NETCOMmunications:

SSAD: System and Software Architecture Description

Rev Number	Rev Date	Rev Description
0.1	March 24, 2002	Preliminary Design Decisions
0.5	April 1, 2002	Component Interface Definitions
0.9	April 5, 2002	Integrated Image and Text Components
1.0	April 7, 2002	Final Editing for Design Phase Submission
1.1	April 24, 2002	Revision for Implementation Phase
1.2	April 30, 2002	Implementation-Phase Submission

Team Members:

Nelson Brand (nab27)
Eric Poirier (edp29)
Tecuan Flores (tf180)
Mikhail Litvin (myl16)
Omar Siddiqui (oss8)
Charles O'Donnell (cwo4)

TABLE OF CONTENTS

1 Introduction

The NETCOM MDSCloud implementation is made of three stand-alone components: the MDS node, responsible for searching and meta-data storage, the MDSProxy, responsible for XML parsing, load balancing, and forwarding of incoming connections, and the Graphical Statistics Viewer, responsible for displaying usage statistics of the MDSCloud. Every PClient and Pserver will always connect to an MDSProxy, which they already know the IP and port of. Messages from the PClient or PServer will be forwarded through the MDSProxy to an MDS node, which handles the actual searching. The Graphical Statistics Viewer is also started as a separate program and communicates with all of the MDSProxies and MDS nodes to gather usage information. More information about the design and motivations will be given in the design choices section of this document, which will help the reader more fully understand the NETCOM MDSCloud.

2 Design Choices

2.1 Component Interaction

The three stand-alone components in the NETCOM MDSCloud inter-operate via messenger objects (*requests*) serialized over TCP/IP sockets. Object serialization using very compact request classes was chosen for its advantage to provide a high speed of transfer over a network, while also transforming primitive network functions such as byte transfer and parsing into a more Object Oriented model. This allows for Object Oriented design without the complexity or unneeded overhead of other methods, such as RMI or CORBA. Using this serialization model universally between components promotes well-defined interfaces and aids in separation of development tasks. In only one case, that of a *Bloom Update* discussed later, was it deemed necessary to deviate from object serialization and instead use UDP Packets to communicate between different MDS nodes. UDP packets are lightweight and can be easily routed which suites our need to broadcast many packets to many hosts without the overhead of socket creation and destruction. With the choice of Java as the implementing language, these UDP packets still fit rather nicely into our Object Oriented Model.

When handling request routing, it was decided to use two different methods based on whether the originating request was from a PClient or from a PServer. Basically, a PClient login creates a constant connection between PClient and MDSProxy, while any search requests across that socket will generate a new socket connection to the least loaded MDS node, send a request object, and close the socket connection. This allows for every search, even from the same PClient, to be load balanced to whichever MDS node will give optimal speed. To handle routing, the initial PClient to MDSProxy socket, as well as a MDSProxy identifier is added to the search request. All socket connections deeper within the MDSCloud then use the PClient to MDSProxy socket as a unique key for indexing. A logout of a PClient simply removes the connection from the MDSProxy and has no effect on any deeper MDS nodes.

It was decided that PServers should maintain a constant connection to a single MDS node, via the MDSProxy, so as to ease handling of meta data storage. If PServers connected to different MDS nodes with every *add-share* request, this would not only require much more

complicated *remove-share* and *PServer-Logout* routines, but also *might* effect the speed of certain searches when dealing with real-world data (for example, if we make a decent assumption that different PServers tend to hold *similar* types of data (eg. .mp3 versus .doc), then a search upon that type would be spread throughout the entire cloud instead of just a few PServers). To handle this, the original PClient to MDSProxy socket is again saved with the request to act as a unique key for indexing, and the connection would be handled as follows. Upon login a PServer (A) makes its first connection to a MDSProxy (B). A MDS node (C) is chosen according to load balancing criteria, and a new socket connection from the MDSProxy to a MDS node is created. This socket is held in a Hashtable in the MDSProxy which is indexed by the initial PServer socket information. With this a pipe from A to B is created which always has data forwarded to the same pipe between B and C, giving a *logical pipe* from A to C. Further, the MDS node will use the original PServer socket to index its own Hashtable of sockets, allowing returned requests to be sent to the appropriate MDSProxy. Upon logout, all these sockets are simply destroyed recursively and their Hashtable entries removed.

2.2 Network Topology

The NETCOM MDS nodes will form a decentralized network based on the idea of *friends* or neighbors which whom each node shares a socket connection. In best efforts to simulate and allow for a large-scale network of MDS nodes, each node will have a minimum and a maximum number of friends. When a MDS node joins the cloud it will contact LDAP and create connections with the maximum number of friends. As connections are broken as MDS nodes go on or offline, the number of friends will decrease until it reaches the critical minimum number. If any further friends are lost below the minimum number, the MDS node will contact LDAP and refill its friends until there are the maximum number of connections once again.

2.3 Searching

The NETCOM MDSCloud is designed to handle two different types of searching through the cloud, to be determined on a per node basis. The first is a *Breadth-First-Driven (BFD) Search* which focuses on high reliability and flexibility of search criteria, while the second approach, a *Bloom Search,* which focuses on high speed returns with marginally limited search criteria. In both cases we chose to *Return-Fast* all matches so that a PClient would have at least some returns as soon as possible. Further, all returned matches do not follow the initial path the search first traversed even if it might be slightly faster. Instead all returned matches are sent directly to the initiating MDSProxy, removing the overhead of reverse-path traversal.

The BFD Search involves sending a search request to a single MDS node from a MDSProxy. That node checks its local data repository and immediately returns any matches. It then passes the search request to all of its friends, who do the same, continuing. Such a method requires all search and return requests to have identifying numbers and origination information which is set unmodifiable at creation. Each Request maintains a history of all nodes visited, allowing MDS nodes to identify when a Request has already been seen and only allow any single request to be propagated once, thus eliminating the possibility of looping.

The Bloom Search uses *Bloom Filters* to give every individual MDS node a summary

cache of the contents of every other MDS node on the system. For some background, a Bloom Filter is a data structure which holds information through the use of hashing functions and reuse of datapoints by multiple elements. A Bloom Filter can have an item (usually a hashable string) *inserted* into it at one point, and then be checked later to determine if that specific item has or has not already been inserted. This can be useful as a kind of *compressed summary* where elements can be entered into a filter at one location, the relatively small filter can then be transported, and a second location can test to see if the first location has certain elements. More technically this is accomplished by the multiple hashing of strings to different points in a bitset. If one wants to check if a string has already been entered in into the filter, one needs only to hash a string using the same functions and see if *all* of the resulting hash points in the bitset have been set true. This allows for a zero possibility of false-negatives and a small possibility of false-positives, which are acceptible in our search scheme. Unfortunately this structure cannot have items *removed* from it, but its benefits outweight this small cost. More information can be found at http://www.cs.wisc.edu/~cao/papers/summary-cache/share.html

This Bloom Search method is accomplished by having every MDS node send tiny UDP update packets to every other node known on the system. This, as well as theoretical limitations, results in slightly unreliable summary caches. However, these summary caches allow for any search to be completed in at most, 2 hops. A single MDS node receives a search request, and return any of its local matches. The MDS node then checks all of the local summary caches, and propagates the search to all of the nodes which *should* contain a match. These nodes then return the full matching information which is forwarded back to the Proxy. This second search ensures that, while there might be incorrect matches with the local bloom filters (false-positives), an incorrect match will never be returned. In a large-scale network, this method will prove impressively faster than a BFD Search which would require *many* hops to return data, while the only cost of this speedup is MDSNode server memory.

Finally, both search methods can be applied on a per-node basis as added configurability, and careful attention is paid so that all searches will be handled according to each individual MDS node's settings. To do this we chose to always enforce UDP update packets to be generated so that every node had the *possibility* of Bloom Searching. However, each MDS node decides itself whether a request is propagated or not. For example: Node A is using a Bloom Search and finds a possible match according to node B's summary cache. Node A forwards the search request to Node B, and Node B returns any matches. However, if Node B is using a BFD Search, then it will take that search request, and propagate it to all of its friends.

2.4 Data Storage

The NETCOM MDS nodes must hold information for possibly thousands of shares for each of many PServer connections. Therefore efforts were made to reduce the memory used per share by methods such as use of primitives, and, for example, holding name-list pairs as an array of Lists instead of a two-dimensional array, which must be square, or a Vector of Vectors which have an overhead of two levels of Object usage instead of more "primitive" arrays. Further, searches through PServer shares data should be as fast as possible, so all shares could be held in multiple red-black trees, sharing objects, and using different *names* of the name-list pairs as indexing keys.

2.6 Concurrency

Both the MDS node and MDSProxy components use a multithreaded scheme to handle input and output requests on the socket connections. To begin, threads are used to monitor each of the types of socket connections, such as accepting new sockets, and reading current sockets. This scheme then breaks every incoming request into a *Task* which is then completed by various pools of threads. This allows for a polling scheme across all the socket connections because data is simply read from a socket and given to a separate thread to be processed, allowing the polling thread to quickly move on to the next poll. Each thread has access to *core* data structures (data structures which many objects and threads share), which have been created carefully to avoid any concurrent access exceptions. This scheme was chosen because it allows for constant processing without the dangerous problem of infinite thread spawning. This makes great leaps to alleviate any bottlenecks in the NETCOM MDSCloud.

2.8 XML Parsing

In the NETCOM MDSCloud XML is only used when receiving messages from PClients and PServers, while all other internal communications are handled differently as outlined already. The MDSProxy component is solely responsible for all XML interpretations between MDS node components and PClient/PServers as well as the authentication of users. The MDSProxy begins by accepting any outside socket connection and creating a task, which will handle the initialization of the session. This task, when run by the pool of threads, takes incoming XML, and parses it to usable variable fields using the JDOM Library (www.jdom.org). This task will either accept non-login requests (eg. newuser), followed by a disconnection, or accept a login request and enter the socket into the component's collection of *validated* connections. In the case of incorrect XML being sent, and error response is returned to the PClient/PServer and the socket is closed. The MDSProxy then constantly polls the validated connections and creates new tasks which handling incoming XML and parse it into valid Requests to be interpreted by the MDS node components. Logouts and dead connections are handled simply by removing a socket from the validated connection collection.

2.9 Service Availability (Crash Detection)

The NETCOM MDSCloud is designed to handle the possiblity of MDS nodes and MDSProxies terminating due to abnormal conditions. An MDS node can go down with no hinderence to search requests, and will only affect PServers directly connected to the afflicted node, who must reload their share information to a new MDS node. This is possible because every MDS node contact is done through the MDSProxy. With every attempt to contact an MDS node, an MDSProxy updates a *failure count* for that specific MDS node in the LDAP listing. When a successful connection is made, that failure count is set to zero, while if there is an error, that failur count is incremented. Once the failure count is past a certain threshold, the MDSProxy removed the MDS node from the LDAP listing. However, when a MDSProxy fails, there is a bigger problem. The Graphical Statistics Viewer can detect when a MDSProxy has failed, but because PClients and PServers do not *receive* connections from the MDSCloud, they cannot be told to redirect. This is a severe case and clients must find another MDSProxy,

however the rest of the MDSCloud continues functioning normally for there is in essence no central point that can bring down the entire system, except for catastrophic LDAP failures.

3 Use Cases

3.1 Login/Logout Use Cases

Use-Case: PClient Login to MDSProxy (LoginCloud):
Scenario: SSAD-UC-01
Reference: MDS-UC1 in SSRD
Sequence Diagram: SSAD-SD-1

1. PClient sends LoginCloud XML request to MDSProxy
2. MDSProxy parses XML
3. MDSProxy creates AuthenticationEntry from username and password XML tags
4. LDAPCommunicator is called to verify AuthenticationEntry
5. LDAPCommunicator searches the LDAP server for AuthenticationEntry
6. MDSProxy accepts and saves the connection socket from PClient
7. MDSProxy creates XML AcceptLogin response and sends it to Pclient
8. Log entry is added to the Logger
9. Pclient receives AcceptLogin response from MDSProxy

Extensions:

4a. Password or username are of invalid format. MDSProxy creates XML ExInvalidLogin response, sends it back to PClient and closes the connection socket.

5a. LDAPCommunicator is not able to find a AuthenticationEntry. MDSProxy creates XML ExInvalidLogin response, sends it back to PClient and closes the connection socket.

Use-Case: PClient Creates New Account (NewUser):
Scenario: SSAD-UC-02
Reference: MDS-UC2 in SSRD

1. PClient sends NewUser XML request to MDSProxy
2. MDSProxy parses XML
3. MDSProxy creates AuthenticationEntry from username and password XML tags
4. LDAPCommunicator is called add AuthenticationEntry
5. LDAPCommunicator verifies that AuhenticationEntry does not exist on LDAP server
6. LDAPCommunicator saves AuhenticationEntry on LDAP server
7. MDSProxy creates XML AcceptNewUser response and sends it to PClient
8. Log entry is added to the Logger
9. PClient receives AcceptNewLogin response from MDSProxy

Extensions:

4a. Password or username are of invalid format. MDSProxy creates XML ExInvalidUser response, sends it back to PClient and closes the connection socket.

5a. LDAPCommunicator finds an AuthenticationEntry on LDAP server MDSProxy creates XML ExInvalidUser response, sends it back to PClient and closes the connection

socket.

Use-Case: PClient Logout of MDS (LogoutCloud):
Scenario: SSAD-UC-03
Reference: MDS-UC5 in SSRD

1. PClient sends LogoutCloud XML request to MDSProxy
2. MDSProxy parses XML
3. MDSProxy deletes the connection socket for PClient
4. Log entry is added to the Logger

Use-Case: PServer Login to MDSCloud (LoginCloud):
Scenario: SSAD-UC-04
Reference: MDS-UC6 in SSRD

1. PServer sends LoginCloud XML request to MDSProxy
2. MDSProxy parses XML
3. MDSProxy creates AuthenticationEntry from the given username
4. LDAPCommunicator is called to verify AuthenticationEntry
5. MDSProxy accepts and saves the connection socket from PServer
6. LDAPCommunicator is searched for the least loaded MDS that is assigned to the PServer
7. MDSProxy creates s connection socket between MDS and MDSProxy
8. The failure count of the NodeEntry for the MDS is updated on LDAP
9. MDSProxy creates XML AcceptLogin response and sends it to PServer
10. Log entry is added to the Logger
11. PServer receives AcceptLogin response from MDSProxy

Extensions:

4a. Username alreasy exists on LDAP server
MDSProxy creates XML ExInvalidLogin response sends it back to PServer and closes the connection socket.

7a. Connection between MDS and MDSProxy fails. The failure_count of the NodeEntry for the MDS is updated on LDAP. Another MDS is selected.

Use-Case: PServer Logout from MDSCloud (LogoutCloud):
Scenario: SSAD-UC-05
Reference: MDS-UC9 in SSRD
Sequence Diagram: SSAD-SD-2

1. PServer sends LogoutCloud XML request to MDSProxy
2. MDSProxy parses XML
3. MDSProxy creates AuthenticationEntry from the given username
4. LDAPCommunicator removes AuthenticationEntry from LDAP server
5. MDSProxy creates a RemoveShareRequest
6. MDSProxy sends RemoveShareRequest to MDS that PServer is designated to
7. MDS removes share to internal storage database
8. MDS updates BloomFilter

9. MDSProxy terminates the connection sockets for the PServer
10. Log entry is added to the Logger

Use-Case: MDS Binds to MDSCloud:
Scenario: SSAD-UC-06
Reference: MDS-UC10 in SSRD
Sequence Diagram: SSAD-SD-5

1. MDS sends request to MDSCloud to connect to cloud
2. MDS binds itself to LDAP, thus becoming part of the MDSCloud

Extensions:

2a. LDAPCommunicator does not recognize user

3.2 Search/Share Use Cases

Use-Case: PClient Searches MDSCloud (Breadth-First-Driven (BFD) Search):
Scenario: SSAD-UC-07
Reference: MDS-UC3 in SSRD
Sequence Diagram: SSAD-SD-3, SSAD-SD-3A

1. PClient sends XML request to MDSProxy for search
2. MDSProxy parses XML into SearchRequest
3. MDSProxy sends SearchRequest to MDS that PClient is designated to
4. MDS searches all local PSDatas, match found and returns a ReturnRequest
5. MDS propagates SearchRequest to all friends
6. Remote MDS recieves SearchRequest it has not seen
7. Remote MDS searches all local PSDatas, match found and returns a ReturnRequest
8. Remote MDS propagates SearchRequest to all friends, goes to step 5.)

Extensions:

4a. Remote MDS searches all local PSDatas, no match found so does not return a ReturnRequest, but continues.
6a. 6a. Remote MDS has already seen this SearchRequest, so does nothing more for this request
7a. Remote MDS searches all local PSDatas, no match found so does not return anything, but continues.

