Exercise 1: Client/Server application

Matthew Leon Dailis

March 9, 2018

Contents

1 Definitions of Components

The central component of this system is the server. It contains the database, and provides a message-queue interface for clients to connect. There are arbitrarily many clients. All components are running on the same machine.

2 Communication protocol

Communication between components is via message queues. Every process opens its own message queue for receiving messages. To send a message to another process, the sending process opens the message queue of the receiving process, writes to it, and closes it.

3 Server

The server is multithreaded and stateless (which is to say, connectionless). It controls access to a shared database. Every client request creates a new worker thread. The server guarantees atomic operations to the database by using mutual exclusion between the worker threads.

4 Naming

The server's message queue will be called *"/SERVER"*. Each client can choose its own unique name for its message queue and provide that name to the server. (See Section 5).

5 Message Design

There is a single message type for both requests and responses. It has 5 fields and is structured as follows:

Field	Size
char type/status	1 byte
int key/count	4 bytes
char value1[256]	256 bytes
float value2	4 bytes
char return_queue[35]	35 bytes
Total	**300 bytes**

1. *char* **type/status**

   (a) Requests

   The type indicates the requested operation for the server. The type indicates which other fields are relevant to this message.

#	command	Explanation
0	init	This is the call to make sure that a server is up and running. All other fields are ignored.
1	set_value	Requests to store **value1** and **value2** associated with a new key, **key**.
2	get_value	Retreive the information associated with **key**.
3	modify_value	Store **value1** and **value2** under the existing key, **key**.
4	delete_key	Delete information associated with key, **key** and remove the key from the database
5	num_items	Request the number of items in the database

   (b) RESPONSE: This is the status of the request

   - For *init*, *set_value*, *modify_value*, and *delete_key*, this is the only relevant field in the message.

0	success
-1	error

   - For *get_value*, if the **status** is 0, the requested values are stored in the **value1** and **value2** fields of the message.
   - For *num_items*, if the **status** is 0, the returned count is stored in the **key/count** field of the message.

2. *int* **key/count**

   (a) Request: this is the key of the element requested

   (b) Response: (only applicable to *num_items*) The number of items in the database

3. *char* **value1** [256]

   This represents the string value that is stored or will be stored in the database.

4. *float* **value2**

   This represents the float value that is stored or will be stored in the database.

5. *char* **return_queue** [35]

   (a) Request: the message queue to which the server should write its response

   (b) Response: ignored

6 My protocol

The server is the passive entity. Every interaction consists of a **request** sent by the client to the message queue of the server. The request contains the name of the client's queue, to which the server writes the **response**.

- The first message sent by the client should be *init* - this ensures that the server is ready to receive other messages.
- Since the server is stateless, there is no need to end a connection.

7 Security

There is currently a loophole: Sending the server a message with the server's own queue in the **return_queue** field forces the server into an infinite loop.

As far as data, there is no security in this design. Any client can ask for any information from the server.

8 Implementation

This system is implemented using C99 using POSIX message queues.

8.1 Messages module

To reuse the code for sending and receiving messages, there is the messages library, which defines the absract datatype *Connection*, which is a wrapper over the *mqd_t* type. The library provides the following functions:

Return type	Name	Arguments	Comment
Connection	create_connection_read	char *name	Given the string name of a message queue, create and open that queue read only.
Connection	open_connection_write	char *name	Given the string name of an existing message queue, and open it for writing.
int	send_message	Connection, message	Send the message to the given destination.
int	receive message	Connection, buffer	Block until you receive a message in the given message queue.

8.2 Server

The server is an infinite loop that receives messages from the *"/SERVER"* message queue. On every received message, it starts a new thread. The first thing each thread does is atomically copy the received message and release the mutex, so the server can continue processing messages. This message is then categorized by its **type** field, and the relevant processing function is invoked.

8.3 Database

The **database.h** header file provides a list of general purpose functions on an abstract datatype called *Database*. It assumes data is stored by a unique key, but assumes nothing about the data itself (arbitrary binary data, with a provided size). There are two implementations of this database.

8.3.1 Treemap

The treemap is a binary search tree sorted by the keys stored inside of it, and supporting operations to insert and delete keys.

8.3.2 TODO Filemap

8.4 Client

The client side abstracts access to the remote database using **keys.h**.

8.5 How to compile

Invoke *make* in the topmost directory to compile the entire project.