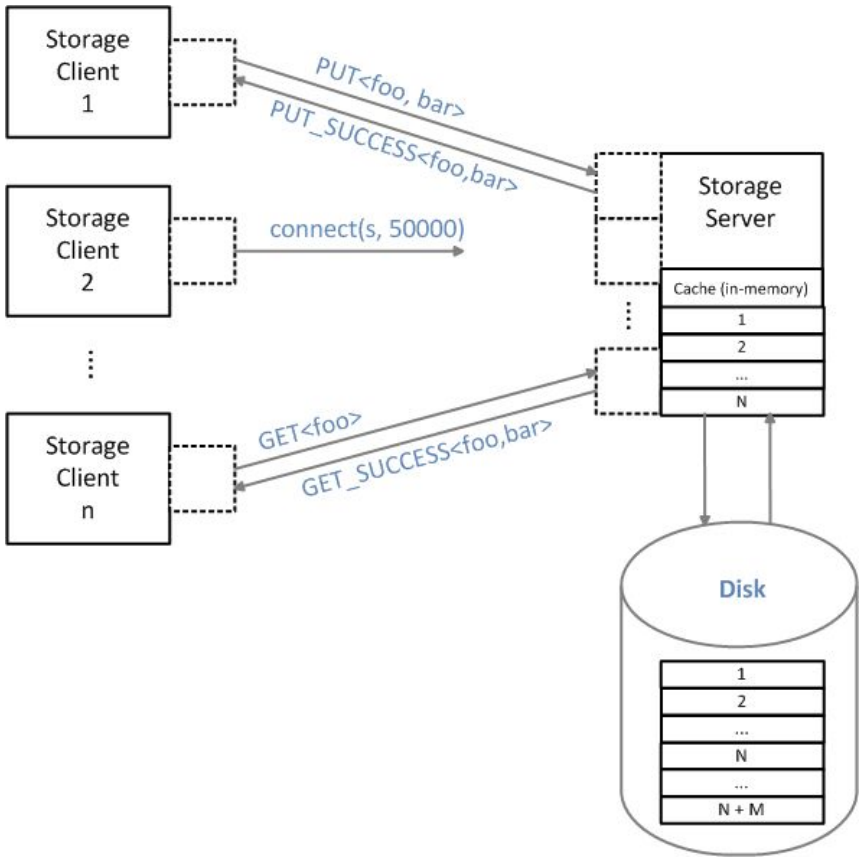
Milestone 1: Client and Persistent Storage Server

Overview

The **objective** of this milestone is to implement a **storage server** that persists data to disk (i.e., to a file). The storage server should provide a typical key-value query interface; the interface is specified in detail below. In addition, a small fraction of the data, i.e., a **configurable** amount of key-value pairs should be cached in the server's main memory. These tasks require the development of a client, a communication protocol and a suitable message format. To ease development, we provide a reference implementation of a fully functional and executable echo client and server as well as the skeleton code for the storage server, which is to serve as the basis for the design and implementation of the storage server of this milestone.

In this milestone, the storage server is to serve requests from multiple clients, whereas in the next milestone, we'll experiment with deploying a number of storage servers as full-fledged **storage service** based on the artifacts developed in this milestone. The logical client-server architecture of this mission is depicted below.



Learning objectives

With this milestone, we pursue the following learning objectives.

- Understand the client-server paradigm and get exposed to more advanced socket programming from the client's and server's perspective
- Differentiate between client-side application, client-side communication stub, messages, basic notions of protocol, and server
- Understand how multiple clients interact with a single server
- Learn to develop a basic protocol made up of messages between client and server
- Implement cache replacement strategies for offloading data to disk (i.e., FIFO, LRU, LFU)
- Use Java tools for logging (log4j) and building (ant)
- Learn about and rigorously apply unit testing and religiously deal with failures
- Conduct performance evaluations

We will build on these concepts in subsequent assignments.

Detailed milestone description

Provided code

Together with this handout that specifies the storage server interface, we provide you with the implementation of an echo client and an echo-server documented in a separate specification, which are to serve as reference code that illustrates how a client communicates with a server that echos any messages sent from the client. In addition, we provide you with a project stub including a set of JUnit test cases, which your program should pass before submitting, some interfaces (see below) and the build script for building and testing the program. You must use this stub to start your implementation.

