University of British Columbia

Electrical and Computer Engineering 411

Design of Distributed Software Applications

Project Phase 02
Key Value Store

Group S5
Scott Hazlett #63215065
Sahay Ishan #60038106

6th March 2014

Deployment

PlanetLab Nodes:

- `planetlab2.cs.ubc.ca:11111`
- `cs-planetlab4.cs.surrey.sfu.ca:11111`
- `planetlab03.cs.washington.edu:11111`

Design Decisions

The following are main design decisions and our motivations behind them. Please find attached another PDF file containing a UML Class Diagram of our *Service*.

Thread Pool and Limited Concurrent Sockets

For **scaleability** to clients, our *Service* instantiates one *WorkerThread* per accepted TCP client socket. This thread is tied to that client socket and handles one command. This *WorkerThread* instantiates a Command object and passes this to *HandlerThread* via a message queue.

We keep track of active sockets by maintaining a semaphore of active TCP sockets. When a client socket has been created, we decrement the semaphore. Later, when the WorkerThread has closed the socket, the semaphore is incremented.

The Service thread remains in a tight loop listening for incoming client connections and reserves two TCP sockets for rejecting clients. This provides **availability**.

To ensure that we stayed within PlanetLab's restriction of 50 concurrent TCP connections, we decided to err on the side of not exceeding 30. We reserved two TCP sockets for rejecting clients, 1 TCP socket for our personal SSH connection to the server, 1 TCP socket for the *ServerSocket* itself, and 5 more for any TCP sockets that may be in use, e.g. performing "RSYNC". In summary, our server socket listens on one TCP socket and can concurrently serve 42 clients as well as reject 2 clients.

Message Queue

We achieved **correctness** by enforcing in-order execution of commands via a queue.

Concurrent Database with Message Queue

We used a *HashMap* for our database because it provides constant time insert and retrieval.

We relied on a *HandlerThread* to be the sole accessor of this database. Each *WorkerThread* produces to the message queue and the one *HandlerThread* consumes from the message queue. A benefit on this design is that it relatively quick to add or remove elements from the queue, therefore reducing the synchronization overhead of concurrent access.

As well, because *HandlerThread* is the bottleneck in our design by being the sole accessor of the database, we minimized the code in *Command::execute()* which *HandlerThread* executes. For instance, we relegated the construction of the reply byte sequence to *Command::getReply()* which each *WorkerThread* executes.

This achieves better **performance**.

Command Design Pattern

This design pattern abstracts the implementation of each command from the executing Thread. To add a new command, we merely create a new class that extends Command and provide its implementation in *Command::execute()* method. The only other changes needed for this new command would be in *WorkerThread* that handles piping the results of the Command through the TCP connection. This cannot be avoided.

In this design pattern, *WorkerThread* is both the Client and Invoker, while *HandlerThread* is the Receiver.

This provides **scaleability** in development and **reduced maintenance**.

