Python Object-Oriented Program with Libraries

Unit 3: Web Programming

CHAPTER 2: PROTOCOL

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Socket is a low-level programming struct. In this chapter, we are focused on using sockets to develop a communication protocol.
- This protocol is named polling protocol (from UCI) for case study
- Use Connect 4 program to illustrate the design flow of a networked program
- Project: Networked Tic Tac Toe program

Protocols

LECTURE 1

Protocols

- When you write a program that will store data in a file and read it back again later, you have to decide on a file format, which specifies, in detail, what the data will look like once it's stored in the file. Sometimes, it's as simple as just storing text, but if you want to do anything with the data other than display it exactly as it's stored, there's a good chance you'll need to consider a way to organize it within the file.
- For common problems like storing images or videos, there are existing file formats, such as the JPEG format for image files, but you can define your own, too, if a pre-existing format isn't appropriate for your particular use.

Protocols

- When you write programs that communicate with one another using **sockets**, you have a similar problem. The program on each side of the connection will be sending data to the other. Without an agreement about what that data will look like, the data sent from one program won't make sense to the other one. So, when programs communicate via sockets, you will always need them to agree on a protocol, which specifies what each program will send and what it will expect to receive.
- As with file formats, there are well-known protocols already defined for specific purposes — like the HTTP protocol that describes how data is transferred over the web, or the SMTP protocol that is used to send email — but you can also define your own protocol if you need something specific for your particular use. What's important is that both programs implement the same protocol, and that each program knows its role in that protocol.

The Polling protocol

LECTURE 2

The Polling protocol

- In lecture, we wrote a client program that interacted with a Polling server, a program that I built that allows users to answer multiple-choice questions, while tracking the number of times a user picked each choice. The server is the program that keeps track, as time goes on, of what the questions are and who's answered them how; other programs can interact with it by connecting to it via a socket and then sending and receiving text in a predefined format called the Polling protocol.
- The Polling protocol, like other protocols, governs what each program — a Polling client and the Polling server — is required to send and what it can expect to receive in return.

The Polling protocol

- The Polling protocol is what is sometimes known as a **request-reply** protocol. Lines of text are sent back and forth between the client and server, with each interaction being driven by the client sending a request and the server sending a corresponding reply.
- Every line is terminated with a newline sequence, which is made up of the Python string '\r\n' — technically, these characters are known as a carriage return and a line feed — without which the receiving program won't know that the sender has sent a complete line of text.

Polling Protocol

- Using the Polling protocol, the interactions between a Polling client and the Polling server are expected to work as described below.

[A] Client Connects

- The client connects to the Polling server

[B] Server Accept

- The Polling server accepts the connection

[C] Client Sends Polling_Hello

- The client sends a line consisting only of the word POLLING_HELLO, followed by a space, followed by a UCInetID (i.e., a UCI email address without the @uci.edu). So, for example, for someone whose UCI email was boo@uci.edu, the client would send POLLING_HELLO boo.

[D] Server Validates the User

- The server checks the UCInetID against a list of the ones that it will accept — everyone in this course should be on the list, among others. If the server accepts the UCInetID, it responds with **HELLO**; otherwise, it responds with **NO_USER**, followed by a space, followed by the UCInetID.
  - The purpose of this initial interaction is to establish that the client implements the Polling protocol and not some other one; it's quite common for protocols to begin with some kind of "hello" or "handshake" sequence like this.

[E] Server Responds

- Once the server has accepted the UCInetID, the client is considered "logged in." There is no password or other authentication required. From that point, the client can send one of five commands, each of which would result in a different kind of repsonse:

[E-1] POLLING_QUESTIONS

- The line **POLLING_QUESTIONS**, in which case the server will respond with two things:
  - The line **QUESTION_COUNT** ***number_of_questions***, where *number_of_questions* is a non-negative integer such as **10**. This specifies how many questions the server has available to vote on.
  - If *number_of_questions* is the number *n*, this will be followed by *n* lines, each of which is **QUESTION**, followed by a space, followed by a *question ID* (a sequence of non-space characters), followed by a space, followed by the text of the question.

