Detailed specification

Contents

Detailed implementation description

Storage Server Functionality

The storage server is organized into *tables*, and each table contains a collection of *records*, where each record is identified by a unique *key*. For this version of the storage server, each record is a simple string with a maximum number of characters.

For example, suppose you wish to store your marks in a table called `marks`, using the course code as the key to identify each record. The `marks` table may look like this.

Key	Value
ECE100	78
ECE143	92
ECE297	87

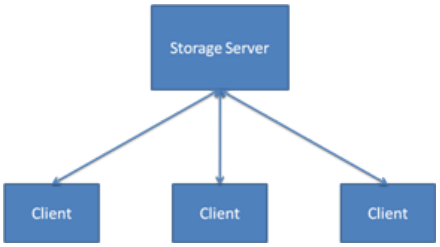
The storage server supports the following functionality.

- Retrieve an existing record. (Use the `storage_get()` function.)

- Insert a new record. (Use `storage_set()` with a key that does not already exist in the table.)
- Update an existing record. (Use `storage_set()` with a key that already exists in the table.)
- Delete a record. (Use `storage_set()` with a NULL value.)

The detailed specification of the `storage_get()` and `storage_set()` functions are described below.

The storage server retains all the keys and records in main memory and all records are lost when you restart the server. Also, the server must respond to operations from clients which may be running on separate machines from the server (see the below illustration to clarify this point.)



Storage Server Specification

You must implement the *storage server* and a *client library* that can communicate with the server. All the details of communicating with the server are encapsulated within the client library. This way, any client can communicate with your server by using the client library. Information about the tables in the server and the location of the server are specified in a configuration *file*.

The storage server and the client library must be derived from the skeleton code distribution we provide for you (see below on how to retrieve the skeleton code.)

Note: you are allowed to start your codebase using one teammate's existing M1 code. Starting from scratch is acceptable as well, but you will need a fully working command-line shell and logging for the midterm.

Configuration file

The configuration file contains one configuration parameter per line, with each line containing the parameter name, followed by one or more spaces, followed by the parameter value. The file may contain empty lines, which should be ignored. Also, any lines that start with a pound "#" should be treated as a comment and should also be ignored.

We will use the following configuration parameters in this assignment.

Name	Value
server_host	A string that represents the IP address (e.g., 127.0.0.1) or hostname (e.g., localhost) of the server.
server_port	An integer that represents the TCP port the server listens on.
username	A string that represents the only valid username for clients to access the storage server.
password	A string that represents the encrypted password for the `username`.
table	A string that represents the name of a table. There may be more than one `table` parameter, one for each table.

Here's a sample configuration file.

```
server_host localhost
server_port 1111
username admin
password xxxnq.BMCifhU
table marks
```

Your client and server may use additional parameters of your choice but must not rely on them. We will use our own configuration files during marking, and your program must not assume additional parameters will be present.

In case of duplicate configuration parameters or table names (i.e., the same parameter or table name is present more than once in the configuration file), the server should exit with an error.

Here's a sample for a bad configuration file (duplicate `server_port`).

```
server_host localhost
server_port 1111
server_port 2222
username admin
password xxxnq.BMCifhU
table marks
```

Here's another sample for a bad configuration file (duplicate table and missing username and password).

```
server_host localhost
server_port 1111
table marks
table marks
```

Server

The storage server will be a standalone program that accepts commands to access the tables from clients over a TCP connection.

When the server is run, it is given the path to the configuration file as the first argument.

```
> ./server default.conf
```

The server should parse the configuration file.

The server should then listen on port `server_port` for any requests from clients.

When a request arrives from a client that requires the server to modify a table, the server should modify the table entry before notifying the client that the operation has succeeded. This is more important for future assignments as the server may be shutdown anytime after a client request completes.

Upon any error (e.g., configuration file isn't given, the listen port is unavailable, etc.) the server should exit with a non-zero return value. However, the server should not exit if a key or a table is not found. (It is UNIX convention to exit a program with a return value of 0 on success.)

It is permissible for the server to print status or debug information while it is operating. However, this will not be marked.

It is also up to you how the server handles requests from clients. A simple design will process each request before moving onto the next, as opposed to processing the requests in parallel.

Client library

Whereas the server is a standalone program that you can run, you are not required to develop a standalone client (we actually provide a simple client code for you which you already extended in Milestone 1). Instead you are required to implement a *client library* that provides an interface to communicate with the server. Storage clients will use the interface functions in the client library to retrieve and store data in the storage server. This way the storage clients don't need to know anything about how to talk to the server; all the details are encapsulated in the client library functions.

