Distributed Systems, Spring 2013
Assignment 2: The Comm-Layer: Messages, Protocol, Sessions, RPC and Pub/Sub Due: Feb 22

Introduction

The goal of this lab is to get us working on our underlying communications layer and primitives for our game. We will be building on the net.[ch] that we currently have to progessively construct:

1. A simple generic game protocol definition
2. A simple session abstraction for a stream of protocol messages
3. A basic messaging model associated with a session
4. Using the protocol messages we will implement a simple RPC like commmunication facility
5. Finally we will and support for Event channels that support a pub/sub like model for group communication.

1 Hand Out Instructions

My suggestion, as always, is that you meet early and organize yourselves and the tasks to be done. Don't forget to document all your meetings.

1.1 Protocol

In `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/lib/protocol.h` you will find a header file that defines a simple generic game protocol. The protocol defines a message format and a set of message types. The protocol structures have been specified with game implementations in mind but are not specific to any given game. When construct a game you will specialize and utilize various aspects of the protocol format and features for a specific game.

All messages consist of at least a header who's structure is defined by:

```
typedef struct {
  int                version;
  Proto_Msg_Types    type;
```

```
  Proto_StateVersion sver;
  Proto_Player_State pstate;
  Proto_Game_State   gstate;
  int                blen;
} __attribute__((__packed__)) Proto_Msg_Hdr;
```

The generic semantics for these fields are:

version: This field is used to mark every message with a protocol version. A protocol version can be used to indicate a specific set of message types and associated interpretations. In this way clients and servers multiple protocol versions and be able to negotiate and identify a specific protocol version for their communication and potentially even on a message by message basis. Our expectation is that all games will define a basic protocol version that all servers and clients must at least support. Exteensions and advance features can be negotiated during session setup by agreeing on protocol versions that are greater than the basic protocol version number that both the server and client support. For example the support for chat messages might not be in the basic game protocol version but an advanced protocol version might add such a message type.

type: The protocol message type that this message is of. The basic example message types that currently form the 0 or base protocol version are defines as follows:

```
  // Requests
  PROTO_MT_REQ_BASE_RESERVED_FIRST,
  PROTO_MT_REQ_BASE_HELLO,
  PROTO_MT_REQ_BASE_MOVE,
  PROTO_MT_REQ_BASE_GOODBYE,
  // RESERVED LAST REQ MT PUT ALL NEW REQ MTS ABOVE
  PROTO_MT_REQ_BASE_RESERVED_LAST,

  // Replys
  PROTO_MT_REP_BASE_RESERVED_FIRST,
  PROTO_MT_REP_BASE_HELLO,
  PROTO_MT_REP_BASE_MOVE,
  PROTO_MT_REP_BASE_GOODBYE,
  // RESERVED LAST REP MT PUT ALL NEW REP MTS ABOVE
  PROTO_MT_REP_BASE_RESERVED_LAST,

  // Events
  PROTO_MT_EVENT_BASE_RESERVED_FIRST,
  PROTO_MT_EVENT_BASE_UPDATE,
  PROTO_MT_EVENT_BASE_RESERVED_LAST

} Proto_Msg_Types;
```

These are just a starting point it is expected that we will evolve this as we implement any given game.

However, there are some basic semantics and typical messages that are useful to observe and utilize. The message types are broken into two main categories:

RPC Messages that will be used for specific RPC exchanges a given RPC will be implemented as a pair of messages each with a unique type for it's request and matching reply. All RPC's will have a reply even if it is just to indicate a return code for the operation.

