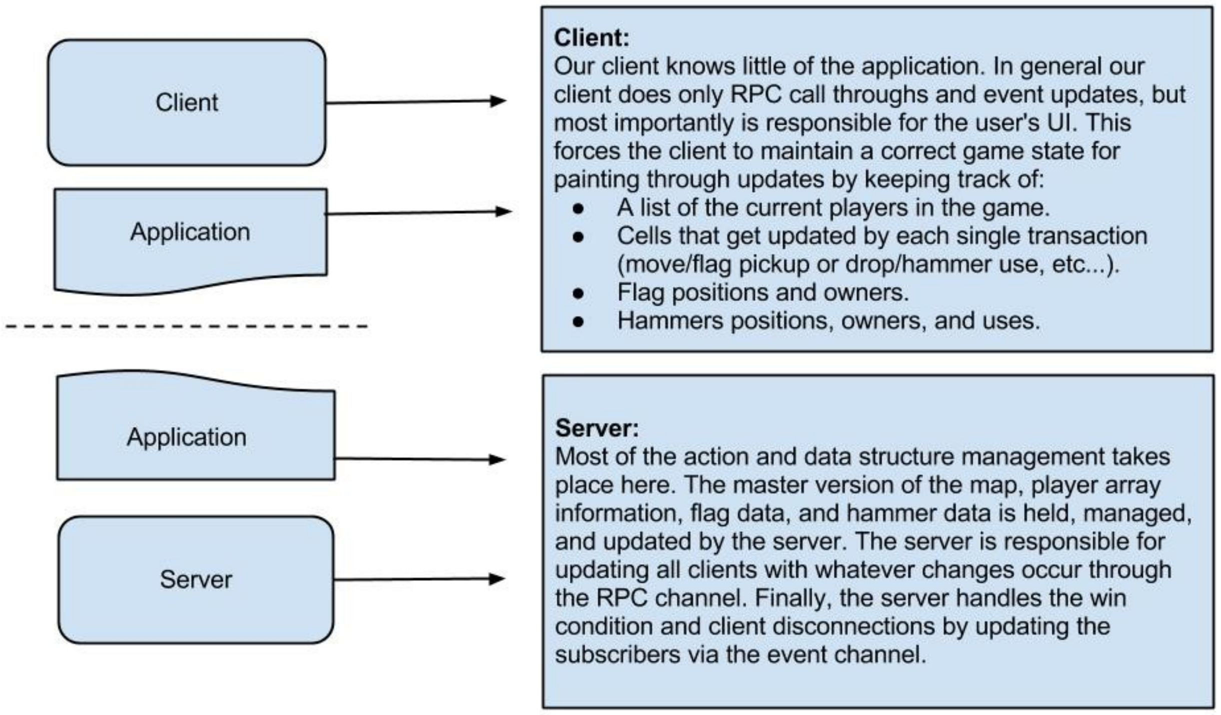
Ryan Conrad
Jeffrey Giardina
William Ang
5-6-2013

