Large Scale Information Systems

Project 1b

Karthik Bellur (kvb26)

Sumukh SP (ss3345)

Prashanth Basappa (pb476)

Solution.zip Contents

- Proj1b.war: WAR file containing binaries to be deployed on Tomcat
- Proj1b: Exported Eclipse workspace containing complete source code that can be directly imported into Eclipse
- Screenshots: Folder which contains screenshots taken while testing the application on EBS with K=1 (on 3 total servers) and K=2 (on 5 total servers)
- ClassDiagram.jpg: UML Class Diagram containing all the classes used in the project along with the relationships between them

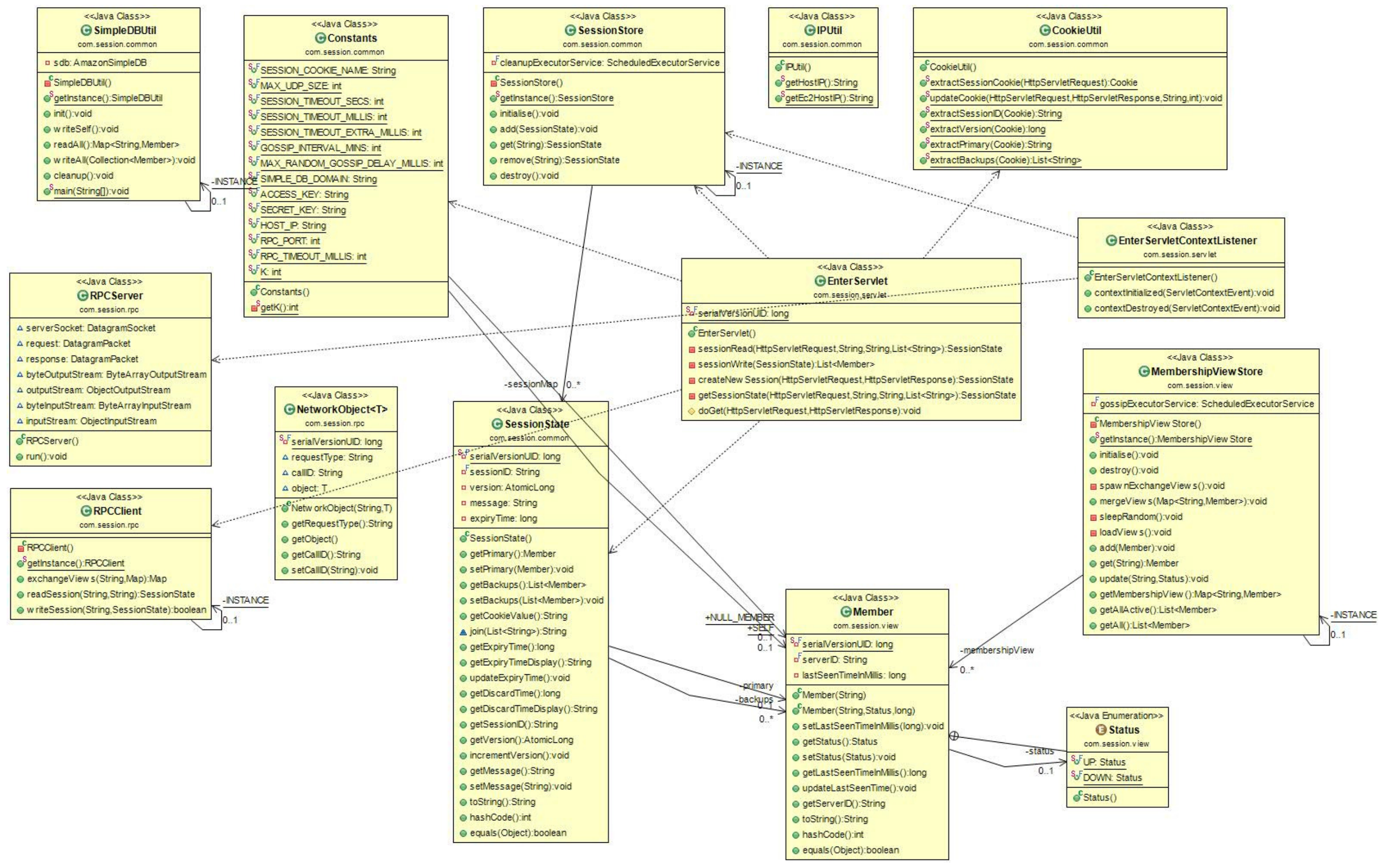
How to run on EBS?

