2 The Universe Teachpack: Client/Server Interactions

The goal of this afternoon is to learn to design interactive programs using the *universe* teachpack in DrScheme, where several players (clients) compete or collaborate using a server to administer the communications between them and act as an arbiter when necessary.

Our players (clients) each are designed as *WORLD*s, the server that manages the worlds is the *UNIVERSE*.

You may want to read through the lecture notes for this afternoon, examine the sample code provided here, but should make sure that you design at least one part of the interactions on both the server and the client side yourself.

If you have time left over, focus on the systematic design of a single-player interactive game that is somewhat more complex than what you have had a chance to do this morning.

2.1 Designing the client

The context for our example code is a children's card game of war. Every player shows the top card of his or her deck and the highest card takes all cards played. The gam eends when one player runs out of cards.

We focus on a very simplified version - with no ties allowed (everyone takes back their card), and only two players in the game. Actually, we do not even complete this much — you have to fill in some pieces yourself.

Look at the file `war-player-simple.ss`. We have made this intentionally very simple. The player starts with a deck of cards, and shows the card on the top of the deck. When the player hits the space bar, the top card is moved to the bottom of the deck and we can see the next card in the deck. The game stops when the deck is empty.

The state of the world is our deck of cards — a list of `String`s of the form `"Kd"` for the King of Diamonds, or `"8s"` for Eight of Spades.

We now add two new features to our game. First, we have to tell the universe what is our top card. Next, the universe will send us cards to add to our deck, if we win the turn. So, we need to learn how to send a message and how to receive a message.

Receiving a message

This code is actually already there. The function `receive` consumes the current `world` and the `message` that has been sent and produces a new `world`.

We decide that the each message should be a list of cards. That way we may get no cards at all, or get two cards (or even more, if we extend the game to several players).

Alternately, when the game ends, the message will be the symbol `'done`.

The following code does the work:

```
;; receive a message: if 'done - stop the world
;; else append the card you won to the end of your list
;; receive-world: (World [Listof String] -> World
(define (receive-world w msg)
  (cond
    [(symbol? msg) 'stop-the-world]
    [else (append w msg)]))

;; test receive-world:
(check-expect (receive-world deck 'done) 'stop-the-world)
(check-expect (receive-world deck empty) deck)
(check-expect (receive-world deck (list "4s" "Jc"))
              (list "Kh" "Qd" "3s" "8c" "4s" "Jc"))
```

Sending a message to the server

In the simple program, on key event we just moved the top card to the bottom. We now want to send this card to the server and remove it from our deck.

Originally, the `on-key-event` function has been defined as:

```
;;---
;; on key event (space bar)
;; if space bar is pressed
;;   remove the top card from the deck and put it on the bottom
;;   the next card will now be seen by us
;; ignore all other keys, or when our deck is empty
;; on-key-event: World KeyEvent -> World
(define (on-key-event alist ke)
  (cond
    [(string=? ke " ")
    (cond
      [(empty? alist) 'stop-the-world]
      [else (append (rest alist) (list (first alist)))])]
    [else alist]))

;; test on-key-event:
(check-expect (on-key-event deck "right") deck)
(check-expect (on-key-event empty " ") 'stop-the-world)
(check-expect (on-key-event deck " ")  (list "Qd" "3s" "8c" "Kh"))
```

When the world wants to send a message, it needs to make a `package` as the result of the `on-key-event` function (or any other function that before produced a new state of the world. A `package` combines the new state of the world with the message that will be sent to the server.

All messages (whether from the server to the client of from the client to the server) must be plain S-Expressions. They cannot be *struct*s, because there is no way to communicate the structure definitions across the client-server connections.

So, we decide that, considering we are always sending only one card to the server, the message will be a simple `String` that represents the desired card.

So, the modified function `on-key-event` will be:

```
;;---
;; on key event (space bar)
;; if space bar is pressed
;;   remove the top card from the deck
;;     and send it to the server
;;   the next card will now be seen by us
;; ignore all other keys, or when our deck is empty
;; on-key-event: World KeyEvent ->
;;               (Package: World String) | World
(define (on-key-event alist ke)
  (cond
    [(string=? ke " ")
    (cond
      [(empty? alist) 'stop-the-world]
      [else (make-package (rest alist) (first alist))])]
    [else alist]))

;; test on-key-event:
(check-expect (on-key-event deck "right") deck)
(check-expect (on-key-event empty " ") 'stop-the-world)
(check-expect (on-key-event deck " ")
              (make-package (list "Qd" "3s" "8c") "Kh"))
```

Run the code in the file war-player.ss. It does not worry about sending a message unless it is connected to the server.