Team: UrbanSafari

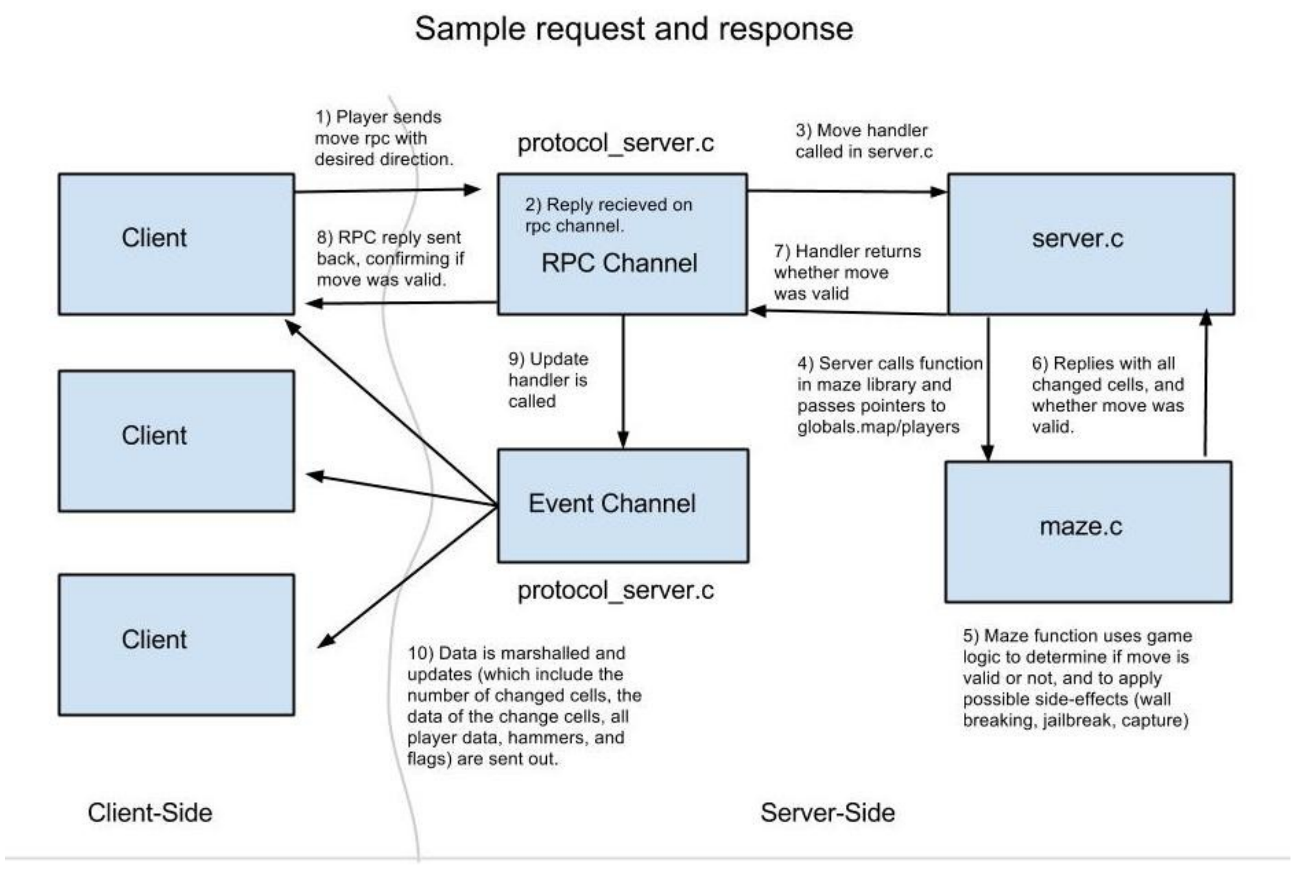
Architecture for CTF



Client:
Our client knows little of the application. In general our client does only RPC call throughs and event updates, but most importantly is responsible for the user's UI. This forces the client to maintain a correct game state for painting through updates by keeping track of:

- A list of the current players in the game.
- Cells that get updated by each single transaction (move/flag pickup or drop/hammer use, etc...).
- Flag positions and owners.
- Hammers positions, owners, and uses.

Server:
Most of the action and data structure management takes place here. The master version of the map, player array information, flag data, and hammer data is held, managed, and updated by the server. The server is responsible for updating all clients with whatever changes occur through the RPC channel. Finally, the server handles the win condition and client disconnections by updating the subscribers via the event channel.


Sample request and response
1) Player sends move rpc with desired direction.
protocol_server.c
3) Move handler called in server.c
Client
2) Reply recieved on rpc channel.
RPC Channel
8) RPC reply sent back, confirming if move was valid.
7) Handler returns whether move was valid
server.c
9) Update handler is called
4) Server calls function in maze library and passes pointers to globals.map/players
6) Replies with all changed cells, and whether move was valid.
Client
Event Channel
maze.c
Client
protocol_server.c
10) Data is marshalled and updates (which include the number of changed cells, the data of the change cells, all player data, hammers, and flags) are sent out.
5) Maze function uses game logic to determine if move is valid or not, and to apply possible side-effects (wall breaking, jailbreak, capture)
Client-Side
Server-Side

Server:
The server should serve as the game moderator. It will:

1. Hold the master instance of the map, players, and other needed globals to keep track of all clients.
2. Enforce the game rules and logic.
3. Notify clients of events.

Initial connection to the server is done from a socket connection from the client, and followed by a hello RPC to let the user into the game and initialize the client's Player. We will not worry about Denial of Service attacks for this project.

Server should handle/accept RPC's defined below in the client section.