[E-2] POLLING_CHOICES

- The line **POLLING_CHOICES *question_id***, where *question_id* is expected to be one of the question numbers of an existing question that the server has available to vote on. If it's not, the server will respond with **NO_QUESTION**, followed by a space, followed by the question number sent to it. If it is a valid question number, the server will respond with two things:
  - The line **CHOICE_COUNT *number_of_choices***, where *number_of_choices* is a non-negative integer such as **10**. This specifies how many choices the chosen question has available.
  - If *number_of_choices* is the number *n*, this will be followed by *n* lines, each of which is **CHOICE**, followed by a space, followed by a *choice ID* (a sequence of non-space characters), followed by a space, followed by the text of the choice.

[E-3] POLLING_VOTE

- The line **POLLING_VOTE *question_id choice_id***, where *question_id* is expected to be one of the existing question IDs and *choice_id* is expected to be one of the existing choice IDs for that question. There are a few possible responses you might get, depending on the circumstance, but you'll only get one of them.
  - **VOTED**, if the vote was accepted.
  - **ALREADY_VOTED**, if the user already voted on this question.
  - **NO_QUESTION *question_id***, if the question did not exist.
  - **NO_CHOICE *choice_id***, if the choice did not exist for the specified question.

[E-4] POLLING_RESULTS

- The line **POLLING_RESULTS *question_id***, where *question_id* is expected to be one of the question IDs of an existing question that the server has available to vote on. If it's not, the server will respond with **NO_QUESTION**, followed by a space, followed by the question ID sent to it. If it is a valid question ID, the server will respond with two things:
  - The line **RESULT_COUNT *number_of_choices***, where *number_of_choices* is a non-negative integer such as **10**. This specifies how many choices the chosen question has available.
  - If *number_of_choices* is the number *n*, this will be followed by *n* lines, each of which is **RESULT**, followed by a space, followed by a *choice ID*, followed by a space, followed by the number of votes for that choice so far, followed by a space, followed by the text of the choice.

[E-5] POLLING_GOODBYE

- The line POLLING_GOODBYE, in which case the server will respond with GOODBYE and then close the connection; the interaction between the client and the server is now over.

An example session follows:

Client	*Server*
initiates a connection	
	accepts the connection
POLLING_HELLO boo	
	HELLO
POLLING_QUESTIONS	
	QUESTION_COUNT 1
	QUESTION 1 Who is your favorite Pekingese?
POLLING_CHOICES 1	
	CHOICE_COUNT 1
	CHOICE 1 Boo
POLLING_VOTE 1 1	
	VOTED
POLLING_VOTE 1 1	
	ALREADY_VOTED
POLLING_GOODBYE	
	GOODBYE
	closes the connection
closes the connection	

What we want to build?

LECTURE 3

What we wanted to build

- The protocol described above is not intended for human use, any more than the HTTP protocol — which governs how web browsers download web pages and other data — is intended for people. A web browser has the knowledge of the HTTP protocol embedded within it; behind the scenes, when you visit a web page, a conversation between your web browser and a web server commences, with HTTP defining what that conversation will look like. But the conversation itself is invisible to users of a web browser; someone using a browser simply sees some kind of progress indication and, ultimately, the web page.
- Similarly, we might like to build a Polling client, whose job is to provide a user with the ability to use the Polling service without having to know the details of hosts, ports, sockets, and protocols, so they can simply look at a list of questions and vote on them.

Taking the opportunity to think about design

- As programs get larger, we're best off separating them into modules that contain related subsets of functionality. When writing this program, we quickly find that there's a natural separation between the part of the program that implements the protocol (i.e., the part that communicates with the Polling server) and the program's user interface. Isolating each of these into its own module makes each of those modules simpler, and also provides other benefits (e.g., keeping a larger, complex program organized; allowing us to put more than one "outer shell" around the protocol code if, for example, we wanted to also write a graphical user interface).
- So this program is probably best written with two modules, which we'll call polling (the protocol implementation) and polling_ui (the user interface).

Taking the opportunity to think about design

- Similarly, the functions in each module are broken into progressively smaller functions, with meaningful names and well-named parameters.
- This code example, in my view, is a good example of why we should want to do that, because there's a fair amount of complexity here that's worth isolating, so we can think about one thing at a time instead of everything.

