CS5300 S14 **TR 11:40-12:55** **Olin 255**

Project 1b: Scalability and Fault-Tolerance

1. Introduction

This is the second half of CS5300 Project 1. It will use AWS Elastic Beanstalk (documented here) together with UDP networking, to build a distributed, scalable and fault-tolerant version of the website you built as Project 1a. AWS Elastic Beanstalk will be used to create and maintain a load-balanced set of Application Servers running Apache Tomcat Version 7. The servers will run Java Servlet (and/or JSP) code implementing your site, together with additional Java code implementing a distributed, fault-tolerant, in-memory session state database similar to SSM.

Recall that this project is for **groups of 3 or 4 persons**. For 4-person groups the "extra credit" feature described in **Section 5** is mandatory.

2. Preliminaries

For Project 1a you downloaded and installed Apache Tomcat Version 7 on your own machine, and used it to develop Servlets and/or JSPs. For Project 1b your Web application will run in the AWS Cloud. It will be managed by AWS Elastic Beanstalk and your code will keep track of appproximate group membership in an AWS SimpleDB database. The "getting started guides" at Amazon will tell you how to sign up for the AWS services you need.

For this project you should download and install the AWS Toolkit for Eclipse if you haven't already done so. This includes the AWS SDK for Java, documented here, which includes client stubs that Java code can use to access many AWS services (and includes many useful code examples as well).

You can also use the AWS Management Console to deploy and control your website under Elastic Beanstalk.

3. Architecture

Your Project 1a solution had one or more Servlets or JSPs to handle requests; it also had an in-memory session data table (the "SessTbl", probably a HashTable or HashMap) with some mechanism for garbage-collecting expired entries. You are now going to expand Project 1a to a distributed system. The system will have a load balancer (provided by Amazon Elastic Beanstalk), and two or more identical server nodes (with Apache Tomcat 7 running Java code that you provide). These components will cooperate to implement a 1-resilient distributed website. You will write the server code, consisting of:

- Servlets/JSPs for processing client requests -- essentially the code from Project 1a.
- A distributed session state database analogous to SSM: each server node will have a local in-memory session data table ("SessTbl" for short) exposed through a Remote Procedure Call server interface to be discussed in Section 3.9 below, and each server will have a client RPC "stub" to access the SessTbls of other server nodes.
- A gossip-based approximate group membership service, so every server knows the address of sufficiently many other servers to maintain 1-resilience.

3.1 Server Identifiers, Session Identifiers and Session Cookies

Server IDs. Each server will be identified by a value we will call a SvrID. Suppose the HTTP TCP port and the RPC UDP port (to be discussed below) are the same for every server and are globally known. Then a SvrID can just be the server's IP address

a SvrID can just be the server's IP address.

In the discussion below we shall often abuse notation by using a term like "$\text{SvrID}_{\text{foo}}$" to denote either a SvrID or the server itself; for example,

$\text{SvrID}_{\text{local}}$ sends a message to $\text{SviID}_{\text{primary}}$.

The intended meaning should be clear from context.

There is a distinguished "null" server ID that identifies no server. This is denoted $\text{SvrID}_{\text{NULL}}$; internally it can be represented by an IP address of all zeroes. Your protocols need to check for $\text{SvrID}_{\text{NULL}}$ and treat it specially in some cases; for example, it makes no sense to send a message to $\text{SvrID}_{\text{NULL}}$.

Getting Your Own IP Address / SvrID. This is surprisingly difficult. During development, when you are testing your code in a Tomcat server running on your own machine rather than on an EC2 instance, you will want the real IP address of your machine, probably *not* the "localhost" address 127.0.0.1. You can Google how to get your machine's IP address from Java; as is often the case, a reasonable discussion can be found at StackOverflow. Unfortunately, because of the unique Amazon networking environment, the same technique *will not work* to enable a server running on an EC2 instance to determine its own IP address.