Use-Case: PClient Searches MDSCloud (Bloom Search):
Scenario: SSAD-UC-08
Reference: MDS-UC4 in SSRD
Sequence Diagram: SSAD-SD-3, SSAD-SD-3B

1. PClient sends XML request to MDSProxy for search
2. MDSProxy parses XML into SearchRequest
3. MDSProxy sends SearchRequest to MDS that PClient is designated to
4. MDS searches all local PSDatas, if match found returns a ReturnRequest

5. MDS compares SearchRequest against all BloomFilters
6. A match is found, SearchRequest is propagated to Remote MDS corresponding to the matching BloomFilter
7. Remote MDS recieves SearchRequest
8. Remote MDS searches all local PSDatas, match found and returns a ReturnRequest

Extensions:

6a. No Bloom match is found, MDS does nothing more for this request

8a. Remote MDS searches all local PSDatas, no match found so does not return anything

Use-Case: PServer Adds Shared File to MDS (Addshare):
Scenario: SSAD-UC-09
Reference: MDS-UC7 in SSRD
Sequence Diagram: SSAD-SD-4

1. PServer sends XML request to MDSProxy
2. MDSProxy parses XML into a AddShareRequest
3. MDSProxy sends AddShareRequest to MDS that PServer is designated to
4. MDS adds share to internal storage database
5. MDS updates BloomFilter and spawns new BloomUpdate packets

Use-Case: PServer Removes Shared File from MDS (RemoveShare):
Scenario: SSAD-UC-10
Reference: MDS-UC8 in SSRD
Sequence Diagram: SSAD-SD-5

1. PServer sends XML request to MDSProxy
2. MDSProxy parses XML into a RemoveShareRequest
3. MDSProxy sends RemoveShareRequest to MDS that PServer is designated to
4. MDS removes share name from internal storage database
5. MDS rebuilds BloomFilter and spawns new BloomUpdate packets

Extensions:

4a. MDS cannot find file name to be removed. However, no action is taken by MDS, and no error is sent to PServer

3.3 Logging and Statistics Use Cases

Use-Case: MDSNode (MDS or MDSProxy) adds a LogEntry to the Logger:
Scenario: SSAD-UC-11

1. MDS creates a new LogEntry
2. MDS adds LogEntry to the Logger
3. Logger puts LogEntry in the FileWriter's queue
4. Logger updates the number of LogEntries

5. FileWriter dequeues the LogEntry
6. FileWriter opens output file
7. FileWriter writes the LogEntry to the end of the output file
8. FileWriter closes the output file

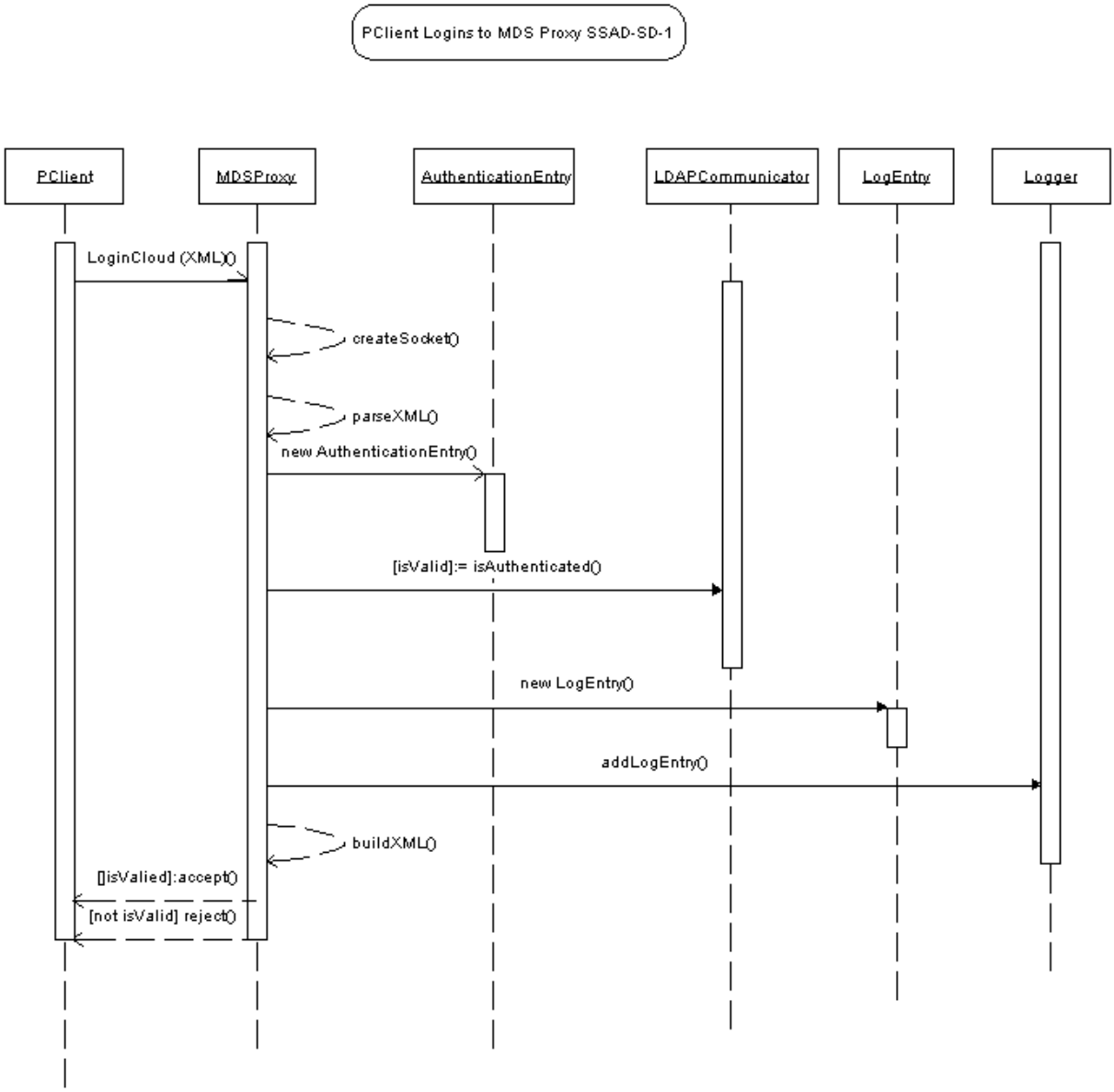
Extensions:
6a. Output file is missing. New file is created
8a. More LogEntries are in the queue. Step 7 is repeated until queue is empty.

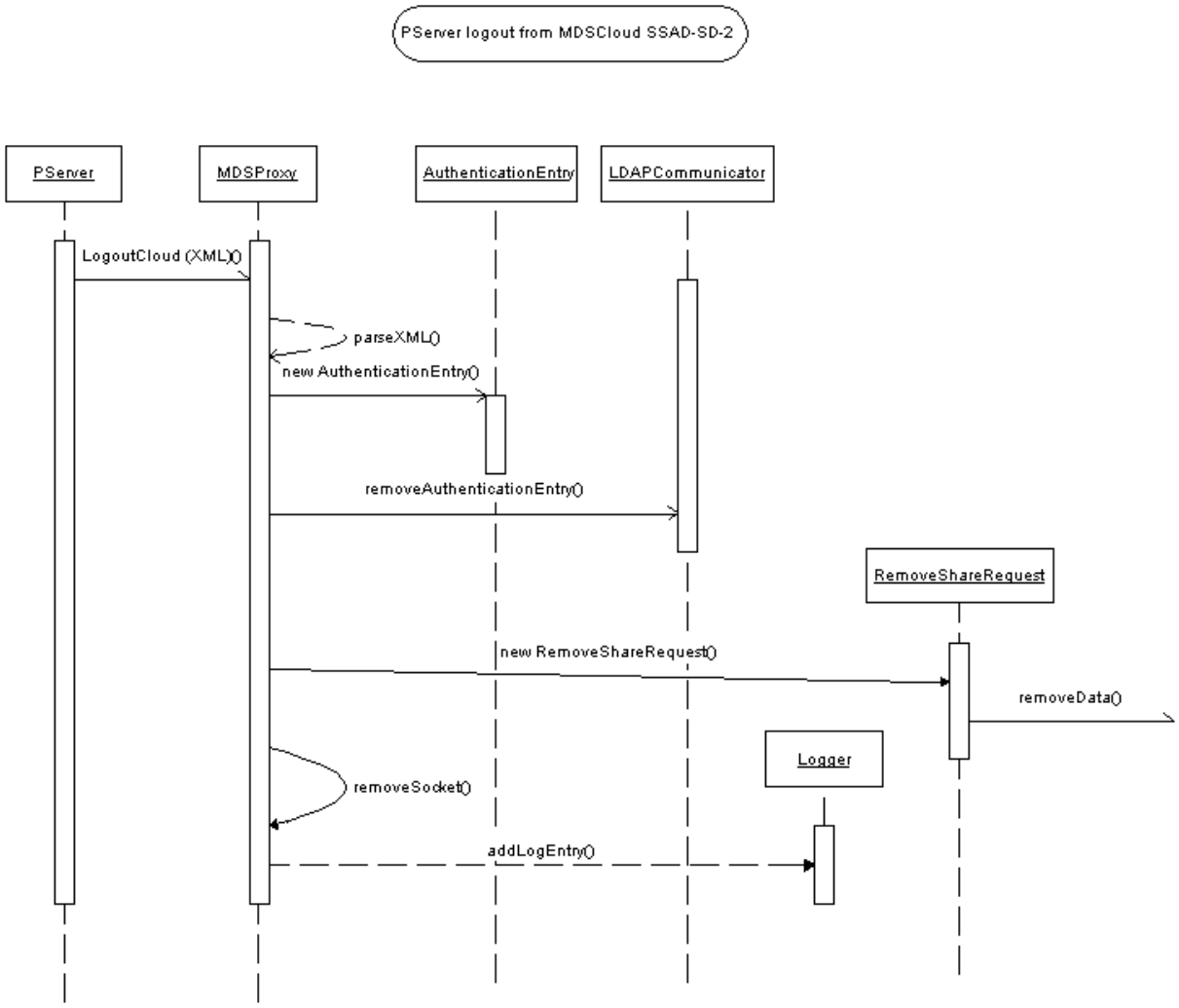
Use-Case: User clicks 'refresh' menu item to update the statistics
Scenario: SSAD-UC-12
Sequence Diagram: SSAD-SD-6

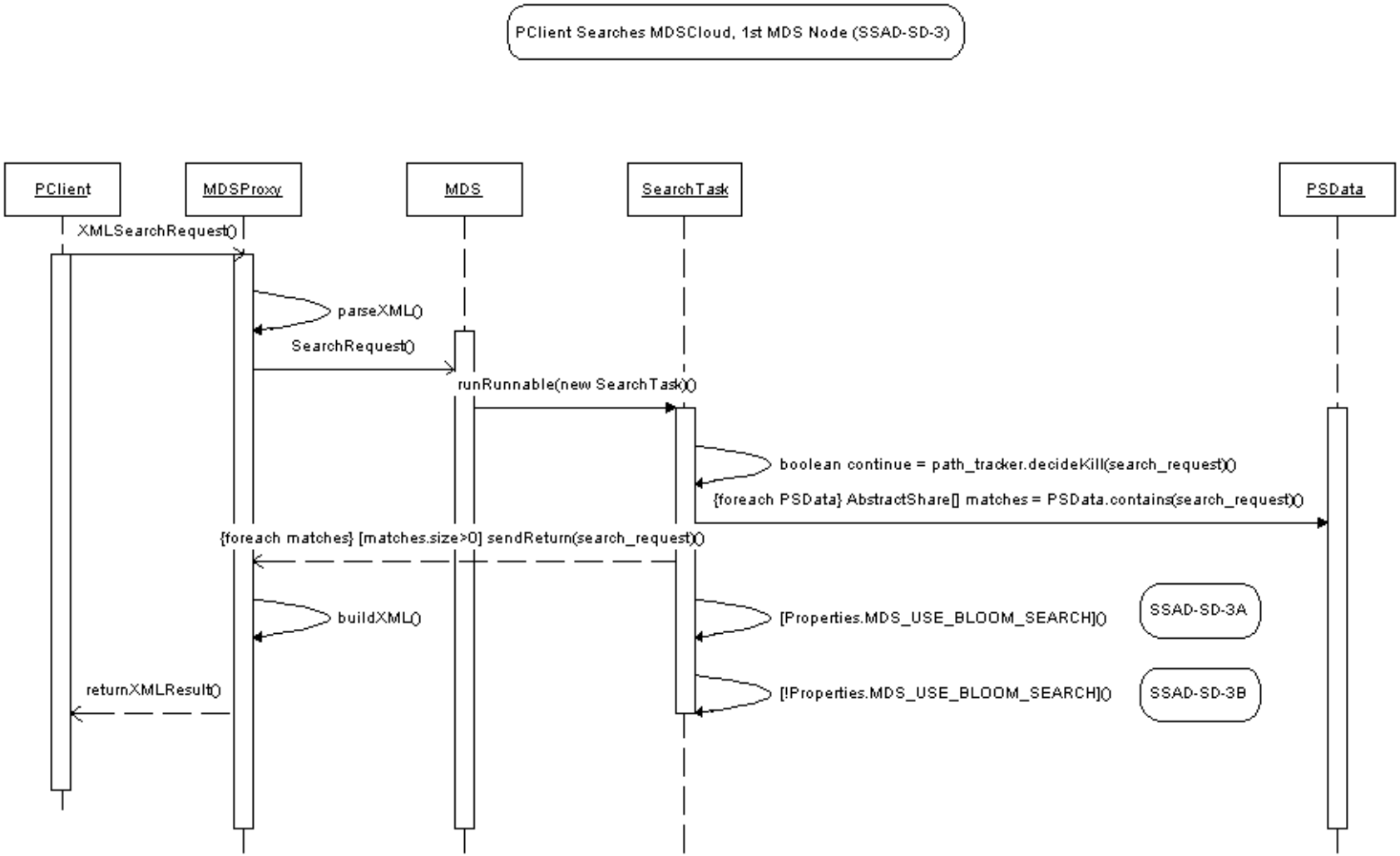
1. Mainform uses LDAPCommunicator to verify that the required node is still registered withLDAP
2. Mainform creates and LogMessage request
3. Mainform adds LogMessage to GUIClient
4. GUIClient sends LogMessage to MDS or MDSProxy
5. Logger receives the results
6. Logger retrieves required information
7. Logger sends results back to GUIClient
8. GUIClient receives LogMessage with results from MDS or MDSProxy
9. Mainform gets LogMessage and updates the cached data
10. Mainform repaints for all the visible components

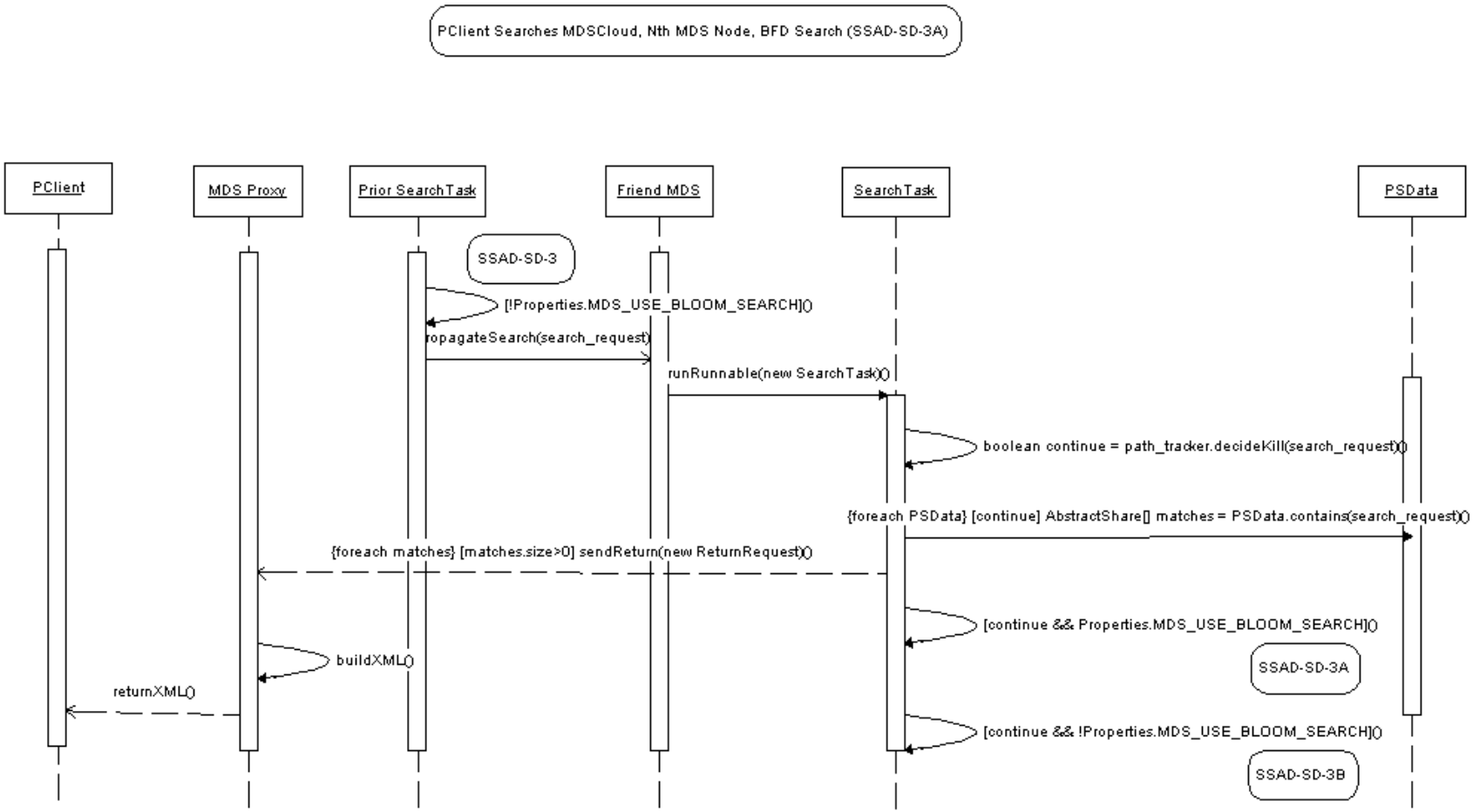
Extensions:
1a. The node is no longer in LDAP list. Error message is displayed