The client library interface is declared in the `storage.h` header file given to you as part of the skeleton distribution. **You must not modify this header file**. Here are the key parts of the client library interface.

```
// You should ignore error codes and constants that are not
// defined in the handout as they are not needed for this
// assignment. However, please do not modify them as they are
// required for the other assignments.

// Configuration constants.
#define MAX_CONFIG_LINE_LEN 1024
#define MAX_USERNAME_LEN 64
#define MAX_PASSWORD_LEN 64
#define MAX_HOST_LEN 64
#define MAX_PORT_LEN 8

// Storage server constants.
#define MAX_TABLES 100
#define MAX_RECORDS_PER_TABLE 1000
#define MAX_TABLE_LEN 20
#define MAX_KEY_LEN 20
#define MAX_VALUE_LEN 800

// Error codes.
#define ERR_INVALID_PARAM 1
#define ERR_CONNECTION_FAIL 2
#define ERR_NOT_AUTHENTICATED 3
```

```
#define ERR_AUTHENTICATION_FAILED 4
#define ERR_TABLE_NOT_FOUND 5
#define ERR_KEY_NOT_FOUND 6
#define ERR_UNKNOWN 7

struct storage_record {
        char value[MAX_VALUE_LEN];
        uintptr_t metadata[8];
};

void* storage_connect(const char *hostname, const int port);
int storage_auth(const char *username, const char *passwd, void *conn);
int storage_get(const char *table, const char *key, struct storage_record *record, void *conn);
int storage_set(const char *table, const char *key, struct storage_record *record, void *conn);
int storage_disconnect(void *conn);
```

The five functions above will be implemented in the client library. The required behavior of each function is described below.

Function	Parameters	Description	Return value
storage_connect	`hostname`: The IP address or hostname of the server. `port`: The TCP port of the server.	Establish a connection to the server running at `hostname` on port `port`.	If successful, return a pointer to a data structure that represents a connection to the server. Otherwise return NULL.
storage_auth	`username`: Username to access the storage server. `passwd`: Password in plain text form. `conn`: A connection to the server.	Authenticate the client's connection to the server.	Return 0 if successful, and -1 otherwise.
storage_get	`table`: A table in the database. `key`: A key in the table. `record`: A pointer to a record structure. `conn`: A connection to the server.	The record with the specified key in the specified table is retrieved from the server using the specified connection. If the key is found, the record structure is populated with the details of the corresponding record. Otherwise, the record structure is not modified.	Return 0 if successful, and -1 otherwise.
storage_set	`table`: A table in the database. `key`: A key in the table. `record`: A pointer to a record structure. `connection`: A connection to the server.	The key and record are stored in the table of the database using the connection. If the key already exists in the table, the corresponding record is updated with the one specified here. If the key exists in the table and the record is NULL, the key/value pair are deleted from the table.	Return 0 if successful, and -1 otherwise.
storage_disconnect	`connection`: A pointer to the connection structure returned in an earlier call to`storage_connect`.	Close the connection to the server.	Return 0 if successful, and -1 otherwise.

Upon any error, the client library should set the global `errno` variable to one of the error values defined in the header file. For example, if the `storage_get()` function is called with a NULL connection parameter, `errno` should be set to `ERR_INVALID_PARAM`. We will be checking whether you are setting at least the following error conditions properly.

- ERR_INVALID_PARAM: This error may occur in any of the five functions if one or more parameters to the function does not conform to the specification.
- ERR_CONNECTION_FAIL: This error may occur in any of the five functions if they are not able to connect to the server.
- ERR_AUTHENTICATION_FAILED: This error may occur if the client provides wrong username and password to `storage_auth()`. Note that valid values for these parameters are those specified in the server's config file.
- ERR_NOT_AUTHENTICATED: This error occurs if the client invokes `storage_get()` or `storage_set()` without having successfully authenticated its connection to the server.

- ERR_TABLE_NOT_FOUND: This error may occur in the `storage_get()` or `storage_set()` functions if the server indicates that the specified table does not exist. Note that since the client library does not have access to the database, it is only the server that knows whether the table exists.
- ERR_KEY_NOT_FOUND: This error may occur in the `storage_get()` function if the server indicates that the specified key does not exist in the specified table. It is also used when trying to delete a non-existing key.

You should use ERR_UNKNOWN for any other errors that may occur such as out of memory errors or network errors.

It is up to you whether you want to establish a connection to the server when `storage_connect()` is called, and use this open connection in subsequent `storage_get()` and `storage_set()` calls. Alternatively, you can establish and tear down a connection to the server each time `storage_get()` or `storage_set()` is called. Your decision here will affect the implementation of the `storage_connect()`, `storage_disconnect()` and `storage_auth()` functions as well as what you store in the `conn` parameter.

Note that regardless of whether you reuse or create a new connection for each invocation of `storage_get()` and `storage_set()` the client is required to call `storage_auth()` **only once**, prior to these invocations. Furthermore, if the client fails to successfully authenticate with the server, all subsequent calls to `storage_get()` and `storage_set()` must fail (return -1 and setting errno to ERR_NOT_AUTHENTICATED). Hint: The cleanest way to do this is to use a data structure to store a client's connection information. This information may include the username and password that the client provides through `storage_auth()` as well as the file descriptor of the server socket (created after `storage_connect()`).

The implementation of the `storage_get()` and `storage_set()` functions will communicate with the server. You will need to design a protocol to send the client's request to the server and receive the server's response (including errors) at the client. The skeleton code given to you includes a simple protocol in which the server simply echos back requests from the client. You will need to modify this.