Here is one (admittedly baroque) approach to getting the IP address or other instance metadata of an EC2 instance from code running on that instance. The EC2 Instance Metadata Query Tool is automatically installed as

/opt/aws/bin/ec2-metadata

on the EC2 instances created for you by Elastic Beanstalk. So for example running the shell command

/opt/aws/bin/ec2-metadata --public-ipv4

on such an instance would print

public-ipv4: ww.xx.yy.zz

on the standard output. Java code running in your Tomcat server can actually execute this command and examine the output, by using `Runtime.exec()` to execute the command and capturing the command output in a Java String. A fairly complete discussion of this, which has been tested recently, can be found here. The `ec2-metadata` script actually issues an HTTP request to the EC2 infrastructure, so it's quite expensive. Thus, you should call it only once and cache the result.

Session IDs. In your system, each server will need the ability to generate a globally unique session ID. As we discussed briefly in lecture, this can be done by including a SvrID as part of the session ID; that is, a session ID can be a pair

$$\text{SessID} = \langle \textit{sess\_num}, \text{SvrID} \rangle$$

where SvrID identifies the server that originally created the session ID, and *sess_num* is the value of a global

variable in that server that is incremented each time a new session ID is created there. With multiple servers, the *sess_num* values will not be globally unique; but the SessIDs, which include the server's unique SvrID, will be globally unique as required.

A Session Version Number is just a number that is incremented for every request within a session; it uniquely identifies a version of a given session's data.

Session Cookies. As in Project 1a, a `CS5300PROJ1SESSION` cookie will be sent with each client request in an existing session. It must contain the SessID and Session Version Number. In addition, because you are

implementing a distributed session state data base, you will need to put some "location metadata" into your session cookies. For a 1-resilient system, the metadata will consist of a pair of SvrIDs identifying the "primary" and "backup" locations that hold the session state data:

$$< \text{SvrID}_{\text{primary}}, \text{SvrID}_{\text{backup}} >$$

Thus, a request in an existing session will include a `CS5300PROJ1SESSION` cookie whose value contains the SessID, version and location metadata:

$$< \text{SessID}, \textit{version}, < \text{SvrID}_{\text{primary}}, \text{SvrID}_{\text{backup}} > >$$

This tuple-of-tuples must be encoded as a string for transmission as a cookie value. As in Project 1a, you can "canonicalize" the tuples or just restrict them to "safe" characters (letters, digits, ".-_" are all safe). For example, if you use underscore characters to separate fields the string value of a cookie might look like

2_192.168.1.2_8_192.168.1.2_192.168.1.3

This would represent an instance of the second session created at server 192.168.1.2, session version number 8, currently stored at servers 192.168.1.2 and 192.168.1.3.

The session's expiration time is determined by the `MaxAge` parameter of the session cookie, so it doesn't need to be stored explicitly in the cookie value.

Because the session expiration time and version change with every client request, every response needs to include a new cookie with the `MaxAge` and session version number set appropriately.

Just as in Project 1a, a client request arriving at your home page without a `CS5300PROJ1SESSION` cookie will cause a new session to be created. This is discussed in Section 3.5 below.

3.2 Remote Procedure Call (RPC) Communication

Your distributed session state database will consist of multiple server instances that communicate with one another using a simple Remote Procedure Call (RPC) mechanism that you will implement. For simplicity, you should do your network communication using the User Datagram Protocol (UDP), which has some nice features for a project like this. The appropriate Java classes for UDP are DatagramSocket and DatagramPacket. Your RPC servers should all listen at the same well-known port number; port 5300 would be a good choice.

UDP is an unreliable packet-oriented protocol (in contrast to TCP, which provides *reliable byte streams*). The basic primitivies of UDP are

- Send a packet to a specified IP adddress and port;
- Receive a packet (*any* packet) sent to my own IP address and a specified port.

This project has a 512-byte maximum length for session data (inherited from the specification of Project 1a). This maximum length ensures that any procedure call or reply message you need to send will fit in a single UDP packet. You should truncate session data values received from a client if they exceed this maximum length. Your RPC messages may be lost or reordered, but they will not be truncated, and you may assume they will not be corrupted either. UDP has a simple receive timeout mechanism -- see the `setSoTimeout(int msec)` method of DatagramSocket for a description of this mechanism. You can use receive timeouts to decide that another machine is not responding. **You are not expected to implement retries of lost RPC messages** -- in fact, this is discouraged.