Storage Server API

The storage server should provide the following interface:

```
/**
 * Establishes the connection to the KV Server.
 * @throws Exception if connection could not be established.
 */
public void connect() throws Exception;

/**
 * disconnects the client from the currently connected server.
 */
```

```
public void disconnect();

/**
 * Inserts a key-value pair into the KVServer.
 * @param key the key that identifies the given value.
 * @param value the value that is indexed by the given key.
 * @return a message that confirms the insertion of the tuple or an error.
 * @throws Exception if put command cannot be executed
 *     (e.g. not connected to any KV server).
 */
public KVMessage put(String key, String value) throws Exception;

/**
 * Retrieves the value for a given key from the KVServer.
 * @param key the key that identifies the value.
 * @return the value, which is indexed by the given key.
 * @throws Exception if put command cannot be executed
 *     (e.g. not connected to any KV server).
 */
public KVMessage get(String key) throws Exception;
```

whereby KVMessage is defined by the following interface:

```
public enum StatusType {
     GET,             /* Get - request */
     GET_ERROR,       /* requested tuple (i.e. value) not found */
     GET_SUCCESS,     /* requested tuple (i.e. value) found */
     PUT,             /* Put - request */
     PUT_SUCCESS,     /* Put - request successful, tuple inserted */
     PUT_UPDATE,      /* Put - request successful, i.e. value updated */
     PUT_ERROR,       /* Put - request not successful */
     DELETE_SUCCESS,  /* Delete - request successful */
     DELETE_ERROR     /* Delete - request successful */
}

/**
 * @return the key that is associated with this message,
 *         null if not key is associated.
 */
public String getKey();

/**
 * @return the value that is associated with this message,
 *         null if not value is associated.
 */
```

```
public String getValue();

/**
 * @return a status string that is used to identify request types,
 * response types and error types associated to the message.
 */
public StatusType getStatus();
```

Assigned development tasks and deliverables

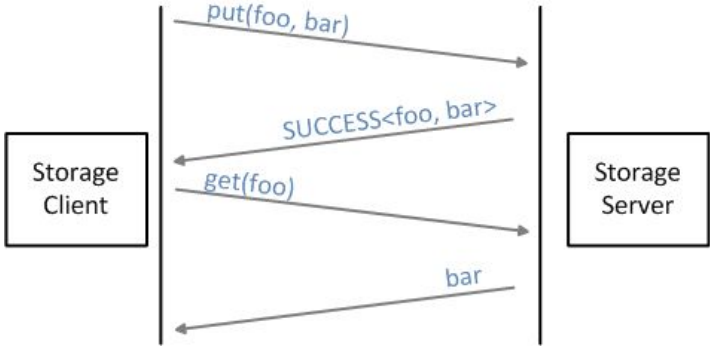
In this milestone, the following components need to be developed:

- Communication logic
    - Design message formats for requests (get & put) and server replies (requested tuple, status (e.g., “success”, “error”)).
    - Implement marshalling and unmarshalling of different messages.
    - Design and implement a communication protocol to support queries on the storage server.
- Storage server comprised of
    - Multi-threaded storage server (i.e., it should handle multiple client connections concurrently (while this can be achieved without threads, we require that you implement concurrency via multi-threading by leveraging our skeleton code of the provided echo server.))
    - Failure handling to gracefully deal with errors, exceptions and failure conditions.
    - Caching mechanism (i.e., in memory data maintenance) for a configurable amount of key-value pairs.
    - Implementation of a persistence mechanism for key-value pairs that stores them permanently on disk
    - Implementation of 3 different cache replacement strategies (FIFO, LRU, LFU).
    - Separate logging capability to log server behaviour and interactions.
- Storage client program comprised of
    - Application logic: Command line shell to interact with the storage server and support queries to it.
    - Communication logic: Socket-based communication by reading and writing from/to a byte buffer.
    - Clean separation of application logic and communication logic
    - Separate logging capability to log client behaviour and interactions.
- JUnit Tests
    - Integration of the JUnit library into your project.
    - Automate the running of the given test cases
    - Add at least 10 test cases of your choice that cover relevant system behavior not yet tested in the given test cases

    - Compile a short test report about all test cases, especially your own cases (submit as appendix of your design document).
- Performance evaluation
    - Provide a performance report that evaluates your storage servers get (read) and put (write) performance for the difference caching strategies and different cache sizes while varying the ratio of puts to gets (e.g., evaluate latency and/or throughput for at least 80% puts, 20% gets to 50%/50%, and 20%/80% puts vs. gets, respectively.)
- A design document that is no longer than 3 pages detailing your design, any design decisions you made and includes the performance evaluation; the documentation of your test cases is not counted among the page count.