Events Messages that define types of events that maybe published on an event channel from a server to all clients subscribed to the channel. This will style of communication will can be used for asynronously updateing clients with game changes even when they have not explicity requested it.

sver: It is common that a version number that represent the current "state/time" of the server and clients view of the state be maintained. This allows both clients and servers to identify if a given message or their current view of the game is uptodate.

pstate: As an optimization the header leaves space for 4 32 bit values to be used by message types to encode player relevant state and or state changes. It is a common technique to encode important common data in the message header since it maybe required for many message types and it avoids the need to have a body associated with many messages. For the moment these have no specific meaning. You may use them as you see fit when defining and playing with protocols.

gstate: Similarly to pstate the header leaves space for 3 32 bit values to be used by message types to encode relevant game state or state changes.

blen: Final field of the header specifics the length of a body that a specific message may use to send aditional information associated with the message in the body. Thus our message format supports a message body of variable size. Remember however given the pstate and gstate fields you may find that often you may not need a body at all and blen will be set to 0.

1.2 Sessions

In `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/lib/protocol_session.h` you will find ageneric session abstraction. A session maintains the state of a connection between a specific client and the server. Much of your intellectual programming challenge will be to understand how effectively utilize this code (and implement it) and the ideas behind it. A session maintains the file descriptor of the socket that this session is associated with and seperate send and receive headers and buffers for bodies associated with messages being exchanged on the session. A number of utility functions need to be defined that "marshallel" the data into the session so that the message can be sent. You will need to figure out what this means.

1.3 Messaging

In `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/lib/protocol_session.h` you will find that delarations for the two basic functions for message exchange:

```
extern int  proto_session_send_msg(Proto_Session *s, int reset);
extern int  proto_session_rcv_msg(Proto_Session *s);
```

The basic idea is that on a given session code will prep the session with message data to be sent then invoke the send function. In addition to the session that the send is to occur on a reset specification is provided. This indicates if once the send is done if the message should be clearer from the session send header and buffers. If set to one then the session should reset the send fields if not it should leave them so that the same message can potentially be sent again.

Similarly when code wants to read a message from the session it would invoke the rcv function. The session code will then block reading in a message from the session and return once the message is loaded in the sessions receive fields.

The session layer is intended to be generic and provides the notion of a protocol message type handler. The next layer of session code `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/lib/protocol_server.h` and `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/lib/protocol_server.h`. Builds on the protocol_session interfaces to provide the this extensible model.

The protocol_server code extends the protocol_session code to provide the ability to develop server side code that is implemented as call-backs to handlers for each rpc message type. It should provide the following function on receipt of a message it should parse the message type of the message and call the handler associated with it. By default the session layers should have a dummy handler associated with all the message types. Code using this layer will hook themselves in to the protocol processing by defining message handlers and registering them at initialization time. `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/lib/protocol_server.c` Provides you with a template for this code.

Similarly the protocol_client code extnds the protocol_session code to provide the client side code to be developed. Its provides the client functional interface to the supported rpc's and also the ability for the client to associated a handler for an event channel session it may have connected to. See `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/lib/protocol_server.c` for a template of this code.

1.4 RPC

Using the above implement the necessary functions in protocol_session, protocol_server, and protocol_client to implement rpc exchanges for the message. On the server side the server will need a thread associated with each session that is listening for rpc messages and then dispatches the messages to the handler as discussed above.

On the client side you will need to construct the functional interface in which send the RPC request messages and then receives the associated reply.

1.5 Pub/Sub: Events Channels

Similar to the rpc communications support fill in the necessary code to implement server provided event channels.

1.6 Putting it all Together: Tic-Tac-Toe

Inside you `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/server/server.c` and `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/client/client.c` provides a dummy client and server that uses the a functions discussed above in a trivial way. Your first task should be implment everything to get these working.

Once you have this working then design and implement a tic-tac-toe game using the code you have developed. See `http://www.cs.bu.edu/~jappavoo/Resources/451/a2/tic-tac-toe.pdf` for additional information.

Note the code for the client and server provided are crude implementations Your's should be better and be more robust.

You may find my directory structure and Makefiles useful. They can be found in `http://www.cs.bu.edu/~jappavoo/Resources/451/daGame/TheDoctor.tar`

2 Evaluation

I will expect to be able to look at your repo and find an a2 branch that I can evaluate.

I will look at your source, LOG file, and commit logs. I will review your specifications and architecture documents.

You will be expected to demo your tic-tac-toe game and be able to explain its design, function, and problems encountered.