1. On your AWS page, click on Elastic BeanStalk
2. Create New Application. Be sure to make the below choices:
   - Choose Tomcat Webserver and Load Balancing mode
   - Create environment inside an RPC
   - Use your key-value pair
   - Use a VPC security group which permits UDP and TCP traffic on appropriate ports. We use port 5305 for UDP and 8080 for TCP. If possible, choose the option to permit all traffic incoming/outgoing on all ports only for the duration of the test
3. On the Configuration Page, configure the following:
   a) Under "Scaling", configure Total number of instances required for testing
   b) Under "Software Configuration", configure AWS_ACCESS_KEY_ID and AWS _SECRET_KEY used for SimpleDB access
   c) Under "Software Configuration", set the required 'K' value using PARAM1
4. You can now access the application using: http://<env_name>/EnterServlet
   For example: http://kvbsessionapp-env.elasticbeanstalk.com/EnterServlet

Resiliency Testing on EBS

1. Navigate to your EC2 page from AWS where you can see your EC2 instances running. EBS will run as many instances as you configured earlier in the environment
2. Choose the appropriate instances you want to kill based on your test case. Select those instances and do Actions->Instant State->Terminate
3. EBS will now terminate those instances and bring up new ones in their place
4. Wait for a few mins until gossip completes execution on all machines. Gossip interval is set to 1 min for ease of testing. This can be changed in the Constants.java file if required. You can also monitor the tomcat logs to check if gossip has happened on a particular node and if so, the resultant membership view post gossip.
5. Now, click "Refresh" and you should be able to see the updated Membership View Table. The updated table will have the new members added with a Status UP and the terminated members shown in RED with a Status DOWN

