CS 262 Assignment 1 Design*

Protobufs: David Ding and Lily Tsai Custom: Dan Fu and Ross Rheingans-Yoo

March 2, 2016

1 High Level Design

We have a single server and multiple clients that communicates with the server via RPC calls. The server stores all the persistent state in our system in a SQL database. This database has tables for users, groups, user group pairs (storing a pair of an user id and a group id which indicates that the user belongs to the group), virtual groups (to be explained latter), and messages. The server does not store any state relating to a specific client-server session; all such state (the logged in user, messages that have already been fetched) is stored by the client, and this state vanishes when the client finishes running.

Users in this system have an unique user id and an unique username. Logging in consists of checking to make sure the specified username exists in the database. Groups in this system have an unique group id and an unique group name. The group id is always odd; this design decision will be explained later. Each group could have zero or more users, and membership in groups is stored in the user group pairs table. Group membership is unaffected by changes in group name.

A message in this system consists of a `to_id`, a `from_id`, a message id (`m_id`), and the actual contents of the message. To simplify handling of user and group messages, we elected to have all messages be group messages. The `from_id` is the user id of the user who sent the message, the `to_id` is the group id of the group to whom the message is sent. and the message id is a 64 bit integer that always increments. To handle messages between pairs of users, we create a *virtual group* consisting of those two users alone. These virtual groups have a group id (which is always even) along with the two ids of the users that this group contains. Messages sent from user A to user B will be stored as a message from user A to vgroup(A, B). Virtual groups are created lazily: whenever A wants to send a message to B, if the virtual group does not exist yet, a new virtual group consisting of A and B will be created. Virtual groups can be distinguished from real groups in that their ids are even, whereas those of real groups are odd. Virtual groups have no name and cannot be listed by the client. In all other respects, virtual groups are identical to real groups.

All required commands of the chat applications can be implemented as SQL queries. For example, group and user listings is a simple `SELECT` query, creating new users or groups is an `INSERT` query, deleting users or groups is a `DELETE` query, adding a user to a group is an `INSERT` query to the user group pairs table, and changing the group name is an `UPDATE` query. Getting the members of a group with given group name is a `SELECT` query on the `JOIN` of the users table, groups table, and user group pairs table. Fetching messages a certain user can see is a `SELECT` query on the `JOIN` of the user group pairs table and the messages table (i.e., messages an user can see is the same as messages addressed to a group that the user belongs to). Finally, sending messages consists of an `INSERT` query to the messages table, with a `SELECT` query to the virtual groups table for individual messages and possibly an `INSERT` query if the user does not exist.

Both sets of protocols implement RPC calls with the following interface:

```
void client_run(port)
error send_message(self, from_name, dest_id, msg):
    '''Returns None on success, or an error string'''
message_list fetch_messages(self, to_id, checkpoint=0):
    '''Returns a list of Messages addressed to to_id'''
group create_group(self, g_name=""):
    '''Returns a Group object with g_name=g_name'''
user create_account(self, username=""):
```

*Code: https://github.com/tslilyai/cs262chat

```
    '''Returns a User object with username=username'''
error remove_account(self, username=""):
    '''Returns None or error string'''
error edit_group_name(self, old_name="", new_name=""):
    '''Returns None or error string'''
error remove_group_member(self, g_name="", membername=""):
    '''Returns None or error string'''
error add_group_member(self, g_name="", membername=""):
    '''Returns None or error string'''
group_list list_groups(self, pattern="%"):
    '''Returns a list of Group objects (empty list if none match)'''
user_list list_accounts(self, pattern="%"):
    '''Returns a list of User objects (empty list if none match)'''
user_list list_group_members(self, g_name=""):
    '''Returns a list of User objects (empty list if none)''
```

The client operates as follows. First, the user must login on the client (or create a new account). This consists of the user specifying an username, and the client checking with the server (via the `ls_users` RPC call) to see if the user exists. After logging in, the client starts a thread which polls the server for new messages for the client via the `get_messages` RPC call. Because the user is logged in, the client has an user id to pass to the server. To prevent the user from fetching the same messages repeatedly, the client stores the message id of the last message returned by the server (a value we will call the *checkpoint*). In the `get_messages` call, the client will pass in the checkpoint, and the server will only return messages with message id greater than checkpoint. The other commands that the client supports are the following:

```
*must be logged in
 > login [username]
 > mk-user [username]
 > delete-account*
 > ls-groups [pattern (optional)]*
 > ls-users [pattern (optional)]*
 > ls-group-members [groupname]*
 > mk-group [groupname]*
 > add-group-member [groupname] [username]*
 > remove-group-member [groupname] [username]*
 > talk-with [group/user] [group or user name]*
 > logout*
```

All other commands are straightforward; for example, if the client wants to list all groups whose names matches a pattern the client just has to make a `ls_groups` RPC call. Figuring out the RPC-calls for the other commands is left as an exercise for the reader.