Client Authentication and Password Encryption

As you may have noticed already, the server reads an encrypted form of the password from the configuration file. For example, if the plain text password is `dog4sale`, its encrypted form may look like `xxxnq.BMCifhU`. To compute the encrypted form of a password from a command line, we have provided a stand-alone program called `encrypt_passwd` that you can compile and run:

```
> cd storage/src
> make encrypt_passwd
> ./encrypt_passwd dog4sale
  xxxnq.BMCifhU
```

You can use this output to update the encrypted password field in the configuration file. Note that it is in general a bad idea to use dictionary words in your password. In the above example, we disobeyed this rule to make the plain text password distinguishable from its encrypted form.

It is important to note that on the client side, `storage_auth()` is invoke by passing the password in its plain text form. This password must be encrypted first before being sent to the server for authentication. The skeleton code provided to you handles this as part of `storage_auth()` by invoking the below utility function (defined in utils.h):

```
char *generate_encrypted_password(const char *passwd, const char *salt);
```

For now, you can simply ignore the second argument and invoke the method by passing the plain text password (as first argument) and `NULL` (as second argument).

While we have provided most of the code needed on the client side, it is your responsibility to implement the logic on the server side to compare the received username and encrypted password with the values read from the configuration file. If they match, authentication is successful, otherwise, authentication fails and the client call to `storage_auth()` must return -1 and errno must be appropriately set.

Table, key, and value formats

In any use of tables, keys, and values above, valid table and key names must consist of alphanumeric characters, while valid values may contain both alphanumeric characters and spaces. In all cases, the alphabetic characters should be case sensitive. The table below lists some examples of tables, keys, and values that are allowed and not allowed.

Type	Allowed	Not allowed
Table	marks Marks Marks2009	my marks Marks*2009
Key	commdesign CommDesign 297	Comm and Design ECE-297

	ECE297	
Value	aplus A Plus 98	A+ a_plus

Performance evaluation

A major component of this assignment is to perform a quantitative evaluation of the performance of your storage server.

You must evaluate different *configurations* of the storage server, -- namely the logging policies, -- measure different *metrics,* and vary the *workload*.

Server configurations to evaluate

You should measure the performance of the storage server under three configurations:

- A configuration without logging,
- logging enabled to *stdout*, and
- logging enabled to *a file*.

Performance metrics

The evaluation should quantify two metrics: The *end-to-end execution time* of the workload and the *server processing time*. You must modify your code to measure these metrics; this is referred to as *profiling* or *instrumenting* your code.

End-to-end execution time

The end-to-end execution time is the duration it takes to execute a workload (a set of get or set calls). This includes the time for the client to send the associated commands over the network connection to the server, the server to execute the commands, and the client to receive the response from the server. For example, the client might include code that looks like this:

```
struct timeval start_time, end_time;

// Remember when the experiment started.
gettimeofday(&start_time, NULL);

// Perform the workload operations.  (Issue the storage_* calls.)
// ...

// Get the time at the end of the experiment.
gettimeofday(&end_time, NULL)

// Calculate difference in time and output it.
// ...
```

Server processing time

You should also measure the time the server spends processing the commands in the workload. This duration should not include the time to receive or send messages from the client, nor the time to parse the message from the client or construct the reply message to be sent back to the client. For example, the server code may include something like this:

```
struct timeval total_processing_time = 0;

int handle_command(int sock, char *cmd)
{
    if (is_a_get_command(cmd)) {
        // Extract the parameters from the command
        // ...

        struct timeval start_time, end_time;

        // Remember when the processing started.
        gettimeofday(&start_time, NULL);

        // Perform the get operation.
        // ...

        // Get the time at the end of the processing.
        gettimeofday(&end_time, NULL)

        // Calculate difference in time and add it to total_processing time.
        // total_processing_time += ...

        // Send the response back to the client.
        // ...
    }

    // ...
```

```
}
```

Workload

The workload consists of the *contents of the tables* stored in the storage server and the get or set *operations* invoked by the client.

As table content, you should use the following table specification about population information from the 2006 Canadian Census by Statistics Canada:

- The table name should be `census`.
- The key is the name of the city region.
- The value is the `Population` count of the city as of 2006. (An integer)
- The table should be populated with the data from the census subdivisions. (Comma-separated and tab-separated versions of the data are available.)

Note: if you are unable to use the above link, try this mirror.

To populate the table, you will need to scrub the raw data from Statistics Canada. For example here's what some of the raw CSV data looks like:

```
3520005,"Toronto (Ont.)",C  ,F,2503281,2481494,F,F,0.9,1040597,979330,630.1763,3972.4,1
2466023,"Montréal (Que.)",V  ,F,1620693,1583590,T,F,2.3,787060,743204,365.1303,4438.7,2
5915022,"Vancouver (B.C.)",CY ,F,578041,545671,F,F,5.9,273804,253212,114.7133,5039.0,8
3519038,"Richmond Hill (Ont.)",T  ,F,162704,132030,F,F,23.2,53028,51000,100.8917,1612.7,28
```

You'll notice that the city region and province are included in one field, and the province is abbreviated and in parentheses. You'll have to extract the necessary columns (`City` and `Population`) from the raw data and store them in your table. As well, since your storage server can only handle alphanumeric characters for keys, you should strip any special characters in the city names, such as spaces or dashes. You can replace accented characters (e.g., the *é* in *Montréal*) with their non-accented equivalents, but it is perfectly acceptable if you simply strip out these characters too.