Taking the opportunity to think about design

- Finally, within each module, we were fastidious about separating the public functions (i.e., the ones we expect would be needed by code in other modules) from the private ones (i.e., the ones that are only useful within that module). This separation provides at least two benefits: Making it easier to understand how to use a module, by limiting how many functions a user of that module needs to know about; and leaving open the possibility that certain aspects of how a module is implemented might change without affecting the code that calls it.
- As long as other modules use only the public parts of our module, we can feel free to change the private ones without having a negative effect on the others.

Case Study – Connect4

LECTURE 4

Steps to create a networked game

1. Create a simulated standalone game and make it event-driven
   - connectfour.py
   - play.py (stand-alone game launcher)
2. Design the communication protocol between Clients and Server
3. Implement the basic socket module
4. Implement the network version play.py (`connectfour_console.py`)
5. Implement the networked version of the play (`connectfour_network.py`)
6. Write the server program (`local_server.py`). You may write a Networked-version and then, upload to a server.
7. Upload server program to web-site with proper Server software. Or, use client program to talk to the server.

Connect 4 Game

https://en.wikipedia.org/wiki/Connect_Four

Connect 4

- This project allows you to take a first step into a more connected world by introducing you to the use of sockets, objects in Python that represent one end of a connection between one program and another — the other might be on the same machine, on another machine in the same room, or on a machine halfway across the world. You'll learn about the importance of protocols in the communication between programs, and will implement a game that you can play either standalone (on your own computer) or via your Internet connection.

Connect 4

- Along the way, you'll also be introduced in more detail to the use of **modules** in Python, and to writing programs that are made up of more than one module, a technique that we'll revisit repeatedly as the size and complexity of the programs we write begins to increase.
- You'll find that the design decisions you make, such as keeping functions small and self-contained, organizing your functions and other code by putting it into appropriate modules, will be an important part of being able to complete your work. Additionally, you'll use a small library that I'm providing in order to seed your work on the project.

The program

For this project, you'll implement a Python-shell-based game that you will initially be able to play on your own computer, but will extend so that you can play via the Internet by connecting to a central, shared server.

The game

- For this project, you'll implement a Python-shell-based implementation of a game called Connect Four. The rules of the game are straightforward and many of you may already know them; if you're not familiar with the rules of the game, or haven't seen them in a while, Wikipedia's Connect Four page is as good a place to go as any to become familiar with it.
- Note that our implementation will include not only the traditional rules regarding dropping pieces into columns, but also the "Pop Out" variation discussed on the Wikipedia page. Connect Four boards come in a variety of sizes; while the default is 7x6 (i.e., seven columns and six rows), ours can support as many as 20 columns and 20 rows.
- Also, one very minor wrinkle that we're adding to the rules on the Wikipedia page is that the red player always moves first.

A starting point: the connectfour module

- Unlike the previous project, this project begins with a *starting point*, in the form of a library that I've already implemented that contains the underlying game logic. You will be required to build your game on top of it (and you will not be allowed to change it), which will be an instructive experience; learning to use other people's libraries without having to make modifications (and, in a lot of cases, without being allowed to make them) is a valuable real-world programming ability.
- Before proceeding much further with the project, it might be a good idea to spend some time reading through the code, its docstrings, and its comments to get an understanding of what's been provided. You can also try some focused experimentation in the Python interpreter so you can understand how the provided module works; you'll need that understanding in order to complete your work. Don't worry if not all of it makes complete sense initially, but do get a feel for what's available at the outset, then gradually fill in the details as you move forward.

The first program:

A Python shell version of Connect Four

- One of your two programs will allow you to play one game of Connect Four using only Python shell interaction (i.e., no networks or sockets). The user will need to specify the size of the board, after which the game begins. The user is repeatedly shown the current state of the game — whose turn it is and what the board looks like. The board should always be shown in the following format:

```
1  2  3  4  5  6  7  8  9  10 11 12
.  .  .  .  .  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .  .  .
.  .  .  .  .  .  .  .  .  .  .  .
.  .  R  .  .  .  .  .  .  .  .  .
.  .  Y  R  .  .  .  R  .  .  .  .
.  R  R  Y  .  Y  .  Y  .  Y  .  .
```

The first program:

A Python shell version of Connect Four

- Of course, there may be a different number of columns and a different number of rows, but this is the format you'd want to follow. (Notice how the columns whose numbers are above 10 are handled just slightly differently than the others; you'll need to do that, if there are at least 10 columns.)
- You have some latitude in how you ask the user to specify the board's size and make each move, but it needs to be clear what the user should do. Do not assume that your user will know what to type; tell them (briefly). Columns should be selected by typing a number between 1 for the far-left column and the appropriate number — say, 12, if there are 12 columns — for the far-right.