Class Diagram



<<Java Class>> SimpleDBUtil com.session.common
sdb: AmazonSimpleDB
SimpleDBUtil()
getInstance():SimpleDBUtil
init():void
writeSelf():void
readAll():Map<String,Member>
writeAll(Collection<Member>):void
cleanup():void
main(String[]):void
-INSTANCE
0..1
<<Java Class>> Constants com.session.common
SESSION_COOKIE_NAME: String
MAX_UDP_SIZE: int
SESSION_TIMEOUT_SECS: int
SESSION_TIMEOUT_MILLIS: int
SESSION_TIMEOUT_EXTRA_MILLIS: int
GOSSIP_INTERVAL_MINS: int
MAX_RANDOM_GOSSIP_DELAY_MILLIS: int
SIMPLE_DB_DOMAIN: String
ACCESS_KEY: String
SECRET_KEY: String
HOST_IP: String
RPC_PORT: int
RPC_TIMEOUT_MILLIS: int
K: int
Constants()
getK():int
<<Java Class>> SessionStore com.session.common
cleanupExecutorService: ScheduledExecutorService
SessionStore()
getInstance():SessionStore
initialise():void
add(SessionState):void
get(String):SessionState
remove(String):SessionState
destroy():void
-INSTANCE
0..1
<<Java Class>> IPUtil com.session.common
IPUtil()
getHostIP():String
getEc2HostIP():String
<<Java Class>> CookieUtil com.session.common
CookieUtil()
extractSessionCookie(HttpServletRequest):Cookie
updateCookie(HttpServletRequest,HttpServletResponse,String,int):void
extractSessionID(Cookie):String
extractVersion(Cookie):long
extractPrimary(Cookie):String
extractBackups(Cookie):List<String>
<<Java Class>> EnterServletContextListener com.session.servlet
EnterServletContextListener()
contextInitialized(ServletContextEvent):void
contextDestroyed(ServletContextEvent):void
<<Java Class>> EnterServlet com.session.servlet
serialVersionUID: long
EnterServlet()
sessionRead(HttpServletRequest,String,String,List<String>):SessionState
sessionWrite(SessionState):List<Member>
createNewSession(HttpServletRequest,HttpServletResponse):SessionState
getSessionState(HttpServletRequest,String,String,List<String>):SessionState
doGet(HttpServletRequest,HttpServletResponse):void
<<Java Class>> RPCServer com.session.rpc
serverSocket: DatagramSocket
request: DatagramPacket
response: DatagramPacket
byteOutputStream: ByteArrayOutputStream
outputStream: ObjectOutputStream
byteInputStream: ByteArrayInputStream
inputStream: ObjectInputStream
RPCServer()
run():void
<<Java Class>> NetworkObject<T> com.session.rpc
serialVersionUID: long
requestType: String
callID: String
object: T
NetworkObject(String,T)
getRequestType():String
getObject()
getCallID():String
setCallID(String):void
<<Java Class>> RPCClient com.session.rpc
RPCClient()
getInstance():RPCClient
exchangeViews(String,Map):Map
readSession(String,String):SessionState
writeSession(String,SessionState):boolean
-INSTANCE
0..1
-sessionMap
0..*
<<Java Class>> SessionState com.session.common
serialVersionUID: long
sessionID: String
version: AtomicLong
message: String
expiryTime: long
SessionState()
getPrimary():Member
setPrimary(Member):void
getBackups():List<Member>
setBackups(List<Member>):void
getCookieValue():String
join(List<String>):String
getExpiryTime():long
getExpiryTimeDisplay():String
updateExpiryTime():void
getDiscardTime():long
getDiscardTimeDisplay():String
getSessionID():String
getVersion():AtomicLong
incrementVersion():void
getMessage():String
setMessage(String):void
toString():String
hashCode():int
equals(Object):boolean
<<Java Class>> MembershipViewStore com.session.view
gossipExecutorService: ScheduledExecutorService
MembershipViewStore()
getInstance():MembershipViewStore
initialise():void
destroy():void
spawnExchangeViews():void
mergeViews(Map<String,Member>):void
sleepRandom():void
loadViews():void
add(Member):void
get(String):Member
update(String,Status):void
getMembershipView():Map<String,Member>
getAllActive():List<Member>
getAll():List<Member>
-INSTANCE
0..1
+NULL_MEMBER
+SELF
0..1
0..1
-membershipView
0..*
-primary
-backups
0..1
0..*
<<Java Class>> Member com.session.view
serialVersionUID: long
serverID: String
lastSeenTimeInMillis: long
Member(String)
Member(String,Status,long)
setLastSeenTimeInMillis(long):void
getStatus():Status
setStatus(Status):void
getLastSeenTimeInMillis():long
updateLastSeenTime():void
getServerID():String
toString():String
hashCode():int
equals(Object):boolean
-status
0..1
<<Java Enumeration>> Status com.session.view
UP: Status
DOWN: Status
Status()

This web-application has been built using MVC style architecture. It is composed of a single view (Enter.jsp), a singleton controller (EnterServlet), several data models, caches and Utility classes. A brief description of each component and its working is presented below:

Member

A server instance in the network is modeled as a Member object. Member is Plain Old Java Object (POJO) consisting of 3 attributes:

i. ServerID: IP Address of the server
ii. Status: defined as an enum of 2 values – UP, DOWN
iii. LastSeenTime: this is the time that this Member was last seen communicating with another Member

SimpleDBUtil

This is a utility class used to manage all CRUD (Create, Read, Update, Delete) operations related to SimpleDB. The entire membership view is persisted into SimpleDB and periodic gossip with members of the network keeps the data updated with the latest status of the members. Members are stored as replaceable items with attributes. If the item with the given name is also present, then a *write()* replaces the old item updating its attributes. Supports 3 main operations:

i. *writeSelf()*: Every server calls this method on startup to register itself with SimpleDB
ii. *readAll()*: Returns membership view as stored on SimpleDB
iii. *writeAll()*: Writes a collection of members into SimpleDB (replaces an item if already present)

Note:- We use the atomic operation *batchPutAttributes()* to implement *writeAll()*. This guarantees that all/none of the writes succeed. It is not clear if *batchPutAttributes()* allows for concurrent writes. Anyhow, gossip eventually takes our system to a point of consistency. There we haven't used conditional updates while implementing *writeAll()*.