After scrubbing, the above four entries should look something like this in your table:

Key	Value
Toronto	2503281
Montral	1620693
Vancouver	578041
RichmondHill	162704

Be aware that there are almost 700 entries in the raw data, and you shouldn't try to convert this data manually! To help automate the conversion of the raw data, you can use popular tools described in the Parsing Section such as `awk` and `sed` (or a tool of your liking, maybe `perl`.) If you wish, you may even write a program to convert the raw data; you can write this program in any language you wish, including C, Perl, Python, and Java.

Think also about how you will bulk load these entries into the table. Will you write these entries directly into the server's table data structures in memory, or will you issue a number of `storage_set` calls to populate the table? Note that since tables are memory-resident only, issuing a number of `storage_set` calls may be the only way to populate these tables, or do you see alternatives?

Once you have converted the raw data to a format your storage server can process, place a copy of this data in a data directory called `storage/data/census`. Be sure to `svn add` and `svn commit` this directory and its contents to the SVN repository so it is included in your submission. We will verify that this directory to ensure you did process the census data.

You should also include a sample configuration file in `storage/data/census.conf` that includes a table specification for the census table. Remember that this table should be called `census` and contain the columns specified above.

Performance evaluation report

You must write a report that describes your evaluation *methodology*, *results*, and *analysis*. This will be handed in to Turnitin.com at the code submission deadline but will be considered as part of the Design Document mark.

The methodology section should be a *brief* description of the evaluation setup. This includes

- the metrics you measured
- the workload including the tables used, the number of table entries, and the operations.

The results section should present the quantitative data from running the experiments. You can present the results in a table or chart.

Finally, in the analysis section, you should draw some conclusions you derive from the evaluation results. Summarize the relative benefits and tradeoffs you observe. How much is the server's processing performance improved or degraded? Is this a significant portion of the end-to-end processing time?

The evaluation report should contain no more than 500 words (roughly one page), excluding tables and figures.

Coding Requirements

You are given a skeleton distribution that includes incomplete code for the storage server and client library, as well as Makefiles and test suites. (See here for more on working with the skeleton distribution.)

For the most part, you can modify the code given to you as you like. However, you must observe the following constraints.

- **The code must be written in C**. You cannot use any C++ features such as classes, or the iostream `cin` and `cout` objects.
- The `storage.h` file should not be modified. This file declares the interface of the client library.
- The server must be buildable by running `make server` in the `src` directory. The server executable must be called `server`. (The directory layout of the code is described below.)
- The client library must be buildable by running `make libstorage.a` in the src directory. The client library must be called `libstorage.a`.

You're allowed to change any of the code (other than storage.h), add new source or header files, or modify the Makefiles (subject to the constraints above).

As a minimal test, you should make sure the test suites in the skeleton run and pass.

Useful Concepts

To complete this assignment you will need to become familiar with marshalling and network communication. This section gives you a brief introduction to these concepts. You can also find more information in the Course Reader.

Marshalling and parsing

A large aspect of this assignment involves transforming between a data structure in memory and a format that can be sent over the network. For example, an integer in memory might be represented as a sequence of 32 bits, but when you want to send it over the network, you might want to first convert it to its base-10 string representation. On the other hand, you may decide to store an integer in a more compact binary representation. This process of converting from and to a data structure in memory is called marshalling and unmarshalling, respectively. Often, unmarshalling is trickier to implement since you need to *parse* an arbitrary data stream to make sure it conforms to some expected structure.

You will need to perform marshalling and unmarshalling in at least two components:

- Reading the configuration file. The server needs to read parameter names and values from the configuration file. The format of the configuration file was defined here.
- Protocol. The server and client library must communicate over a network connection. Any requests or responses must be written to a string (or some binary representation) and read at the other end. Again, you must design the protocol such that it is unambiguous, preferably easy to parse, and possibly human readable to ease debugging.

The code given to you already includes some examples of parsing. For example, in `utils.c`, the `process_config_line()` function uses the `sscanf()` function to extract the two words in each configuration file line. It also uses the `atoi()` function to convert a string to an integer.

You can lookup how to use these and other functions in their man pages (e.g., enter `man atoi`). There are also pointers in the Course Reader and some of this material is covered in lectures.

Socket communication

The storage server and client communicate over a TCP network connection. Your program will use a *socket* abstraction in which the communicating end-points (i.e., an IP address and port) are represented as sockets.

A server that must listen on a given IP address and port, must perform a number of steps:

- First, it must create a socket. This is done using the `socket()` function.
- Next, the socket must be bound to an IP address and port. This is done using the `bind()` function.
- Then, the server tells the operating system to listen for incoming connections on this address and port. This is done using the `listen()` function.
- Finally, the server must wait for a new incoming connection. This is done using the `accept()` function.

In the final step above, the `accept()` function will return a *new* socket that uniquely represents the connection. At this point, the server can send and receive data on this socket using the `send()` and `recv()` functions, respectively.