Communication logic

The client-server communication protocol is mainly comprised of request-reply interactions (see below), where messages consist of various attributes formed from byte sequences which are arranged in a predefined order. The figure below shows an excerpt of a possible communication session between a client and the storage server.



The first two messages represent a typical put interaction (i.e., request and reply). Essentially, the put request message consists of at least the two parameters “key” and “value” and a flag that identifies the actual operation (i.e., put in this example). After the tuple has been successfully inserted into the storage server, the request is going to be acknowledged by the server (SUCCESS<key, value>).

The second message pair represents the retrieval functionality. The client issues a message that contains at least the query key and again, the type of operation (i.e,, get in this case). As a result, the server looks up the value for the desired key and hands it back to the client. If the server is not able to find the requested value, it will return an error message.

Messages

Since the communication between client and server must only rely on sending and receiving bytes, it is necessary to differentiate between the whole message and its components (i.e., message attributes, values and/or flags). This requirement exposes the definition of message

formats, which describe the structure and content of particular messages. A challenge, here, is to develop a suitable, efficient, and extensible serialization mechanism that enables the transformation from Java objects, representing the messages, to a stream of bytes and vice versa. Basically, there are several ways to achieve this. Before deciding for a particular approach, try to identify alternative approaches and compare them with respect to the criteria mentioned above.

Before designing your message format, you should think about the data that actually needs to be transferred between client and server. Then, set up the message format and implement the marshalling and unmarshalling methods, respectively.

Protocol

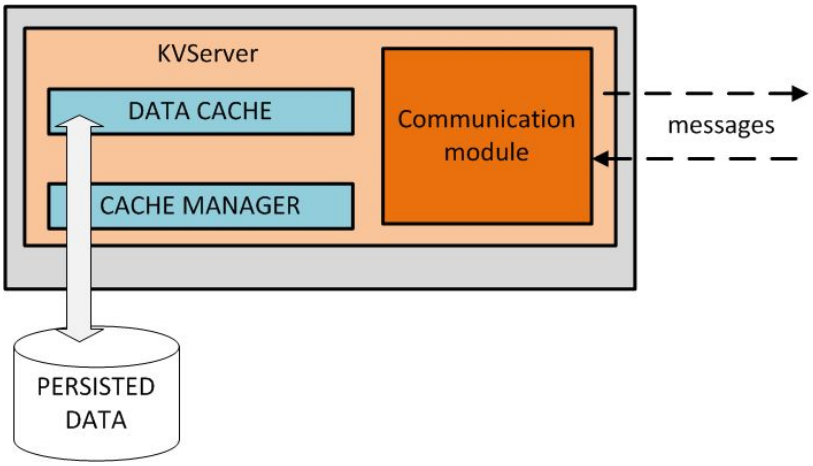
Besides the various message formats a protocol also needs to specify the order in which particular messages have to be exchanged to meet the protocol specification. In order to set up a suitable communication protocol you should first define the message format which also involves the serialization mechanism for the byte transfer and then think about an appropriate interaction model. Make sure that the protocol covers the request functionality, - the get and put- as well as the reply functionality that serves as application-level acknowledgement. Among others, consider the following scenarios:

- During a put request:
    - The server successfully inserted a tuple into its data structure and confirms this to the client.
    - The server may be unable to insert the tuple due to some unpredictable event (e.g., error), hence, it should inform the client about this situation.
- During a get request:
    - The tuple may not be present at the storage server. Hence, an error has to be passed back.
    - In the case of a successful lookup, the data is returned.

Think of other possible failure scenarios and model them accordingly in your protocol design. Note, it is important that the client is able to correlate the server reply with a particular request it previously issued.

Storage server

The storage server is responsible for maintaining the key-value pairs and providing access to clients. As already indicated above, data must be stored durably on disk. In addition to storing each key-value pair on disk, a configurable amount of tuples should also be stored in memory to speed-up the lookup process for GET requests (parameter ‘cacheSize’ of KVServer) . The general architecture for a storage server is depicted in the figure below.



Persistence & Caching

Think of an efficient data-structure for caching as well as an efficient file-structure for disk persistence to manage key-value pairs (i.e., fast insertion and lookup) and implement the following three strategies to displace data from cache to disk.

