Design & Professional Skills — Pacman Protocol Specification Assignment

1 Introduction

In this assignment you do not need to write any code; instead you will write a technical specification. You have been given the source code for an alpha-release of a multi-player networked Pacman game. The original Pacman game predates networked games, but we have extended the game to allow you to compete with other players for food and to try and tempt the ghosts to attack them.

Our version of this game was originally intended to be a sociable if somewhat silly game to be played by two players in the same room. If your Pacman goes down the tunnel, it will disappear from your screen and reappear on your opponent's screen, but is still controlled by you. You have to look at their screen to play.

Unfortunately in 2020, many of us can not be in the same room as one another, so a remote mode has been added (this is enabled using the "-r" command line flag). In remote mode, the game also displays a small version of their Pacman screen, making it possible to play against remote opponents. This does, however, reduce the sillyness somewhat.

The current version of the network protocol used to communicate between the two computers was implemented by a professor who shall remain anonymous, late one night after a long evening in the pub. It works, but it is an ugly quick and dirty solution, which does not perform especially well, and which has potential security vulnerabilities.

Your challenge is to design a better protocol and to write a specification of that protocol that is complete and unambiguous enough for another programmer to be able to implement an interoperable implementation in Python. You do not need to write an implementation of your protocol, though to understand the current protocol you may need to add some debugging print statements to the existing code.

2 Running Pacman

In the ENGF0002 github repository, in assignments/assignment5/multi-player/src there is source code for a multi-player version of Pacman.

There are several ways to run the code.

Client-Server Mode

In the simplest method, one player's computer acts as a "server" and the other as a "client". The server must run first, and then the client connects to the server. Once the network connection has been established, there is no difference between how the server and client behave.

To run the server:

```
python3 pacman.py -r -s -p <passwd>
```

The server pacman will start, will display its IP address on screen, and will wait for the client to connect. Substitute a password of your choice for `<passwd>`.

To run the client:

```
python3 pacman.py -r -c <ip_address> -p <passwd>
```

The IP address must be that of the server pacman, and `<passwd>` must be the same one they chose. The password is there to give some minimal control over who can connect to a server pacman.

You can then play multi-player pacman. If your pacman goes through the tunnel, it appears on the screen of your opponent, and can eat the food, eat the powerpills, eat frightened ghosts, and die. At present, the two pacmen do not directly interact. They simply pass straight through each other.

Client-server mode is useful when both players are on the same local network, or when the server is on a computer that is reachable from the public Internet, rather than behind a firewall. Unfortunately in 2020, it is rare that two players will be in the same room, but you can run both client and server on the same computer and play against yourself, which isn't very entertaining, but is useful for testing.

3 Two clients, one relay server

In `assignments/assignment5/pacman-server/` there is a simple relay server that passes messages between two clients. You can run the relay server as:

```
python3 pacman_server.py
```

Once the server is running, two clients can both connect to the server:

```
python3 pacman.py -r -c <ip_address> -p <passwd>
```

Again, both clients much choose the same password to be connected to each other. Many pairs of clients can all use the same server - only pairs of clients using the same password will be connected to each other. The IP address must be the address of the computer running the server.

You can run your own server on any computer that is reachable from your clients. Alternatively, we will run a server on a computer at UCL (IP address will be posted on Piazza), but the latency connecting from remote parts of the world may affect the play.

4 About the code

The code is a form of model-view-controller, roughly similar to that used for Frogger.

Your pacman's interactions with its environment are always modelled on your computer, even when it is visiting the remote screen. This allows fast interaction between keypresses and motion in the model, which is necessary to turn corners precisely. The display of your pacman on the remote screen may lag slightly if the network is not performing well. Your pacman is always shown in yellow, whereas your opponents is always pink.

Ghosts's motion and strategy are always modelled on their home computer. Ghosts cannot traverse the tunnels and visit the remote screen.

We use the following terminology to distinguish between visiting pacman and various game objects:

- LOCAL: the game object is a local game object, and is currently on the local screen.
- AWAY: our pacman is current away on the remote screen.
- REMOTE: a game object on the remote screen that our AWAY pacman might interact with.
- FOREIGN: the other player's pacman, when it is visiting our screen.