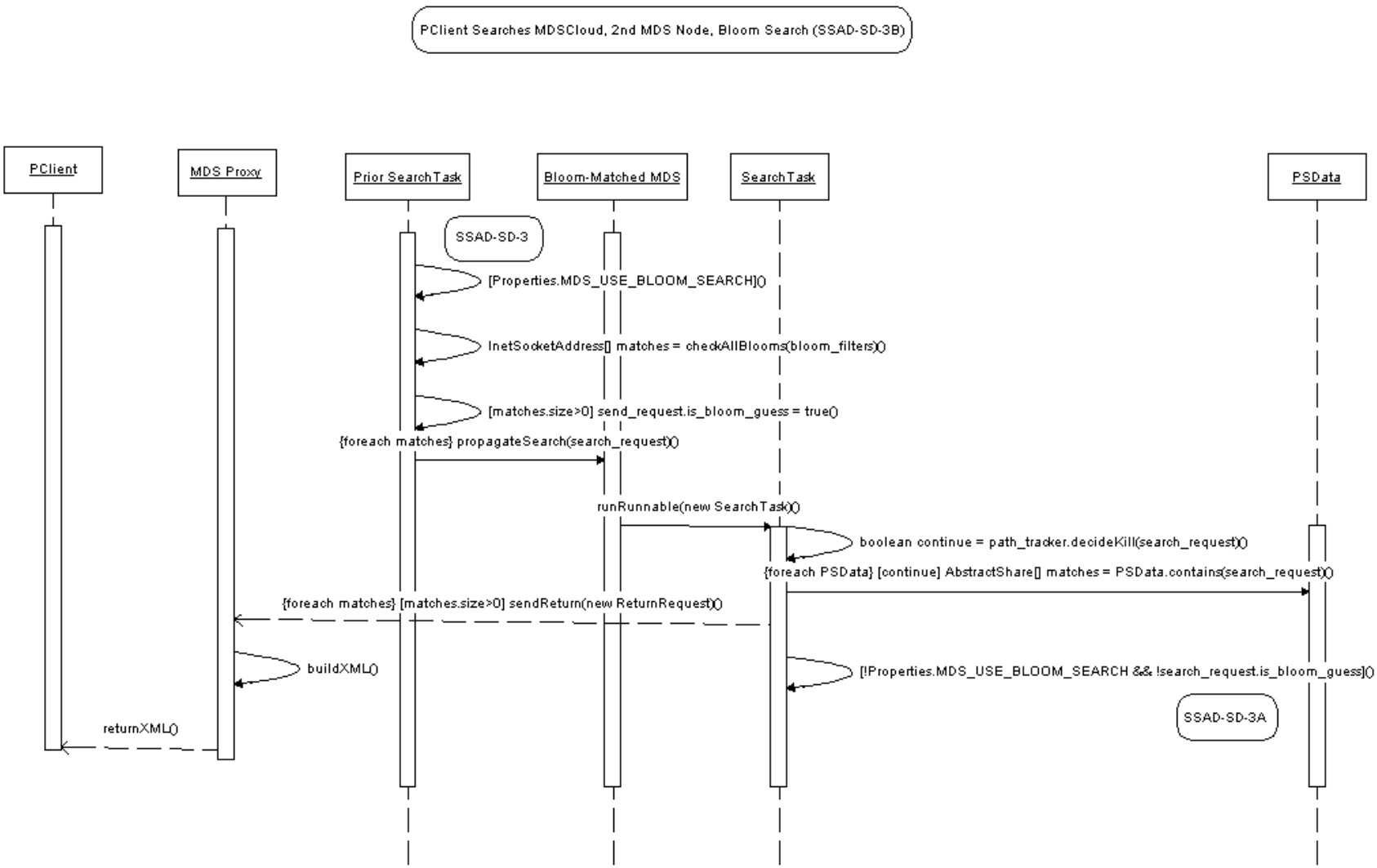
4 Sequence Diagrams

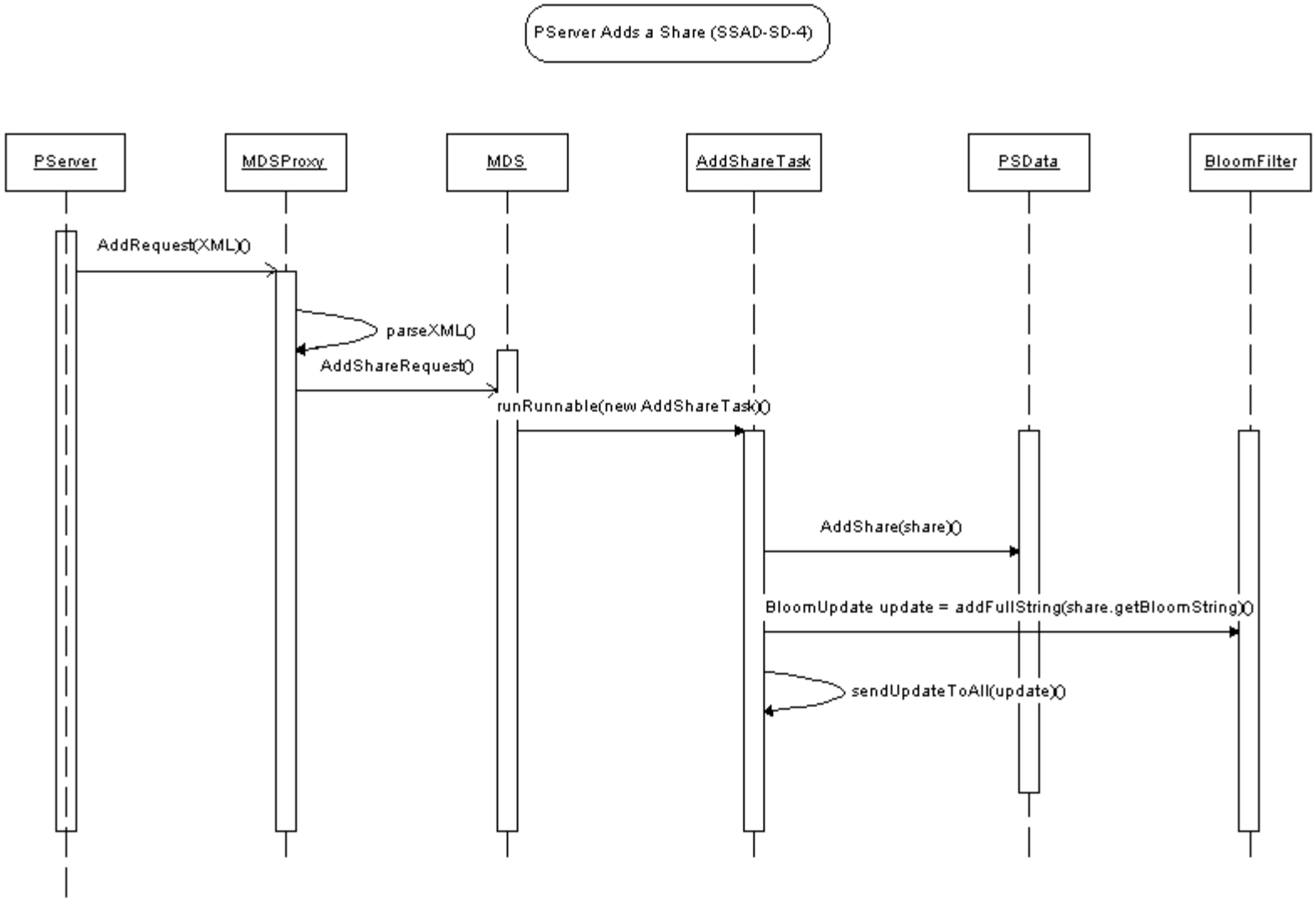

PClient Logins to MDS Proxy SSAD-SD-1
PClient
MDSProxy
AuthenticationEntry
LDAPCommunicator
LogEntry
Logger
LoginCloud (XML)()
createSocket()
parseXML()
new AuthenticationEntry()
[isValid]:= isAuthenticated()
new LogEntry()
addLogEntry()
buildXML()
[]isValied]:accept()
[not isValid] reject()

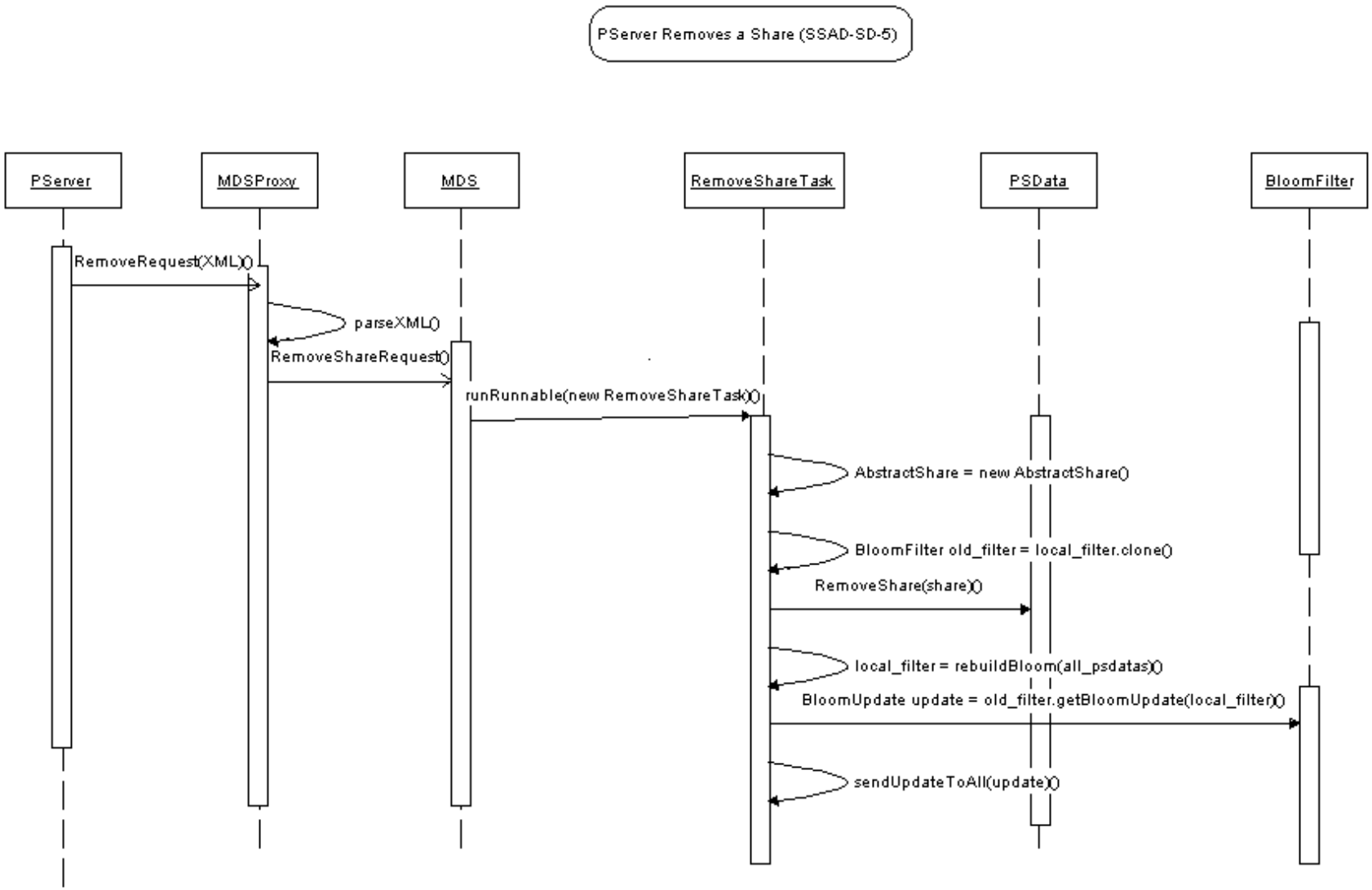

PServer logout from MDSCloud SSAD-SD-2
PServer
MDSProxy
AuthenticationEntry
LDAPCommunicator
RemoveShareRequest
Logger
LogoutCloud (XML)()
parseXML()
new AuthenticationEntry()
removeAuthenticationEntry()
new RemoveShareRequest()
removeData()
removeSocket()
addLogEntry()

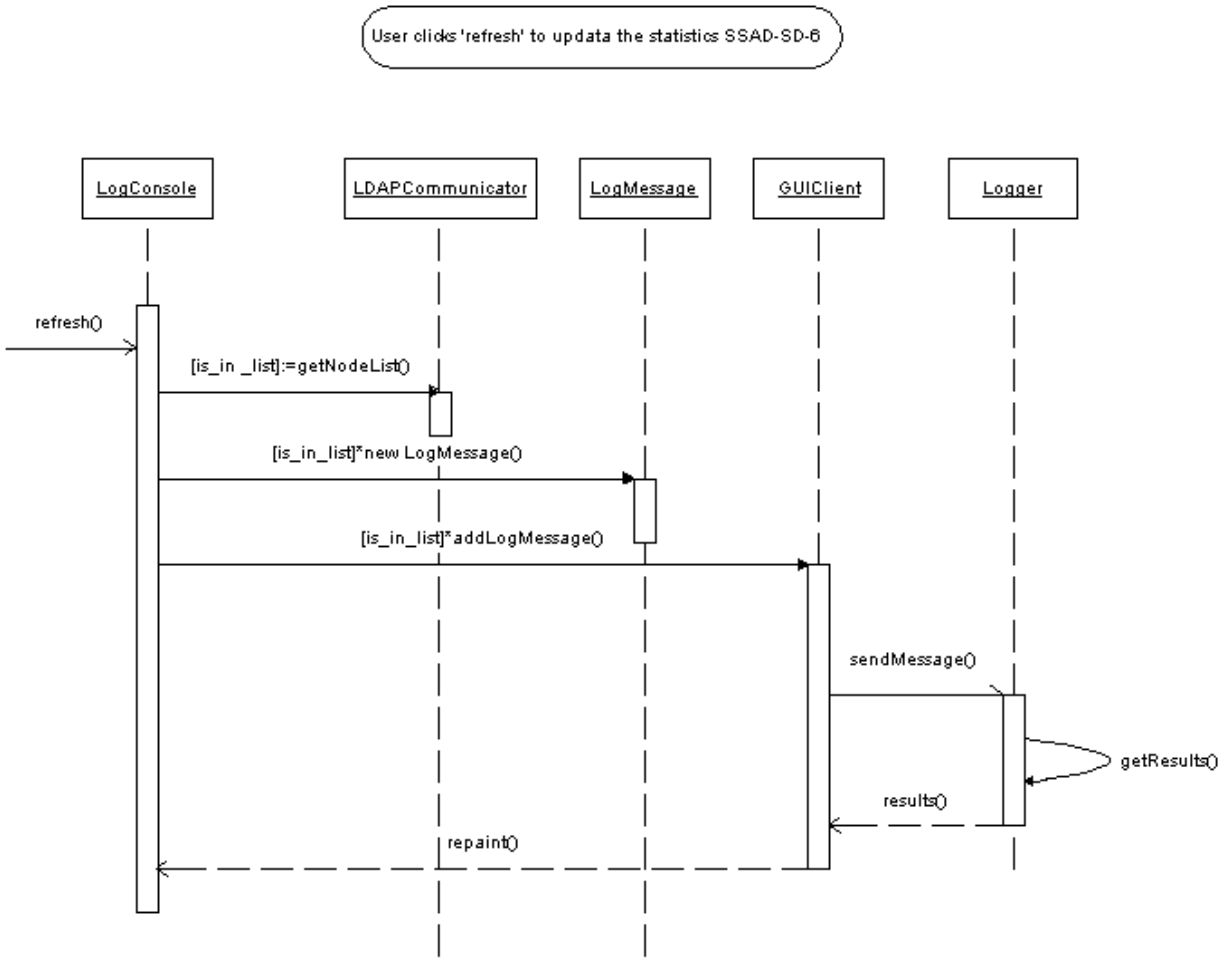

PClient Searches MDSCloud, 1st MDS Node (SSAD-SD-3)
PClient
MDSProxy
MDS
SearchTask
PSData
XMLSearchRequest()
parseXML()
SearchRequest()
runRunnable(new SearchTask())
boolean continue = path_tracker.decideKill(search_request)()
{foreach PSData} AbstractShare[] matches = PSData.contains(search_request)()
{foreach matches} [matches.size>0] sendReturn(search_request)()
buildXML()
[Properties.MDS_USE_BLOOM_SEARCH]()
SSAD-SD-3A
returnXMLResult()
[!Properties.MDS_USE_BLOOM_SEARCH]()
SSAD-SD-3B