An RPC protocol is basically just an agreed-upon format for call and reply messages. For example, a call message could consist of

- a unique *callID* for the call.
- an *operation code*.
- zero or more *arguments*, whose format is determined by the operation code.

A reply message could consist of

- the *callID* of the call to which this is a reply.
- zero or more *results*, whose format is determined by the operation code in the call.

You *could* encode these messages using Java's `Serializable` mechanism, but this is probably overkill for a simple system that exchanges only tuples of scalar values (i.e., no cyclic structures); so use of Serialization is definitely *not* required. You can easily encode the messages as fixed or variable length strings. Unlike cookie values, UDP packets do not need to be "canonicalized."

The purpose of the callID is to enable you to ignore received packets that are not valid replies to the current call (e.g. packets that are delayed replies to previous calls). So if you send the same call to several servers concurrently, (as in the original SSM protocol discussed in class), the requests could all use the same callID; but the callIDs of logically different calls must be different. Note this does not require that the callID be a globally unique value. It is sufficient for each server maintain a counter that is incremented every time the server performs a remote call.

Your protocol may exploit RPC parallelism, sending a number of requests to different servers, then waiting for the first (or the first few) responses and discarding any remaining responses. Doing this efficiently requires the ability for a thread to wait for multiple incoming messages at once. You can't just wait for the individual responses in the order in which the requests were sent, since there is no reason to expect them to arrive in that order. Indeed, some responses might not arrive at all, due to a dropped UDP packet or a failed server. Waiting for multiple responses is a difficult trick with TCP -- it requires multiple TCP connections and lots of threads and synchronization, or use of the complicated Java NIO mechanism. But with UDP it is quite easy, since a single UDP DatagramSocket is capable of receiving data from multiple senders.

3.2.1 RPC Clients

Here is pseudocode of the RPC client stub for a `SessionRead(...)` call (described in Sections 3.3 and 3.11 below). A `SessionRead(...)` call may be sent to one or more destinations, but only the first response is significant.

```
//
// SessionReadClient(sessionID, sessionVersionNum)

//   sending to multiple [destAddr, destPort] pairs
//   using a single pre-existing DatagramSocket object rpcSocket
//
callID = generate unique id for call
byte[] outBuf = new byte[...];
fill outBuf with [ callID, operationSESSIONREAD, sessionID, sessionVersionNum ]
for( each destAddr, destPort ) {
  DatagramPacket sendPkt = new DatagramPacket(outBuf, length, destAddr, destPort)
  rpcSocket.send(sendPkt);
}
byte [] inBuf = new byte[maxPacketSize];
DatagramPacket recvPkt = new DatagramPacket(inBuf, inBuf.length);
try {
  do {
    recvPkt.setLength(inBuf.length);
    rpcSocket.receive(recvPkt);
  } while( the callID in inBuf is not the expected one );
} catch(InterruptedIOException iioe) {
  // timeout
  recvPkt = null;
} catch(IOException ioe) {
  // other error
```

```
  ...
}
return recvPkt;
```

Note we have left out some exception handling (`try ... catch` blocks). Also, a "real" implementation might want to retry the `receive` operation after an IOException other than timeout.

Now consider thread synchronization. Suppose two or more threads try to execute the above RPC code concurrently *using the same rpcSocket*. Technically, DatagramSocket is thread-safe; but when multiple threads read from the same DatagramSocket, each incoming packet is delivered to exactly one of the reading threads, chosen by the OS at random (or at least chosen in a way we can't possibly predict). Thus, each calling thread would be likely to receive (and discard) some of the other thread's reply packets. This would be a Bad Thing. Fortunately, there is a simple solution: create a new DatagramSocket object for each call, and close it at the end of the call. The above code becomes

```
//
// SessionRead(sessionID, sessionVersionNum)
//   sending to multiple [destAddr, destPort] pairs
//   creating new DatagramSocket object rpcSocket
//   and closing it when done
//
DatagramSocket rpcSocket = new DatagramSocket();
callID = generate unique id for call
... (same as before)
rpcSocket.close();
return recvPkt;
```

This ensures that all concurrent outgoing RPCs use different `DatagramSocket` objects. The zero-argument constructor for `DatagramSocket` gives the new socket an arbitrary, currently unused port number. This is fine -- nothing in the RPC protocol depends on the client port number; the server simply sends the response to whatever port the client used when it sent the request. The client's port (and IP address) can be obtained from the received DatagramPacket. When the client has received "enough" responses, it closes the socket. UDP packets sent to a socket that has been closed are automatically discarded by the OS.

