to the Storage Server (master) to fetch/store data

Checkpoint Hints

- How to use Go RPC?
  - RPC Client: Refer to TribClient code
  - RPC Server: Refer to the comments in the rpc package in starter code (below from *rpc/tribrpc/rpc.go*)

```
tribServer := new(tribServer)

// Create the server socket that will listen for incoming RPCs.

listener, err := net.Listen("tcp", fmt.Sprintf(":%d", port))

// Wrap the tribServer before registering it for RPC.
err = rpc.RegisterName("TribServer", tribrpc.Wrap(tribServer))

// Setup the HTTP handler that will server incoming RPCs and
//  serve requests in a background goroutine.
rpc.HandleHTTP()
go http.Serve(listener, nil)
```

Checkpoint Hints

- What should the keys (in the key-value store) look like?
  - Use *`util/keyFormatter.go`* to format
    - User key, Post Key for tribble, Subscribers List userkey, Tribbles List user key
  - No other specific key formats required for the project.

Checkpoint Hints

- TribServer: How to implement CreateUser function?
  - FormatUserKey
  - Do a Get on the LibStore instance to check if the user exists
    - If yes, set status to Exists and return
  - Do a Put on the LibStore instance and set status OK if success

Final (Post Checkpoint)

Due: December 1 at 11:59PM

Late Submission Due: December 3 at 11:59PM

Post Checkpoint ToDo's

- Request Routing in libstore
- Setting up the consistent hashing ring
- Caching & Leasing on Libstore
- Leasing on Storage server
- Performance improvements

- You need to make design decisions on your own for P3 (not like Raft or LSP)
  - Write your decisions on Report!

Post Checkpoint Hints

- For Storage Servers, we will check for performance including
  - Wall clock time
  - Number of calls to storage server when things can be cached
  - Calls that Must go to storage server should not be served from cache
  - And more on the same lines..

Post Checkpoint Hints

- Request routing is easy, start with that to get the flow
- You can use Always Leasing mode for debugging
- Handle timeout based revoking properly
- Cache & Reuse connections, assume failures
- Maintain freshness of LibStore Cache
- Maintain freshness of Storage Server Lease metadata - remove on lease expiry

Post Checkpoint Hints

- Handle concurrency in LibStore and Storage Server
  - Libstore & Storage Server data
  - Lease conflicts
  - Fine grained locking on users - many approaches

Thank You