CS 2500, Lab 10—Universe

- Work in pairs
- Change roles often!
- Follow the design recipe and/or the abstraction recipe for every problem.

In the beginning...

In this lab, you'll be converting a chat client that only works on a single machine to interact with a chat server and multiple other clients. Download the initial chat client to get started.

Altering the Chat Client

The Universe teachpack allows for client/server programs– that is, programs where a server manages multiple clients, and a client and the server communicate through *messages*.

To start our client, we use the big-bang form that we've been using all semester, while the server is started (by your freindly TAs) using a new universe form. A longer description of client/server programming can be found on the main Universe HelpDesk page.

Clients

Client programs look very similar to the kinds of interactive programs we've already written. There are just a few changes involved:

1. A `name` clause to identify the client,
2. A `register` clause to connect to a server, and
3. New protocols for sending (*packages*) and receiving (*messages*) to/from the server.

Connecting to the server

To connect we use the clause `(register String)`, where the String is either a *hostname* or an *IP address*. When `big-bang` is executed, it attempts to connect to the `universe` running on the given machine. If the connection fails the program (i.e., the client) will run locally without sending messages.

In addition, a client should specify a name using `(name String)` to identify it to other clients. In our chat program, we will use it as the nickname of each connected user.

Sending messages to the server

To send messages to the server client functions that originally returned a new `World` can instead (optionally) return a package. For example, the `on-key` contract was originally:

```
;; on-key :  World KeyEvent -> World
```

But now we'll treat it as:

```
;; on-key :  World KeyEvent -> [Or World Package]
;; An [Or X Y] is one of:
;; - X
;; - Y
```

A `package` consists of two items: a new World (as before) and a *message* to send to the server.

```
;; A Package is:  (make-package World Message)
;; A Message is an SExpr
```

To make a package use `make-package` (yup), and as with Worlds, what makes up a *Message* is up to the server and clients. At first, we will simply use Strings.

Receiving messages from the server

To receive messages from the server, include the on-receive clause. It takes a function with the following contract:

```
;; World Message -> [Or World Package]
```

When a message is received, the function is passed the current local World state and the message from the server. The result of the function is either just the next World state, or a Package that contains the next World state and a Message to send back to the server.

Modifying the Client

First things first... your name's not `Alfred E. Neumann` right? Modify your *nick* to something reasonable (keep it at least PG-13 please). Now look through the code, figure out where we've marked ****** MODIFY HERE ******, and try to get an idea of what the code does.

Now that you have a background on the World, Universe, and the client code, we can start setting things up. Your basic chat client currently echoes what you've typed to your screen whenever you hit *Enter*. What we need to do is make the client connect to the server, and start sending and receiving messages.

Exercise 1: Update the `handle-key` client function to return a `Package` whenever the you hit enter. Unlike the local implementation it should not add text to the buffer, but instead just send it as a message to the server.

Exercise 2: Design the function `handle-msg` that can be installed with `on-receive`. The function should take a Message (i.e., a String) and add it to the line buffer of the client. When you're done, add `(on-receive handle-msg)` to the `big-bang` form.

Exercise 3: Adjust the `big-bang` form to register with the server.

Now run it! It should now function as a little chat client. You may see other people connecting, and you should be able to send them messages. The TAs should be starting up a client up on the projector (as you read this) so you can see people connecting, sending messages, and disconnecting as they work on the lab.

Altering the protocol

Right now the server and its clients send around Strings as messages. It would make sense to make the protocol more extensible to deal with future changes, additions, etc. We do this by defining a Message be a *non-empty* `[Listof String]`. The first String defines the *type* of the Message and the meaning of the rest of the strings in the list depend on the message type.

To avoid *spoofing*, the following **protocol** will be used by the server to communicate to the clients. The server sends messages of format (list String String String), where the first String is the type of message (ex. "MSG" or "ACTION"), the second String is the name of the client, and the third String is the content of the message. When clients send a `‘‘MSG’’` Message to the server, they do not send a name, the server will add the appropriate client name before forwarding the message to other clients. In other words, the only Message sent from clients to the server at this point is `(list ‘‘MSG’’ <msg>)`. All other Messages will be generated by the server for connections and disconnections.