3.2.2 RPC Servers

A useful UDP-based RPC server can be just a single thread in a loop. Each time around the loop it receives from a single DatagramSocket, computes a response, and sends a reply using the same DatagramSocket. The pseudocode looks something like this:

```
DatagramSocket rpcSocket = new DatagramSocket(portPROJ1BRPC);
serverPort = rpcSocket.getLocalPort();
...
while(true) {
  byte[] inBuf = new byte[...]
  DatagramPacket recvPkt = new DatagramPacket(inBuf, inBuf.length);
  rpcSocket.receive(recvPkt);
  InetAddress returnAddr = recvPkt.getAddress();
  int returnPort = recvPkt.getPort();
  // here inBuf contains the callID and operationCode
  int operationCode = ... // get requested operationCode
  byte[] outBuf = NULL;
  switch( operationCode ) {
      ...
      case operationSESSIONREAD:
              // SessionRead accepts call args and returns call results
              outBuf = SessionRead(recvPkt.getdata(), recvPkt.getLength());
              break;
      ...
  }
  // here outBuf should contain the callID and results of the call
  DatagramPacket sendPkt = new DatagramPacket(outBuf, outBuf.length,
      returnAddr, returnPort);
```

```
        rpcSocket.send(sendPkt);
    }
```

Again we have left out some distracting `try ... catch` blocks, but the intended behavior should be clear.

Here `rpcSocket` is created using the one-argument constructor that accepts an explicit port number; the port is the well known port used for your RPC servers; 5300 might be a good choice.

3.3 Retrieving Session State

When a user clicks one of your website's buttons, a request is sent through the AWS Elastic Beanstalk load balancer, and is routed to one of your servers. Unless it is the first request in a new session, or its session has timed out, the request will include a `CS5300PROJ1SESSION` cookie as discussed above:

< SessID, *version*, < $\text{SvrID}_{\text{primary}}$, $\text{SvrID}_{\text{backup}}$ > >

If either $\text{SvrID}_{\text{primary}}$ or $\text{SvrID}_{\text{backup}}$ is the receiving server's own SvrID (call that "$\text{SvrID}_{\text{local}}$"), then the requested session state is in the server's local SessTbl and should be retrieved directly from there with no network traffic. Otherwise, there are two options:

Option 1. Send a

SessionRead(SessID, version)

RPC request to $\text{SvrID}_{\text{primary}}$ and wait for a successful response. If the response times out or fails (i.e., returns "not found"), send another

SessionRead(SessID, version)

request to $\text{SvrID}_{\text{backup}}$ and wait for a successful response. This approach minimizes network traffic at the expense of possibly higher latency, especially in the (infrequent) case that $\text{SvrID}_{\text{primary}}$ has failed.

Option 2. Send two

SessionRead(SessID, version)

RPC requests concurrently to $\text{SvrID}_{\text{primary}}$ and $\text{SvrID}_{\text{backup}}$, wait for the first successful response, and discard the other response (if any). This approach minimizes latency but generates some unnecessary network traffic.

Using either approach, a successful response will contain the requested session data. However, you may be unable to elicit a successful reponse, possibly because the requested session has timed out, or because both the primary and backup servers have failed. In that case you should return an HTML page with a message saying the session timed out or failed (you will be able to tell the difference between these in some but possibly not all cases), and make sure the cookie for the timed-out-or-lost session is deleted from the browser.

3.4 Storing Session State

The Normal Case. To process a client request, a server performs some request-specific computation, which may involve updates to the data layer as well as computing a new session state data value. We assume the updates to the data layer are done in (a sequence of) atomic transactions, each of which leaves the system in a consistent state. Logically this should be followed by an atomic update of the session state to its next version, and finally returning a successful result (HTML page and updated session cookie) to the client.