To accept another connection from a client, the server must call the `accept()` function again. However, this poses a problem because the `accept()` function blocks the server (i.e., suspends execution) until a new connection arrives, and the `recv()` functions blocks the server until data arrives on that connection. It seems that a server must either choose to communication with a client on an existing connection, or wait for new connections, but not both. One way to address this problem is for the main server program to only listen for new connections, and then create a new process to manage each connection to a client. For this assignment, however, you may assume that there will be at most one client connected to the server at any given time, so you don't have to solve this problem just yet. But you may want to start to think about how to design your server to support multiple connections.

The client side is a little easier to implement.

- First, it needs to create a socket as in the server. Unsurprisingly, this is done using the `socket()` function.
- Then, it needs to make a connection to the server. This is done using the `connect()` function.

Note that there is no need to call `bind()` as `connect()` will automatically bind the socket to an available port on the client machine.

At this point, as with the server, the client can use the `send()` and `recv()` functions to communicate with the server.

The skeleton code given to you already includes a simple server and client library that implements all of the above steps. You should read it carefully and understand what's going on. Good places to start are `server.c` and `storage.c`.

Please remember that it's good practice when you use any system calls or library functions to check the return status and handle any errors that may occur. For example, every time you call `send()`, you should check if the send succeeded, and implement error handling logic.

You can find many excellent socket programming guides online, some of which are listed in the Course Reader. Also, use the man pages for details on how to use individual socket functions.

Set Up Your Code and Development Environment

Note: this section is different from Milestone 1. It includes instructions about SVN and unit tests that are required for Milestone 2.

Customize Your Environment

Here are some little tweaks to setup your development environment to your liking.

You can change the default editor that Subversion uses by setting the `EDITOR` environment variable. In the example below, you can replace pico with your favorite text editor.

```
> setenv EDITOR pico
```

Depending on how you login, you might not be able to use CTRL+C to terminate programs. You need to do this, for example, because the skeleton storage server given to you doesn't have a clean way to shut it down other than killing it. You might be able to get CTRL+C working by configuring the terminal properly:

```
> stty intr '^C'
```

If you don't like the default command line prompt, you can customize it by setting the `prompt` variable in CSH. Other shells will have different ways of changing the prompt. Here's an example.

```
> set prompt = "%n@%m:%~%# "
```

If you like the above customizations, you can put them in you shell's startup script (`.cshrc` for the CSH shell).

Get the Skeleton Code

Create a `ece297` directory in your home directory to put everything in.

```
> mkdir ~/ece297
> cd ~/ece297
> tar zxf /cad2/ece297s/public/storage-skeleton.tgz
```

It is important that you do not change the location or name of the `ece297` directory as it will be used to test your code.

Browse the Skeleton Code

The files in the distribution are shown in the tree below. You can also find doxygen documentation of the code in Doxygen Documentation.

```
storage
|-- Makefile
|-- doc
|   |-- Makefile
|   `-- doxygen.conf
|-- src
|   |-- Makefile
|   |-- default.conf
|   |-- client.c
|   |-- encrypt_passwd.c
|   |-- server.c
|   |-- storage.c
|   |-- storage.h
|   |-- utils.c
|   `-- utils.h
`-- test
    |-- Makefile
    |-- Makefile.common
    `-- a1-partial
        |-- conf-duplicateport.conf
        |-- conf-onetable.conf
        |-- conf-twotables.conf
        |-- conf-threetables.conf
        |-- Makefile
        |-- main.c
        `-- md5sum.check
```

There are subdirectories for the source code (under `src`), documentation (under `doc`), and automatic test suites (under `test`). Each directory contains a makefile which you should at some point examine and try to understand. You can find some resources on the make utility in the Course Reader.

The `src` directory contains a basic framework for you to implement your storage server and client library. Some of the notable files include:

- `storage.h` declares the client library functions such as `storage_get()` and `storage_set()` that you need to implement.
- `storage.c` contains incomplete implementations of the client library functions. However, it does include some code to communicate with the server which may be instructive.
- `server.c` includes an incomplete implementation of the storage server. As with `storage.c`, there is some socket programming code here that might be helpful.
- `client.c` includes a very simple sample client that interacts with the server using the client library.
- `default.conf` is a sample configuration file.
- `utils.h` and `utils.c` contain some utility functions that you may find useful.

The files above are well commented, and you should spend the time to try and understand the code.

The `doc` directory includes some files to generate Doxygen documentation of your source code. You can also commit your design documents or any other documentation files to the `doc` directory. This way, you can easily share these files among your team members.

The `test` directory is where you should add code to test your storage server and client library. The tests given to you use the Check unit test framework which you will learn more about later in the term. Grab the tests for M2 as outlined below and ignore the `a1-partial` directory.

It's a good idea to make sure all the given tests pass before submitting your assignment. You are also highly encouraged to run the `testsubmitece297s` script to make sure that all files are included in your submission and all the tests pass. Refer to the Test Submitting Assignment 2 section below to learn about how to run the `testsubmitece297s` script.

Setup the Subversion Repository

Now create a Subversion code repository in your team's shared directory and import the skeleton code into it. *Only one member of the team should do this step.*

```
> mkdir /groups/ECE297S/$TEAMID/svnroot
> svnadmin create /groups/ECE297S/$TEAMID/svnroot
> svn import -m "Initial checkin." ~/ece297/storage-skeleton file:///groups/ECE297S/$TEAMID/svnroot/trunk/storage
```

Note that you need to substitute `$TEAMID` in the lines above with the group identifier (e.g., `cd-000`) assigned to you on the group registration website.