HashMap<ByteArrayWrapper, byte[]>

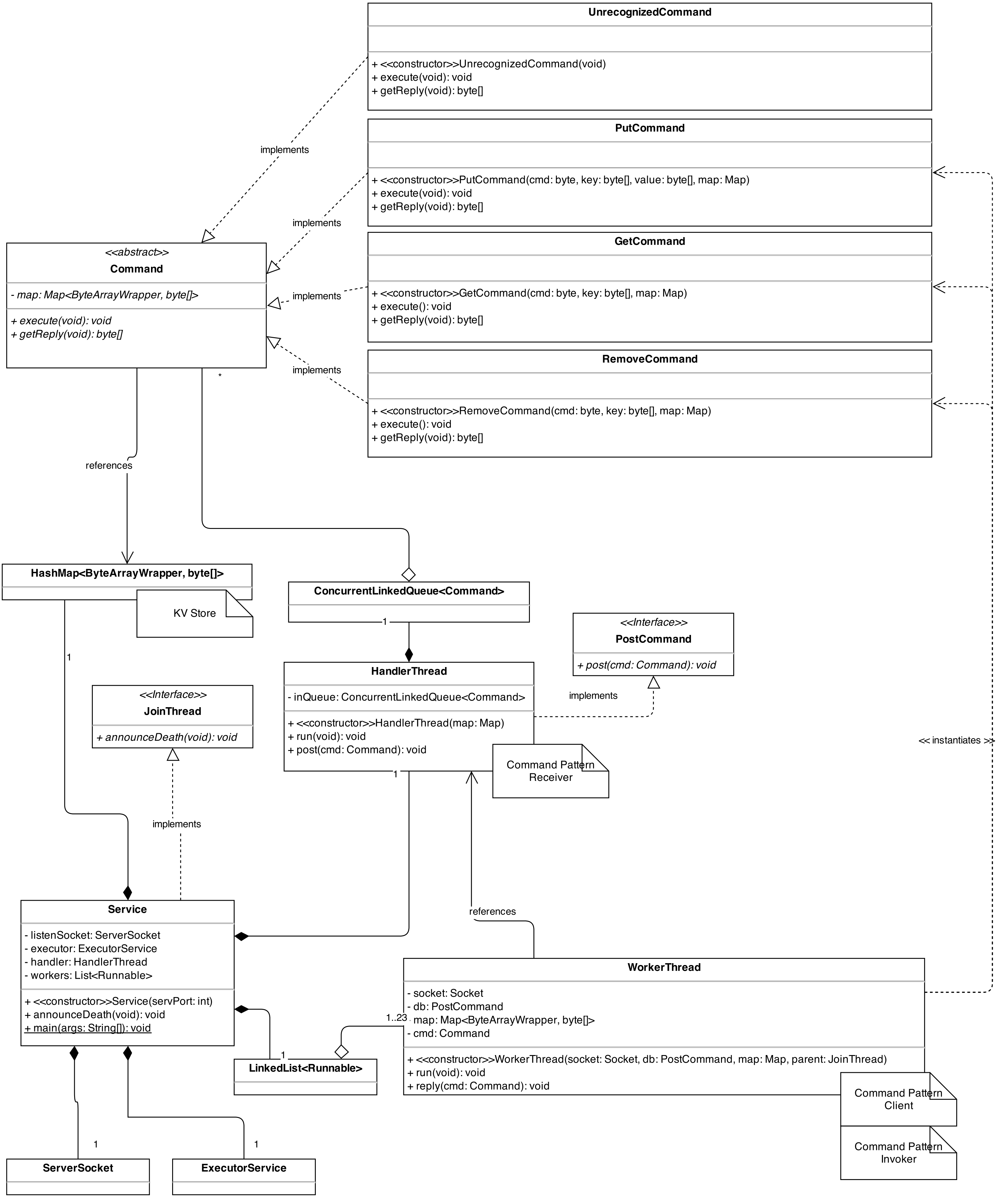
In Java, byte primitive arrays (*byte[]*) are treated as objects. Therefore, it had seemed feasible to implement a *HashMap* using *byte[]* as the key. However, in Java, you cannot guarantee equality with *byte[]* because you cannot override the *equals()* and *hashcode()* methods. Therefore, we wrapped a *byte[]* into *ByteArrayWrapper*[1] specifically so that we could implement our own comparator. The *byte[]* reference in *ByteArrayWrapper* is saved as a private variable to prevent mutation. We did not implement an accessor as we are only concerned with reproducing the value from the map by matching the corresponding key.

As well, by using our own *ByteArrayWrapper* we can add functionality easily in the future.

TCP Timeout

All TCP connections have a 5 second timeout from the moment that the *WorkerThread* is instantiated. This provides **timeouts** and **robustness** so that a TCP socket resource can be relinquished.

[1] This code obtained from http://stackoverflow.com/questions/1058149/using-a-byte-array-as-hashmap-key-java.


UnrecognizedCommand
+ <<constructor>>UnrecognizedCommand(void)
+ execute(void): void
+ getReply(void): byte[]
PutCommand
+ <<constructor>>PutCommand(cmd: byte, key: byte[], value: byte[], map: Map)
+ execute(void): void
+ getReply(void): byte[]
GetCommand
+ <<constructor>>GetCommand(cmd: byte, key: byte[], map: Map)
+ execute(): void
+ getReply(void): byte[]
RemoveCommand
+ <<constructor>>RemoveCommand(cmd: byte, key: byte[], map: Map)
+ execute(): void
+ getReply(void): byte[]
implements
implements
implements
implements
<<abstract>>
Command
- map: Map<ByteArrayWrapper, byte[]>
+ execute(void): void
+ getReply(void): byte[]
*
references
HashMap<ByteArrayWrapper, byte[]>
KV Store
ConcurrentLinkedQueue<Command>
1
<<Interface>>
PostCommand
+ post(cmd: Command): void
implements
<< instantiates >>
1
HandlerThread
- inQueue: ConcurrentLinkedQueue<Command>
+ <<constructor>>HandlerThread(map: Map)
+ run(void): void
+ post(cmd: Command): void
Command Pattern Receiver
1
<<Interface>>
JoinThread
+ announceDeath(void): void
implements
references
Service
- listenSocket: ServerSocket
- executor: ExecutorService
- handler: HandlerThread
- workers: List<Runnable>
+ <<constructor>>Service(servPort: int)
+ announceDeath(void): void
+ main(args: String[]): void
WorkerThread
- socket: Socket
- db: PostCommand
map: Map<ByteArrayWrapper, byte[]>
- cmd: Command
+ <<constructor>>WorkerThread(socket: Socket, db: PostCommand, map: Map, parent: JoinThread)
+ run(void): void
+ reply(cmd: Command): void
1..23
1
LinkedList<Runnable>
Command Pattern Client
Command Pattern Invoker
1
1
ServerSocket
ExecutorService