Assume a client request arrives at server $SvrID_{local}$ with the usual session cookie

$$< SessID, version, < SvrID_{primary}, SvrID_{backup} > >$$

and we reach the point of storing the updated session state and returning.

The new session state version is just

$$new\_version = version + 1.$$

To maintain 1-resilience, the new session state must be stored in the SessTbl of at least two servers.

To minimize network traffic, the primary site for the new session state can be $SvrID_{local}$ itself. So $SvrID_{local}$ stores a copy of the new session state into its own SessTbl.

The new backup site can be any other site that is "known" by $SvrID_{local}$ (that is, in the local server's "view" -- see Sections 3.8 below). So $SvrID_{local}$ chooses a new backup site $SvrID_{new\_backup}$ and sends a

SessionWrite(SessID, new_version, new_data, discard_time)

RPC to $SvrID_{new\_backup}$. How to compute the *discard_time* will be discussed in Section 3.6. After sending this request, $SvrID_{local}$ *must wait* for a successful response. If the response times out, $SvrID_{local}$ chooses a different new backup server and retries the call. If it eventually manages to elicit a successful response, it constructs the new session cookie

$$< SessID, new\_version, < SvrID_{local}, SvrID_{new\_backup} > >$$

and returns the result (HTML page with the new session cookie) to the client. If no new_backup can be found, $SvrID_{NULL}$ can be used in place of the backup location, so the new session cookie is

$$< SessID, new\_version, < SvrID_{local}, SvrID_{NULL} > >$$

That is, the session is *non-replicated*. In this case the system is not 1-resilient, at least for the given session, but if there are no further failures the system will continue to operate correctly.

Because the above process is supposed to persist until it finds a backup site that is alive, it is not absolutely necessary to choose *new_backup* at random each time. To avoid having "obsolete" copies of (old versions of) a session cluttering up your SessTbls, you might first try whichever of $SvrID_{primary}$ and $SvrID_{backup}$ is not equal to $SvrID_{local}$ as your new backup. That way, the new version of the session data will replace the old version in the SessTbl. See Section 3.7 on Garbage Collection.

3.5 Creating New Sessions

A client request arriving at your home page without a `CS5300PROJ1SESSION` cookie must cause a new session to be created. The executing server $SvrID_{local}$ constructs a new SessID value by incrementing its global *sess_num* variable and constructing

$$SessID = < sess\_num, SvrID_{local} >$$

It then uses the procedure described in Section 3.4 above to store the new session data at two locations.

Newly Booted Servers. A newly-booted server does not yet "know about" any servers other than itself. The server should immediately try to initialize its "membership view" from the global "Bootstrap View" as described in Section 3.8 below. If the server receives a new-session client request (a client request without a

session cookie) before it has initialized its view, the server will be unable to store two copies of the new session data, so 1-resilience will not be guaranteed. You might be tempted to return an error in this case, or wait until the view becomes nonempty. However, such a policy would prevent you from running and debugging a system configuration that had only a single server instance. Since that is a *very* convenient thing to do, you should just create a non-replicated session cookie in this case.

3.6 Implementing Session Timeouts

Inactive sessions time out, and the data for a timed-out session must eventually be "garbage collected" from all the servers.

Assume there is a global constant `SESSION_TIMEOUT_SECS`, which you can think of as the *minimum* amount of time a session is required to remain accessible after the last client request. The system should make the following guarantee: if a client issues a request in session *s*, then waits for an interval strictly less than `SESSION_TIMEOUT_SECS`, then issues another request in session *s*, the session data for *s* will still be available. In other words, sessions are guaranteed not to time out prematurely. The system does not need to enforce the converse guarantee that sessions time out as early as possible. That is, if a client issues a request in session *s*, then waits for an interval greater than `SESSION_TIMEOUT_SECS` and then issues another request in session *s*, the session data *may or may not* still be available.

Assuming the server clocks are approximately synchronized, this "one-sided" guarantee is easy to achieve. There are two interesting values. One is the MaxAge of the session cookie that will be returned to the client. The other is the timestamp on the session's entry in the SessTbls (which is effectively the session *discard_time* passed in the SessionWrite call described in Section 3.4 above).