The first program:

A Python shell version of Connect Four

- When the user is asked to specify a move but an invalid one is specified (such as dropping into a column that is full), an error message should be printed and the user should be asked again to specify a move. In general, erroneous input at the Python shell should not cause the program to crash; it should simply cause the user to be asked to specify his or her move again.
- The game proceeds, one move at a time, until the game ends, at which point the program ends.

The second program:

A networked version of Connect Four

- Your second program will instead allow you to play a game of Connect Four via a network, by connecting to a server that I've provided. Your program always acts as a client.
- When this program starts, the user will need to specify the host (either an IP address, such as 192.168.1.101, or a hostname, such as www.ics.uci.edu) where a Connect Four server is running, along with the port on which that server is listening.
- Additionally, the user should be asked to specify a username. The username can be anything the user would like, except it cannot contain whitespace characters (e.g., spaces or tabs). So, for example, boo or HappyTonight are legitimate usernames, but Hello There is not.

The second program:

A networked version of Connect Four

- Additionally, the user should be asked to specify a username. The username can be anything the user would like, except it cannot contain whitespace characters (e.g., spaces or tabs). So, for example, boo or HappyTonight are legitimate usernames, but Hello There is not.

The second program:

A networked version of Connect Four

- Once the user has specified where to connect, the program should attempt to connect to the server. If the connection is unsuccessful, print an error message specifying why and end the program. If, on the other hand, the connection is successful, the game should proceed, with the client acting as the red player (and moving first) and the server — which acts as an artificial intelligence — acting as the yellow player. (Of course, the user will need to specify the size of board before the game starts.) For red player moves, the user should specify the move at the Python shell, as in your first program; for yellow player moves, the program should instead communicate with the server and let the server determine what move should be made.
- As in the first program, the game proceeds, one move at a time, until the game ends, at which point the program ends.

The second program:

