KV store

TABLE OF CONTENTS

In this part, you will implement a basic key-value (KV) store by introducing two new RPCs, `PUT` and `GET`. You may find it helpful to follow the steps you used to implement the `ECHO` RPC. Keep in mind that the only files you should be editing in this part are `kv_store.x`, `server.c`, and `kv_store_client.c`.

Protocol specification

In `kv_store.x`, define two new RPCs: `PUT` and `GET`.

- `PUT` will need to take in a key and value, each of which is a variable-length array of bytes (`char`'s). To take in multiple arguments, you will need to wrap all of your arguments in a single XDR structure and have that be the argument to your RPC.
- `GET` will need to take in a key (a variable-length `char` array) and return a value (also a variable-length `char` array).
- Keep in mind that keys and values are not necessarily NULL terminated.

Note: You may want to use type definitions to simplify your code. We have already defined the following type in `kv_store.x` for you to use for your keys and values:

```
typedef char buf <>;
```

As you can find in `kv_store.h`, this is compiled to the following C code:

```
typedef struct {
        u_int buf_len;
```

```
        char *buf_val;
} buf;
```

Data structures

You will need to add state to your server to store key/value pairs. Since C doesn't have many built in data structures, we have provided you with GLib in `server.c`.

You may define state using global variables and initialize it in the `main` function. Since the RPC server is not multi-threaded, you do not need to worry about synchronization.

> You will need to use `GHashTable` and `GBytes` to store the "database" as well as the keys/values.
>
> Below is a snippet of example code to show how you might use these data structures:

```
GHashTable *ht;

void init() {
  ht = g_hash_table_new(g_bytes_hash, g_bytes_equal);
}

void add() {
  GBytes *key = g_bytes_new("key", strlen("key"));
  GBytes *value = g_bytes_new("value", strlen("value"));
  g_hash_table_insert(ht, key, value);
}

void lookup() {
  GBytes *key = g_bytes_new("key", strlen("key"));
  GBytes *value = g_hash_table_lookup(ht, key);

  g_bytes_unref(key);

  if (value != NULL) {
    long unsigned int len;
    const char *data = g_bytes_get_data(value, &len); /* Sets len = 5. */
    printf("%.*s\n", (int) len, data); /* Outputs first `len` characters of `data` ("value"). */
```

```
  }
}
```

RPC implementation

Implement the server-side stubs for the `PUT` and `GET` RPCs in `server.c`. You can find the relevant function signatures in `kv_store.h`.

Your stubs should look something like this, though the types in question are dependent on how you defined your protocols in `kv_store.x`:

```
void *put_1_svc(put_request *argp, struct svc_req *rqstp) {
  static void *result;

  /* TODO */

  return &result;
}

buf *get_1_svc(buf *argp, struct svc_req *rqstp) {
  static buf result;

  /* TODO */

  return &result;
}
```

If you `malloc` anything, you should free it the next time the stub is called. This is because the `result`s are static, so they are preserved between function calls. For example, if we wanted to `malloc` the `buf` `result` in `get_1_svc`, we would write:

```
free(result.buf_val);
result.buf_val = malloc(5);
```

For `GET`, if there is no value corresponding to the provided key, return a `buf` with length 0.

WARNING

> Make sure to return a valid pointer from `put_1_svc` as shown in the code above. Due to some weird internals of `rpcgen`, returning a NULL pointer may cause your code to hang.

Once you are done, you should be able to put key/value pairs into your store by modifying `kv_store_client.c` to call the `PUT` and `GET` client stubs. Test your implementation using the instructions from the Usage section.

A working implementation of `ECHO`, `PUT`, and `GET` should give you a full score on the autograder.