The MaxAge parameter on the session cookie can just be `SESSION_TIMEOUT_SECS` to ensure that the client does not discard the cookie prematurely.

> Sadly, Versions 8 and earlier of Internet Explorer support the (obsolete) "expiration time" cookie parameter but not the MaxAge parameter. Ii is okay if you ignore this problem and just use a modern browser.

The discard_time passed in the SessionWrite call (which is the expiration timestamp stored in each SessTbl) should be at least

$$\text{discard\_time} = \texttt{now} + \texttt{SESSION\_TIMEOUT\_SECS} + \Delta$$

where `now` denotes the system clock value at the server making the SessionWrite call, and Δ is a constant to account for the maximum allowable difference between any pair of server clocks plus the maximum allowable clock drift over a session timeout interval plus the communication and processing time associated with the SessionWrite calls. All replicas of a given version of a given session's state should be given the same discard_time.

A server *may not* garbage collect a session from a SessTbl before its discard_time has arrived, and it *should* garbage collect the session as soon after that time as is convenient.

Choosing a large values for Δ wastes some server memory but is otherwise benign. Choosing an aggressively small value helps you to test your session timeout implementation.

3.7 Garbage Collection

Just as in Project 1a, you can do garbage collection with a thread that periodically wakes up and scans the local SessTbl looking for session data objects whose discard_time is past. As we mentioned in Section 3.4, you can reduce the expected load on the garbage collector by preferentially storing each new version of a session's data at one of the servers that held the previous version -- the new version overwrites the previous version and thus the previous version never needs to be garbage collected. To make this possible, the SessTbl should be keyed by SessID only, not <SessID, version>.

3.8 Implementing Group Membership

Basic View Rules. In a couple of places in the protocol, a server chooses another server at random. To make this possible, each server maintains a membership "View," a set containing the SvrIDs of some servers it "believes" are up. To help maintain the View, each server follows some simple rules.

- A SvrID is inserted into the View whenever an RPC request or reply message is received directly from that SvrID. This can be implemented at a very low level, since the Java DatagramPacket class has methods that return the sender's IP and port.
- A SvrID is removed from the View whenever an RPC sent to that SvrID times out.

As we discussed in lecture, this basic view maintenance protocol is too conservative: it adds a server to the

View only when there is direct evidence that it is running, and deletes a server from the View only when there is direct evidence that it has failed. In particular, using only this protocol, a network partition will persist forever.

So, *in addition to* the above rules, we will use a gossip protocol to maintain server Views, as well as to maintain a "bootstrap view" at a well-known location in the Amazon SimpleDB.

Basic Gossip Protocol. Like the protocol in the Lightweight Probabilistic Broadcast paper discussed in lecture, rather than trying to keep track of the entire server set, we pick a nominal "view size," ViewSz, and each server attempts to maintain a View containing ViewSz SvrIDs chosen at random from all the active SvrIDs. Your choice of ViewSz must be big enough to guarantee your desired degree of resilience, and in addition, as discussed in the paper, should be $\Omega(\log N)$ to maintain connectivity of the entire group.

> **Suggestion:** ViewSz = 5

A View is essentially just a set of SvrIDs; but to be more precise, here is a list of operations your code might need to perform on views:

- shrink(View v, int k)
  while v contains more than k entries, delete an entry chosen uniformly at random.
- insert(View v, SvrID s)
  insert s into v if not already present
- remove(View v, SvrID s)
  remove s into v if it is present
- SvrID s = choose(View v)
  return s chosen uniformly at random from v
- union(View v, View w)
  set v to the union of v and w (eliminating duplicates)

With these primitives, the basic gossip protocol can be described fairly simply. Each server *s* periodically chooses a server *t* at random from its View and sends an RPC call

```
GetView()
```

to *t*. A successful reply will return t.View, the current View of *t*, which can be stored into a local variable temp. Now *s* can merge this into its own View by executing

```
union(temp, s.View)
remove(temp, s)
shrink(temp, ViewSz)
s.View = temp
```

That is, *s* replaces its own View with a new View of the desired size chosen at random from the set of all SvrIDs known to either *s* or *t*.