In this document, when these terms are capitalized, they have these specific meanings.

When our pacman is AWAY, the local model needs to know about everything it can interact with. At game start or restart, each computer sends the other a copy of its maze. The game ships with three different mazes, though more can be added. A player can choose a maze by selecting "-m <mazenum>" on the command line, where mazenum is an integer, typically from 0 to 2. If no maze is specified, it is selected randomly. The maze that is sent includes the location of all the food and powerpills.

The model running on your computer keeps two mazes in memory - the LOCAL one and the REMOTE one.

To keep the copies of the maze running on your computer and the remote computer synchronized, each computer continuously informs the other of actions. The actions include the pacman moving (position, direction and speed), the LOCAL ghosts moving, ghosts changing state, eating food or powerpills, ghosts getting eaten, and the player dying.

On receipt of these messages from the remote computer, the local computer updates the relevant maze - this might be the local maze if the other pacman is currently FOREIGN (ie it is visiting our maze), or it might be the remote maze if the other pacman is REMOTE.

When our pacman visits the remote screen, it becomes AWAY. Our computer first sends a "pacman arrived" message, so the remote computer can initialise any state. Whenever our pacman moves, our computer sends "pacman update"

messages to the remote computer, giving the current position, direction and speed of our pacman. It does this irrespective of whether the pacman is LOCAL or AWAY.

Whenever our LOCAL or AWAY pacman eats food or powerpills, this is detected by the model running on our own computer, using its copy of the LOCAL or REMOTE maze as appropriate. Our computer then sends an "eat" update message to the remote computer informing it that food or a powerpill has been eaten. Thus, even when our pacman is AWAY, interactions between it and food are still handled by the local model.

Our computer also sends "ghost update" messages whenever the LOCAL ghosts move. These give the position, direction, speed, and mode of each ghost. Amongst other things, mode includes whether the ghost is in "FRIGHTEN" mode (having turned blue, and being edible). The remote computer also sends such "ghost update" messages to our computer. Our model uses this information to update a local model of the REMOTE ghosts to determine if our pacman was either killed by one, or has eaten one.

If our model detects that our AWAY pacman has eaten a REMOTE ghost (while it was in FRIGHTEN mode), it sends a "foreign pacman ate ghost" message to update the remote system.

If our model detects that our AWAY pacman was killed by a REMOTE ghost, it sends a "foreign pacman died" message.

If our model detects that our AWAY pacman has traversed the tunnel again, and is now LOCAL, it sends a "foreign pacman left" message. The remote side will stop displaying the pacman on its main screen (and display it again on the smaller remote screen if enabled).

Some events require that our AWAY pacman be forcibly sent home. This happens when the level is completed on the remote screen, for example. The remote system sends a "pacman go home" message. Our system then resets our pacman to LOCAL, and sends a "foreign pacman left" message in reply.

Whenever our pacman's score changes, whether our pacman is LOCAL or AWAY, our system sends the remote system a "score update" message.

The local game board also has states associated with it. These are defined the class GameMode in `pa_model.py`:

- `STARTUP`
- `CHASE`
- `FRIGHTEN`
- `GAME_OVER`
- `NEXT_LEVEL_WAIT`
- `READY_TO_RESTART`

Changes between these states are communicated using "status update" messages.

Gameplay only happens in `CHASE` and `FRIGHTEN` state (the difference being whether a powerpill has recently been eaten). The software is in `STARTUP` state while playing the startup jingle.

If either player loses their last life, the game ends. The losing player's computer goes to `GAME_OVER` state, and sends a status update message. The other side then also moves to `GAME_OVER` state.

From `GAME_OVER` state, if the local player presses "r" to restart, the local computer goes to `READY_TO_RESTART` state and sends an update message. The game restarts when the second player also presses "r", and sends a replying "`READY_TO_RESTART`" status update.

When a level is cleared on a screen, that screen's system goes to `NEXT_LEVEL_WAIT` while it plays the jingle and the player gets ready. Completing a level does not affect the level being played on the other screen, except the pacmen positions are reset.

