# Data Store Client – Server Project

This project will involve constructing a client-server system that communicates over TCP/IP sockets. The server will store, retrieve, delete, & list named data sets on the server. Each data set is associated with a string name provided in the client’s request to the server. The server is responsible for storing this data in a persistent fashion for later retrieval. Data is persistently stored in files in a directory under the project directory.

The project deliverables include:

1. An executable Datastore Server application implementing the protocol described in the section “DatastoreClient Protocol”. This server will listen for incoming socket connections at port 10023.
2. A Client-side API that can be used to build applications that connect to the server over a TCP socket and stores, retrieves, deletes, and lists named datasets.

The delivered Datastore Server will be capable of executing on a PC and responding to requests from a remote PC running a client that implements this protocol i.e. a client provided by the grader and not the client implemented by the team.



## Datastore Server

Think of the Datastore Server as a very simple database server (a datastore) that stores and retrieves data based on unique names. The Datastore Server is an application that executes on a machine. Its purpose is to listen at a specific port for incoming socket connection requests. Upon accepting a connection, the server will listen for and act on Datastore Protocol messages sent by the client application. These messages will requests to store, retrieve, delete, or list data sets (blocks of binary data) on the server’s machine. The contents / formats of the messages are defined in this document’s section Datastore Protocol. In terms of message processing, the server is responsible for accepting and process request messages sent by the client and producing response messages that are return to the client.

## Datastore Client API

The Datastore Client API is a library that can be linked into an application and used by the application to store, retrieve, delete, and list data sets based on unique names. An interface that defines this API is provided in the following section (Datastore Client API). The library API will implement the client-side of the Datastore Protocol defined in the document. In terms of message processing, the client is responsible for creating and sending request messages to the server and for receiving and processing response messages that are return by the server.

# Datastore Client Interface

To illustrate the operations both the client and server implement, the following is a Java definition of the DatastoreClient interface that needs to be implemented.

public interface DatastoreClient

{

void write(String name, byte data[]);

byte[] read(String name);

void delete(String name);

List<String> directory();

}

The write operation will save the data contained in the byte[] associated with the given name. See the protocol section for a description of the message structure. The server will store the data in a file using the data’s name for later retrieval.

The read operation will retrieve from the server data previously saved using the provided name. If the name is not found, the response message will contain an error message. See the protocol section for a description of the message structure.

The delete operation will remove data associated with the given name from the service repository. See the protocol section for a description of the message structure.

The directory operation will respond with a list of names that are currently stored by the service. See the protocol section for a description of the message structure.

# Server Socket Communications

This project is to use TCP/IP sockets to implement communications between a client and server project. Using the Java network library class ServerSocket, an application can be created that listens for connection requests at a given port number. The server process will block at the ‘accept()’ method call until a connection request is received. The accept() method returns a socket that can be used to exchange data between the server process and the client process making the connection request. Communication is implemented as input & output streams at each end of the socket connection.

# Client-Server Messaging

The TCP Socket connection between the client and server processes described above simply provides a means of reliably passing binary data between the two processes. It is up to a client-server system to establish a message protocol over this connection. That is, the client and server must implement a message protocol that allows either side to understand and act upon the data they receive from the other.

This project defines four operations that must be implemented by the client & server. Described in the section “Data Store Service Protocol”, each of these operations is defined by both the request and response messages. Review each operation’s message protocols and observe that:

1. Each data store operation is implemented as two messages: A request message sent from the client to the server 2) A response message sent by the server to the client.
2. Request messages contain operation names that describe which of the four Datastore operations is being requested. Operation names are transferred as ASCII text. Message lengths and other integer values are transferred as ASCII representations of the numbers.
3. The data exchanged and persisted will be binary data i.e. the binary bytes are not translated into ASCII strings.
4. Each operation’s request and response message is divided into sections. Each message section is delimited (separated) by a single new line (\n) character.
5. Every request message starts with the operation’s name e.g. read\n, write\n, delete\n, or directory\n. NOTE that the ‘\n’ denotes a newline character that is used to mark the end-of-line between message sections.
6. Each response message starts with an “ok\n” if the operation was successful or some error text that describes the problem the server experienced fulfilling the client’s request.
7. A response message’s ‘ok\n’ is followed by the binary data used to communicate the results of the operation (if any data is required by the message protocol). For example, in the case of the read operation, the response message contains the number of bytes of data to be returned (as ASCII), followed by a newline, and finally by the N bytes of binary data.

# Client / Server Design

This section provides a basic design that can be followed when implementing both the server and client API portions of this project.