The last command above imports the code into the Subversion repository under the `trunk` directory. This follows the Subversion convention of putting the main code under `trunk` to allow for branches and tags. You can learn about branches and tags in the Subversion manual referenced in the Course Reader.

The team member who creates the directory should then make sure the team has read/write permissions; otherwise, the other team members might get permission denied errors.

To set the proper read/write permissions, run the following command:

```
> chmod -R g+rw /groups/ECE297S/$TEAMID/svnroot
```

Note that in the commands above (and some below) you should replace `$TEAMID` with the ID assigned to your team. If you can't wait for an ID and want to try the Subversion parts of this handout right away, replace the repository directories in the examples with one in your home directory. However, once you have an ID, you'll have to redo these steps to setup the repository in the proper directory so everyone in your team can use it.

Now, let's delete the code you just imported and do a fresh checkout from the repository.

```
> rm -rf ~/ece297/storage-skeleton
> cd ~/ece297
> svn checkout file:///groups/ECE297S/$TEAMID/svnroot/trunk/storage
```

And finally, convince yourself that you really have checked out the code from the repository.

```
> cd ~/ece297/storage
> svn info
```

The `svn info` command will (not surprisingly) print out some information about where the repository is, who made the last modification, and so on.

Get Familiar with Subversion

Subversion is a source code version control system. It manages the source files of a software package so that multiple programmers may work simultaneously. Each programmer has a private local copy of the source tree (or part of it) and makes modifications independently. Subversion attempts to merge multiple people's modifications and highlight potential conflicts. Since you'll be working in teams, this functionality is very useful.

We've already used Subversion to import and checkout code from a repository above. This section will demonstrate some of the common Subversion features.

View file modifications

Edit the file src/server.c, and add a comment with your name in it.

You can use the Subversion `diff` command to show you what changes you've made.

```
> cd ~/ece297/storage/src
> svn diff server.c
Index: server.c
===================================================================
--- server.c    (revision 1)
+++ server.c    (working copy)
@@ -45,6 +45,8 @@
 /**
  * @brief Start the storage server.
  *
+ * @author John Smith
+ *
  * This is the main entry point for the storage server.  It reads the
  * configuration file, starts listening on a port, and processes
  * commands from clients.
```

The above listing shows that two lines were added (note the plus signs at the left), and also shows some of the code surrounding the new lines.

View file status

If you're happy with the changes you've made you should commit them to the repository so the other members of your team can get your new code. Before doing that, let's use the Subversion `status` command to see which files we've modified.

```
> svn status
?      server
?      libstorage.a
M      server.c
```

This shows that `server.c` has been modified (the `M` prefix). It also indicates that `server` and `libstorage.a` exist in your local directory but are not in the repository (the `?` prefix). This is fine because `server` and `libstorage.a` are binary files that are generated as part of the build process. As a general rule, you should not add generated or binary files to the repository.

Commit files

Now let's commit the changes you made to server.c.

```
> svn commit
```

The Subversion `commit` command will open a text editor for you to enter a description of the changes you're committing. It's a good idea to always write a reasonable commit description in case you need to review the history of a file. Enter your description at the top of the file and close your text editor. The changes to `server.c` will now be saved in the repository.

View file commit history

You can see the history of commits to a file with the Subversion `log` command.

```
> svn log server.c
------------------------------------------------------------------------
r2 | jsmith | 2009-01-01 10:16:10 -0500 (Thu, 01 Jan 2009) | 2 lines

Added myself as the author of the main() function.


------------------------------------------------------------------------
r1 | jsmith | 2009-01-01 10:01:08 -0500 (Thu, 01 Jan 2009) | 101 lines

Initial checkin.
------------------------------------------------------------------------
```

Notice that in addition to the commit description, you can see who committed each version of the file, and at what time.

Retrieve updates

Along with `commit`, the command you're most likely to use is `update`. The Subversion `update` command retrieves any changes committed by your team members and applies them to your local copy.

```
> svn update
At revision 2.
```

Since no one else has committed anything, Subversion simply confirms that your local files contains all modifications up to version 2 of the repository. (Each commit to any file increments the repository's version.)

You should get into the habit of doing an `update` often so your code is kept up-to-date with commits from your team members.

Revert file modifications

Suppose you make some changes that breaks your code. Go and delete the first half of the `storage.h` file. If you try to build the code now it will fail.

```
> make clean build
```

Play with the `svn status` and `svn diff` commands to see how the file was changed. Now let's use the Subversion `revert` command to discard any changes you've made to a file, and replace the file with the last committed version.

```
> svn revert storage.h
```

Look at the `storage.h` to see that the deleted lines are there again, and try to build the code again.