MembershipViewStore

- A singleton cache built to store the membership view of a server.
- Stores members in a concurrent hash table keyed by ServerID
- This membership view table is sent across to other members in the view during gossip
- Provides methods to read, add & update members in the view
- On initialization, the MembershipViewStore performs the following actions:
    - Write Self to SimpleDB
    - Loads the membership view from SimpleDB
    - Spawns an executor thread (referred to as GossipThread) which runs in the background and gossips with other members by calling *exchangeViews()*
- The GossipThread sleeps for GOSSIP_INTERVAL + random duration of time to avoid convoys
- Merges its local membership view with other member's view received during *exchangeViews()* such that the merged view represents the latest view of the network. This is done using the same approach given in the problem description
- The GossipThread executor is terminated just before the server goes down as part of the cleanup procedure

SessionState

A SessionState object holds the state of a client's session at any instant in time. There is 1 instance per session of a client. It is made Serializable since SessionState objects are passed around the network using RPC. It encapsulates the following attributes:

i. SessionID: string identifier used to uniquely identify a session
ii. Version: AtomicLong which is incremented on every request from a particular client
iii. Message: latest updated message text string
iv. ExpiryTime: time in millis when the session cookie will expire. Session ExpiryTime is set to 60 mins by default, but can be updated in the Constants file to the desired value
v. DiscardTime: time in millis when the SessionState object stored on servers will be garbage collected
vi. Primary: The primary server assigned to store this SessionState object currently
vii. Backups: A list of backup servers assigned to store this SessionState object currently

SessionStore

- A singleton cache built to store the SessionState objects on a server.
- Stores SessionState in a concurrent hash table keyed by SessionID
- Provides methods to read, add & update SessionState objects
- On initialization, the SessionStore spawns a background thread (referred to as the GarbageCollector thread) to clean up expired sessions from the local session table. Before the server goes down, this executor thread is terminated as part of a cleanup operation
- The GarbageCollector thread periodically iterates over the Session Table and removes every session entry for which: Current Time >= SessionEntry.discardTime
- The Session Table is protected from issues resulting from concurrent access of the data structure by using a "ConcurrentHashMap", which provides atomic operations (get/put) and fail-safe iterators, thereby protecting the session data from concurrency issues.

NetworkObject

An instance of NetworkObject is used to exchange data when performing RPC requests. For this reason NetworkObject is made Serializable. It includes 3 fields:

i. RequestType: Supported request types are SessionRead, SessionWrite and ExchangeViews
ii. CallID: Unique ID used while making an RPC call. This will be used to identify and match a response with its corresponding request
iii. Object: Any Serializable object which will be used as network payload to pass arguments which making RPC class

RPCServer

The RPC Server is a stub on the Application server which services requests from other Application Servers.
It is implemented as a class which implements a Runnable interface. A server thread is spawned when the application server boots up. The server communicates by exchanging UDP packets.
The RPC Server listens continuously on port 5305 and services requests received from RPC Clients running on other Application servers. There are 3 kinds of request types supported:-

1. SessionRead
    - On receiving a SessionRead request from a client, the server unpacks the NetworkObject which contains request_type=SessionRead and argument being SessionID for a session read request.
    - Performs a Lookup on its local session table for the SessionID and replies with a NetworkObject containing the SessionState object. If the object is not found it returns a null session object

2. SessionWrite
    - On receiving a SessionWrite request from a client, the server unpacks the NetworkObject which contains request_type=SessionWrite and argument being a SessionState object
    - Stores the given SessionState object into its local SessionState table with a status encapsulated inside the NetworkObject status if the operation succeeded

3. ExchangeViews
    - On receiving an ExchangeView request from a client, the server unpacks the NetworkObject which contains request_type=ExchangeView and argument being a Membership View table
    - Updates the local Membership View Table by merging the other member's view as defined by the gossip protocol.
    - Returns the updated view back to the client which uses it to update its own membership view.