The final piece of the puzzle is the implementation of the RPC layer. We have two different implementations, a gRPC implementation that uses Protocol Buffers for the wire protocol, and a custom wire protocol for RPC calls; both are described below.

2 Protobuf Protocol

First, we have a gRPC implementation, which uses Google's gRPC framework[1] to automatically generate stubs that the client can call to make RPC call (these stubs are responsible for sending the requisite network packets) and a server that listens for these RPC-calls and delegates the tasks to the database layer. Google's gRPC uses Protocol Buffers to serialize messages and sends them using the HTTP/2 protocol with method POST. Protocol buffers serializes messages as a key value pair, where the key encodes the field name and type, and the value is the actual data for that field. To save space, the key uses an id number that is specified by the programmer. Protocol buffers can robustly handle different versions of the protocol if programmers never delete fields from the .proto file and instead always assign new id numbers. When parsing protocol buffer binary data, the runtime will simply ignore

[1] http://www.grpc.io/docs/guides/wire.html

fields it does not understand, and it is up to the programmer to ensure that all necessary fields are included in the message.

Both the protocol buffer messages and the RPC interface is specified in a .proto file that can be compiled with the Protocol Buffer Compiler:

```
service ChatApp {
    rpc rpc_send_message(CMessage) returns (Response) {}
    rpc rpc_get_messages(User) returns (stream CMessage) {}
    rpc rpc_create_conversation(UserPair) returns (Group) {}
    rpc rpc_create_group(Group) returns (Group) {}
    rpc rpc_create_account(User) returns (User) {}
    rpc rpc_remove_account(User) returns (Response) {}
    rpc rpc_edit_group_name(Group) returns (Response) {}
    rpc rpc_remove_group_member(Group) returns (Response) {}
    rpc rpc_add_group_member(Group) returns (Response) {}
    rpc rpc_list_group_members(Group) returns (stream User) {}
    rpc rpc_list_groups(Pattern) returns (stream Group) {}
    rpc rpc_list_accounts(Pattern) returns (stream User) {}
}
message CMessage {
     int64 m_id = 1;
     int64 to_id = 2;
     string from_name = 3;
     string msg = 4;
}
message User {
     int64 u_id = 1;
     string username = 2;
     string password = 3;
     int64 checkpoint = 4;
}
message UserPair {
    string username1 = 1;
    string username2 = 2;
}
message Group {
     int64 g_id = 1;
     string g_name = 2;
     string edit_member_name = 3;
     string new_name = 4;
}
message Response {
     int32 errno = 1;
     string msg = 2;
}
message Pattern {
     string pattern = 1;
}
```

Below, in the Comparison section, we evaluate and discuss the pros and cons of using Protocol Buffers and gRPC.

3 Custom Protocol

Our primary concern for our custom protocol was making it maintainable for future developers. To that end, we elected to abstract away annoying details where possible and make the design of the protocol simple without losing expressiveness of the protocol. At the same time, for the sake of the educational value of this assignment, we tried to avoid design choices that too closely mimicked other well-known communication protocols where possible. For

example, making our custom protocol just be REST would have made it very easy for future developers to grasp how our protocol worked - but would provide relatively little educational value.

The first question we had to answer was how to send bits over the wire. We could have chosen either to write our own library to handle wire communications, which would have involved considerable socket programming, or we could use an existing library. We chose to communicate using HTTP, for two main reasons - we could rely on a well-known, well-tested Python library instead of rolling our own and introducing the potential for countless bugs and buffer overflows; and the learning curve for future developers would be much lower than anything else we could introduce. On the server side, it takes about five lines of code to get a server running, and the abstractions for request handling are well-documented and easily accessible. On the client side, it allows us to use the Requests library, which has a very simple interface and is also well-documented. We are fairly confident that anything we could have come up with would not have had as simple an interface, would not have been as well-tested, and would not have had documentation as complete as those of the libraries we used.