Add new files

You'll notice that the skeleton code doesn't have a README file. It is good practice to include a README file with any source code, so let's create one.

```
> pico ~/ece297/storage/README
```

(Of course you can use any text editor you want.) Include in the README file a brief description, in your own words, of what's in the `src`, `doc`, and `test` directories. Also describe the steps to build and run the storage server. When you're done, commit the file to the repository.

```
> cd ~/ece297/storage
> svn commit README
svn: Commit failed (details follow):
svn: '/nfs/ugsparcs/j-l/j-l/jsmith/ece297/storage/README' is not under version control
```

Subversion gives an error because you need to add new files to the repository before committing them. Use the Subversion `add` command to add the README file, and then check it's status.

```
> svn add README
A         README

> svn status
A      README
```

Notice that the `svn status` command shows that the README file is scheduled to be added to the repository (the `A` prefix). This doesn't mean the file is in the repository yet. You still need to `commit` the file.

```
> svn commit README
```

Again, enter a reasonable description for this commit when prompted.

Please remember that you should not add any generated files (such as the `libstorage.a` file) to the repository. This increases the size of the repository and interferes with the build process for your team members.

You can find links to more information about Subversion in the Course Reader.

Build and Run the Code

Now let's build the skeleton storage server and client library.

```
> cd ~/ece297/storage/src
> make
```

If there are no errors, you should see something like this:

```
cc -g -Wall -c storage.c -o storage.o
cc -g -Wall -c utils.c -o utils.o
ar rcs libstorage.a storage.o utils.o
cc -g -Wall -c server.c -o server.o
cc -g -Wall server.o utils.o -o server
cc -g -Wall -c client.c -o client.o
cc -g -Wall client.o libstorage.a -o client
```

The first two commands compile some C source files, and the third one *archives* them into a library file. The last command links a number of object files to create the server executable.

Try running the server now.

```
> ./server
Usage ./server <config_file>
```

Oops, you need to specify a configuration file. Let's try again.

```
> ./server default.conf
Server on localhost:1111
```

The skeleton server listens on the specified port (port 1111 above). For now, it simply reads a line of input on a connection and sends it back out on the connection. You can see what's going on by using a *fake* client. Keep the server running in one window, and enter the following in another terminal.

```
> nc localhost 1111
hello
hello
```

`nc` is the netcat program and is a powerful utility to add to your developer's arsenal. Here you used it to make a connection to the server. Now, any text you enter to netcat will be sent to your server. In the above sample, the user entered the first`hello`, and the second one was what was sent back by the server. Also, look back at the first window where the server is running and see what's going on there.

You can kill the netcat program and server when you're done.

We have also provided a simple client that you can build and run.

```
> ./client
```

The client connects to the server and invokes a series of storage operations: `storage_auth()`, `storage_get()`, `storage_set()` and finally `storage_disconnect()`. Note that the initial skeleton code does not fully implement the server and client libraries. As a result, the client you run may not be fully functional - at least in the beginning.

Run Some Tests

Some of the marking tests are available at `/cad2/ece297s/public/assignment2/a2-partial.tgz`. You can try running them as follows.

```
> cd ~/ece297/storage/test
> tar zxf /cad2/ece297s/public/assignment2/a2-partial.tgz
> cd a2-partial
> make clean run
```

The first test `test_sanity_filemod` just makes sure the `storage.h` file hasn't been modified. This test currently passes and should always pass as you should not modify the `storage.h` file.

The following tests: `test_config_onetable`, `test_config_twotables`, `test_config_threetables`, and `test_conn_basic` start/connect to the server with different but valid parameters. These tests currently pass and should always pass.

Since the client library and storage server have not been fully implemented, the rest of the tests fail and you should see something like this:

```
Running suite(s): a2-partial
45%: Checks: 11, Failures: 6, Errors: 0
main.c:162:P:sanity:test_sanity_filemod:0: Passed
main.c:190:P:config:test_config_onetable:0: Passed
main.c:197:P:config:test_config_twotables:0: Passed
main.c:204:P:config:test_config_threetables:0: Passed
main.c:215:F:config:test_config_duplicateport:0: Server should not run with duplicate port numbers in the config file.
main.c:238:P:conn:test_conn_basic:0: Passed
main.c:245:F:conninvalid:test_conninvalid_disconnectinvalidparam:0: storage_disconnect with invalid param should fail.
main.c:264:F:getmissing:test_getmissing_missingtable:0: storage_get with missing table should fail.
main.c:283:F:setinvalid:test_setinvalid_badkey:0: storage_set with bad key name should fail.
main.c:313:F:oneserver:test_oneserver_onetable:0: Got wrong value.
main.c:360:F:restartclient:test_restartclient_onetable:0: Got wrong value.
```

Before you submit your code, you should at the very least make sure that the above tests pass:

```
> cd ~/ece297/storage/test/
> make run TESTS=a2-partial
```

Generate Code Documentation

When looking through the code, you might have noticed some funny looking tags in the comments. These comments are formatted in such a way that the Doxygen program can automatically extract them and generate code documentation. Let's do that now.

```
> cd ~/ece297/storage/doc
> make
```

This will generate documentation in two formats: HTML in `doxygen/html` and Latex in `doxygen/latex`. You can browse the HTML documentation by opening the `doxygen/html/index.html` file in your browser. And you can view the Latex documentation by opening the `doxygen/latex/refman.pdf` file in a PDF viewer. (Latex is a powerful document typesetting tool but has a steep learning curve, and you aren't expected to learn to use it for this course.)