RPCClient

The RPC Client is a stub on the Application Server which sends requests to RPC Servers running on other Application Servers. The client and the server exchange information using a NetworkObject which encapsulates the request_type and the argument for the operation. Also, the RPC Client uses a CALL_ID to match a response with its corresponding request. The arguments and returns for each request type are indicated below:

Request Type	Argument	Return
SessionRead	SessionID	SessionState
SessionWrite	SessionState	Boolean Result
ExchangeViews	MembershipView	MembershipView

EnterServletContextListener

A ServletContextListener is invoked just after Tomcat starts up to initialize any deployed applications and just before Tomcat goes down to enable deployed applications to perform cleanup. We override 2 methods to handle both these cases:
contextInitialized(): initialize the SessionStore, MembershipViewStore and the RPCServer Thread
contextDestroyed(): cleanup SessionStore and MembershipStore and terminate any running background executor threads before the server goes down

EnterServlet

- Singleton Application Controller. A single instance of this object receives HTTP requests on multiple threads over the scope of the application. Therefore, it is designed to not hold any state within it.
- On receiving an HTTP request, performs the required processing and generates an HTTP Response containing a session cookie with an appropriate value stamped on it. This session cookie will be presented by the client on subsequent requests. The sequence of actions performed by this servlet on receiving an HTTP request is stated below:
    1. If there is no cookie included in this request, then creates a new session state object.
    2. If there is a cookie in the request, then it uses the cookie value to extract the primary and backup servers for this session. Attempts to read the session state from the primary/backup servers until a successful read returns the session state. Increments the version on the session state, updates the message if required, updates the expiry time.
    3. Writes the new session state object to the local session table and to K other randomly selected servers in its view using RPCClient. Each RPC is performed in a serial and synchronized fashion.
    4. This information related to the new primary and backups is stamped on both the session state object and the cookie. In case of a logout request, the corresponding session state is removed from the local session state table (if present) and the session cookie is also removed from the response
    5. Stores relevant attributes into request scope and dispatches the request to the JSP where this data is neatly rendered
- After each RPC returns successfully/times out, the status of the corresponding Member is updated in the local Membership View Table and eventually propagated to other members via gossip
- Handles exceptional cases (ex: primary and all backups down) and returns an appropriate message to the user

Application Constants

All application constants are stored in a file named Constants.java and can be modified to suit testing needs. For instance, cookie maxAge is set to 1hr by default but can be changed to test garbage collection of old sessions. If any of the constants are changed, then a new WAR file has to be generated from Eclipse and re-deployed onto EBS.

Utility Classes

IPUtil is used to obtain the IP Address of a server on EC2 using the approach given in the problem description. Also, the IP Address of a machine is computed only once and stored as an Application Constant that can used throughout the scope of the application.
CookieUtil is class which contains utility methods for cookie management – extracting data out from cookies and updating attributes on cookies like maxAge/value etc.

Enter.jsp

- Simple HTML form consisting of a text input field and 3 buttons (Replace, Refresh & Logout) as specified
- Any button click results in an HTTP request being sent to the EnterServlet along with the corresponding request parameters in the request URL
- Pulls data required for rendering from request scope using Expression Language (EL) syntax. Ex:- ${message}
- For testing and debugging purposes, we also display the following data:
    i. Session Cookie Value
    ii. The server which handled the current request
    iii. The server on which session data was found
    iv. The new Primary/Backups assigned to store the client's session data
    v. SessionExpiryTime and SessionDiscardTime
    vi. Neatly formatted table representing the Membership View of the server which handled the current request

Cookie format

When the EnterServlet receives request from a client for the first time, it generates a cookie containing the session information as a key-value pair. The format of the key-value pair is as follows:
CS5300PROJ1SESSION = <sessionId_versionNum_primary_backup1_backup2_....._backupK>
Ex: CS5300PROJ1SESSION = a1eaa031-5558-43fe-b5b5-966ddad579d9_5_192.168.0.105_192.168.1.110_192.56.44.1
Note that the cookie value contains only "safe" characters such as letters, digits and ".-_" and is restricted to less than 512 bytes so it can fit into a single UDP packet.

Supporting K-resiliency

The application has been built to support K-resiliency. Every session state is stored on 1 primary and K backup servers. The primary and list of backups are also stored down in the cookie sent back to the client each time.
Every change to the session state results in SessionWrite RPC call to K backup servers.
Every time a session is accessed, its session state can be retrieved from its primary or any 1 of the K backups incase the primary and any backups are down. Thus the system can tolerate upto K server failures.