Design & Professional Skills — Pacman Protocol Specification Assignment

1 Running Pacman

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

To run the program in multi-player mode, one player needs to act as a "server" and the other as a "client". Once the network connection has been established, there is no different between how the server and client behave.

To run the server:

```
python3 payman.py -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 -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, get killed by ghosts, eat the powerpills, and eat frightened ghosts. At present, the two pacmen do not directly interact. They simply pass straight through each other.

2 About the code

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

Your pacman is 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.

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, each computer sends the other a copy of its maze. The mazes of the client and server differ slightly, to give the game a little variety. The maze that is sent includes the location of all the food and powerpills.

The local model keeps two mazes in memory - the LOCAL one and the REMOTE one.

When the REMOTE pacman eats food or powerpills on the remote screen, the remote computer sends "eat" update messages. Our computer receives these update messages, and uses them to update its copy of the REMOTE maze. It does this even when our own pacman is LOCAL.

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. Our computer now sends "pacman update" messages to the remote computer, giving the current position, direction and speed of our pacman.

If our AWAY pacman eats food or powerpills on the remote screen, this is detected by the model running on our own computer, using its copy of the REMOTE maze. Our computer then sends an "eat" update message to the remote computer informing it that food or a powerpill has been eaten.

While our pacman is AWAY, the remote computer sends "ghost update" messages. 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). 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, 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.

Some events require than our AWAY pacman be forcibly sent home. This happens when the level is competed 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 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 different 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 arrived**
6. **pacman go home**
7. **pacman update**
8. **ghost update**
9. **ghost was eaten**
10. **foreign pacman ate ghost**
11. **eat**
12. **score update**
13. **status update**

2.1 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 connection for communication, uses verbose message names, and uses pickle to encode and decode payloads. You should probably pay attention to the code 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 is sending multiple messages for each frame, including for example four 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.

3 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.
- Your protocol may use TCP, UDP, or both.
- Your protocol should use either a text-based encoding or a binary encoding. You should not mix the two without good reason (if you have a good reason, you should explain it).
- You need to specify how to encode all 11 existing message types.
- 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.
- 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.

4 Marking

This task is worth 15%. You will be arranged in groups, and will assign a mark to each other specification from your group. You will each have to implement the protocol from your group that receives the best mark, but that is a separate assignment.

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.

More complete marking guidance will follow.