The complete list of messages in the current version of the protocol is therefore:

1. **maze update**
2. **pacman arrived**
3. **pacman left**
4. **pacman died**
5. **pacman go home**
6. **pacman update**
7. **ghost update**
8. **ghost was eaten**
9. **foreign pacman ate ghost**
10. **eat**
11. **score update**
12. **lives update**
13. **status update**

4.1 Existing networking code

The existing networking code is in `pa_network.py`. The protocol follows the above description, but is not a good implementation. It uses a TCP[1] connection for communication, uses verbose message names, and uses `pickle` to encode and decode payloads. You should probably pay attention to the code

[1] we will discuss TCP and UDP in course materials

in `check_for_messages()`, which ensures that when multiple messages are received by one call to `recv()`, the additional messages are not missed, but are kept and processed.

This protocol is less than ideal for a number of reasons:

- The encoding is python specific. Your brief is to produce a specification that could be implemented using any programming language.
- Pickle is not robust to malicious input. Your brief is to produce a specification that indicates how received values should be sanity-checked. For example, a ghost number of 5 would be illegal.
- The protocol is a strange mixture of binary encoding for message length, text encoding for message type, and binary pickle encoding. This is really ugly! Your brief is to write a specification that is clean. You can produce either a text-based protocol, or a binary-encoded one, but be consistent.
- The protocol is chatty. For example, it sends multiple messages for each video frame, including, for example, four separate ghost update messages. If you wish to combine multiple updates into one message, you may do so.
- The protocol only uses TCP. For some messages, such as sending the maze, this is sensible. For others, UDP would provide more timely delivery. Your protocol can use TCP, UDP, or both. Bear in mind though that if you use both, you may have to worry about message ordering between UDP and TCP connections (UDP messages may overtake TCP ones for example). If you choose UDP only, you must state how missing packets will be handled.

5 Your Task

- Your task is to write a protocol specification for a protocol to replace the one above. You must not directly use any existing protocol, but you can modify such a protocol if you wish. However, your protocol specification cannot simply reference an external protocol specification - it should be possible to implement your protocol without reading any other specification other than TCP or UDP.
- Your protocol may use TCP, UDP, or both.
- Your protocol should use either a text-based encoding or a binary encoding. You should probably not mix the two without good reason (if you have a good reason, you should explain it).
- Your protocol must not use pickle.
- You need to specify how to encode the information currently sent in all 12 existing message types, though your protocol does not necessarily need to have 12 distinct messages.

- You need to specify any additional processing the receiver should perform on receipt of these messages *that is not already performed in the existing code*. You do not need to specify processing that the existing code already performs, such as what the Model does with a particular message type. Examples of additional processing might be retransmitting lost data when using UDP.
- You may use text from this document if it helps.
- Hint: if you don't understand how a particular message type should work, or what the values included are, try adding print statements to `pa_network.py` to show the message contents before encoding or after decoding.
- You may ask on Piazza for more information about how things work, but I will only answer public questions, so everyone has the same information. It is a normal part of writing a specification to gather information, and I will be happy to clarify anything that is unclear about how the existing protocol works, or how the game works.
- If there are differences between how the python code of the existing protocol works and how it is described in this document, the python code is authoritative and takes priority (there are no deliberate differences, but errors may have crept in).

6 Marking

This task is worth 17% of the course marks. You will be arranged in groups, and will assign a mark to each other person's specification from your group. When marking, you should consider the difficulty you would have understanding the specification well enough to implement it in python.

When you are marking, you are not marking the quality of the English, so long as it is intelligible and unambiguous.

You should assign marks for:

- Conciseness. Don't waffle. Be specific.
- Correctness. Will the protocol fail if implemented as specified.
- Unambiguous. Do you understand how to code what is specified in all cases?
- Completeness. Are some things missing?
- Examples. Although examples are not part of the specification (in technical terms, they're "non-normative"), use of a few examples may be useful in complex cases. But not to the extent this severely contradicts conciseness.

The expectation is that for most groups, a median mark for this coursework will be between 60 and 70%, representing a 2:1 grade. More complete marking guidance will follow.