

TIC TAC TOE PROTOCOL

AN IMPLEMENTATION BY:

Satya Mohanty

Ravi Mohammed

Geoff Rothman

December 13, 1999

TIC TAC TOE PROTOCOL OUTLINE

- I. Game Descriptor
- II. Sending & Receiving Challenges
- III. Gameplay
 - a. Initiate move or receive opponents.
 - b. Decrypt packet and process move message.
 - i. Check move history.
 - ii. Check bounds of move.
 - c. Display move on board.
 - d. Choose random move and display.
- IV. Outcome Procedure
- V. Conclusions
 - a. Bakeoff
 - b. Future work

GAME DESCRIPTOR

We have chosen to initially focus on the game descriptor because it is the main focus of our Tic Tac Toe Protocol implementation. Our data structure consists of:

```
struct game_descriptor {  
    u_char version;  
    u_char flags;  
    u_short game_type;  
    char challenger_name[28];  
    u_short challenger_port;  
    u_short challenger_game;  
    char responder_name[28];  
    long responder_game_no;  
}; // from file example.h
```

The game descriptor is necessary in order to organize how a game will be played. If any one of the first six fields is missing, then the responder will not know how to properly play the game.

According to TTTTP version 1.0 specifications, these fields are defined as:

Version. This byte contains the version number of the TTT Protocol, in binary. This specification defines Version 1 of the protocol, so the contents of this byte should be 0x01.

Flags. This byte contains flags indicating various options that apply to the game. The flag bits are numbered from 0 (most significant, i.e. 0x80) to 7 (least significant, i.e. 0x01).

- **AUTH.** Bit 0. If this flag is 1, it indicates that the game is an authenticated game. Otherwise the game is not authenticated.
- **Challenger First.** Bit 1. If this flag is 1, the first move of this game is made by the Challenger. If this flag is 0, the first move is the Responder's.
- **Reserved.** Bits 2-7. These bits **MUST** be set to zero in this version of the protocol.

Game Type. The Game Type field is a non-zero 16 bit integer which describes the type of game being played. This is implemented in order to allow this protocol to include other types of games as well. For our purposes, a Tic Tac Toe game is defined as 0x01 (decimal 1).

Challenger Name. The Unique Name of the challenger, left-justified in the field, and padded with bytes containing 0 to a length of 28 bytes. Because Unique Names do not exceed 27 bytes in length, this field always contains at least one pad byte.

Challenger Port. The TCP port number used by the challenger in playing the game, in network (big-endian) byte order.

Challenger Game #. This field is set to zero in a Challenge message. In all other messages, it is set to a nonzero 16-bit integer (in network byte order) chosen by the challenger to identify this game. The challenger is responsible for ensuring that the 32-bit combination of the port number and game # is unique across all games played with this name acting as challenger.

Responder Unique Name The Unique Name of the responder, left-justified in the field and padded with bytes containing zero to a length of 28 bytes. This field always contains at least one pad byte.

Responder Game # A four-byte integer, different from zero, in network byte order, chosen by the responder to be unique across all games played with the above unique name as the responder.

- Each Game Descriptor is exactly 68 bytes in length.

SENDING & RECEIVING CHALLENGES

All the declarations for the functions during gameplay exist in the example.h file. In order to perform different aspects of the game simultaneously, we have chosen to use the fork() command to spawn a child and parent process. They have been declared like this:

```
if((pid=fork())==0){  
    rcv_bcast_msg(fp); //receive challenges from others  
    // child process  
}  
else challenger(1, 0,fp); //send challenge messages.h  
    // parent process  
// code from challenger.c
```

As defined above, the child process will receive all the broadcast messages from others wanting to challenge, and the parent process will continually send out challenge messages. For each challenge we send, our implementation accepts only three responses; this saves system resources because each extra response requires another fork(). The challenger() procedure performs a sleep() for 10 seconds and then send out a new challenge message; both of these functions are defined in "mcast.c". The challenge message is formed in form_challenge() in the message.c file. A new available port is set as the port