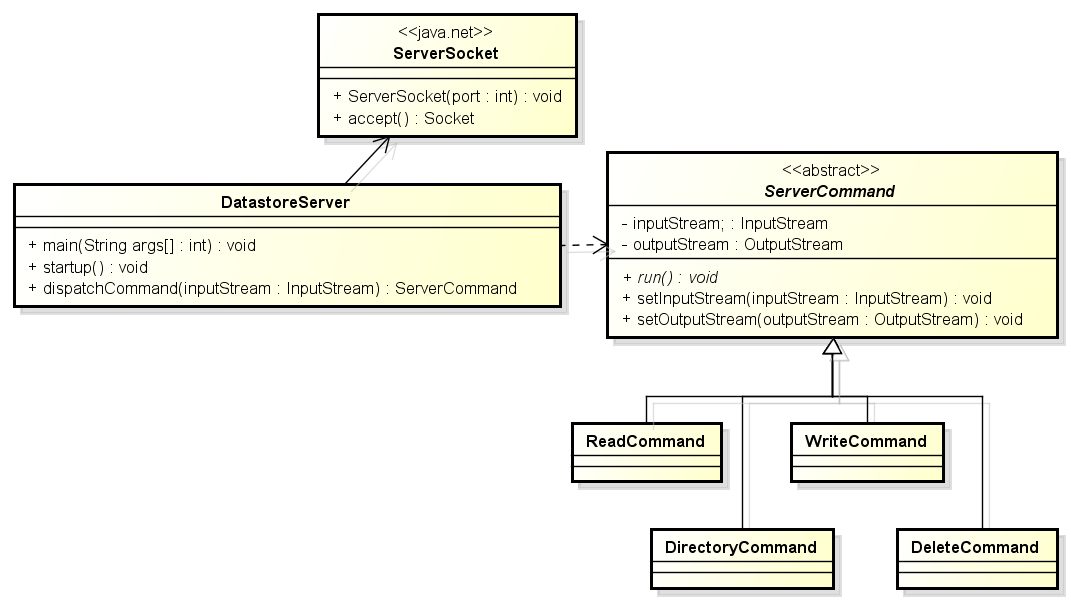


Figure : Datastore Server Design

The class DatastoreServer design requires a standalone application which will host the ServerSocket instance that implements the socket listener for incoming connection requests from the client. The dispatchCommand() method is responsible for parsing the command string that is the first several bytes of every command presented in the section Datastore Protocol. Based on this initial command string, the dispatchCommand will build the appropriate Command class (subclasses of ServerCommand). During the initialization of the ServerCommand subclass, the dispatchCommand() method will initialize the InputStream and OutputStream variables obtained from the opened socket. The dispatchCommand then calls the run() method on the newly created Command which will finish the parsing of the needed information from the input block and will then execute specific command’s behavior. This behavior includes producing and sending the corresponding response message back to the client.