Using HTTP introduces the question of what verbs to use, and what information should be conveyed through the requested URL. We decided to use GET for read requests (`fetch_messages`, `list_groups`, etc) and POST for the other requests. This conforms to how GET and POST are typically used in HTTP. We decided to pass parameters to GET requests using query strings, which is standard practice since GET requests do not come with a payload. However, we decided not to convey any information to the server through the path itself. The primary reason we made this design choice was to de-couple the functions on the client side from functions on the server side - if the server does not have to worry about the path, it can ignore the path during processing.

The next question is what information to actually send in the payloads - in other words, how to encode Python objects into a wire format. Again, we could have written our own library to do this, but using an existing one lowers the learning curve for future developers and decreases the amount of new, potentially buggy code we needed to write. We set out with the objective of not using JSON for our encoding protocol to avoid looking too much like existing protocols, but we soon found that most of Python's other encoding libraries had critical bugs that eliminated them from consideration. Pickle, for example, has critical bugs that would have allowed bad actors to make the server execute arbitrary code. We found that our choices basically came down to XML or JSON. Given that choice, JSON is the clear winner - it takes less bandwidth, is human-readable, and is very well-known.

In our requests, we sent a JSON object that had at least two and up to six properties. Each object contains a version field that tells which version of the protocol we are using and an action field that tells what method we are invoking. In this first version, there is a one-to-one mapping between the client methods and the action names, but we purposefully gave the actions different names from the client methods to de-couple the client methods and action names. We allow four other properties to be passed in the JSON object as well - actor, setting, target, and value. We choose to only allow four properties so that we can de-couple the objects being passed from the methods on the client or server. Furthermore, only having four properties encourages future developers to make new actions simple - if actions can only have four pieces of information, there is only so much complexity they can have. This encourages simple, well-defined actions that do one thing.

In summary, our communication protocol is basically an HTTP and JSON protocol. We send information in JSON-encoded protocols where possible, and we try to use as few HTTP-specific features as possible. We came to this design by prioritizing the re-use of existing libraries to make our codebase more robust and reduce the learning curve for future developers.

4 Comparison

4.1 Protocol Buffers

The chief advantage of Google's gRPC framework is that it makes writing RPC Services simple. After specifying the messages and the RPC interface, the protocol buffer compiler generated all the message serialization/deserialization methods and network communication logic for us, so for the purposes for implementing the server and client, we can almost pretend that we are making ordinary function calls.

One disadvantage of the gRPC framework is its newness, meaning that support is low and bugs relatively plentiful. The documentation online is quite helpful when we were implementing the client and server, but it was almost useless for installation. The installation instructions were scattered across multiple repositories, and dependencies were not very clear. To install the gRPC tools such as the compiler and gRPC runtime, we had to compile from source multiple times, play trial-and-error to find necessary dependencies, and figure out quirks of the build system; we summarized the steps needed to install the framework in our `install.sh` script, which hopefully installs all necessary dependencies and builds everything in the right order. The gRPC framework is also not very portable. For instance, we gave up on installing gRPC for Macs, because as it turns out, in the Mac OS X distribution for the gRPC runtime,

the code attempts to dynamically load a binary file compiled for Linux, and quite shockingly, that file was unable to run on the Mac OS X operating system. It comes without saying that gRPC support for Windows is even poorer. Right now, the easiest system to install gRPC for is Ubuntu, and even for Ubuntu, the installation process is lengthy and error prone. We expect as gRPC matures, the installation process will become more straightforward.

The newness of the gRPC framework also implies that it probably has a few bugs. To Google's credit, we were unable to find substantial bugs with gRPC, but we did catch this amusing compiler bug, in which the compiler generated a Python function that looks like

```
def beta_create_ChatApp_stub(channel, host=None, metadata_transformer=None, pool=None,
        pool_size=None):
  import service_pb2
  import service_pb2
  import service_pb2
   ...
  import service_pb2
  import service_pb2
  import service_pb2
  request_serializers = {
   ...
  }
```

Fortunately, because of the way the Python interpreter caches imports, this bug is innocuous, but it leads us to wonder if there are other less amusing quirks with the compiler.