A sample server message: `(list ‘‘MSG’’ ‘‘Nick’’ ‘‘Sample Message’’)`

We are also running an advanced server at the address given by the TAs. You can test your advanced client by connecting to it. It supports all the messages described below.

Exercise 4: Change the protocol/message send functions as described above (minor adjustments to contracts and functions). Make sure you handle `‘‘MSG’’` messages (you can discard all other types for now).

Exercise 5: Add this functionality to your implementation:

When a client connects, it will receive a message from the server with information about how many clients are connected to the server and what are the names of the clients.

The format is `(list ``CLIENTS'' <number> <clients>)`. Let's also make it possible for the client to obtain that information anytime, by typing ``` ``/clients'' ```.

Exercise 6: Add *action messages* to the protocol. The user specifies actions by starting a message with ``` ``/me `` ``` (not part of the actual action). For example, if the client ``` ``vlad'' ``` types: `/me watches TV` and hits Enter, the message that will be displayed to all clients should be `*vlad watches TV`.

Hint: It will be useful to first design a `string-prefix?` function, that takes two strings and checks to see if the second string is a prefix of the first, and a `string-remove-prefix` function that removes a number of characters from the front of a string (`substring` is useful for both).

Exercise 7: Update the client so that each client will have a "status". As opposed to actions or regular messages, a client is associated to a status as long as it is connected. Of course, an active client can change its status, but it will always have one. When a client first enters the chat room, its status is ``` ``Available'' ```. At any time, the client can change its status by typing: ``` ``/newstatus 'something''' ```, where 'something' can be any String. A client can inquire about another client's status by typing `"/status?  clientname"`. If the client is not connected, the status will be ``` ``Offline'' ```.

Use messages of the form `(list ``STATUS'' <user>)` when asking for user's status and messages of the form `(list ``UPDATE-STATUS'' <new-status>)` when updating your own status.

Expect to receive messages of the form `(list ``NEW-STATUS'' <user> <status>)` when user updates their status or when you send the command ``` ``/status? username'' ```. You should display the messages as ``` ``clientname status: the-new-status'' ```.

Writing the server

To write a server, we will be using the universe form. It is similar to big-bang, and we'll be interested in the following events: *new connections*, *disconnections*, and *messages*. Here's how we'll be using those events:

Server events

- *New* connection events happen when a client connects to the server, and are handled by the `on-new` clause.

  The contract for functions given to `on-new` is:

  ```
  UniverseState IWorld -> Bundle
  ```

- *Disconnections* happen when a client disconnects from the server, and are handled by the `on-disconnect` clause.

  The contract for functions given to `on-disconnect` is:

  ```
  UniverseState IWorld -> Bundle
  ```

- *Messages are sent from the clients to the server, and are handled by the* `on-msg` *clause.*

  The contract for functions given to `on-msg` *is:*

  ```
  UniverseState IWorld Message -> Bundle
  ```

UniverseState

The `UniverseState` is very similar to the `World` State for local programs, only that it keeps track of the State of the entire Universe, not just a World. At the very least, the `UniverseState` will have to contain all the client connections (a.k.a. IWorlds), to be able to send Messages from any client to any client.

IWorld

An `IWorld` is a datatype internal to the universe teachpack that describes a client connection.
For the purpose of this lab, there is one operation you can perform on an IWorld, which is `iworld-name`. `iworld-name` takes an IWorld and returns a String which represents the name of the client in the specific client connection (a.k.a. IWorld).
`iworld-name` will return, for a given IWorld (a.k.a. client connection), exactly the name specified by the name clause of big-bang in the specific client.

Bundle

A Bundle is a collection of three things:

- the next UniverseState
- a list of Mails (described later) to be sent to clients
- a list of IWorlds to be disconnected

For our purposes, the list of IWorlds to be disconnected will either be empty (in most cases) or the singleton list (for handling disconnections).
A `Bundle` is created using the `make-bundle` function, whose contract is:

```
;; make-bundle:  UniverseState [Listof Mail] [Listof IWorld] -> Bundle
```

How the basic chat server works

The server we design will keep track of the connected clients, and any time a client sends a message to the server, which will be a String, the server sends a *Mail* to each connected client (including the sending client) that contains a String containing both the client's name and the message sent by the client.