A networked version of Connect Four

127.0.0.1 – Localhost

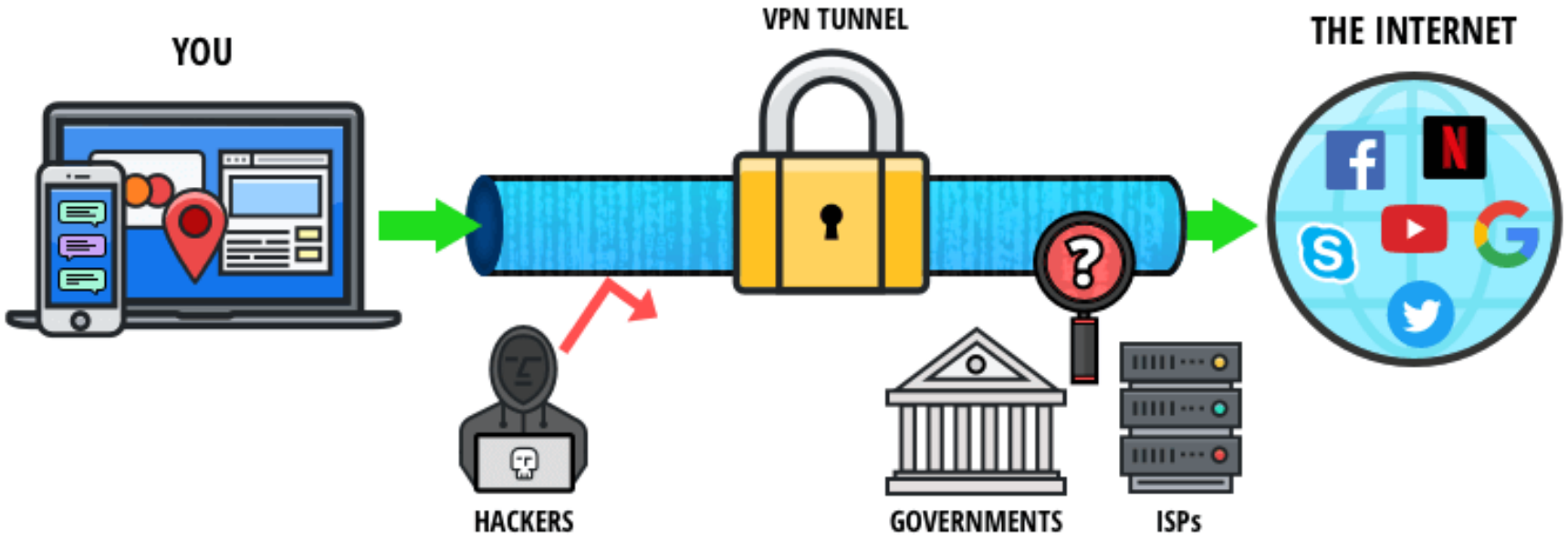
(Loopback) Address

- Using localhost for loopback testing:

The second program:

A networked version of Connect Four

- Remote VPN Server



The second program:

A networked version of Connect Four

- Where is the ICS 32 Connect Four server?
- Information about where the server is running will be distributed via email; I'll also keep you posted about planned downtime (e.g., if I need to fix a problem, if I know that the machine where it's running will be down, etc.). It may be necessary to move the server from time to time; when I do that, I will let everyone know via email.

The second program:

A networked version of Connect Four

- Please note that the ICS 32 Connect Four server is running on a machine on the ICS network that is not exposed to the open Internet. In order to connect to it, you will need to be connected to the campus network, which means you'll either need to be on campus or you'll need to be connected to something called the campus VPN, which allows you to access the campus' network from off-campus. Note, also, that certain residential areas are not connected to a part of the campus network that will allow you direct access to the ICS 32 Connect Four server, so you'll need to use the campus VPN in those cases. In general, if you're not able to connect to the ICS 32 Connect Four server, the first thing you should try is using the campus VPN.

Protocol

LECTURE 5

The ICS 32 Connect Four Server Protocol (I32CFSP)

- I32CFSP conversations are relatively simple. They are predominantly centered around sending moves back and forth, with the assumption being that both conversants will be able to determine the game's state simply by applying these moves locally; for this reason, the game's state is not transmitted back and forth.
- I32CFSP conversations are between two participants, which we'll call the server and the client. The server is the participant that listens for and accepts the conversation; the client is the participant that initiates it. (You'll be implementing the client in this project; I've already implemented the server.) The client is always the red player and the server is always the yellow player; this means that the client always moves first. I32CFSP conversations proceed in the following sequence.

•The server awaits a connection from a client
•The client connects to the server
•The server accepts the client's connection
•The client sends the characters **I32CFSP_HELLO**, followed by a space, a *username* that identifies the user, followed by an end-of-line sequence. End-of-line sequences are always '\r\n', in Python terms. (This two-character sequence, which is common in Internet protocols, is called a *carriage return and line feed* sequence.)
•The server sends the characters **WELCOME**, followed by a space, followed by the username sent by the client, followed by an end-of-line sequence.
•The client requests a game against an artificial intelligence by sending the characters **AI_GAME *columns rows***, followed by an end-of-line sequence, where *columns* is the number of columns on the board and *rows* is the number of rows on the board. Both *columns* and *rows* must be at least 4.
•The server indicates that it's prepared to play a game by sending back the characters **READY**, followed by an end-of-line sequence.
•From here, the client and the server alternate sending moves, with the opposite participant moving each time. This continues until the game has ended.

- When a participant wants to drop a piece into a column, the characters **DROP *col***, followed by an end-of-line sequence, are sent, where *col* is the column number (1 being the far left column, 2 being the column to the right of that one, and so on) into which a piece is to be dropped
- When a participant wants to pop a piece from the bottom of a column, the characters **POP *col***, followed by an end-of-line sequence, are sent instead, where *col* is the column number (1 being the far left column, 2 being the column to the right of that one, and so on) from which a piece should be popped
- More specifically, things proceed as follows:
  - The client sends its first move, such as **DROP 3**.
    - If this is a valid move, the server responds with **OKAY**, followed by an end-of-line sequence, unless the game is now over, in which case the server responds with **WINNER_RED** or **WINNER_YELLOW**, followed by an end-of-line sequence, depending on whether the winner was the red player or the yellow player.
    - If this is not a valid move, the server responds with **INVALID**, followed by an end-of-line sequence.
  - If the game is not yet complete, the server now takes its turn to move. It sends its move in the format specified above (e.g., **DROP 5** or **POP 2**).
    - Immediately following this, the server sends **READY** if the game is still in progress, or either **WINNER_RED** or **WINNER_YELLOW** if there is a winner.