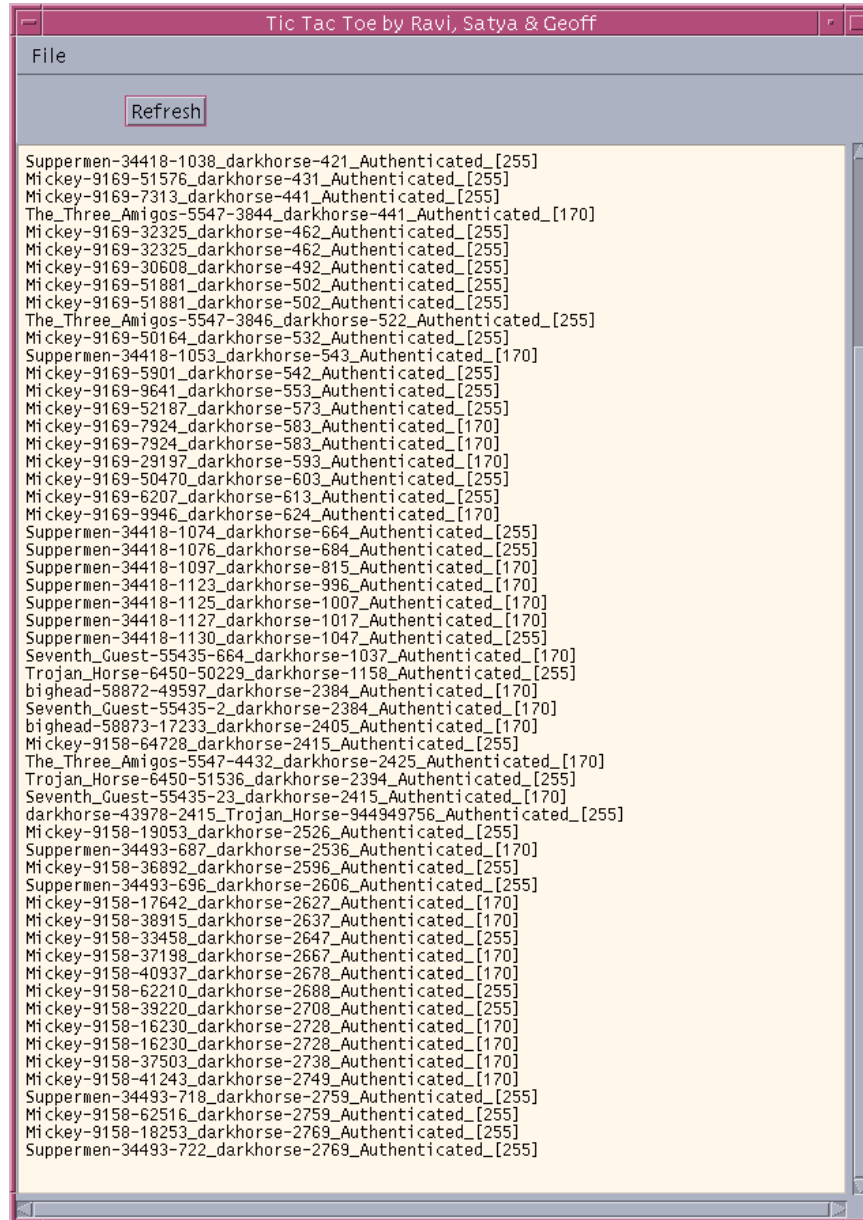
for playing the game (TCP port). The port allocated is obtained using the `getsockname()` function. Note that the sending and receiving of challenge messages is done over an IP multicast channel.