Notice how th bigbang clauses have been expanded:

```
;; run the world
(big-bang deck
          (stop-when end-the-world?)
          (on-receive receive-world)
          (on-key on-key-event)
          (register "127.0.0.1") ;; LOCALHOST
          (on-draw show-card))
```

For now, comment out the line that registers this player with the universe. If needs to know the IP address of the server. You can play the game on one machine — in that case the several running programs communicate over what is know as LOCALHOST, with the IP address being always "127.0.0.1".

2.2 Designing the server:

The code for the universe is actually more complex than it needs to be - as we were trying to make sure there are two players in the game, no more, and no less.

Let's start from the beginning.

We first have to decide what information does the universe need to run the game. It will always need a list of all worlds that are currently connected to it. The rest depends on the game we are working on. We have decided to record the number of cards the first and the second players have, and a pair of the two cards that have been played in this round.

The universe needs to know what to do if a new world joins the universe, and how to *process* a message from the world. Of course, this may involve *sending* a message (reply) to the world, so we need to know how to do that as well.

Processing a message

We do not have a complete code here — we should make sure that after the player sends us a card to play, we will not accept anothercard until the other player has sent his card and the turn is completed. In a real game, once we get the second card of the current turn, we should send both cards to the winner and an empty list of cards to the looser.

We decided to make the task of replying o the player's message very simple: we just send back to every player the card we have received. On the player's side this will result in placing the card the player has sent us back on the bottom of the player's deck.

The function that processes a message needs to know the current state of the `universe`, it needs to know which `world` sent the message, and, of course, the contents of the `message`. In turn, it needs to produce a `bundle` that consists of three parts: the new state of the `universe`, a list of `mail`s to be send to various `world`s, and a list of `world`s that should be disconnected at this time.

Each `mail` consists of the `world` to which the message should be delivered, and the contents of the `message`.

So, our `process` function is actually quite simple – we just send the card we have received back to the `world` that has sent it to the `universe`':

First let us see the data definitions for the `universe`] and for the `bundle`:▌

```
;; A Play is (make-play String String)
(define-struct play (card1 card2))

;; examples of play:
(define one-only (make-play "" "Kd"))
(define two-only (make-play "8h" ""))
(define none-yet (make-play "" ""))
(define one-wins (make-play "8h" "2d"))
(define two-wins (make-play "8h" "Jd"))

;; Universe state:
;; Number Number Play [Listof World]
;; Interpretation: number of cards each player has;
;;                 the pair of cards played by each
(define-struct war (p1 p2 played worlds))

;; initial universes
(define war-none (make-war 0 0 none-yet empty))
(define war-one (make-war 20 0 one-only (list iworld1)))
(define war-two (make-war 20 12 two-wins (list iworld1 iworld2)))▌

;; sample two decks to give to the players
(define deck1 (list "Kh" "Qd" "3s" "8c"))
(define deck2 (list "Jh" "Ad" "9s" "5c"))

; Bundle is
; (make-bundle UniverseState [Listof mail?] [Listof iworld?])▌
```

We can now look at the function that processes a message from the `world`:

```
;; process a message
;; just send the card back to the player for now
;; process: UniverseState World Message -> Bundle
(define (process a-war iw a-card)
  (make-bundle a-war
               (list (make-mail iw (list a-card)))
               empty))

;; test the fake process function:
(check-expect
  (process war-two iworld2 "Ks")
  (make-bundle war-two
               (list (make-mail iworld2 (list "Ks")))
               empty))
```

Adding a new World

When the `world` sends the `register` request to the `universe`, the `universe` may or may not accept the connection, and it may need to send messages to either the newly added `world` or possibly also to the `worlds` already connected.

So, the function `add-world` consumes the current state of the `universe` and the `world` that is requesting to be registered, and, again, produces a `bundle`.

Of course! — the `bundle` contains the new state of the world, in which the list of `worlds` now has the new `world` added (if we allowed it to join the game). It also contains a list of messages we want to send at this time - to both the new `world` and to the `worlds` already connected, and finally a list of `worlds` to disconnect. If we do not accept the new world, the `bundle` we produce will include the new `world` among those to be disconnected.