PClient Searches MDSCloud, Nth MDS Node, BFD Search (SSAD-SD-3A)
PClient
MDS Proxy
Prior SearchTask
Friend MDS
SearchTask
PSData
SSAD-SD-3
[!Properties.MDS_USE_BLOOM_SEARCH]()
ropagateSearch(search_request)
runRunnable(new SearchTask())
boolean continue = path_tracker.decideKill(search_request)()
{foreach PSData} [continue] AbstractShare[] matches = PSData.contains(search_request)()
{foreach matches} [matches.size>0] sendReturn(new ReturnRequest)()
[continue && Properties.MDS_USE_BLOOM_SEARCH]()
SSAD-SD-3A
buildXML()
returnXML()
[continue && !Properties.MDS_USE_BLOOM_SEARCH]()
SSAD-SD-3B

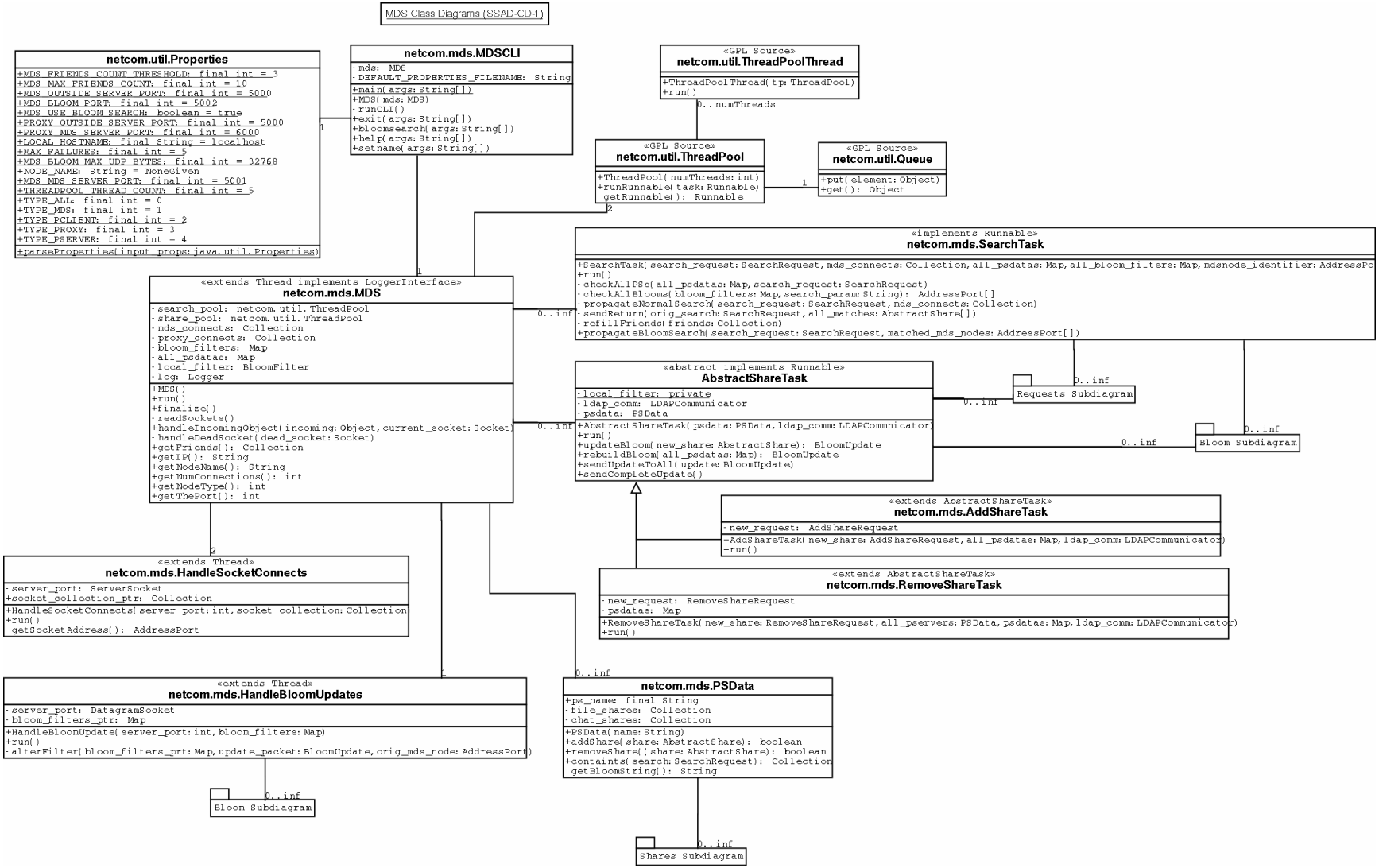

PClient Searches MDSCloud, 2nd MDS Node, Bloom Search (SSAD-SD-3B)
PClient
MDS Proxy
Prior SearchTask
Bloom-Matched MDS
SearchTask
PSData
SSAD-SD-3
[Properties.MDS_USE_BLOOM_SEARCH]()
InetSocketAddress[] matches = checkAllBlooms(bloom_filters)()
[matches.size>0] send_request.is_bloom_guess = true()
{foreach matches} propagateSearch(search_request)()
runRunnable(new SearchTask)()
boolean continue = path_tracker.decideKill(search_request)()
{foreach PSData} [continue] AbstractShare[] matches = PSData.contains(search_request)()
{foreach matches} [matches.size>0] sendReturn(new ReturnRequest)()
buildXML()
[!Properties.MDS_USE_BLOOM_SEARCH && !search_request.is_bloom_guess]()
SSAD-SD-3A
returnXML()

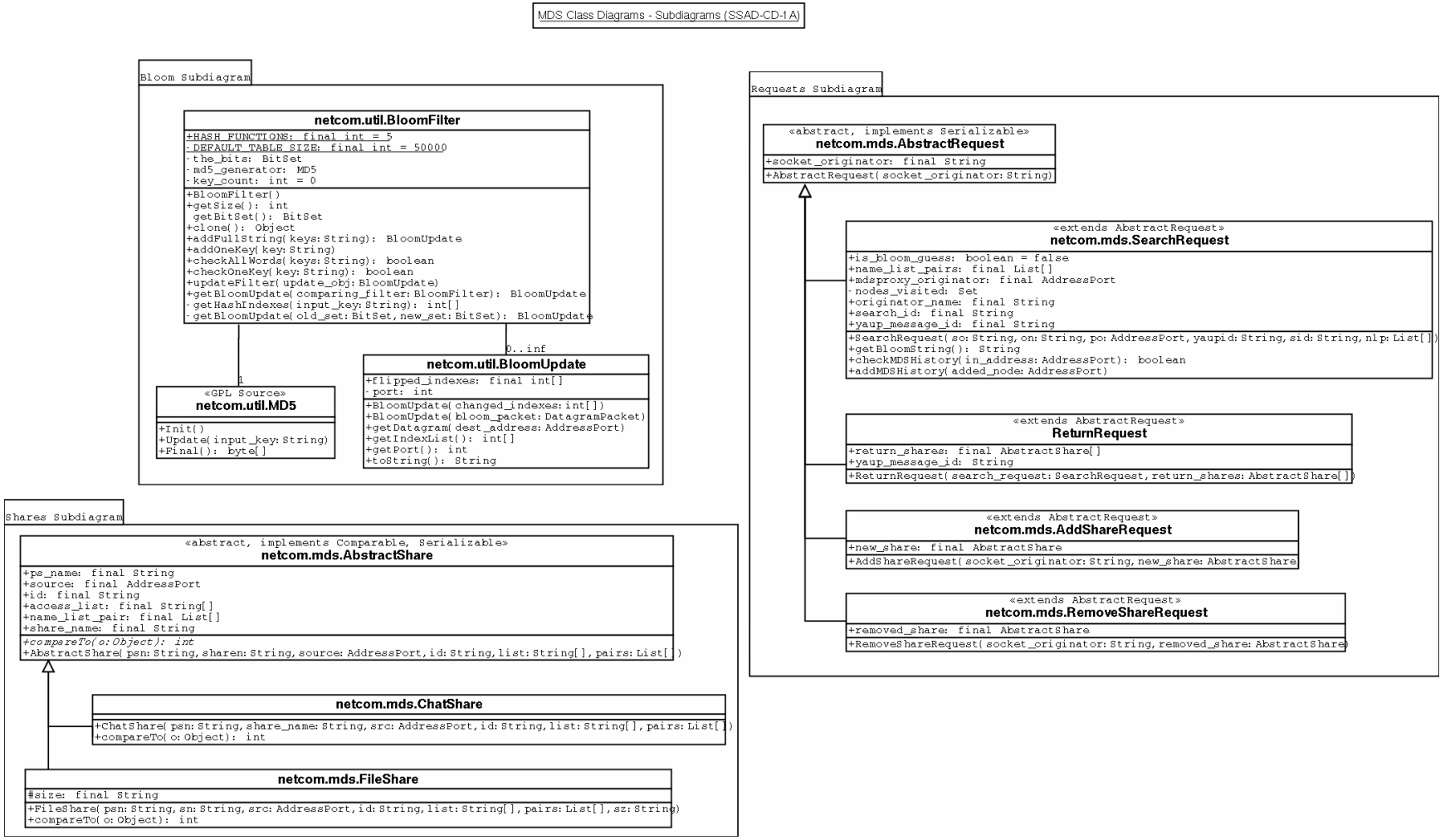

PServer Adds a Share (SSAD-SD-4)
PServer
MDSProxy
MDS
AddShareTask
PSData
BloomFilter
AddRequest(XML)()
parseXML()
AddShareRequest()
runRunnable(new AddShareTask)()
AddShare(share)()
BloomUpdate update = addFullString(share.getBloomString)()
sendUpdateToAll(update)()

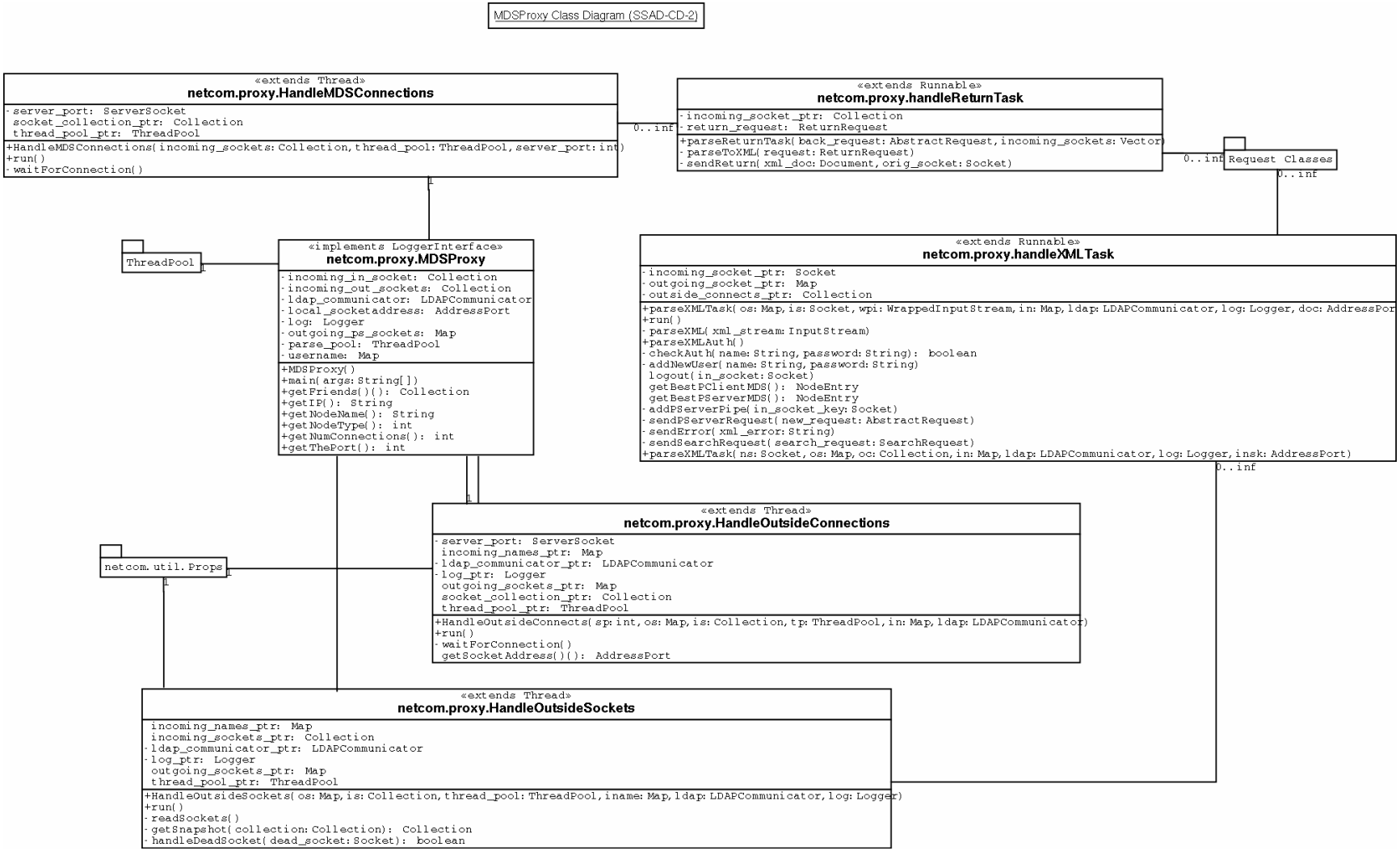

PServer Removes a Share (SSAD-SD-5)
PServer
MDSProxy
MDS
RemoveShareTask
PSData
BloomFilter
RemoveRequest(XML)()
parseXML()
RemoveShareRequest()
runRunnable(new RemoveShareTask)()
AbstractShare = new AbstractShare()
BloomFilter old_filter = local_filter.clone()
RemoveShare(share)()
local_filter = rebuildBloom(all_psdatas)()
BloomUpdate update = old_filter.getBloomUpdate(local_filter)()
sendUpdateToAll(update)()