- FIFO (First In First Out)
- LRU (Least Recently Used)
- LFU (Least Frequently Used)

Cache replacement of key-value pairs is necessary in two distinct cases:

1. If a GET-request by a client causes a cache miss, the key-value pair is looked-up on disk and transferred to the cache. If the cache is already full, a tuple is evicted, following the cache replacement policy selected.
2. If a PUT-request by a client is received by the server and the cache is already full, then a tuple must be selected for eviction, following the cache replacement policy selected.

Concurrency

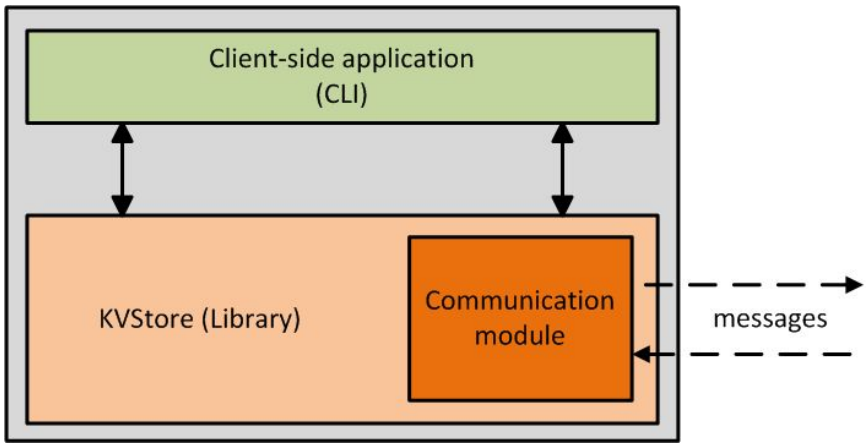
Furthermore, the concurrent handling of multiple client connections is an inherent prerequisite for our storage server and for other server implementations in general. A generic way to realize concurrency can be found in the implementation of our echo server code. Here, each client connection is handled by its own thread. Besides message handling, this thread also accesses the shared data of the storage server process. Note, that it is important to guarantee the consistent manipulation of shared data in the face of concurrent accesses.

The server should also contain a logging facility that logs all the relevant program behavior during its execution to a specific log file (logs/server.log). It is sufficient that the log level of the server is controlled by a commandline parameter (i.e., while starting the server application from the command line).

The port number (int), which the storage server should listen at, the cache size (int), and the cache replacement strategy (String), i.e., either “FIFO”, “LRU”, or “LFU”, should be passed as a command line parameter as well.

Storage server client

In order to guarantee the highest possible degree of reusability, the client should be implemented in a modular manner. Altogether, it should be comprised of at least three components (cf. the figure below).



First, the **client-side application** is mainly comprised of a command line interface that manages the interaction with the user and utilizes KVStore. **Adapt the provided echo client** such that it supports the commands listed in the table below.

Second, **KVStore** serves as a library for client applications and encapsulates the complete functionality to use the storage server from anywhere on the Internet. Note that this library should be independent from any particular client-side application. Thus, it has to provide a well-defined, minimal, and universally applicable interface to application programmers.

Finally, the **communication module** encapsulates the complete communication logic. Predominantly, this incorporates connection management, message sending, and message receiving. Here, you should be able to re-use much of the code from the provided echo client, but try to define a proper interface for this module.

Don't forget to also set up logging for the client as well.

The table below specifies the commands supported by the storage server client.

Command	Informal description	Parameters	Shell output
`connect <address> <port>`	Tries to establish a TCP- connection to the storage server based on the given server	**address**: Hostname or IP address of the storage server.	**server reply**: Once the connection is established, the client program should

	address and the port number of the storage service.	**port**: The port of the storage service on the respective server.	give a status message to the user. (*Note, if the connection establishment failed, the client application should provide a useful error message to the user*);
`disconnect`	Tries to disconnect from the connected server.	-	**status report**: Once the client got disconnected from the server, it should provide a suitable notification to the user. (*Note that the connection might also be lost due to a connection error or a break down of the server*);
`put <key> <value>`	**Inserts** a key-value pair into the storage server data structures. **Updates** (overwrites) the current value with the given value if the server already contains the specified key. **Deletes** the entry for the given key if <value> equals null.	**key**: arbitrary String (max length 20 Bytes) **value**: arbitrary String (max. length 120 kByte)	**status message**: provides a notification if the put-operation was successful (SUCCESS) or not (ERROR)
`get <key>`	Retrieves the value for the given key from the storage server.	**key**: the key that indexes the desired value (max length 20 Bytes)	**value**: the value that is indexed by the given key if present at the storage server, or an error message if the value for the given key is not present.
`logLevel <level>`	Sets the logger to the specified log level	**level**: One of the following log4j log levels: (ALL | DEBUG | INFO | WARN | ERROR | FATAL | OFF)	**status message**: Print out current log status.
`help`		-	**help text**: Shows the intended usage of the client application and describes its set of commands.