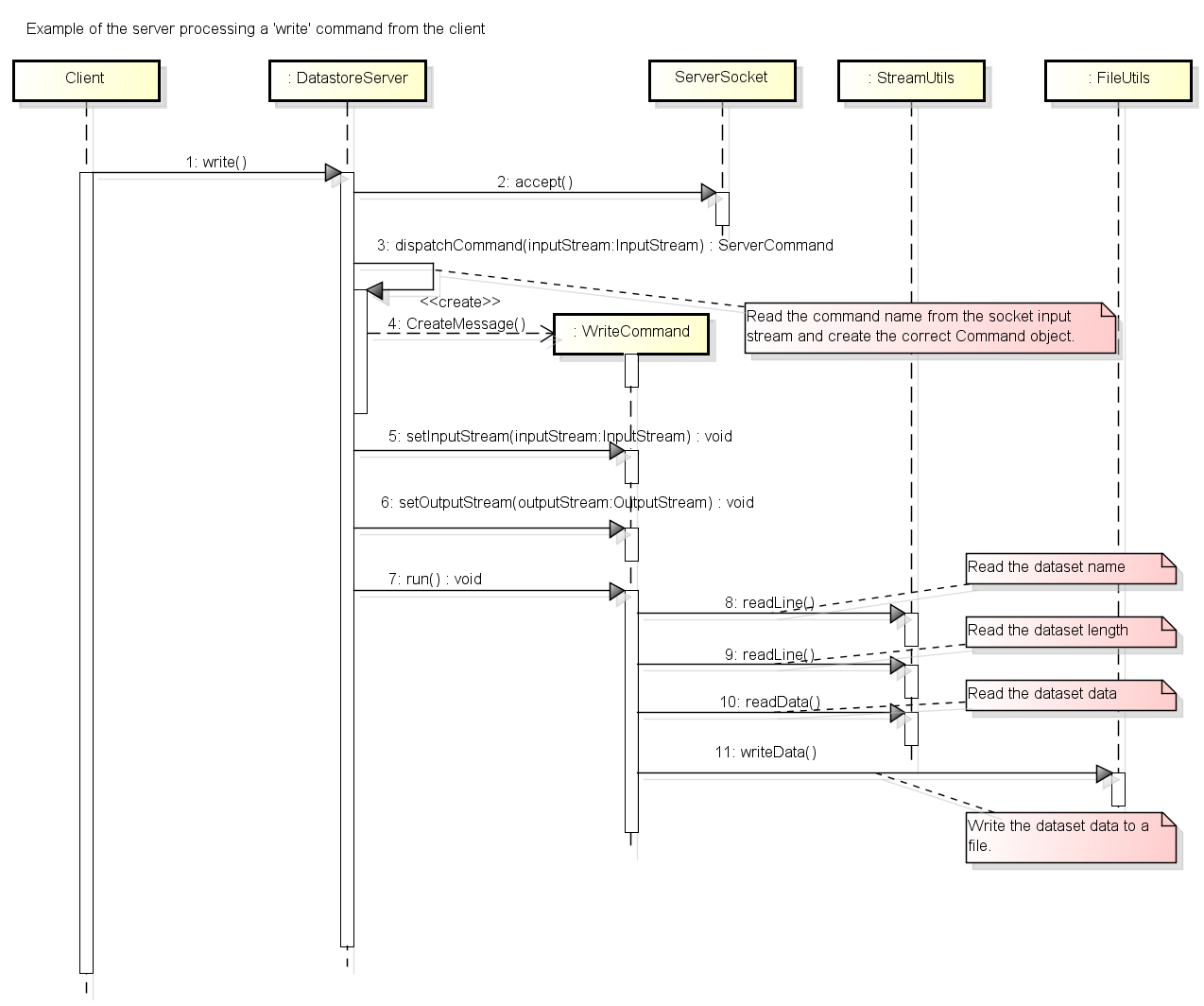


Figure : Example of DS Server processing a "WRITE" command

The sequence diagram presented above describes the processing that is performed by the server in response to a ‘write’ message from the client. The server accepts the client’s connection request though Java’s ServerSocket class. The server then parses the name of the command from the first line read from the socket’s input steam. The name is used to determine which ServerCommand subclass to create and execute. The Command’s run() method is then executed which performs the remaining write command-specific operations such as 1) reading the dataset’s name 2) reading the dataset’s length 3) reading the actual N bytes of the dataset 4) writing the dataset data to a file named by the dataset name.

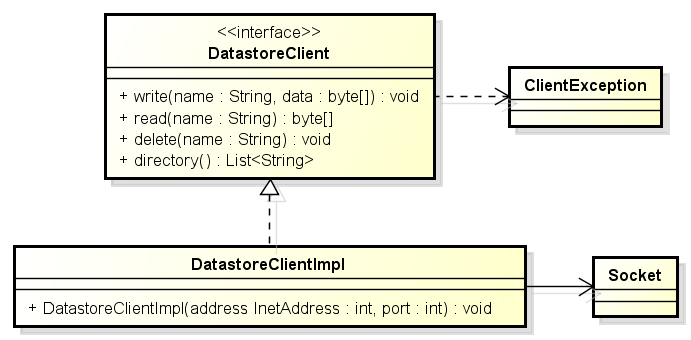


Figure : Data Store Client Design

The DatstoreClient provides an API (methods) that are used to communicate with and execute the four datastore commands on the connected server. These methods are defined on the DatastoreClient interface and implemented by DatastoreClientImpl.

The client code is responsible for opening a socket connection to the running and waiting DatastoreServer application. This involves configuring the server’s IP address and listening port number which is passed to the implementation’s constructor.

For this implementation, each of the four API methods should 1) open a socket to the server 2) build and send the correct request message to the server 3) wait for and process the response message 4) close the socket connection. Note that read() and directory() methods return a result to the object utilizing the client implementation.

# Datastore Protocol

This section describes the protocol that is to be implemented between the DatastoreClient and DatastoreServer. Each of the four sections presents the request message sent by the client to server, and the response message sent from server to client.

**NOTE**: The symbol ‘\n’ represents a new line character in either the request or response messages. The presence of the newline is used to delimit the various parts of each message and must be included as indicated by the following operation protocols.

**NOTE**: The angle brackets <> in the following operation templates are provided to mark parameter descriptions and will not be included in the message implementation.

## Write Operation

Persists the binary data provided in the message.

### Request Message

write\n

<name>\n

<data size in ANSII>\n

<N Bytes of binary data>

### Write Request Details

The write request message is provided in four sections delimited by three newlines (\n). If the name is used twice, the second write operation will overwrite the data provided by the first. The message is in ASCII except for the data (forth section) which is binary.

### Response Message

<response code>\n

### Write Response Details

The string “ok\n” (no quotes in the actual message) if the data was written else a text message describing the problem. For example, if the socket closes before N bytes is read from the client.

## Read Operation

Returns the contents of a named data that was stored in a write operation, as N bytes of binary data.