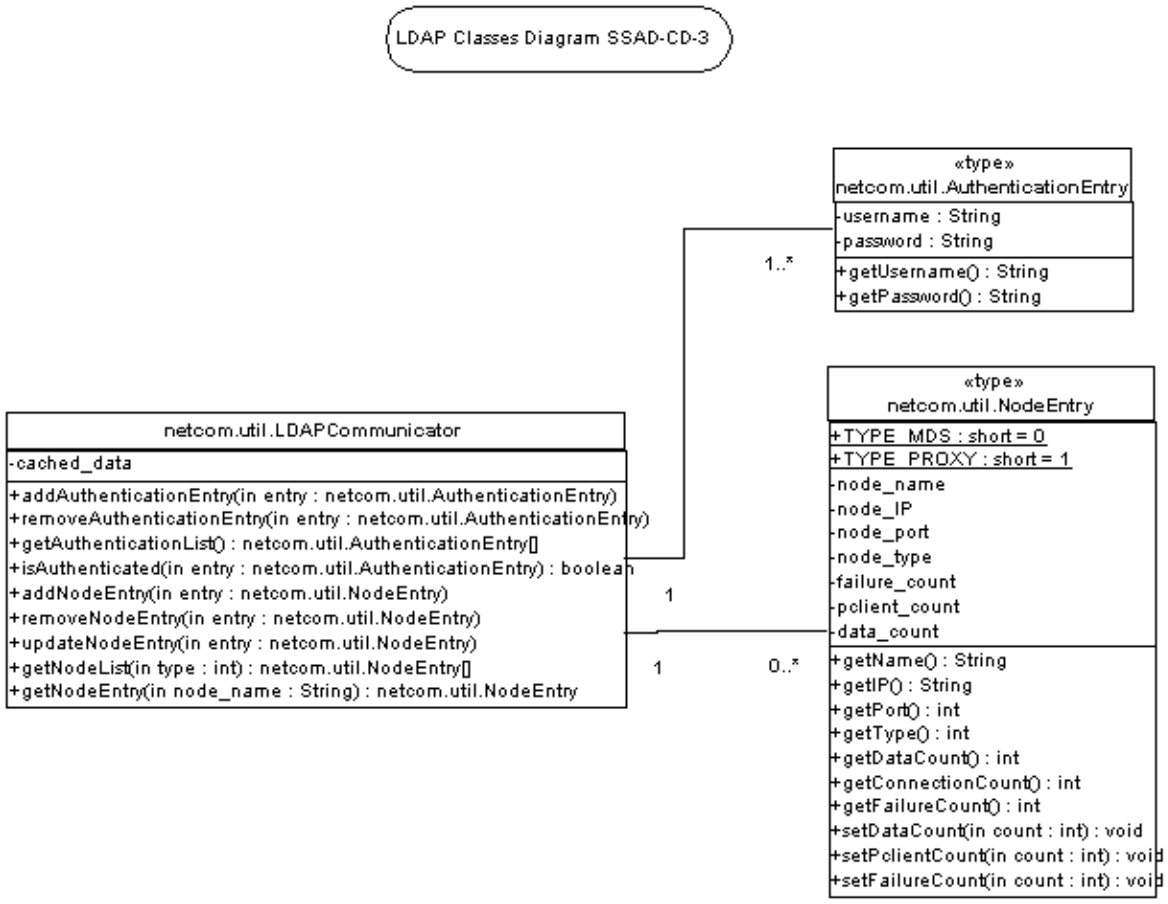


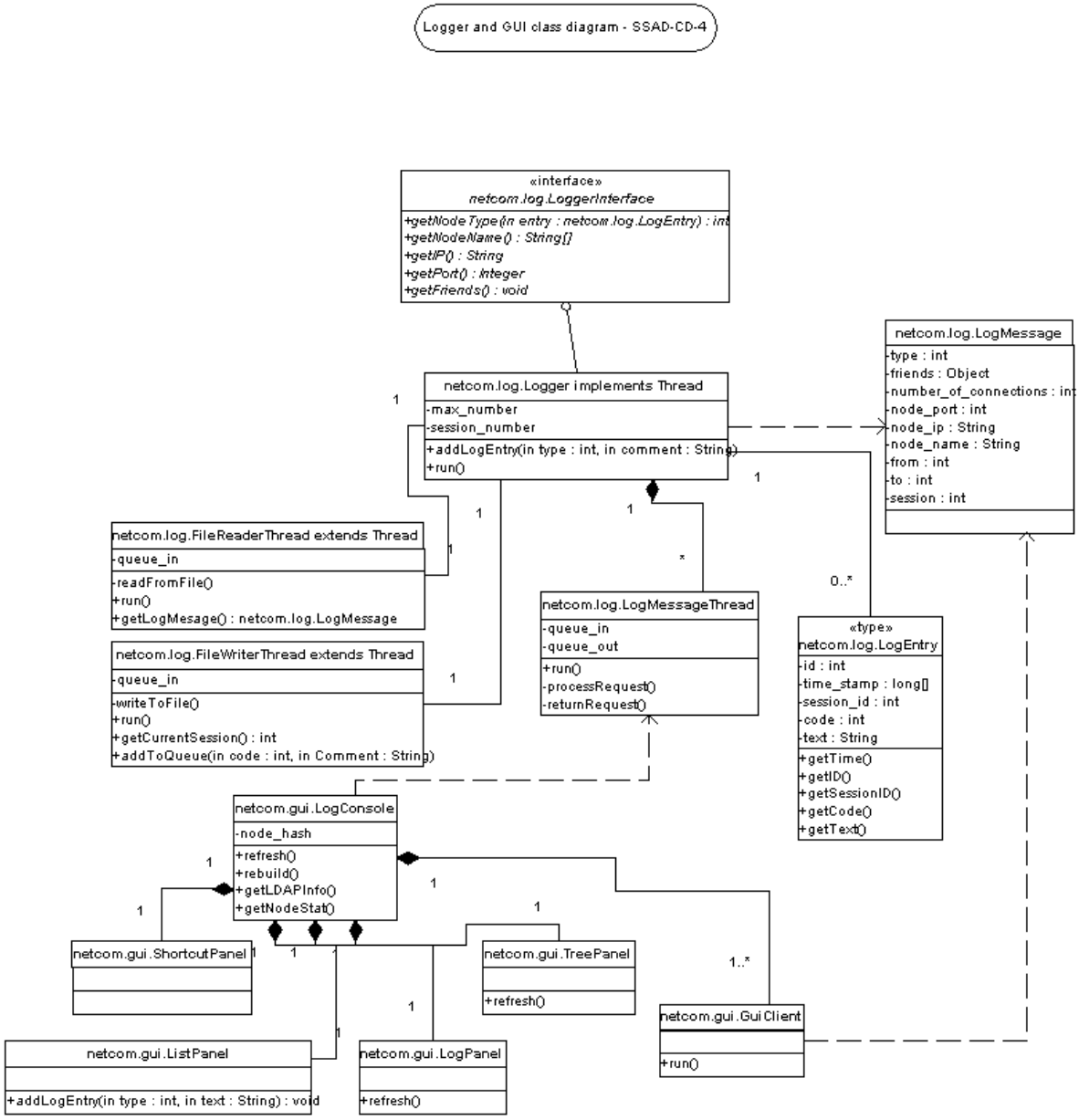
5 Class Diagrams


MDS Class Diagrams (SSAD-CD-1)
netcom.util.Properties
+MDS_FRIENDS_COUNT_THRESHOLD: final int = 3
+MDS_MAX_FRIENDS_COUNT: final int = 10
+MDS_OUTSIDE_SERVER_PORT: final int = 5000
+MDS_BLOOM_PORT: final int = 5002
+MDS_USE_BLOOM_SEARCH: boolean = true
+PROXY_OUTSIDE_SERVER_PORT: final int = 5000
+PROXY_MDS_SERVER_PORT: final int = 6000
+LOCAL_HOSTNAME: final String = localhost
+MAX_FAILURES: final int = 5
+MDS_BLOOM_MAX_UDP_BYTES: final int = 32768
+NODE_NAME: String = NoneGiven
+MDS_MDS_SERVER_PORT: final int = 5001
+THREADPOOL_THREAD_COUNT: final int = 5
+TYPE_ALL: final int = 0
+TYPE_MDS: final int = 1
+TYPE_PCLIENT: final int = 2
+TYPE_PROXY: final int = 3
+TYPE_PSERVER: final int = 4
+parseProperties(input_props: java.util.Properties)
netcom.mds.MDSCLI
- mds: MDS
- DEFAULT_PROPERTIES_FILENAME: String
+main(args: String[])
+MDS(mds: MDS)
- runCLI()
+exit(args: String[])
+bloomsearch(args: String[])
+help(args: String[])
+setname(args: String[])
«GPL Source»
netcom.util.ThreadPoolThread
+ThreadPoolThread(tp: ThreadPool)
+run()
0..numThreads
«GPL Source»
netcom.util.ThreadPool
+ThreadPool(numThreads: int)
+runRunnable(task: Runnable)
getRunnable(): Runnable
«GPL Source»
netcom.util.Queue
+put(element: Object)
+get(): Object
«implements Runnable»
netcom.mds.SearchTask
+SearchTask(search_request: SearchRequest, mds_connects: Collection, all_psdatas: Map, all_bloom_filters: Map, mdsnode_identifier: AddressPo
+run()
- checkAllPSs(all_psdatas: Map, search_request: SearchRequest)
- checkAllBlooms(bloom_filters: Map, search_param: String): AddressPort[]
- propagateNormalSearch(search_request: SearchRequest, mds_connects: Collection)
- sendReturn(orig_search: SearchRequest, all_matches: AbstractShare[])
- refillFriends(friends: Collection)
+propagateBloomSearch(search_request: SearchRequest, matched_mds_nodes: AddressPort[])
«extends Thread implements LoggerInterface»
netcom.mds.MDS
- search_pool: netcom.util.ThreadPool
- share_pool: netcom.util.ThreadPool
- mds_connects: Collection
- proxy_connects: Collection
- bloom_filters: Map
- all_psdatas: Map
- local_filter: BloomFilter
- log: Logger
+MDS()
+run()
+finalize()
- readSockets()
+handleIncomingObject(incoming: Object, current_socket: Socket)
- handleDeadSocket(dead_socket: Socket)
+getFriends(): Collection
+getIP(): String
+getNodeName(): String
+getNumConnections(): int
+getNodeType(): int
+getThePort(): int
«abstract implements Runnable»
AbstractShareTask
- local_filter: private
- ldap_comm: LDAPCommunicator
- psdata: PSData
+AbstractShareTask(psdata: PSData, ldap_comm: LDAPCommunicator)
+run()
+updateBloom(new_share: AbstractShare): BloomUpdate
+rebuildBloom(all_psdatas: Map): BloomUpdate
+sendUpdateToAll(update: BloomUpdate)
+sendCompleteUpdate()
Requests Subdiagram
Bloom Subdiagram
«extends AbstractShareTask»
netcom.mds.AddShareTask
- new_request: AddShareRequest
+AddShareTask(new_share: AddShareRequest, all_psdatas: Map, ldap_comm: LDAPCommunicator)
+run()
«extends AbstractShareTask»
netcom.mds.RemoveShareTask
- new_request: RemoveShareRequest
- psdatas: Map
+RemoveShareTask(new_share: RemoveShareRequest, all_pservers: PSData, psdatas: Map, ldap_comm: LDAPCommunicator)
+run()
«extends Thread»
netcom.mds.HandleSocketConnects
- server_port: ServerSocket
+socket_collection_ptr: Collection
+HandleSocketConnects(server_port: int, socket_collection: Collection)
+run()
getSocketAddress(): AddressPort
«extends Thread»
netcom.mds.HandleBloomUpdates
- server_port: DatagramSocket
- bloom_filters_ptr: Map
+HandleBloomUpdate(server_port: int, bloom_filters: Map)
+run()
- alterFilter(bloom_filters_prt: Map, update_packet: BloomUpdate, orig_mds_node: AddressPort)
Bloom Subdiagram
netcom.mds.PSData
+ps_name: final String
- file_shares: Collection
- chat_shares: Collection
+PSData(name: String)
+addShare(share: AbstractShare): boolean
+removeShare((share: AbstractShare): boolean
+containts(search: SearchRequest): Collection
getBloomString(): String
Shares Subdiagram


MDS Class Diagrams - Subdiagrams (SSAD-CD-1 A)
Bloom Subdiagram
netcom.util.BloomFilter
+HASH_FUNCTIONS: final int = 5
-DEFAULT_TABLE_SIZE: final int = 50000
-the_bits: BitSet
-md5_generator: MD5
-key_count: int = 0
+BloomFilter()
+getSize(): int
getBitSet(): BitSet
+clone(): Object
+addFullString(keys:String): BloomUpdate
+addOneKey(key:String)
+checkAllWords(keys:String): boolean
+checkOneKey(key:String): boolean
+updateFilter(update_obj:BloomUpdate)
+getBloomUpdate(comparing_filter:BloomFilter): BloomUpdate
-getHashIndexes(input_key:String): int[]
-getBloomUpdate(old_set:BitSet,new_set:BitSet): BloomUpdate
1
0..inf
«GPL Source»
netcom.util.MD5
+Init()
+Update(input_key:String)
+Final(): byte[]
netcom.util.BloomUpdate
+flipped_indexes: final int[]
-port: int
+BloomUpdate(changed_indexes:int[])
+BloomUpdate(bloom_packet:DatagramPacket)
+getDatagram(dest_address:AddressPort)
+getIndexList(): int[]
+getPort(): int
+toString(): String
Shares Subdiagram
«abstract, implements Comparable, Serializable»
netcom.mds.AbstractShare
+ps_name: final String
+source: final AddressPort
+id: final String
+access_list: final String[]
+name_list_pair: final List[]
+share_name: final String
+compareTo(o:Object): int
+AbstractShare(psn:String,sharen:String,source:AddressPort,id:String,list:String[],pairs:List[])
netcom.mds.ChatShare
+ChatShare(psn:String,share_name:String,src:AddressPort,id:String,list:String[],pairs:List[])
+compareTo(o:Object): int
netcom.mds.FileShare
#size: final String
+FileShare(psn:String,sn:String,src:AddressPort,id:String,list:String[],pairs:List[],sz:String)
+compareTo(o:Object): int
Requests Subdiagram
«abstract, implements Serializable»
netcom.mds.AbstractRequest
+socket_originator: final String
+AbstractRequest(socket_originator:String)
«extends AbstractRequest»
netcom.mds.SearchRequest
+is_bloom_guess: boolean = false
+name_list_pairs: final List[]
+mdsproxy_originator: final AddressPort
-nodes_visited: Set
+originator_name: final String
+search_id: final String
+yaup_message_id: final String
+SearchRequest(so:String,on:String,po:AddressPort,yaupid:String,sid:String,nlp:List[])
+getBloomString(): String
+checkMDSHistory(in_address:AddressPort): boolean
+addMDSHistory(added_node:AddressPort)
«extends AbstractRequest»
ReturnRequest
+return_shares: final AbstractShare[]
+yaup_message_id: String
+ReturnRequest(search_request:SearchRequest,return_shares:AbstractShare[])
«extends AbstractRequest»
netcom.mds.AddShareRequest
+new_share: final AbstractShare
+AddShareRequest(socket_originator:String,new_share:AbstractShare)
«extends AbstractRequest»
netcom.mds.RemoveShareRequest
+removed_share: final AbstractShare
+RemoveShareRequest(socket_originator:String,removed_share:AbstractShare)


MDSProxy Class Diagram (SSAD-CD-2)
«extends Thread»
netcom.proxy.HandleMDSConnections
- server_port: ServerSocket
socket_collection_ptr: Collection
thread_pool_ptr: ThreadPool
+HandleMDSConnections(incoming_sockets: Collection, thread_pool: ThreadPool, server_port: int)
+run()
- waitForConnection()
0..inf
«extends Runnable»
netcom.proxy.handleReturnTask
- incoming_socket_ptr: Collection
- return_request: ReturnRequest
+parseReturnTask(back_request: AbstractRequest, incoming_sockets: Vector)
- parseToXML(request: ReturnRequest)
- sendReturn(xml_doc: Document, orig_socket: Socket)
0..inf
Request Classes
0..inf
ThreadPool
«implements LoggerInterface»
netcom.proxy.MDSProxy
- incoming_in_socket: Collection
- incoming_out_sockets: Collection
- ldap_communicator: LDAPCommunicator
- local_socketaddress: AddressPort
- log: Logger
- outgoing_ps_sockets: Map
- parse_pool: ThreadPool
- username: Map
+MDSProxy()
+main(args: String[])
+getFriends()(): Collection
+getIP(): String
+getNodeName(): String
+getNodeType(): int
+getNumConnections(): int
+getThePort(): int
«extends Runnable»
netcom.proxy.handleXMLTask
- incoming_socket_ptr: Socket
- outgoing_socket_ptr: Map
- outside_connects_ptr: Collection
+parseXMLTask(os: Map, is: Socket, wpi: WrappedInputStream, in: Map, ldap: LDAPCommunicator, log: Logger, doc: AddressPor
+run()
- parseXML(xml_stream: InputStream)
+parseXMLAuth()
- checkAuth(name: String, password: String): boolean
- addNewUser(name: String, password: String)
logout(in_socket: Socket)
getBestPClientMDS(): NodeEntry
getBestPServerMDS(): NodeEntry
- addPServerPipe(in_socket_key: Socket)
- sendPServerRequest(new_request: AbstractRequest)
- sendError(xml_error: String)
- sendSearchRequest(search_request: SearchRequest)
+parseXMLTask(ns: Socket, os: Map, oc: Collection, in: Map, ldap: LDAPCommunicator, log: Logger, insk: AddressPort)
0..inf
«extends Thread»
netcom.proxy.HandleOutsideConnections
- server_port: ServerSocket
incoming_names_ptr: Map
- ldap_communicator_ptr: LDAPCommunicator
- log_ptr: Logger
outgoing_sockets_ptr: Map
socket_collection_ptr: Collection
thread_pool_ptr: ThreadPool
+HandleOutsideConnects(sp: int, os: Map, is: Collection, tp: ThreadPool, in: Map, ldap: LDAPCommunicator)
+run()
- waitForConnection()
getSocketAddress()(): AddressPort
netcom.util.Props
«extends Thread»
netcom.proxy.HandleOutsideSockets
incoming_names_ptr: Map
incoming_sockets_ptr: Collection
- ldap_communicator_ptr: LDAPCommunicator
- log_ptr: Logger
outgoing_sockets_ptr: Map
thread_pool_ptr: ThreadPool
+HandleOutsideSockets(os: Map, is: Collection, thread_pool: ThreadPool, iname: Map, ldap: LDAPCommunicator, log: Logger)
+run()
- readSockets()
- getSnapshot(collection: Collection): Collection
- handleDeadSocket(dead_socket: Socket): boolean


LDAP Classes Diagram SSAD-CD-3
«type»
netcom.util.AuthenticationEntry
-username : String
-password : String
+getUsername() : String
+getPassword() : String
1..*
«type»
netcom.util.NodeEntry
+TYPE_MDS : short = 0
+TYPE_PROXY : short = 1
-node_name
-node_IP
-node_port
-node_type
-failure_count
-pclient_count
-data_count
+getName() : String
+getIP() : String
+getPort() : int
+getType() : int
+getDataCount() : int
+getConnectionCount() : int
+getFailureCount() : int
+setDataCount(in count : int) : void
+setPclientCount(in count : int) : void
+setFailureCount(in count : int) : void
netcom.util.LDAPCommunicator
-cached_data
+addAuthenticationEntry(in entry : netcom.util.AuthenticationEntry)
+removeAuthenticationEntry(in entry : netcom.util.AuthenticationEntry)
+getAuthenticationList() : netcom.util.AuthenticationEntry[]
+isAuthenticated(in entry : netcom.util.AuthenticationEntry) : boolean
+addNodeEntry(in entry : netcom.util.NodeEntry)
+removeNodeEntry(in entry : netcom.util.NodeEntry)
+updateNodeEntry(in entry : netcom.util.NodeEntry)
+getNodeList(in type : int) : netcom.util.NodeEntry[]
+getNodeEntry(in node_name : String) : netcom.util.NodeEntry
1
1
0..*



6 Testing Plan

6.1 Testing Strategy

Testing of the NETCOM MDSCloud will follow a strict and thorough procedure to ensure that absolutely every component is functioning properly. Because of the component-based design of the NETCOM MDSCloud, blackbox testing will be the most important to ensure that every component adheres to its described interface and performs as all other components expect it to. Because of the distributed design of this system, blackbox isolation testing is crucial to every component to aid in a quick integration phase, and to allow for individuals to code and test their separate components without the heavy communications overhead that could occur. Glassbox testing should be used in more specific, mission-critical components where there must be valid data being stored within the component itself.