`quit`	Tears down the active connection to the server and exits the program.	-	**status report**: Notifies the user about the imminent program shutdown.
`<anything else>`	Any unrecognized input in the context of this application.	`<any>`	**error message**: Unknown command; print the help text.

The client should log relevant information to its own file (logs/client.log). Logging should be dynamically controllable (ALL | DEBUG | INFO | WARN | ERROR | FATAL | OFF) by the command `logLevel`. (see table above).

Graceful failure handling

A vital prerequisite for the client is its ability to handle failures gracefully. Please make sure that your program is robust to any kind of wrong or unintended user input (e.g., wrong/unknown commands, number and format of parameters, etc.). Additionally, consider problems that might occur in communication, i.e., handle exceptions properly and watch out for connection failure and (controlled) shutdown of the storage server. In addition to error messages, also print out status messages to the shell, if it makes sense (e.g., if the client is connected or disconnected from the server.)

Server feedback for requests

In the table below, you find a set of interaction schemes that your client-server application should at least support.

Request	Result at server side	Consequence
put <key, value>	tuple successfully inserted	send acknowledgement to client: PUT_SUCCESS<key,value>
put <key, value>	unable to insert tuple	send error message to client: PUT_ERROR<key,value>
get <key>	tuple found	SUCCESS<key, value> to client.
get <key>	tuple not found	send error message to client: GET_ERROR<key>
any	message format unknown, message size exceeded, etc	send error message to client: FAILED<error description>

General considerations

- Document your program properly using JavaDoc comments.
- Stick to the common Java coding conventions ([Link]).
- Pay attention to good program design (e.g., decoupling of UI and program logic.)

Testing with JUnit

1. Create connection / disconnect
2. Set value
3. Get value
4. Update value (set with existing key)
5. Get non-existing value (error message)
6. Test client library methods

Suggested development plan

First of all, download the project files, and base your implementation on the provided package structure. The stub already contains the necessary libraries, the interfaces, the build script and the JUnit test cases. Before you start coding try to set up the communication protocol.

Getting the project files

The project files for Milestone 1 are located in the **`/cad2/ece419s/M1/`** directory on the EECG UG machines. The Milestone 1 starter package has three archive files containing source code: *BasicStorageServer-stub.tgz, echoClient.tgz,* and *echoServer.tgz*. To retrieve these files, make a new directory for Milestone 1 and run the following commands:

```
> cp /cad2/ece419s/M1/* ./
> find -iname \*.tgz -exec tar -xzf {} \;
```

Building and Running the code

To build any of the three projects, run **`ant`** in the project directory.
To run the sample echoServer, use **`java -jar echoServer.jar <port number>`** . Port numbers between 1024 and 65536 should be used (pick a random port to avoid conflicting with other services.) If you cannot start the server because "***Port is already bound!***" use a different port number. Avoid hard-coding port numbers for this reason.
To run the sample echoClient, use **`java -jar echoClient.jar`**
Use the same methods to run either of the storage server apps (ms1-client and ms1-server)

Deliverables & Code submission

By the **deadline**, you must hand in your software artifacts that implement all the coding requirements and include all necessary libraries and the build script. See the above specified deliverables for what needs to be submitted. Submission Instructions can be found here.

Marking guidelines and marking scheme

All the code you submit must be compatible with the build scripts, interfaces and test cases that we provide with the respective milestone. In addition, your code must build and execute on the UG machines Debian 8.2 with Java 1.7.0_111 without any further intervention on our part and provide the specified functionality.

Additional resources

- Java SE API: http://docs.oracle.com/javase/6/docs/api/java/net/Socket.html
- Log4j: http://logging.apache.org/log4j/2.x/
- JUnit: http://www.junit.org/
- Ant build tool: http://ant.apache.org/
- ASCII format: http://tools.ietf.org/pdf/rfc20.pdf

Document revisions

Major changes to the milestone handout after posting it are tracked here.

Date	**Change**
None	