You are expected to comment your code using the Doxygen commenting format. This format is mostly self-explanatory from looking at the examples in the code, but you can also refer to the Doxygen manual referenced in the Course Reader.

Please review and improve the current documentation as well. Your code documentation will be graded for language, technical correctness and completeness.

Suggested Development Plan

Working in teams makes it especially important that you identify the tasks required to complete the assignment, and develop a plan on who will work on each task and when each task must be completed.

Here are some of the major tasks involved in this assignment.

- Design considerations
- Initialization
  - Configuration file
- Client/server communication protocol
  - Connection establishment and maintenance
- Implementation of storage_get() and storage_set() functions
  - Data structure for storing tables and records in memory
- Integration and testing
- Performance evaluation and experimentation
- Result analysis and interpretation
- Evaluation plan and workload preparation
- Design document

It is up to you whether you want to divide the tasks as above. Think carefully about how much work each task may take, which tasks build on one another or can be worked on simultaneously, and the skills of the members in your team whom you allocate to each task.

Design Considerations

In this section we summarize key design decisions you will have to make when designing your storage server. These design decisions are:

- What are the data structure to use for storing records and tables in main memory and how are records indexed and retrieved?
- What is the communication protocol between the client library and storage server?
- How is the connection state between client and server represented? Is the connection reused for multiple get and set requests from the client? What are the implications of your design when multiple clients interact with the server at the same time?
- How do the server and the client library handle failures? For example, what happens if there's an error in the configuration file, what if the server is not running, what if an invalid table name is given?

To simplify the assignment, you may assume that clients and servers don't crash in the middle of an operation. For example, you may assume that a client won't send a partial request to a server and leave the server waiting for the remainder of the request. However, note that in reality, on the Internet, for example, this assumption would be detrimental.

Be prepared to argue about why you came to your design decision and discuss the pros and cons of the decision, such as the ease of implementation, robustness to failures, performance implications, and so on. Note, for many decisions you will have to balance trade-offs. Try to understand what they are.

Deliverables

Please note, there are **two deadlines** for this assignment. We first ask you to hand in your design documents, later followed by the code you developed.

Design Document

By the **design document deadline,** each team must hand in the **M2 Design Document** that includes the following content.

- Cover Page
  - Includes document title, team identification, and submission date
- Table of Contents
  - Provides a listing of the document contents
- Introduction
  - Uses a situation-problem-solution structure to contextualize your project. The Introduction should establish the purpose of the document and answer the following: Who will use your server? Why? What currently exists? Where is the gap or opportunity? How does your design fill that need? As part of this discussion, in at least one well-researched paragraph, identify the potential user, and explain why you have chosen this user. How does, or how will, your server meet this user's needs? What benefits does it provide? At this early stage in the project, your choice may be general (a type of industry or business), but as you move toward the next milestone, we expect that your choice will become more specific. Include at least three respectable research sources in this discussion.
- Software Architecture
  - UML Component Diagram, providing an overview of your software system
  - UML Sequence Diagram, showing the interactions between the server and the client
  - A short written explanation of each diagram
- System Requirements
  - Functions, objectives, and constraints
  - Functional requirements and constraints must be introduced with the auxiliary *shall,* and constraints must be defined in verifiable terms.
- Design Decisions
  - Includes all the important M2 design decisions
  - As a guide, view Design Considerations above in this file.
  - Make a case for these decisions using elements of argument. Identify any alternatives you considered, and explain the reasoning behind your choices.
- Conclusion
  - A short "summing up" statement returning your reader to the document's purpose.
- List of References
  - Use IEEE format to document your research sources.
- Appendices
  - Appendices are optional.
  - Include important information, including important figures and tables, in the body of your report. Include "optional reading" in appendices.
- Performance Evaluation Report
  - This report is due with your M2 code (February 16). The instructions are stated above in this file.

The main body of this document may contain no more than seven (7) pages. These seven pages include written text, tables, diagrams and other graphical material, but do not include the cover page, table of contents, list or references, and appendices. The report may contain no more than two (2) pages of appendices. You are expected to use single-spaced block paragraphs, standard font, for your written text, and you must label all tables and figures.

Sections of each design document must have individual author identified by name and student number. In cases where revision/editing/proofreading are performed by teammates other than the original author, please indicate this information, providing names and student numbers.

The text must be submitted to Turnitin.com. For information on registering with Turnitin.com, please view Turnitin Registration Information here.

Your Communication Instructor will use this grading rubric to grade the assignment.

Code

By the **software development deadline**, you must hand in your software artifacts that implement the coding requirements and include at least the following content.

- Your code (you must make no change to `storage.h` file in the skeleton code).
- Your code documentation (i.e., the Doxygen output). **You must also submit the Doxygen PDF through Turnitin.**
- Performance Evaluation Report: One (1) page of text (maximum), not including graphics, must be submitted along with your code (using `submitece297s` script).

You can find instructions to submit the code and Doxygen documentation in the Code Submission Instructions page.

In addition, upon completion and submission of the code, you must submit the following to your Project Manager through Turnitin.com:

- Performance Evaluation Report: One (1) page of text (maximum), not including graphics.

back to top