PClient Login to MDSProxy(Login):
SSAD-UC1 black-box testing

Integration testing

A PClient login script will be written that will emulate the PClient's login actions. We will print xml to standard out of the terminal to determine whether or not the correct XML message was sent or received. LDAP will then be checked for the logged in user. Equivalence testing will be used to determine the inputs (username, password).

Isolation testing

The only modules being tested is the MDSProxy. The individual methods, such as "MDSProxy parses XML" will be tested mostly during implementation.
Test log entries will be sent to the Logger.

PClient Creates New Account (NewUser):
SSAD-UC2 black-box testing

Integration testing

A PClient newuser script will be written that will perform the action of creating a new user account. We will use boolean values to determine whether or not the new user account was created as well as checking the LDAP server for the new user account. Equivalence testing will be used to determine inputs (newusername, password).

Isolation testing

The same modules as in SSAD-UC1 will be tested here.

PClient Logout of MDS (LogoutCloud):
SSAD-UC-05 black-box testing

Integration testing

We will write a program that connects a PClient to the MDScloud via login and send log out requests.

Isolated testing
MDSProxy and Logger will be tested

PServer Login to MDSCloud (LoginCloud):
SSAD-UC-04 black-box testing

Integration testing
A PServer login script will be written that will emulate the PServer's login actions. The responses from the MDSproxy will be returned to the script. Equivalence testing will be used to determine the inputs (username).

Isolation testing
Load balancing will be tested on the MDSproxy which will also test the MDSnodes ability to accept a connection from the proxy.

PServer Logout from MDSCloud (LogoutCloud):
SSAD-UC-05 black-box testing

Integration testing
write a program that logs a PServer into the MDScloud and sends logout request. We will send to STDOUT success or failure messages.

Isolation testing
MDSProxy receiving LogoutCloud XML requests.
Logger is tested with correct log entry.
MDSProxy parsing, creation of RemoveShareRequest, sending RemoveShareRequest and MDS updating BloomFilter will be tested as SSAD-UC-5 have been tested.

MDS Binds to MDSCloud:
SSAD-UC-06 black-box testing

Integration testing
A program will be written that will test MDS binding to MDSCloud. Both extension cases will be tested.

Isolation testing
The MDSnode module is the only thing being tested.

PClient Searches MDSCloud (Breadth-First-Driven (BFD) Search):
SSAD-UC-07 black-box testing

Integration testing
A PClient search script will be written that will emulate the PClient's BFD search method. Results from the search will be placed in a text file.

Isolation testing

MDSProxy's parser and method to find an MDSnode will be isolated. Serialized objects will be sent to individual MDSnodes. ReturnRequests will be sent back to a file.

SSAD-UC-07 glass-box testing

Isolation testing

Serialized objects will be sent to individual MDS nodes. MDS nodes will trace their execution of the search to ensure all local shares are compared against, and all friend node connections are contacted and propogated the search request

PClient Searches MDSCloud (Bloom Search):

SSAD-UC-08 black-box testing

Integration testing

We will use a variation of SSAD-UC7 search script that will test the Bloom search method. Results from the search will be placed in a text file.

Isolation testing

The same modules as SSAD-UC7 test plan will be tested here.

SSAD-UC-08 glass-box testing

Isolation testing

Serialized objects will be sent to individual MDS node. MDS nodes will trace their execution of the search to ensure all local shares are compared against. MDS nodes will also identify if any Bloom Filters were found to match, and will provide information confirming proper search requests were sent to matching Bloom Filter nodes only.

PServer Adds Shared File to MDS:

SSAD-UC-09 black-box testing

Integration testing

A PServer Addshare script will be written that will emulate the Pserver Addshare method. We will then send search requests with the new shares to the MDScloud and expect a positive response.

Isolation testing

MDSProxy will be tested for its ability to create an AddShareRequest. MDSNodes will be sent BloomUpdate packets to test bloomFilter updating.

SSAD-UC-09 glass-box testing

Isolated testing

MDSNodes will be sent serialized AddShareRequest objects. The MDS node will provide a trace showing that the share was added correctly *and* that the local BloomFilter is properly updated and *correct* BloomUpdate packets are formed.

PServer Removes Shared File from MDS:
SSAD-UC-10 black-box testing

Integration testing

A PServer RemoveShare script will be written that will emulate the PServer RemoveShare method. We will then send search request and wait for a specified time for no response.

Isolation testing

The modules tested are the same as in SSAD-UC7 test plan but the MDSProxy will be tested for its ability to create a RemoveShareRequest.

SSAD-UC-10 glass-box testing

Isolation testing

MDSNodes will be sent serialized RemoveShareRequest objects. The MDS node will provide a trace showing htat the share was actually removed *and* that the local Bloom Filter is properly destroyed and rebuilt, forming the correct BloomUpdate packets.

MDSNode (MDS or MDSProxy) adds a LogEntry to the Logger:
SSAD-UC-11 black-box testing

Integration testing

The Logger will be tested as it has been described in Isolation testing of SSAD-UC1 test plan. A more thorough test on the logger will be done. We will test both inputs from MDSProxy and MDSnodes for every kind of input.

Isolation testing

Same as SSAD-UC1 test plan

User clicks 'refresh' menu item to update the statistics
SSAD-UC-12 black-box testing

Integration testing

A user will click on refresh to test for updated statistics from the logger.

Isolation testing

Mainform methods are tested.
GUIClient will be sent LogMessage and send LogMessage to a file

7 Documented Class Source File Stubs

AbstractRequest

```
package netcom.mds;

import java.net.*;
import java.io.Serializable;

/**
 * <p>Title: AbstractRequest</p>
 * <p>Description:  This is a parent class to all Request Classes for a MDS node
```

```
 * </p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 * @see SearchRequest
 * @see ReturnRequest
 * @see AddShareRequest
 * @see RemoveShareRequest
 */
public abstract class AbstractRequest implements Serializable
{
    public final Socket originator = null;
    public final int message_id = 0;
}
```

AbstractShare

```
package netcom.mds;

import java.net.*;
import java.util.*;
import java.io.Serializable;

/**
 * <p>Title: Abstract Share</p>
 * <p>Description: This represents one shared piece of information
 * by a PServer</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 * @see FileShare
 * @see ChatShare
 */
abstract class AbstractShare implements Comparable, Serializable
{
    protected final String name = null;
    protected final InetSocketAddress source = null;
    protected final String id = null;
    protected final String[] access_list = null;
    protected final LinkedList[] name_list_pair = null;

    /**
     * This method is provided so that AbstractShares can be placed into
     * sorted data structures so a search can be performed on them
     * optimally
     *
     * @param o  Object to compare this object against
     */
    public int compareTo(Object o)
    {
        int dummy = 1;
        return dummy;
    }
}
```

AddShareRequest

```
package netcom.mds;


/**
 * <p>Title: Add Share Request</p>
 * <p>Description: This class is a Request to handle the adding
```

```
 * of shared resources by PServers.</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class AddShareRequest extends AbstractRequest
{
    public final AbstractShare new_share = null;

    public AddShareRequest()
    { super(); }
}
```

AddShareTask

```
package netcom.mds;

import netcom.util.*;
import java.util.*;

/**
 * <p>Title: Add Share Task</p>
 * <p>Description: This class performs the task of adding one share to
 * the local MDS node.  It is a Runnable class meant to be run by a ThreadPool.
 * It is responsible for updating the internal database, updating the internal
 * local bloom filter, and sending BloomUpdate packets to all other MDS nodes
 * on the system</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class AddShareTask implements Runnable
{
   /**
    * Constructor
    */
    public AddShareTask(AddShareRequest new_share, Vector all_psdatas, BloomFilter local_filter)
    {}

  /**
   * run() needed to implement Runnable and is where this task begins execution
   */
    public void run()
    {}

   /**
    * This methods adds a share to some PSData object in the MDS node
    * (all_psdatas given to create this object)
    */
    private void addToPS(AbstractShare new_share)
    {}

   /**
    * Method to update the local BloomFilter with the new added share. Destructively
    * modifies the Bloomfilter and return the BloomUpdate signifying the change
    */
    private BloomUpdate updateBloom(AbstractShare new_share, BloomFilter local_filter)
    {
        BloomUpdate dummy = new BloomUpdate(new int[] { 0 });
        return dummy;
    }

   /**
    * Method to send a BloomUpdate to all other MDS Nodes by contacting LDAP for a listing
    */
    private void sendUpdateToAll(BloomUpdate update)
```

```
    {}

   /**
    * Method to send a BloomUpdate which represents a complete BloomFilter to all nodes
    * This complete bloom filter is sent after N many updates to compensate for the error
    * of normal BloomUpdates
    */
    private void sendCompleteUpdate(BloomFilter local_filter)
    {}
}
```

BloomFilter

```
package netcom.util;

import java.util.BitSet;
import java.util.StringTokenizer;
import java.util.Properties;
import java.util.ArrayList;
import java.util.List;

//NOTE: This bloom filter is meant to work with an older proof of concept version
// and may not be up to speed with the design implementation or be working with
// the current other classes

/**
 * <p>Title: Bloom Filter</p>
 * <p>Description: A Bloom filter is basically a bit array which has its bits
 * modified from a one-way hash function of an input string.  (Each string is hashed
 * to 5 different indexes.)  This allows the filter to tell whether certain string
 * 'keys' have been entered into it or not with a very low memory cost. It also allows
 * for a very small false-positive ratio, see
 * <a href=http://www.cs.wisc.edu/~cao/papers/summary-cache/node8.html>this</a> (where k=5)
 * for more details on the false-positive ratio</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@cs.columbia.edu>
 * @version 0.1
 * @see MD5
 * @see BloomUpdate
 */
public class BloomFilter {

   /**
    * This is the number of hash functions used for every key
    * It should always be 5
    * (any changes would require modifying the algorithm, so this is not a
    * properties-loaded constant)
    */
   public static final int HASH_FUNCTIONS = 5;

   /**
    * The table size is critical to the false-positive ratio of the filter
    * The size can be loaded from a file under the key "BLOOMFILTER_TABLE_SIZE"
    * but if that is not supplied, this default is used
    * Ideally we want a size-to-keys ratio greater than 10-1 which results in less
    * than 1% false-positives, however, even a 7-1 ratio still only gives 3% false positives
    * see <a href=http://www.cs.wisc.edu/~cao/papers/summary-cache/node8.html>this</a> (k=5)
    * for exact numbers
    *
    * As a sidenote, a rehash upon size flucuations might be considered, however this
    * would require a separate storage of all origional indexes (before modulus operations)
    * and thus would use much more memory than simply making a larger table
    *
    * Also, this must be constant across ALL MDS nodes for proper hasing, so cannot make this
    * a netcom.util.Properties loaded constant
    */
   private final static int DEFAULT_TABLE_SIZE = 50000;
```

```
/**
 * private BitSet, holds the array of bits which make up the filter
 */
private final BitSet the_bits;

/**
 * private MD5 object, only ever need one, so final
 */
private final MD5 md5_generator;

/**
 * This is the simulated array length for the filter
 */
private final int table_size;

/**
 * This is the number of keys currently in the BitSet
 */
private int key_count;

/**
 * The constructor for a BloomFilter assuming no Properties object
 */
public BloomFilter() {
   this(null);
}

/**
 * The constructor for a BloomFilter only takes 1 argument of a Properties object
 * which holds the value for the filter's size
 *
 * @param defaults  Properties object with default values, can be null
 */
public BloomFilter(Properties defaults) {
   md5_generator = new MD5();
   md5_generator.Init();
   the_bits = new BitSet();
   key_count = 0;

   // messy, but we must search for a BLOOMFILTER_TABLE_SIZE key and handle
   // any possible errors
   // we also want a 'final' table_size for the many benefits, but for the compiler
   // to accept this, we must introduce a temporary variable
   int temp_size = 0;
   if (defaults != null) {
      String table_size_str = defaults.getProperty("BLOOMFILTER_TABLE_SIZE");
      if (table_size_str != null) {
         try {
            temp_size = Integer.parseInt(table_size_str);
         } catch (NumberFormatException ne) {
            temp_size = DEFAULT_TABLE_SIZE;
         }
      } else {
         temp_size = DEFAULT_TABLE_SIZE;
      }
   } else {
      temp_size = DEFAULT_TABLE_SIZE; //default size
   }
   table_size = temp_size;
} //end constructor

/**
 * Getter method to return the number of keys currently held in the filter
 *
 * @return the number of keys in the filter
 */
public int getSize() {
   return this.key_count;
}
```

```
/**
 * This method will parse an input string, and add every 'word' to
 * the bloom filter, separating 'words' by <space>
 *
 * @param keys  String with all words wished to be added
 *
 * @return a BloomUpdate object signifying the changes made to the BitSet
 */
public BloomUpdate addFullString(String keys) {

   keys.trim(); //cleanup input
   BitSet old_set = (BitSet)this.the_bits.clone(); //must save old bitset

   StringTokenizer tok = new StringTokenizer(keys, " ");
   while (tok.hasMoreTokens()) {
      // here we duplicate the code from addOneKey() but do not make the actual
      // call itself because that would add unnecessary and high-overhead
      // processing for creating a BloomUpdate
      int[] indexes = getHashIndexes(tok.nextToken());

      //now for every index, set that bit in the BitSet to a '1'
      for (int i=0; i < HASH_FUNCTIONS; i++) {
         the_bits.set((indexes[i] % table_size), true);
      } //endfor

      key_count++;
   } //end while

   return getBloomUpdate(old_set, this.the_bits);
} //end addFullString()

/**
 * This method will add a single String to the bloom filter,
 * whether spaces exist in it or not
 *
 * @param key  String with or without spaces to be inserted as a single key
 *
 * @return A BloomUpdate object signifying the changes made to the BitSet
 */
public BloomUpdate addOneKey(String key) {

   key.trim(); //cleanup input
   int[] indexes = getHashIndexes(key);
   BitSet old_set = (BitSet)this.the_bits.clone(); //must save old bitset

   //now for every index, set that bit in the BitSet to a '1'
   for (int i=0; i < HASH_FUNCTIONS; i++) {
      the_bits.set((indexes[i] % table_size), true);
   } //endfor

   key_count++;

   return getBloomUpdate(old_set, this.the_bits);
} //end addOneKey()


/**
 * This method checks to see if 'all' of the 'words' in a string are
 * in the bloom filter, separating 'words' by <space>
 *
 * @param keys  String with all words wished to be checked
 *
 * @return A Boolean value of whether 'all' of the words are located in the filter
 */
public boolean checkAllWords(String keys) {

   keys.trim();  //cleanup input
   StringTokenizer tok = new StringTokenizer(keys, " ");

   // for every word, check if that key is in the filter
   while (tok.hasMoreTokens()) {
```

```
      boolean this_key = checkOneKey(tok.nextToken());
      if (!this_key) // if ANY of the words returns false, return false
         return false;
   } //end while

   // if it reached here, every word had to be in the filter, so return true
   return true;
} //end checkAllWords()

/**
 * This method checks to see if an individual key is located in the bloom filter
 *
 * @param key  String with exact key to check in the filter
 *
 * @return A Boolean value of whether the key was in the filter or not
 */
public boolean checkOneKey(String key) {

   key.trim(); //cleanup input
   int[] indexes = getHashIndexes(key);

   //now must ensure that all 5 hash functions match, otherwise its a miss
   for (int i=0; i < HASH_FUNCTIONS; i++) {
      boolean this_index = the_bits.get(indexes[i] % table_size);
      if (!this_index)  //if ANY of the indexes is false, return false
         return false;
   } //endfor

   // if it reached here, every index had to be true, so return true
   return true;
} //end checkOneKey()

/**
 * This method updates this bloom filter according to a supplied BloomUpdate object
 * (ie flips the appropriate bits)
 *
 * @param update_obj  BloomUpdate object representing changes to the filter
 */
public void updateFilter(BloomUpdate update_obj) {
   /*
   List changed_indexes = update_obj.getIndexList();

   for (int i = changed_indexes.size()-1; i >= 0; i--) {
      int index = ((Integer)changed_indexes.get(i)).intValue(); //should always hold Integers
      this.the_bits.flip(index);
   }
   */
} //end updateFilter()

/**
 * This method will return 5 hash numbers from a single input
 * string using the MD5 hash as a base.  These output numbers
 * must then be mod'd to the appropriate table size
 *
 * @param inputKey  the String to be hashed
 *
 * @return int array of size 5 filled with hash numbers
 */
private int[] getHashIndexes(String input_key) {

   final int[] ret_keys = new int[HASH_FUNCTIONS];

   md5_generator.Init();      //clear generator for new input
   md5_generator.Update(input_key);  //plug in string to be hashed
   byte[] ret_bytes = md5_generator.Final();  //perform hash op and return byte array

   // here we form the 5 hash values from the 128bit MD5 value by taking
   // nonoverlapping 3-byte slices, the choice of which/where are arbitrary
   // this method has been shown to form 5 'decent' independent hashes
   // from experimentation
   // each digit represents a 2^8 place, so must multiply by 256 for every place holder
```

```
      ret_keys[0] = ret_bytes[0] + (ret_bytes[1] * 256) + (ret_bytes[2] * 65536);
      ret_keys[1] = ret_bytes[3] + (ret_bytes[4] * 256) + (ret_bytes[5] * 65536);
      ret_keys[2] = ret_bytes[6] + (ret_bytes[7] * 256) + (ret_bytes[8] * 65536);
      ret_keys[3] = ret_bytes[10] + (ret_bytes[11] * 256) + (ret_bytes[12] * 65536);
      ret_keys[4] = ret_bytes[13] + (ret_bytes[14] * 256) + (ret_bytes[15] * 65536);

      // we never want negative hash indexes, so do simple check on every int
      // and multiply by -1 if is negative
      for (int i=0; i< ret_keys.length; i++) {
         if (ret_keys[i] < 0) { ret_keys[i] *= -1; }
      } //endfor

      return ret_keys;
   } //end getHashIndexes()

   /**
    * This method creates a BloomUpdate object representing the 'difference' between the
    * two input BitSets
    *
    * @param old_set  The old bitset
    * @param new_set  The new bitset with modifications
    *
    * @return A BloomUpdate object representing which indexes have been changes from old to new
    */
   private BloomUpdate getBloomUpdate(BitSet old_set, BitSet new_set) {

      BloomUpdate b_upd = new BloomUpdate(new int[] { 0 });

      //want to clone the object so we don't destroy the origional bitset
      BitSet delta_set = (BitSet)old_set.clone(); //we know this is always a BitSet
      delta_set.xor(new_set); //delta_set now has all bits which were different

               ArrayList indexes = new ArrayList();

      // march through and add all set bits to the BloomUpdate object
      for (int i = delta_set.nextSetBit(0); i >= 0; i = delta_set.nextSetBit(i+1))
                     indexes.add(new Integer(i));

               //b_upd.addIndex(i);

      return b_upd;
   } //end getBloomUpdate()

} //end class
```