Upon receiving challenges, we choose to only play authenticated games during the bakeoff so our code reflects that. Once another team receives our challenge message and accepts, a TCP connection is established between the two parties. Both sides exchange their game id numbers and team names over the TCP channel. We have chosen to wait 30 seconds for responses from the other party; if they don't respond, then we abort the connection. Following are the steps the child process follows when receiving challenge messages [as defined in the `example.c` `challenger_play_game()` function]:

1. Check the size of the challenge message. It should be `[sizeof(GAME_DESC) + SIG_SIZE]`. The signature size is specified as 96 bytes, and the size of the `GAME_DESC` field is 68 bytes.
2. Decrypt the challenge message using predefined RSA encryption/decryption methods. Public keys have previously been stored in a data structure in the `challenger.c` file. This file contains a function called `Read_File()` which is implemented in the `read_db.c` file.
3. Check the decrypted message with the `game_descriptor` to make sure everything works out.
4. Extract the challenger's port number from the `game_descriptor` and connect to that port over a TCP connection.
5. Send our `game_descriptor`, which consists of our team name [`darkhorse`] and our game id number. We have chosen to get our unique number from the `time` function. We take the last 16 bits of this 32 bit field and

- sleep one second after getting the time. This ensures that the number will be unique; calculations have been made and this number will remain unique for ~18 hours. This is plenty of time for Bakeoff (the Bakeoff will be explained in Section 5) purposes. Then get the game id of the challenger.
6. Receive the challenger's unique game id number.
 7. If we are the challenger, we accept connections from other responders. Upon creating a new socket with accept, we fork again having the child play the game with the responder. The parent continues to accept other responses, and we have restricted this to 3 responses per challenge message sent. When the 3 responses are obtained, we sleep for 10 seconds and then issue another challenge. [the challenger() function]
 8. In the challenger_play_game() function, we get the responders game id, game, and check to see if they have tampered with our initial challenge message. (by checking the size of the message). Then we send our game id to the responder and depending on who initiates the first move, start playing the game in play_moves().
 9. Depending on who the initiator is, start sending the initial move (play_moves()). We check to see who should go first in chk_initial_mov(). The first bit of the Flags field in the game descriptor tells who goes first.

In Figure 1, we have programmed a game status GUI using the Motif libraries (text1.c), which tells the outcome of games that have been completed. One must click on the refresh button to get the most updated results.



GAMEPLAY

The heart of the Tic Tac Toe game is located in the `play_moves()` function in the `example.c` file. This function is called and is given both the game descriptor and who the initiator is. In `mytictac.c` and `mytictac.h`, we have setup another GUI in Motif, telling the status of the game as can be seen in Figure 2.



Our algorithm for the `play_moves()` function is as follows:

1. Sign my move if I'm the initiator and send it to the responder. If I'm not the initiator, then I receive my opponents move [done in `rcv_mov_msg()` function in `message.c`].
2. Check that the move message size is 106 bytes (10 for the moves and 96 for the signature).
3. Decrypt message and check to see if their move history matches ours. Before sending our moves out, we kept an array called `prev_mov_his`. For example, if our `prev_mov_his` says [0, 2, 4] and their next move message says [0, 2, 4, 6], then that's acceptable whereas [0,4,2,6] would be incorrect. We compare the boards to see

if all indexes up to the current one match. If they don't match up, then we claim that the board was tampered with and then quit the game.

4. Check opponents move "x" to see if ($0 \leq x \leq 8$).
5. In `play_board.c`, we map their move to our game board. To get the x coordinates, we divide x by 3. To get the y coordinates we mod x with 3.
6. Call `display_board()` to show their move on our GUI.
7. Check to see if the game is terminated by a win, loss, draw, or error.
8. Choose a random number and mod it by 3 to get our x and y coordinates for the move. We then check to see if that position is not filled. If not, then we place our move there and update the GUI.
9. Encrypt the board by signing it, and then send it to our opponent in the form [board, signature].
10. Check to see if we terminated the game with our move. Note that we allow 2 minutes for our opponents to respond before we abort the connection with an error.
11. Every signature is stored in an array (`signature *[9]`). The entries of the array alternate with our signature and then the opponent's signature or vice versa.
12. The Tic Tac Toe board is displayed after the games are completed. One board (per game completed) is generated and the board is displayed for 10 seconds.
13. The title of the board contains the challenger's name and id, and the responder's name and id. A label on the board GUI also shows the outcome of the game.