•As soon as the game is over — when the server has sent **WINNER_RED** or **WINNER_YELLOW** — both participants close their connections, as the conversation is now over.

Client	*Server*
I32CFSP_HELLO boo	
	WELCOME boo
AI_GAME 7 6	
	READY
DROP 3	
	OKAY
	DROP 4
	READY
DROP 3	
	OKAY
	DROP 4
	READY
DROP 8	
	INVALID
	READY
DROP 3	
	OKAY
	DROP 4
	READY
DROP 3	
	WINNER_RED

Session Dialog

LECTURE 5

Client-side IDLE

```
Host: localhost
Port: 15000
Enter your username:
Eric
Connecting to localhost (port 15000)...
Connected!
Client-Server Communication Starts...
Server: WELCOME Eric
NEW GAME : CONNECT 4
1 2 3 4 5 6 7
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .

The Current Player is RED
Server RED: READY
Please enter type of move and column (e.x. DROP 5 or POP 4):
```

Server-side CMD (Command Line)

```
C:\Eric_Chou\Python Course\Python Object-Oriented Prog
ramming with Libraries\PyDev\U3 Network\W2 Polling Pro
tocol\project2_networked>python local_server.py
<socket.socket fd=564, family=AddressFamily.AF_INET, t
ype=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1
', 15000), raddr=('127.0.0.1', 63329)>
('127.0.0.1', 63329)
Client:  I32CFSP_HELLO Eric

AI_GAME
```

Client-side IDLE

```
Please enter type of move and column (e.x. DROP
5 or POP 4):
Please enter type of move and column (e.x. DROP
5 or POP 4):Drop 5
Client(Wrong Input):  Please enter type of move
and column (e.x. DROP 5 or POP 4):Drop 5
Please enter type of move and column (e.x. DROP
5 or POP 4):
DROP 5
Client:  DROP 5
Server OK:  OKAY
1 2 3 4 5 6 7
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . R . .

The Current Player is YELLOW
Wait for server...
```

Server-side CMD (Command Line)

```
Client:  I32CFSP_HELLO Eric

AI_GAME

Client:  DROP 5

Server:  OKAY

Enter Server's move:
```

Client-side IDLE

```
Server:  DROP 3
1 2 3 4 5 6 7
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . . . . . .
. . Y . R . .

The Current Player is RED
Server RED: READY
Please enter type of move and column (e.x. DROP
5 or POP 4):
```

Server-side CMD (Command Line)

```
Client:  I32CFSP_HELLO Eric

AI_GAME

Client:  DROP 5

Server:  OKAY

Enter Server's move:
DROP 3
Server:  DROP 3

Server:  READY
```

Repeated Several Times

Client-side IDLE

```
Wait for server...
Server:  DROP 3
1 2 3 4 5 6 7
. . . . . . .
. . . . . . .
. . . . . . .
. . Y . R . .
. . Y . R . .
. . Y . R . .

The Current Player is RED
Server RED: READY
Please enter type of move and column (e.x. DROP
5 or POP 4):
```

Server-side CMD (Command Line)

```
Client:  I32CFSP_HELLO Eric

AI_GAME

Client:  DROP 5

Server:  OKAY

Enter Server's move:
DROP 3
Server:  DROP 3

Server:  READY

Client:  DROP 5

Server:  OKAY

Enter Server's move:
DROP 3
Server:  DROP 3

Server:  READY

Client:  DROP 5

Server:  OKAY

Enter Server's move:
DROP 3
Server:  DROP 3

Server:  READY
```

Client-side IDLE

```
Please enter type of move and column (e.x. DROP
5 or POP 4):
DROP 5
Client:  DROP 5
Server OK:  OKAY
1 2 3 4 5 6 7
. . . . . . .
. . . . . . .
. . . . R . .
. . Y . R . .
. . Y . R . .
. . Y . R . .

Congratulations Player RED
Closing connection...
Closed!
>>>
```

Server-side CMD (Command Line)