Bootstrap View. What we have described so far is a simple "pull" style gossip protocol. As we discussed in lecture, it needs some enhancement to bootstrap newly created servers and to handle network partitions.

When a server first boots, it can't participate in the basic gossip protocol since the only SvrID it knows about is itself. One solution, mentioned in lecture, is to have a crash-recover or even crash-only node at a well-known address taking part in the gossip protocol. Call this node the "Bootstrap View server." Assume it is rebooted, possibly with an empty or incorrrect View, after any crash, and thus is eventually accessible to any correct server. You can implement this behavior using a well-known SimpleDB table name (or "Domain" in the terminology of SimpleDB), and storing a list of up to ViewSz SvrIDs there. As with cookies, you can encode the list as a String of IP addresses separated by '_' characters. You can initialize the Bootstrap View manually, for example by using the SimpleDB Scratchpad to store an empty string or possibly $SvrID_{NULL}$, "0.0.0.0". (The SimpleDB Scratchpad seems to work in Firefox but not in Chrome, Safari or IE.) Your servers can then read and update the Bootstrap View using the operations described here.

Recall, in the basic gossip protocol, each server occasionally reads the View of another server (chosen at random from its own View), and then replaces its own View with a random combination of the two Views. Now that we have added the Bootstrap View in SimpleDB, each server must occasionally read and update it by a procedure like the following:

```
View temp = ... (read Bootstrap View from SimpleDB) ...
remove(temp, self)
union(temp. self.View)
shrink(temp, ViewSz)
self.View = copy(temp)
insert(temp, self)
shrink(temp, ViewSz)
... (write temp to Bootstrap View in SimpleDB) ...
```

Using this scheme, even if the Bootstrap View is empty or completely incorrect, it will not remain so for long, but will (in the absence of further failures) eventually converge to a subset of the correct SvrIDs. Also, you might think you need some synchronization around concurrent SimpleDB accesses made by multiple servers, e.g. use of the "compare-and-swap" functionality provided by SimpleDB. But a little though should convince you it is sufficient to guarantee that concurrent client reads and writes should not get corrupted, and this is guaranteed by the "eventual consistency" SLA of SimpleDB.

A weakness of the scheme is that scalability is limited because the SimpleDB Bootstrap View caan become a "hot spot" -- if the number of servers is very large, the traffic at SimpleDB may grow unacceptably. There are techniques to adaptively change the interval between a server's updates to SimpleDB, but I'm not asking you to worry about this problem. Just choose a sufficiently large interval between Bootstrap View updates so that even with a relatively large number of servers (10?) the expected interval between SimpleDB accesses is at least a few seconds.

Avoiding Convoys. Consider two servers *s* and *t* choosing to gossip with server *u* nearly simultaneously. The single-threaded RPC server implementation described in Section 3.2.2 above processes calls one at a time; thus, the first call arriving at *u* starts being processed immediately, while the other call waits until the first call is done. Effectively, *s* and *t* are now running in lock step, with (say) *t* running one RPC time behind *s*. After the system has been running for a while, it tends to evolve toward a state in which all the servers participating in the gossip protocol form a *convoy* -- there are bursts of synchronized activity, followed by long idle periods.

It is easy to avoid this behavior. Instead of a loop of the form:

```
while(true) {
```

```
    ... gossip with another site chosen at random ...
    sleep( GOSSIP_SECS )
  }
```

that uses a fixed sleep time, you should instead write a loop that generates a random sleep time with the desired mean:

```
  Random generator = new Random()
    ...
  while(true) {
    ... gossip with another site chosen at random ...
    sleep( (GOSSIP_SECS/2) + generator.nextInt( GOSSIP_SECS ) )
  }
```

Now the sleep time is chosen uniformly at random between GOSSIP_SECS/2 and 3*GOSSIP_SECS/2.

3.9 Summary of RPCs

Here is a summary of the RPCs that have been described above.

- SessionRead(SessID, version)

  SessID is a session ID, version is the version number of the requested session. The reply should be a pair

  < found_version, data >

  comprising the stored session data and the version found on the server. If no version of the requested session is found, there should be an explicit "not found" response (rather than just failing to respond). For example, you could return an impossible version number like (-1).