To create a Mail value, the `make-mail` function is used, whose contract is:

```
IWorld Message -> Mail
```

For example, if the client with the name `''vlad''` sends the message `''hello''` to the server, then each client should receive a Mail value from the server that contains the String `''vlad:  hello''`.

In addition, the server should send a Message (as before, a String) to each connected client whenever a client connects or disconnects.

For example, if the client `''chadwick''` connects, each client will receive the message `''chadwick connected.''`, and if it later disconnects, each (still-connected) client will receive the message `''chadwick disconnected.''`

The basic chat server implementation

We start with a basic server implementation (download). The server that you have connected to with your client follows this simple implementation.

- The UniverseState data used by the server should contain at least all the active client connections. The basic server keeps a list of connections (a.k.a. IWorlds) as the UniverseState.
- The function `handle-new` handles new connections to the server.

  ```
  ;; handle-new :  UniverseState IWorld -> Bundle
  ```

For testing functions dealing with IWorld(s), the universe teachpack defines three default IWorlds (a.k.a. client connections): iworld1, iworld2, iworld3, which are all different from one another.

- The function `handle-disconnect` handles clients that disconnect from the server.

  ```
  ;; handle-disconnect :  UniverseState IWorld -> Bundle
  ```

  The IWorld that disconnects has to be added to the list of IWorlds to be disconnected - the third part of a Bundle.

- The function `handle-msg` handles new messages from clients to the server.

  ```
  ;; handle-disconnect :  UniverseState IWorld String -> Bundle
  ```

To launch server the `universe` function is used:

```
(universe INITIAL-UNIVERSE
            (on-msg handle-msg)
            (on-new handle-new)
            (on-disconnect handle-disconnect))
```

INITIAL-UNIVERSE is the first UniverseState, when there are no connected clients.
A good way to test the client-server interaction right after modifications to the client or the server have been done is to run everything on the local machine: the server and a few clients (two is good for starters). The clients will have to specify (register LOCALHOST) in their big-bang form. If everything behaves as expected, you can now test the interaction over a network.

Note: To launch more than one client from the same machine, use `launch-many-worlds`:

```
(launch-many-worlds (run ‘‘chadwick’’) (run ‘‘vlad’’))
```

Possible extensions

Now that clients have the list of connected clients and the protocol is more extensible, you can change the client (and server) in many ways:

Exercise 8: Add `‘‘CLIENTS’’` functionality to your server implementation:
When a client connects, it should receive a message from the server with information about how many clients are connected to the server and what are the names of the clients. This information should be displayed only in the newly connected client's window.
Use the String `‘‘CLIENTS’’` for the Message type.

Exercise 9: Add private messages (messages directed to and received by a single client). The user specifies a whisper by starting a message with `‘‘/msg’’`, followed by the nick to whom the whisper should be sent, followed by the message to be sent (all separated by spaces). This will require changes to the client and server for full functionality. Use the String `''PRIVMSG''` for the Message type.

Exercise 10: Add the ability to change one's nickname. The user specifies a nick change by starting a message with `''/nick''`, followed by the new nickname. Continue using the argument to run as the original nickname, but note that this means that the client name and the displayed nickname can now be different. This will require changes to the client and server. Use the String `''NICK''` for the Message type.

Exercise 11: Challenge: If you find it interesting, go ahead and add an `''Invisible''` status. To implement a correct behavior of the clients and the server in the presence of this status, make sure that when someone becomes `''Invisible''`, the broadcast message will be that the specific client has disconnected. When the client actually disconnects, no message will be sent to other clients anymore. If the client changes its status, the broadcast message will say that it joined.

Exercise 12: Challenge: Add the possibility that a client is only visible to some subset of all other clients. If a client is `''Invisible''`, it should be able to become `''Available''` only to some of the other clients. For example, an `''Invisible''` client would specify `''/newstatus /available-one client-name''` to become visible to client-name.

Exercise 13: Change the GUI to show a list of connected users and their statuses (the `render-strings` may be useful for this).

Exercise 14: A client-only change is to highlight messages that mention the user's nickname. This highlighting can be performed by changing the color of the text for the line including the user's nickname.

Add your own extensions!