OUTCOME

After the games are terminated, we must decide whether to send an outcome message or not. Outcome messages are only broadcasted in case of a win, loss, or a draw. To form the outcome message, we call the `outcome_msg()` function in the `outcome1.c` file. We pass to it the person who won the game, the game descriptor, an array of all the signatures, and the move history (a total of 79 bytes). If the game was unauthenticated, then it won't include any signatures. Therefore, if 6 moves were made in the Tic Tac Toe game, then 6 signatures should be sent. This allows verification of play with the ability to check for tampering with the moves. The message is then sent by IP multicast to the outcome broadcast port. This broadcast allows all players to keep an unofficial count of the game score.

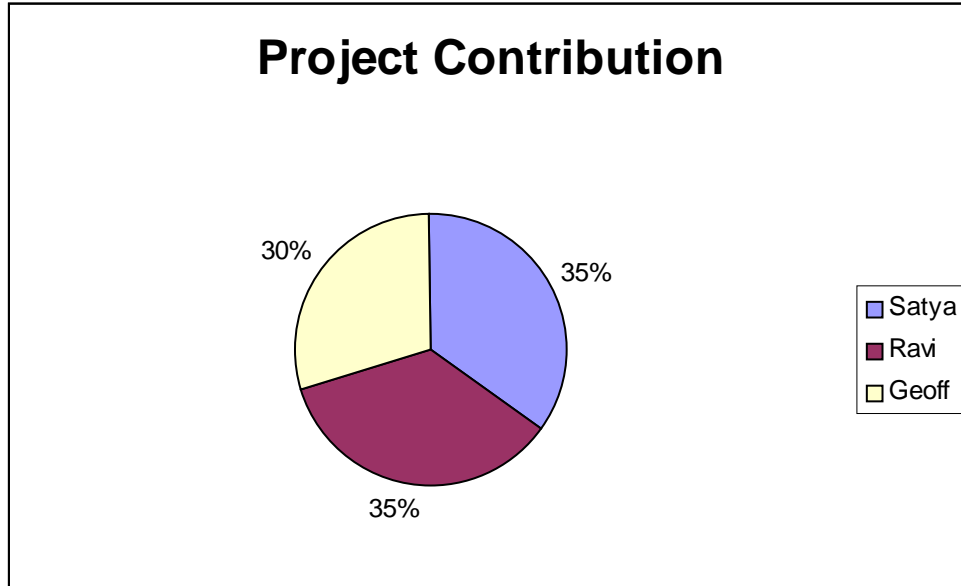
CONCLUSIONS

During the bakeoff held Saturday Dec. 11th, 1999, our implementation was tested with other teams from the CS645 Computer Networks class. The only fault found with our implementation was with forming the outcome message. We were able to play games successfully and both logs show this. However, it

also reports that we sent the array of signatures incorrectly. We didn't win too many games because our "move picker" was random, but we have plans to improve this.

Future work includes making our `play_moves()` function a little more intelligent. Our strategy would be to try and place our moves contiguous to each other until we get three in a row. If the opponent blocked us, then we'd try another contiguous route. After each move, we'd also check to see if the opponent had two contiguous moves. If so, we'd block them with our next move.

We could also keep a history that would include each game we played. If we have won against a certain team frequently, we'll only play games with that team. If we keep losing to a certain team, we'll assume they've guessed our patterns and we'll stop playing them.



Descriptions

Satya: network programming

Ravi: network programming

Geoff: GUI, project writeup