Client:

Client should be dumb and only enforce game logic/rules as a form of efficiency, but not be required to do so. We force all clients to update on event handlers and not RPC replies.

Command Handlers execute an appropriate RPC, OR fetch some sort of static/local game info from the local client data structures.

RPC Commands:
- hello: executed after a connect to the server/port is established) holds you a spot in the server's

connect list, hello is executed as a separate RPC to assign the user a color and number, initialize their session and position in home, and push to other players that a new player has joined
- goodbye/disconnect: kills player session and closes their rpc and event channel
- move: sends player type and number, server pushes event upon a valid move
- drop: Sends player and object to drop and updates the map and refreshes it in an event to user.
-pickup: Sends player and object to pick up and updates the map refreshes it in an event to users;

Event Handlers:
- Need to be registered on initialization with the protocol_client.c
- Will be the only functions to update the client data structures (i.e. rpc replies shouldn't.)

The CTF game server sends out the following events on the event channel.

- Update Players – This event will send out the server master information of all players to all of the clients. This event will get called for whenever a change is made on the game map: player addition or movement, wall destruction, player capturing, jailbreak, picking and dropping objects, and revealing flags. This update sends the number of changed cells, the data of the changed cells, number of players, all player data, the hammers, and the flags.

-Winner- This event occurs when the game logic has determined a winner. The update_winner message is sent with the team color of the winner. The client side receives this message and based on who the winner is displays a specific splash screen.

Data for client:
- Client will hold a local copy of the map that will be updated frequently via event handlers
- Player state, i.e. color, number, in jail or not

Structures:
The structures for the CTF game are written below:

maze.h -- Maze Data Structures

```
typedef enum {JAIL, HOME, FLOOR, WALL} Cell_Type;

typedef enum {RED, GREEN} Color;
```

Pos:

```
typedef struct
{
  int x;
  int y;
} Pos;
```

Hammer:

```
typedef struct
{
  Pos p;
  int charges;
} Hammer;
```

Flag:

```
typedef struct
{
  Pos p;
  Color c;
  int discovered;
} Flag;
```

Map:

```
typedef struct
{
  pthread_mutex_t lock;
  char data_ascii[w*h]; // ascii dump
  int w; // MAPWIDTH
  int h; // MAPHEIGHT
  int num_wall_cells;
  int num_floor_cells;
  Hammer *hammer_1;
  Hammer *hammer_2;
  Flag *flag_red;
  Flag *flag_green;
  Cell cells[sizeof(Cell)*w*h]; // cell
array
} Map;
```

Cell:

```
typedef struct
{
  Pos p;
  Color c;
  Hammer *hammer;
  Flag *flag;
  Cell_Type t;
  int breakable; // cell is breakable? applies to walls
  Player *player; // Pointer to single player that may
occupy this cell
} Cell;
```

Player:

```
typedef struct
{
  pthread_mutex_t lock;
  UI_Player *uip; // pointer for UI info to paint
  int id;
  Pos pos;
  Color team_color; // Color rep. of team int
  int team;
  int hammer; // has hammer or not
  int flag; // has flag or not
  int state; // jailed or free
  int timestamp; // last-updated timestamp
} Player;
```

Constants defined:

```
#define MAPHEIGHT 200
#define MAPWIDTH 200
#define MAXPLAYERS 200
```

server.c -- Server Data Structures

```
struct Globals
{
  Map map;
  char mapbuf[MAPHEIGHT*MAPWIDTH];
  pthread_mutex_t MAPLOCK;
  Player *players[MAXPLAYERS]; // server's array of current
players in the game
  int numplayers;
  pthread_mutex_t PlayersLock;
  int num_red_players;
  int num_green_players;
} globals;
```

Note: Some fields not included that deal with performance testing and keyboard input.

client.c -- Client Data Structures

```
struct Globals
{
  Map map;
  char mapbuf[MAPHEIGHT*MAPWIDTH];
  pthread_mutex_t MAPLOCK;
  Player players[MAXPLAYERS]; //
  int numplayers;
} globals;
```

Note: Some fields not included that deal with performance testing and keyboard input.

```
typedef struct ClientState
{
  void *data; // pointer to the Player object attached to
this client
  Proto_Client_Handle ph; // Handle to the
Proto_Client object attached to this client
} Client;
```

Note: Proto_Client struct not defined here as it was unaltered from skeleton code