BloomUpdate

```
package netcom.util;

import java.util.ArrayList;
import java.util.List;
import java.util.Collections;
import java.net.DatagramPacket;

//NOTE: This class was part of a proof of concept and may not be completely
// synched with the current design

/**
 * <p>Title: Bloom Update Packet</p>
 * <p>Description: This object holds an index list of bits which
 * should be flipped from an origional BloomFilter</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@cs.columbia.edu>
 * @version 0.1
 * @see BloomFilter
 */
public class BloomUpdate {
```

```
   /**
    * This is a non-thread-safe ArrayList that 'Always' should contain 'Integers'
    */
   private int[] flipped_indexes;

   /**
    * Getter to actually get the Datagram Packet representation of this BloomUpdate
    */
   public DatagramPacket getDatagram() {
      return new DatagramPacket(new byte[] { 0},0);
   }

   public BloomUpdate(int[] changed_indexes) {

      flipped_indexes = changed_indexes;
   } //end constructor

   /**
    * This method adds one integer index to the ArrayList of all indexes which
    * have been modified
    *
    * @param mod_idx  The modified index number
    */
   public void addIndex(int mod_idx) {
      //flipped_indexes.add(new Integer(mod_idx));
   }

   /**
    * Getter method to retrieve the ArrayList of flipped indexes
    *
    * @return List of flipped indexes (unmodifiable)
    */
   public int[] getIndexList() {
      //we dont want others having an actual reference to the ArrayList
      //which they could then use to modify, so make it unmodifiable
      return flipped_indexes; //Collections.unmodifiableList(this.flipped_indexes);
   }


} //end class
```

ChatShare

```
package netcom.mds;

/**
 * <p>Title: Chat Share</p>
 * <p>Description: This represents a share by a PServer of a
 * Chat resource</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 * @see FileShare
 */
public class ChatShare extends AbstractShare
{

    public ChatShare()
    {}

   /**
    * This compareTo overwrites that in AbstractShare to give a more meaningful
    * compare for a 'chat' resource
    */
    public int compareTo(Object o)
    {
        int dummy = 1;
```

```
        return dummy;
    }

}
```

FileShare

```
package netcom.mds;

/**
 * <p>Title: File Share</p>
 * <p>Description: This represents a shared File resource by a PServer</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 * @see ChatShare
 */
public class FileShare
{
    protected final String size;

    public FileShare()
    {
        size = "dummy";
    }

   /**
    * This compareTo ovewrites that in AbstractShare to give a more meaningful
    * compare for a 'File' resource
    */
    public int compareTo(Object o)
    {
        int dummy = 1;
        return dummy;
    }

}
```

HandleBloomUpdates

```
package netcom.mds;

import java.net.*;
import java.util.*;
import netcom.util.*;

/**
 * <p>Title: Handle Bloom Updates</p>
 * <p>Description: This class creates a datagram server socket
 * which waits for BloomUpdate packets and then updates the local image
 * of all other MDS nodes' bloom filter.</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class HandleBloomUpdates extends Thread
{
    private DatagramSocket server_port;
    public Vector bloom_filters_ptr;

   /**
    * Constructor
    */
```

```
    public HandleBloomUpdates (Vector boolm_filters)
    {}

   /**
    * Method where this thread begins execution
    */
    public void run()
    {}

   /**
    * Method which loops, simply waiting for incoming packets on the datagram
    * server socket and then processes them.
    */
    private void waitForPacket()
    {}

   /**
    * Method to alter the appropriate local filter image according to a BloomUpdate
    *
    * @param update_packet the BloomUpdate information
    * @param orig_mds_node the InetSocketAddress which serves to index the appropriate
    *                      bloomfilter in the hashtable
    */
    private void alterFilter(BloomUpdate update_packet, InetSocketAddress orig_mds_node)
    {}
}
```

HandleMDSConnections

```
package netcom.proxy;

import java.util.*;
import netcom.util.*;

/**
 * <p>Title: Handle MDS Connections</p>
 * <p>Description: This class handles any new connections coming from the
 * MDSCloud, which should only ever be Search returns</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author Charles O'Donnell <cwo4@cs.columbia.edu>
 * @version 1.0
 */

public class HandleMDSConnections extends Thread {

   public HandleMDSConnections(Vector incoming_sockets, ThreadPool thread_pool) {
   }

   /**
    * Method to begin execution
    */
   public void run() {

   }

   /**
    * Method to wait for connections and then process them
    */
   public void waitForConnection() {
   }
}
```

HandleMDSSockets

```
package netcom.proxy;

import java.util.*;
```

```
import netcom.util.*;

/**
 * <p>Title: Handle MDS Sockets</p>
 * <p>Description: This class polls all sockets which face inward
 * towards the cloud and creates tasks from any data recieved</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author Charles O'Donnell <cwo4@cs.columbia.edu>
 * @version 1.0
 */
public class HandleMDSSockets extends Thread {

   public HandleMDSSockets(Hashtable incoming_sockets, Vector outgoing_sockets, ThreadPool
thread_pool) {
   }

  /**
   * begins the execution of this thread
   */
   public void run() {
   }

   /**
    * Loops through, polling all sockets for data, processes data into a task
    */
   public void readSockets() {
   }
}
```

HandleOutsideConnections

```
package netcom.proxy;

import java.util.*;
import netcom.util.*;

/**
 * <p>Title: Handle Outside Connections</p>
 * <p>Description: This class handles outside connections to the MDSProxy, such
 * as those from PServers and PClients, and adds the socket to the list of
 * sockets to be polled for data</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author Charles O'Donnell <cwo4@cs.columbia.edu>
 * @version 1.0
 */
public class HandleOutsideConnections extends Thread {

   public HandleOutsideConnections(Hashtable incoming_sockets, Vector outgoing_sockets,
ThreadPool thread_pool) {
   }

   /**
    * Method to begin execution
    */
   public void run() {

   }

   /**
    * Method to wait for connections and then process them
    */
   public void waitForConnection() {
   }
}
```

HandleOutsideSockets

```
package netcom.proxy;

import java.util.*;
import netcom.util.*;

/**
 * <p>Title: Handle Outside Sockets</p>
 * <p>Description: This class loops through and handles and data coming in
 * on the sockets facing away from the MDSCloud (PClients and PServers)
 * and processes that data into Tasks</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author Charles O'Donnell <cwo4@cs.columbia.edu>
 * @version 1.0
 */

public class HandleOutsideSockets extends Thread {

   public HandleOutsideSockets(Hashtable incoming_sockets, Vector outgoing_sockets, ThreadPool
thread_pool) {
   }

   /**
   * begins the execution of this thread
   */
   public void run() {
   }

   /**
    * Loops through, polling all sockets for data, processes data into a task
    */
   public void readSockets() {
   }
}
```

HandleReturnTask

```
package netcom.proxy;

import java.util.*;
import java.net.*;
import netcom.mds.*;

/**
 * <p>Title: Handle Return Task</p>
 * <p>Description: Class which is a task to handle returning requests
 * and get them back to the PClient/etc</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author Charles O'Donnell <cwo4@cs.columbia.edu>
 * @version 1.0
 */
public class HandleReturnTask implements Runnable {

   public HandleReturnTask(AbstractRequest back_request, Vector incoming_sockets) {
   }

   public void run() {
   }

  /**
   * Method to parse a returned request back to XML a PClient can understand, then
   * sends it on with sendReturn()
   */
   public void parseToXML(AbstractRequest back_request) {
   }
```

```
   /**
    * Method to send an XML string to an originating socket
    */
   public void sendReturn(String xml_string, Socket orig_socket){
   }
}
```

HandleSocketConnects

```
package netcom.mds;

import java.net.*;
import java.util.*;

/**
 * <p>Title: Handle Socket Connections</p>
 * <p>Description: This class simply listens to a server port and accepts
 * incoming connections to the MDS node. Adding them to a 'core' Hashtable
 * of connected sockets according to their originating key in a Request that
 * must be sent immediately</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class HandleSocketConnects extends Thread
{
    private ServerSocket server_port;
    public Vector friends_sockets_ptr;

   /**
    * Constructor
    */
    public HandleSocketConnects (Hashtable connections_sockets)
    {}

   /**
    * This Method is where this thread begins execution
    */
    public void run()
    {}

   /**
    * Method which loops, waiting for connections and then adding them
    */
    private void waitForConnection()
    {}

   /**
    * Method which adds a Socket to the connections_sockets Hashtable
    */
    private void addSocket(Socket new_socket, Hashtable connections_sockets)
    {}
}
```

HandleXMLTask

```
package netcom.proxy;

import java.util.*;
import java.net.*;
import java.io.*;
import netcom.mds.*;

/**
 * <p>Title: Handle XML Task</p>
 * <p>Description: Class which is a task to handle the parsing of XML into
```

```
 * Requests, doing authentication, and sending Requests along their way</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author Charles O'Donnell <cwo4@cs.columbia.edu>
 * @version 1.0
 */
public class HandleXMLTask implements Runnable {

   public HandleXMLTask(Hashtable outgoing_sockets, Socket incoming_socket) {
   }

   public void run() {
   }

   /**
    * Major method to parse an XML stream (socket) and then call the appropriate
    * other functions to handle the XML message
    */
   public void parseXML(InputStream xml_stream) {
   }

   /**
    * Method to check the authorization of a login and password
    */
   public boolean checkAuth(String name, String password) {
      return true;
   }

   /**
    * Method to talk with LDAP and add a new user
    */
   public void addNewUser(String name, String password) {
   }

   /**
    * Method to handle a logout XML message
    */
   public void logout(Socket in_socket) {

   }

   /**
    * Method to find the best MDS according to PClient load-balancing criteria
    */
   public InetSocketAddress getBestPClientMDS() {
      return null;
   }

   /**
    * Method to find the best MDS according to PServer load-balancing criteria
    */
   public InetSocketAddress getBestPServerMDS() {
      return null;
   }

   /**
    * method to create a new connection to a MDS node for a PServer
    */
   public void addPServerPipe(Socket in_socket) {
   }

   /**
    * Method to send a PServer request to a MDS node
    */
   public void sendPServerRequest(AbstractRequest new_request) {
   }

   /**
    * Method to send and Error back to an originating socket (PClient or PServer)
    */
   public void sendError(String xml_error) {
```

```
   }

   /**
    * Method to send a search request to a MDS node, ie open a socket, send, then drop it
    */
   public void sendSearchRequest(SearchRequest search_request) {
   }
}
```

MD5

```
// long GPL source…
```

MDS

```
package netcom.mds;

import netcom.util.*;
import java.util.*;
import java.io.*;
import java.net.*;

/**
 * <p>Title: MDS</p>
 * <p>Description: This is the central MDS node which holds
 * all of the 'core' data structures which other threads modify. It
 * also checks all sockets for incoming request and sends them off to
 * be processed</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class MDS
{
    private netcom.util.ThreadPool search_pool;
    private netcom.util.ThreadPool share_pool;
    // these data structures are shared by many other threads and objects
    // and consequently will be synchronized or otherwise protected
    private RequestPathTracker path_tracker;
    private Hashtable connection_sockets;
    private Vector friends_Sockets;
    private Hashtable bloom_filters;
    private Vector all_psdatas;
    private BloomFilter local_filter;
    //private LogServer logger;
    private int shares_mod_count;

    public MDS()
    {}

    public void run()
    {}

   /**
    * Loops through, polling every socket for any input,
    * If a request is found it is sent to a new task to be
    * completed by a threadpool
    */
    private void readSockets()
    {}

   /**
    * Method to handle internal data structures when a
    * socket is found to be dead
    */
    private void handleDeadSocket(Socket dead_socket)
```

```
    {}

   /**
    * Method to remove all data associated with a PServer that has
    * been found to be logged out or dead
    */
    private void removePSData(PSData ps_data)
    {}
}
```

MDSCLI

```
package netcom.mds;

/**
 * <p>Title: MDS CLI</p>
 * <p>Description: This Class starts the MDS program and offers
 * a Command Line Interface for global settings to be changed
 * on the fly</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class MDSCLI
{
   /**
    * Starts up MDS objects and runs the CLI
    */
    public static void main(String args[])
    {}

   /**
    * Loops, waiting for user-input and handling it
    */
    private void runCLI()
    {}
}
```

MDSProxy

```
package netcom.proxy;

/**
 * <p>Title: MDS Proxy</p>
 * <p>Description: This is the main MDS proxy class which holds 'core' data
 * structures and hold all threads checking up on ports and processing data</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: </p>
 * @author Charles O'Donnell <cwo4@cs.columbia.edu>
 * @version 1.0
 */
public class MDSProxy {

   public MDSProxy() {
   }
   /**
    * Main method, starts program
    */
   public static void main(String[] args) {
      MDSProxy MDSProxy1 = new MDSProxy();
   }
}
```

PSData

```
package netcom.mds;

import java.util.*;

/**
 * <p>Title: PSData</p>
 * <p>Description: This class represents all the data being held by one
 * PServer on a MDS node
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class PSData
{
    public final String ps_name = null;
    private TreeSet file_shares;
    private TreeSet chat_shares;

    public PSData()
    {}

   /**
    * Adds a share to the PSData inventory
    */
    public boolean addShare(AbstractShare share)
    {
        boolean dummy = true;
        return dummy;
    }

   /**
    * Removes a share from the PSData inventory
    */
    public boolean removeShare(AbstractShare share)
    {
        boolean dummy = true;
        return dummy;
    }

   /**
    * Method to search all share for a PSData object according to
    * a SearchRequest. Returns an array of all shares found or an
    * array of zero if none found
    */
    public AbstractShare[] contains (SearchRequest search)
    {
        AbstractShare[] dummy = new AbstractShare[10];
        return dummy;
    }
}
```

Queue

```
package netcom.util;
// This source from http://www.panix.com/~mito/articles/articles/threadpool/source/

import java.util.*;

public class Queue
{
  private Vector vec = new Vector();

  synchronized public void put( Object o ) {
    // Add the element
    vec.addElement( o );
```

```
    // There might be threads waiting for the new object --
    // give them a chance to get it
    notifyAll();
  }

  synchronized public Object get() {
    while (true) {
      if (vec.size()>0) {
        // There's an available object!
        Object o = vec.elementAt( 0 );

        // Remove it from our internal list, so someone else
        // doesn't get it.
        vec.removeElementAt( 0 );

        // Return the object
        return o;
      } else {
        // There aren't any objects available.  Do a wait(),
        // and when we wake up, check again to see if there
        // are any.
        try { wait(); } catch( InterruptedException ie ) {}
      }
    }
  }
}
```

RemoveShareRequest

```
package netcom.mds;


/**
 * <p>Title: Remove Share Request</p>
 * <p>Description: This is a Request to handle a PServer requesting to remove
 * a share from a MDS node</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class RemoveShareRequest extends AbstractRequest
{
    public final AbstractShare current_share = null;
    public RemoveShareRequest()
    { super(); }
}
```