While the simplicity of protocol buffers and gRPC is appealing, this abstraction away from the implementation details sometimes led us to blindly follow the tutorial code when implementing the gRPC server and client. For example, the compiler's naming schemes are slightly unintuitive (opening a client socket requires calling `implementations.insecure_channel` in the `grpc.beta` module); we reverted to using the gRPC tutorials to quickly detect which variables and names corresponded to the appropriate classes and functions we needed to call.

gRPC and protocol buffers also impose several restrictions on how servers and clients can communicate. Each proto request, for example, can send only one argument: a protobuf message object, or a stream of protobuf message objects of the same type. Similarly, each proto response returns either a protobuf message object, or a stream of the same-type protobuf message objects. This restriction required us to create many different proto message objects or add additional fields to message objects simply to support one client method. For example, in order to support conversations between two users, we could not simply pass in two `User` message objects. Instead, we created a `UserPair` object specifically for this purpose. To support group editing, we had to add an `edit_member_name` field to the `Group` message object, which felt out of place as part of the `Group` object abstraction.

4.2 HTTP/JSON Protocol

In contrast to Google's gRPC framework, there are almost no installation problems for our custom protocol, since it only relies on a few libraries that have been well-tested across multiple systems. In addition, support for both the server and client libraries is plentiful, since HTTP is one of the most common network protocols in use today.

The biggest difference between the custom protocol and the gRPC protocol is the inability to pass complex objects from the client to the server and act as if the client and server were on the same machine. For a simple chat application, this is not a problem - developers must come up with small, well-defined actions that do not require the passing of complicated objects to succeed. For such a simple chat application, the limitations of the HTTP/JSON protocol is actually an advantage - it forced us to define a simple interface between the client and server, and saved us from any temptations that could have arisen from having a too expressive communication interface. That being said, for something like a remote desktop application, the disadvantages of the HTTP/JSON protocol would not render it a viable choice.

4.3 Packet Sizes

We also measured the protocol buffers' packet sizes received and sent by the server over the duration of a typical client-server interaction, which included, among other things, account creation, user conversations, and group listings:

```
> mk-user fding
> login fding
```

```
> ls-users
> ls-groups
> mk-group cs262
> logout
> mk-user lily
> ls-users
> ls-groups cs%
> add-member cs262 lily
> add-member cs262 fding
> delete-account
> logout
> login fding
> talk-with group cs262
    hi i'm david!
    i like distributed systems!
    goodbye
```

It is clear (from Table 1 and Table 2 below) that the wire protocol using protocol buffers sends packets of much smaller sizes; this is as expected, because a protocol buffer message sends only a particular field's number (as opposed to the field's name), which is then decoded using the .proto file. XML or JSON objects, however, have no such compression techniques.

Protocol	Messages sent by server	Messages sent by client
Protobuf	1357	1429
Custom	2254	3150

Table 1: Total size of packets sent by server and client in a session.

Protocol	Messages sent by server	Messages sent by client
Protobuf	73, 93, 111, 87, 30, 13, 85, 13, 45, 13, 78, 13, 83, 13, 48, 13, 45, 13, 38, 9, 13, 84, 13, 49, 13, 19, 30, 13, 91, 9, 13, 80, 13, 51, 9, 13, 0	15, 13, 9, 13, 105, 34, 14, 39, 14, 23, 11, 14, 23, 26, 35, 31, 24, 24, 26, 11, 14, 27, 32, 26, 29, 11, 14, 23, 11, 14, 22, 11, 14, 23, 26, 35, 31, 24, 24, 26, 22, 11, 13, 14, 23, 11, 14, 25, 11, 14, 23, 11, 13, 14, 23, 11, 14, 25, 11, 14, 25, 11, 13, 14, 25, 11
Custom	252, 214, 212, 210, 250, 251, 214, 212, 219, 220	36, 35, 37, 2, 17, 35, 37, 2, 34, 17, 35, 37, 2, 303, 17, 35, 37, 2, 183, 36, 35, 37, 2, 36, 35, 37, 2, 17, 35, 37, 2, 34, 17, 35, 37, 2, 33, 291, 17, 35, 37, 2, 305, 17, 35, 37, 2, 284, 17, 35, 37, 2, 17, 35, 37, 2, 441, 17, 35, 37, 2, 2

Table 2: Size of individual packets sent by server and client in a session.