### Request Message

read\n

<name>\n

### Read Request Details

The message contains two fields delimited by two newlines. Both fields are ASCII. Bothe newlines are needed.

### Response Message

<responseCode>\n

<data size in ASCII bytes>\n

<N bytes of binary data>

### Read Response Details

Response code is the string “ok\n” (no quotes in the actual message) if the file is found else a message describing the problem and no size or data.

The number of data bytes in the response as ASCII integer (not binary) followed by newline.

N bytes of binary data that was previously stored in a Write operation.

## Delete Operation

Remove the data with the given name.

### Request Message

delete\n

<name>\n

### Response Message

<response code>\n

### Delete Response Codes

ok\n if the data was deleted. Otherwise a message describing the problem encountered while attempting to delete the data. For example, a unknown name.

## Directory Operation

List the names of the files currently managed by the service.

### Request Message

directory\n

### Response Message

ok\n

<number of file names as ASCII integer>\n

<name1>\n

<name2>\n

etc.

# Provided Materials

Teams have been provided an exported Eclipse project persistentDataStoreStudent.zip. In this project is a framework on which to build the Datastore server and client implementations. Teams should be able to import the project into their Eclipse workspace and build their implementations. Stub (empty) classes have been provided for DatastoreClientImpl, and DatastoreServer. An abstract base class ServerCommand which can serve as the base of the four command implementations has also been provided.

Important to the success of the project is the packaging of the client API classes. The provided project provides a mandatory package structure of the DatastoreClient interface as well as the DatastoreClientImpl class which should be used to base their client API implementation. Teams must not change this packaging structure of the classes provided in the project.

Also included in this project is the framework for the class DatastoreServer. Teams must complete this class to implement the required message parsing and processing. Note that to be considered a reasonable design, the bulk of the message handling should be implemented as subclasses of the class ServerCommand as described in the section Client / Server Design.

The project also includes a JUnit test case that should be used to test the DatastoreClient implementations. Note that for this test case to pass, both ends (client and server) implemented and the DatastoreServer must be running.

The project also includes two utility classes: StreamUtil and FileUtil. The class StreamUtil has been provided as an example of how to read data from the socket’s input stream. The class provides a number of static methods that send data over an OutputStream or receive data over an InputStream. Notice that some of these methods enforce the ‘\n’ newline delimiter needed to mark message sections.

The provided class FileUtil has a number of methods that can be used to save or retrieve data from files.

# Deliverables

Teams will deliver the following artifacts on a USB thumb drive:

1. An executable jar file that contains the team’s DatastoreServer implementation. This executable jar can be exported from Eclipse using the instructions provided in the document “Working with Eclipse.docx”. Also, here is an instructional video with that demonstrates the process of exporting an executable jar: <https://www.youtube.com/watch?v=mE3rbtKm-pk>.
2. A jar file containing the client library needed to perform Mode 1 Testing described in the next section. If the team packages all of their code in the executable jar, this jar will contain the client library and only a single library is needed.
3. All of the project’s source code. If you wish, you can export your Eclipse project and include the project zip file.
4. A README file (text or Word format) that lists the team number and names of all students that contributed to the execution of the project.

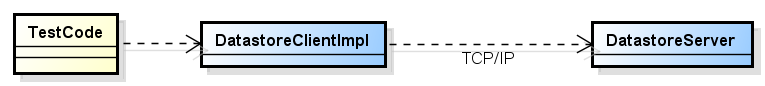
USB Drives will be returned at the final exam.

# Modes of Testing

There will be two levels of testing that determines the difference between 2 and 3 in the section Project Grading. These modes will determine how accurately your teams have implemented the Datastore Protocol outlined above.

### Mode 1 Testing

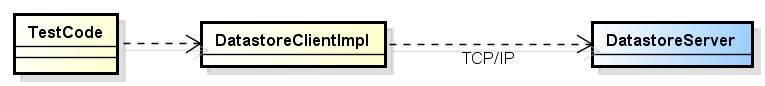
The simplest mode has the grader’s test code executing against the implementation of the Client API and Server executable. This should be the simplest to implement because both client & server are produced by the same team / effort. As shown below, the testing will be performed with test code provided by the grader (yellow) and the client & server components provided by each team (blue).



Note that Mode 1 testing will utilize the executable jar provided by teams for to execute both the server and for the client API against which the test classes DatastoreClientTestCase will be executed.

### Mode 2 Testing

The second, more difficult mode uses both the grader’s test code and the grader’s Client API implementation (yellow) against the DatastoreServer implementation provided by the teams (blue). This testing mode is more difficult because the team’s server implementation must perfectly implement the Datastore protocol.



Note that Mode 2 testing will utilize only the team’s server implementation provided in the executable jar file. The client API will be separate code that implements the server protocol as described in this document.