RemoveShareTask

```
package netcom.mds;

import netcom.util.*;
import java.util.*;

/**
 * <p>Title: Remove Share Task</p>
 * <p>Description: This is a task which handles removing one share from some
 * PSData object contained in this MDS node, then rebuilding a new BloomFilter,
 * and sending out a BloomUpdate to all other MDS nodes</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
```

```
 */
public class RemoveShareTask implements Runnable
{
   /**
    * Constructor:
    */
    public RemoveShareTask(RemoveShareRequest new_share, Vector all_pservers, BloomFilter
local_filter)
    {}

  /**
   * Method which begins execution of this task
   */
    public void run()
    {}

   /**
    * Method which removes a share from a specific PSData
    */
    private void removeShare(AbstractShare removed_share, PSData ps_data)
    {}

   /**
    * Time-intensive method to rebuild a BloomFilter from scratch with all
    * PSDatas on the MDS node. This is necessary because theoretically Bloom Filters
    * cannot handle removes
    */
    private BloomFilter rebuildBloom(Vector all_psdatas)
    {
        BloomFilter dummy = new BloomFilter();
        return dummy;
    }

   /**
    * Method to send a BloomUpdate to all other MDS nodes showing the difference from the
    * origional BloomFilter to the rebuilt BloomFilter
    */
    private void sendUpdateToAll(BloomUpdate update)
    {}

   /**
    * Method to send a BloomUpdate representing an entire BloomFilter to all MDS nodes.
    * Occurrs every N many modifications to take for of unreliability of normal
    * BloomUpdates
    */
    private void sendCompleteUpdate(BloomFilter local_filter)
    {}
}
```

RequestPathTracker

```
package netcom.mds;

import netcom.util.*;
import java.util.*;

/**
 * <p>Title: Request Path Tracker</p>
 * <p>Description: This class keeps track of all incoming Requests and
 * logs their unique identifiers to prevent them from ever being processed
 * twice by the same node.
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class RequestPathTracker
```

```
{
    public static final int MESSAGE_COUNT_ROLLOVER = 2048;
    private Hashtable nodes_lookup;

    public RequestPathTracker()
    {}

  /**
   * Method to decide if a SendRequest has been seen before
   * if not, then it will be entered as being seen for next time
   *
   * @param unkown_search  the request to be checked
   *
   * @return true if the request has been seen, false if it has not
   */
    public boolean decideKill(SearchRequest unknown_search)
    {
        boolean dummy = true;
        return dummy;
    }

   /**
    * Method to decide if a ReturnRequest has been seen before
    * if not, then it will be entered as being seen for the next time
    *
    * @param uknown_search  the return to be checked
    *
    * @return true if the request has been seen, false if it has not
    */
    public boolean decideKill(ReturnRequest unknown_search)
    {
        boolean dummy = true;
        return dummy;
    }
}
```

ReturnRequest

```
package netcom.mds;

import java.net.*;

/**
 * <p>Title: Return Request</p>
 * <p>Description: This represents a return of matching information following
 * a search.</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class ReturnRequest extends AbstractRequest
{
    public final InetSocketAddress originator = null;
    public final InetSocketAddress search_mds_originator = null;
    public final AbstractShare return_share = null;

    public ReturnRequest()
    { super(); }
}
```

ReturnTask

```
package netcom.mds;

import java.util.*;
import netcom.util.*;
```

```
/**
 * <p>Title: Return Task</p>
 * <p>Description: This is a task which handles ReturnRequest objects, checking to see
 * if they're reached their destination, and propagating if they have not</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class ReturnTask implements Runnable
{
    public ReturnTask(ReturnRequest return_request, Vector friends)
    {}

   /**
    * Method where this task begins execution
    */
    public void run()
    {}

   /**
    * Method to return a result to a MDSProxy if the ReturnRequest
    * has reached its originating MDS node
    */
    private void returnResult(ReturnRequest return_request)
    {}

   /**
    * Method to propogate a ReturnRequest to all friends of the
    * MDS Node
    */
    private void propagateResult(ReturnRequest return_request)
    {}

   /**
    * Method to refill the Vector of friend node when it drops below
    * the threshold
    */
    private void refillFriends(Vector friends)
    {}
}
```

SearchRequest

```
package netcom.mds;

import java.util.*;
import java.net.*;

/**
 * <p>Title: Search Request</p>
 * <p>Description: This class represents a search request for a search
 * initiated by a PClient attempting to find a match with name list pairs
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class SearchRequest extends AbstractRequest
{
    public final InetSocketAddress mds_originator;
    public boolean is_bloom_guess = false;
    public final LinkedList[] name_list_pairs;
    public final InetSocketAddress mdsproxy_originator;

    public SearchRequest() {
      super();
```

```
      mds_originator = null;
      name_list_pairs = null;
      mdsproxy_originator = null;
    }

   /**
    * Converts the request to a useful string
    */
    public String toString()
    {
        String dummy = "dummy";
        return dummy;
    }

   /**
    * Converts the request to a string which is useful for a BloomFilter
    * to match against
    */
    public String getBloomString()
    {
        String dummy = "dummy";
        return dummy;
    }
}
```

SearchTask

```
package netcom.mds;

import netcom.util.*;
import java.util.*;
import java.net.*;

/**
 * <p>Title: Search Task</p>
 * <p>Description: This class is a task which handles Search requests entering an
 * MDS node. It performs a local lookup of the Search criteria, a Bloom Check if
 * in Bloom Search Mode, and propogates the SearchRequest</p>
 * <p>Copyright: Copyright (c) 2002</p>
 * <p>Company: NETCOM</p>
 * @author Charles O'Donnell <cwo4@columbia.edu>
 * @author Omar Siddiqui <oss8@columbia.edu>
 * @version 0.1
 */
public class SearchTask implements Runnable
{
   /**
    * Constructor:
    */
    public SearchTask(SearchRequest search_request, Vector friends_sockets, Vector all_psdatas,
RequestPathTracker path_tracker)
    {}

   /**
    * Task begins execution  in this method
    */
    public void run()
    {}

   /**
    * Method to check ALL PSData objects for a match
    */
    private void checkAllPSs(SearchRequest search_request, Vector all_psdatas)
    {}

   /**
    * Method to compare search against all local bloom filters of other nodes
    * returns an array of InetSocketAddresses signifying matches
    */
    private InetSocketAddress[] checkAllBlooms(String search_param, Vector bloom_filters)
```

```
    {
        return (InetSocketAddress[])null;
    }

   /**
    * Method to check an individual bloom filter for a match
    */
    private boolean checkBloom(String search_param, BloomFilter input_filter)
    {
        boolean dummy = true;
        return dummy;
    }

   /**
    * Method propagates a search to all of this MDS Node's friends or to bloom matches
    */
    private void propagateSearch(Vector friends)
    {}

   /**
    * Method to send a search return to the MDSProxy originating the search
    */
    private void sendReturn(ReturnRequest orig_search)
    {}

   /**
    * Method to refill the local friends vector when the friends count goes below a certain
    * threshold
    */
    private void refillFriends(Vector friends)
    {}
}
```

ThreadPool

```
package netcom.util;
// This source from http://www.panix.com/~mito/articles/articles/threadpool/source/

public class ThreadPool
{
  private Queue queue = new Queue();
  private int numThreads;

  public ThreadPool( int numThreads ) {
    this.numThreads = numThreads;

    // Create N threads
    for (int i=0; i<numThreads; ++i)
      new ThreadPoolThread( this );
  }

  public void runRunnable( Runnable runnable ) {
    queue.put( runnable );
  }

  Runnable getRunnable() {
    return (Runnable)queue.get();
  }
}
```

ThreadPoolThread

```
package netcom.util;
// This source from http://www.panix.com/~mito/articles/articles/threadpool/source/

public class ThreadPoolThread implements Runnable
```

```
{
  private Thread thread;
  private ThreadPool tp;
  private boolean active = false;

  ThreadPoolThread( ThreadPool tp ) {
    this.tp = tp;

    thread = new Thread( this, "TPT-"+this );
    thread.start();
  }

  public void run() {
    while (true) {
      try {
        // Get the next task from the parent ThreadPool
       Runnable r = tp.getRunnable();

        // Run the task!
        active = true;
       r.run();
        active = false;
      } catch( Exception e ) {
        e.printStackTrace();
      }
    }
  }

  public boolean active() { return active; }
}
```

AuthenticationEntry

```
/*
 * AuthenticationEntry.java
 *
 * Created on April 8, 2002, 8:03 AM
 */

package netcom.util;

/**
 * A data type which contains login information
 * Consists of a username and a password
 *
 */
public class AuthenticationEntry {

    /** Creates new AuthenticationEntry */
    public AuthenticationEntry(String username,String password) {
    }
    public String getUsername(){return null;}
    public String getPassword(){return null;}

}
```

DataPanel

```
/*
 * DataPanel.java
 *
 * Created on April 8, 2002, 8:31 AM
 */

package netcom.gui;

/**
 * Visually represents the statistical and log data
```

```
 * Multiple DataPanels may be added to the gui by using
 * TabPanel
 *
 * Two ways to represent data:
 *
 * 1)Grid
 * 2)Graph
 */
public class DataPanel {

    /** Creates new DataPanel */
    public DataPanel() {
    }

}
```

FileReader

```
/*
 * FileReader.java
 *
 * Created on April 8, 2002, 9:33 AM
 */

package netcom.log;

/**
 *
 * Reads entries from a log file
 * parses them and puts onout_queue as LogMessage
 * There are 2 threads running: FileReader and FileWriter
 * Wait/Notify algorithm makes sure that only one thread runs at a time
 */
public class FileReader {

    /** Creates new FileReader */
    public FileReader() {
    }
public void run(){}
}
```

FileWriter

```
/*
 * FileWriter.java
 *
 * Created on April 8, 2002, 9:34 AM
 */

package netcom.log;

/**
 * Dumps log entries in a file
 * Takes entries from a in_queue and writs them as text to a log file
 * There are 2 threads running: FileReader and FileWriter
 * Wait/Notify algorithm makes sure that only one thread runs at a time
 */
public class FileWriter {

    /** Creates new FileWriter */
    public FileWriter() {
    }

}
```

GUIClient

```
/*
 * GUIClient.java
 *
 * Created on April 8, 2002, 8:33 AM
 */

package netcom.gui;

/**
 * Handles the communication with a particular MDSNode
 * Communicates via LogMessages.
 * Makes sure that GUI has the up-to-date information about a MDS_Node
 */
public class GUIClient {

    /** Creates new GUIClient */
    public GUIClient() {
    }
    /**
     * The main methods that creates LogMessages and sends them to MDSNodes
     * Receives information from MDSNode, parses it, and saves it to be used
     * by other visual components when needed
     */

    public void run(){}
}
```

LDAPCommunicator

```
/*
 * LDAPCommunicator.java
 *
 * Created on April 8, 2002, 8:12 AM
 */

package netcom.util;

/**
 *  LDAPCommunicator class is responsible for the communication with an LDAP server.
 *  It handles two main data types.
 *  Authentication entries represent the login/logout information for
 *  Pcilents and Pservers. The are validated against the LDAP server database
 *  Node entries contain the information about available MDS and MDSProxies
 */

public class LDAPCommunicator {

    /** Creates new LDAPCommunicator */
    public LDAPCommunicator() {}
    /**
     * Adds new username/password pair if not already exists
     */
    public void addAuthenticationEntry(AuthenticationEntry entry){}

    /**
     * Removes username/password pair
     */
    public void removeAuthenticationEntry(AuthenticationEntry entry){}

    /**
     *  returns list of all authentication entries
     */

    public AuthenticationEntry[] getAuthenticationList(){return null;}

    /**
     * checks if username/password are valid
     */
```

```
    public boolean isAuthenticated(AuthenticationEntry entry){return true;}
    /**
     * Adds new Node to LDAP
     */

    public void addNodeEntry(NodeEntry entry){}

    /**
     * Removes a Node from LDAP
     */

    public void removeNodeEntry(NodeEntry entry){}

    /**
     * updates Node information
     */

    public void updateNodeEntry(NodeEntry entry){}

    /**
     * return all nodes depending on the equested type
     */
    public nodeEntry[] getNodeList(int type){return null;}

    /**
     * returns a NodeEntry with a specified node_name/id
     */

    public nodeEntry getNodeEntry(String node_name){return null;}

}
```

Logable

```
/*
 * Logable.java
 *
 * Created on April 8, 2002, 9:20 AM
 */

package netcom.log;

/**
 *
 *  interface that is implemented by Logger
 */
public interface Logable {

    public void addLogEntry(LogEntry entry);
    public LogEntry[] getLogEntries(int from, int to);
    public int getMaxLogEntryNumber() ;
    public int  getSessionNumber();

}
```

LogEntry

```
/*
 * LogEntry.java
 *
 * Created on April 8, 2002, 9:21 AM
 */

package netcom.log;

/**
 * A data type that contains all the information
 * about a log entry. Provides accesor methods to
```

```
 * retrieve that information
 */
public class LogEntry {

    /** Creates new LogEntry */
    public LogEntry() {
    }

    public String getType(){return null;}
    public String getComment(){return null;}
    public String getData(){return null;}
    public int getCode(){return 0;}
    public int getID(){return 0;}
    public int getSessionID(){return 0;}

}
```

Logger

```
/*
 * Logger.java
 *
 * Created on April 8, 2002, 9:28 AM
 */

package netcom.log;

/**
 * Main logging class that receives LogEntries from MDSNodes
 * And directs all other classes that handle loging information
 */
public class Logger extends Thread implements Logable {

    /** Creates new Logger */
    public Logger() {
    }

    /**
     * Adds new LogEntry
     */
    public void addLogEntry(LogEntry entry) {
    }

    /**
     * returns session_id which is unique for every Node start up
     */

    public int getSessionNumber() { return 0;
    }

    public LogEntry[] getLogEntries(int from, int to) {return null;
    }
    /*
     *return log count
     */
    public int getMaxLogEntryNumber() {return 0;
    }
    /*
     * new task from a client
     */
    public void enqueueMessage( LogMessage lm){}


}
```

LogMessage

```
/*
 * LogMessage.java
```

```
 *
 * Created on April 8, 2002, 9:29 AM
 */

package netcom.log;

/**
 *
 * The instances of this class are used as a communication protocol between gui
 * and  Loggger.
 * It contains both the request parametrs and the result of the request
 */
public class LogMessage {

    /** Creates new LogMessage */
    public LogMessage() {
    }

}
```

LogServer

```
/*
 * LogServer.java
 *
 * Created on April 8, 2002, 9:33 AM
 */

package netcom.log;

/**
 * Communicates with GUIClient
 * Takes requests from GUI parses them and uses Logger to get the required
 * information.
 * Finally sends the results back to GUIClient
 */
public class LogServer {

    /** Creates new LogServer */
    public LogServer() {
    }
    public void run(){}

}
```

Mainform

```
/*
 * Mainform.java
 *
 * Created on April 8, 2002, 8:32 AM
 */

package netcom.gui;

/**
 * The topmost visible component of the Gui.
 * Contains all the coached statistic already retrieved from MDSCloud
 * Is responsible for updating all children component and handling the hash
 * of open sockets (GUIClients)
 */

public class Mainform {

    /** Creates new Mainform */
    public Mainform() {
    }
    /* The most important method that gets connect information from LDAP
```

```
     * Creates new GUIClients that talk to the client nodes
     * Sends LogMessages to every required node
     */
    public void refresh(){}
}
```

NodeEntry

```
/*
 * NodeEntry.java
 *
 * Created on April 8, 2002, 8:05 AM
 */

package netcom.util;

/**
 *  New data type that contains information about
 * an MDSCloud Node ( MDS or MDSProxy)
 * Contains a set of accessor/modifier methods in order
 * to handle every field that is needs to be stored on LDAP
 */

public class NodeEntry {

    /** Creates new NodeEntry */
    public NodeEntry(String name,String ip, int port, int type) {
    }
    public String getName(){return null;}
    public String getIP(){return null;}
    public int getPort(){return 0;}
    public int getType(){return 0;}
    public int getDataCount(){return 0;}
    public int getPClientCount(){return 0;}
    public int getFailureCount(){return 0;}

    public void  setDataCount(int n){}
    public void  setPClientCount(int n){}
    public void  setFailureCount(int n){}



}
```

ShortcutPanel

```
/*
 * ShortcutPanel.java
 *
 * Created on April 8, 2002, 8:31 AM
 */

package netcom.gui;

/**
 * implements the panel that will contain graphical shortcuts
 * (image buttons) that show the statistic that is requested often
 * by a user. The set of shortcuts is dynamically updated, according to the status
 * of MDSCloud
  */

public class ShortcutPanel {

    /** Creates new ShortcutPanel */
    public ShortcutPanel() {
    }

}
```

TreePanel

```
/*
 * TreePanel.java
 *
 * Created on April 8, 2002, 8:32 AM
 */

package netcom.gui;

/**
 *Represents the structure of the MDSCloud using a tree structure
 * Has all the statistic/log types that may be viewed.
 *The main purpose is to simplify navigation and ability to find needed
 *information
 */
public class TreePanel {

    /** Creates new TreePanel */
    public TreePanel() {
    }

}
```