- SessionWrite(SessID, version, data, discard_time)

  SessID is a session ID, version is the version number of the session to be stored, data is the session data to be stored, and discard_time is the time after which the stored session may be garbage collected. The reply is just an acknowledgement. The effect is to store the new session data into the SessTbl at the server. If the server already holds an older instance of that session, it may (should?) be garbage collected immediately.

- GetView( )

  There are no arguments. The reply is the View of the called server, containing at most ViewSz (and possibly fewer) members.

All these procedures should work properly with the basic View rules of Section 3.8. That is, whenever any RPC message (call or reply) is received, its sender should be added to the View if not already present; and when an RPC call times out, the destination should be removed from the View.

4. Your Site's User Interface

To help with grading (and debugging) of your site, you should extend the user interface to report data about server execution. Add the following displayed information:

- The SverID of the server executing the client request.
- For a request in an existing session, report where the session data was found ($SvrID_{primary}$ or $SvrID_{backup}$).
- The $SvrID_{primary}$, $SvrID_{backup}$, session expiration time and discard_time for a new or updated

session.
- The server's entire View.

5. k-Resiliency (optional)

This part is mandatory for groups of 4 persons.

The protocol as described above yields a scalable 1-resilient distributed service. For up to 15 points of extra credit, make your service *k*-resilient, where the value of *k* is a compile-time parameter that may be greater than 1. Note the system cannot possibly be *k*-resilient unless there are at least (*k*+1) live servers. Thus, your system will be *k*-resilient only for sessions created after the $k+1^{st}$ server is booted. This is true even when *k* is 1.

6. Suggestions on how to proceed

(a) Instructions here describe how to create an Eclipse AWS Java Web Project that can be either tested locally or deployed to AWS using ElasticBeanstalk. Make sure you are comfortable with this procedure.

(b) UDP based RPC client and server implementations can be tested in any Java Virtual Machine -- it doesn't need to be an Apache Tomcat instance (though of course it can be). Your RPC client and server code can be fully tested running as separate threads in the same JVM instance, initially using localhost (127.0.0.1) for the IP address. This will let you get comfortable with essential details like packing and unpacking arguments/results in the `DatagramPacket` objects, and handling timeouts properly.

(c) A single instance of your application server should be runnable in a Tomcat instance on your local machine. You can even access SimpleDB and other AWS services from your local machine using the same AWS SDK calls you would use in code running on an EC2 instance. Thus, the group membership protocol from Section 3.8 will work, though of course the View will not grow bigger than a single server.

You can also send HTTP requests directly to such a server instance. Of course, a single server will not be 1-resilient, but it should run correctly.

Using two or more of your group members' machines, you can run multiple copies of your server and they should be able to locate each other using the Bootstrap View in SimpleDB. This will require using "real" IP addresses, not localhost.

(d) At this point you should be ready to deploy a 1-resilient multi-server website running on the Amazon Cloud using Elastic Beanstalk. The major remaining issue is to make sure the EC2 instances in your Elastic Beanstalk auto-scaling group are able to communicate with one another. You will probably need to add some rules to your security group to allow incoming UDP packets sent to any port. This can be done using the AWS Management Console.

7. Submit Your Work

Create a file, in `zip` archive format, named `solution.zip` This archive should contain

- a `README` file.
  This may be in `.pdf` or `.txt` or `.doc` format. This file should include anything we need to know to grade your assignment. In particular, it should briefly describe the overall structure of your solution, including formats of your cookies and RPC messages; and specify what functionality is implemented

  in each source file. It should also include your Elastic Beanstalk setup procedure.

  If you implemented the extra credit option, describe the changes you needed to make..

- your Java and JSP source code.
  Include understandable high-level comments.
  Please include the source code directory structure here! In the past, some students have submitted their `*.java` and `*.jsp` files in a single flat folder. Please don't do this, as it makes it difficult for us to import the files into Eclipse. It is sufficient for you just to include the sources in your `.war` file (see below).
- a deployable `.war` file for your system.

If you wish, you may include additional files in the archive as well, for example, screenshots. Submit your `solution.zip` file using CMS by the specified deadline.