The code for our `add-world` function is quite complex, and so we resorted to our golden rule, *make a wish list if the task is too complex* — there is a helper function `add-if-ok` that adds the given world to our current list of worlds and updates the number of cards the new player has when the player is allowed to join the game; there is a helper function `if-not-ok` that adds the given world to the list of worlds to disconnect, if we already have two players; and a helper method `mail-to` that produces the mail message to the world that has just joined the game, containing the inital deck of cards that has been dealt. (Well, we cheat, and give each player only four specific cards for now.) So, with the helpers out of the way, here is the code for adding the `world` to the `universe`:

```
; add the given world to the universe, if appropriate
;     notify the given world if the request is denied
;     if accepted, send the world its inital deck of cards
; add-world: UniverseState World -> Bundle
;
; Bundle: [add iw to the list of worlds the universe keeps:
;                             --- only two are allowed]
;         [make a mail to iw with its deck]
;         [disconnect a world if it is not allowed to join]
(define (add-world a-war iw)
  (make-bundle (add-if-ok a-war iw)
               (mail-to a-war iw)
               (if-not-ok a-war iw)))

;; test add-world:
(check-expect
  (add-world war-none iworld2)
  (make-bundle (make-war 0 4 none-yet (list iworld2))
               (list (make-mail iworld2 deck2))
               empty))
(check-expect
  (add-world war-one-on iworld2)
  (make-bundle (make-war 4 20 none-yet (list iworld2 iworld1))
               (list (make-mail iworld2 deck1))
               empty))
(check-expect
  (add-world war-two iworld3)
  (make-bundle (make-war 20 12 two-wins (list iworld1 iworld2))
               empty
               (list iworld3)))
```

Read through the helper methods - and rewrite the code so the `universe` always accepts a new connection and when needed disconnects the `world` that has been connected for the longer time.

Disconnecting a world

When a `world` closes up, or the program the the `world` is executing finishes running, the `universe` notices that the `world` is disconnected. This changes the state of the `universe`, and so the `universe` needs to know what to do. The function `disconnect-world` consumes the current state of the `universe` and the `world` that wishes to disconnect and produces, *guess what!* — a new `bundle`. Of course, the `world` that initiated the *disconnect action* should appear in the list of worlds to be disconnected:

```
;; When a world wants to disconnect, just let it do so.
;; Remove it from the list of world the universe keeps
;; and send no messages
;; disconnect-world: Universe World -> Bundle
(define (disconnect-world a-war iw)
  (make-bundle (make-war (war-p1 a-war)
                         (war-p2 a-war)
                         (war-played a-war)
                         (remove (war-worlds a-war) iw))
               empty
               (list iw)))

;; test disconnect-world:
(check-expect
 (disconnect-world war-two iworld1)
 (make-bundle (make-war 20 12 two-wins (list iworld2))
              empty (list iworld1)))
```

Note: You are not wondering what does the `remove` function do - it is yet another helper.

Running the universe

We now have to run the `universe`. To run these programs on one machine, make a copy of the `war-player.ss` so that you have two of them open in your *DrScheme*, and have the `war-universe.ss` open as well.

To run the `universe` we need to run the `universe` function with clauses that provide the functions we have designed:

Remember the `universe` named `war-none` we have defined when we presented the data definitions — it had no players signed up, no cards dealt, and no cards played. That is where we start the initial `universe`:

```
;; A Play is (make-play String String)
(define-struct play (card1 card2))

;; Universe state:
;; Number Number [Listof String] [Listof World]
;; Interpretation: number of cards each player has +
;;                 the list of cards played by each
(define-struct war (p1 p2 played worlds))

(define none-yet (make-play "" ""))
(define war-none (make-war 0 0 none-yet empty))

;;-----------------------------------------------
;; Start the universe with no players signed up
(universe war-none
          (on-new add-world)
;          (check-with cons?)
          (on-disconnect disconnect-world)
          (on-msg process))
```

We commented out the `check-with` clause. It should specify a predicate that will verify that the given piece of data is a properly defined instance of the `universe`.

Note: Design the predicate that will verify that the state of the `universe` in our program has been defined correctly.

Once the universe is up and running, start one of the two `war-player.ss` programs and see that the universe recognized that player. Then start the second version of the `war-player.ss` as well. Now, click on the window showing the card for the first player and hit the space bar. Do the same for the second player. Close the window and observe the disconnect action. Start the program again - and it should get connected again.

Now, try to play with a friend on another machine — all youhave to do is supply the correct IP address in the `register` clause.

2.3 On Your Own

Modify the game *Catch the butterfly* that you have designed in the morning, so that several players are chasing after the same butterfly. When a player wants to catch the butterfly, she sends a message to the server with the location of her net. The server keeps sending messages to all players on each tick, indicating the new position of the butterfly. The game ends when one of the players catches the butterfly.

Alternately, every time a butterfly is caught the old one disappears and a new one appears at a new random location.

Oh, yes, — do have fun again :)