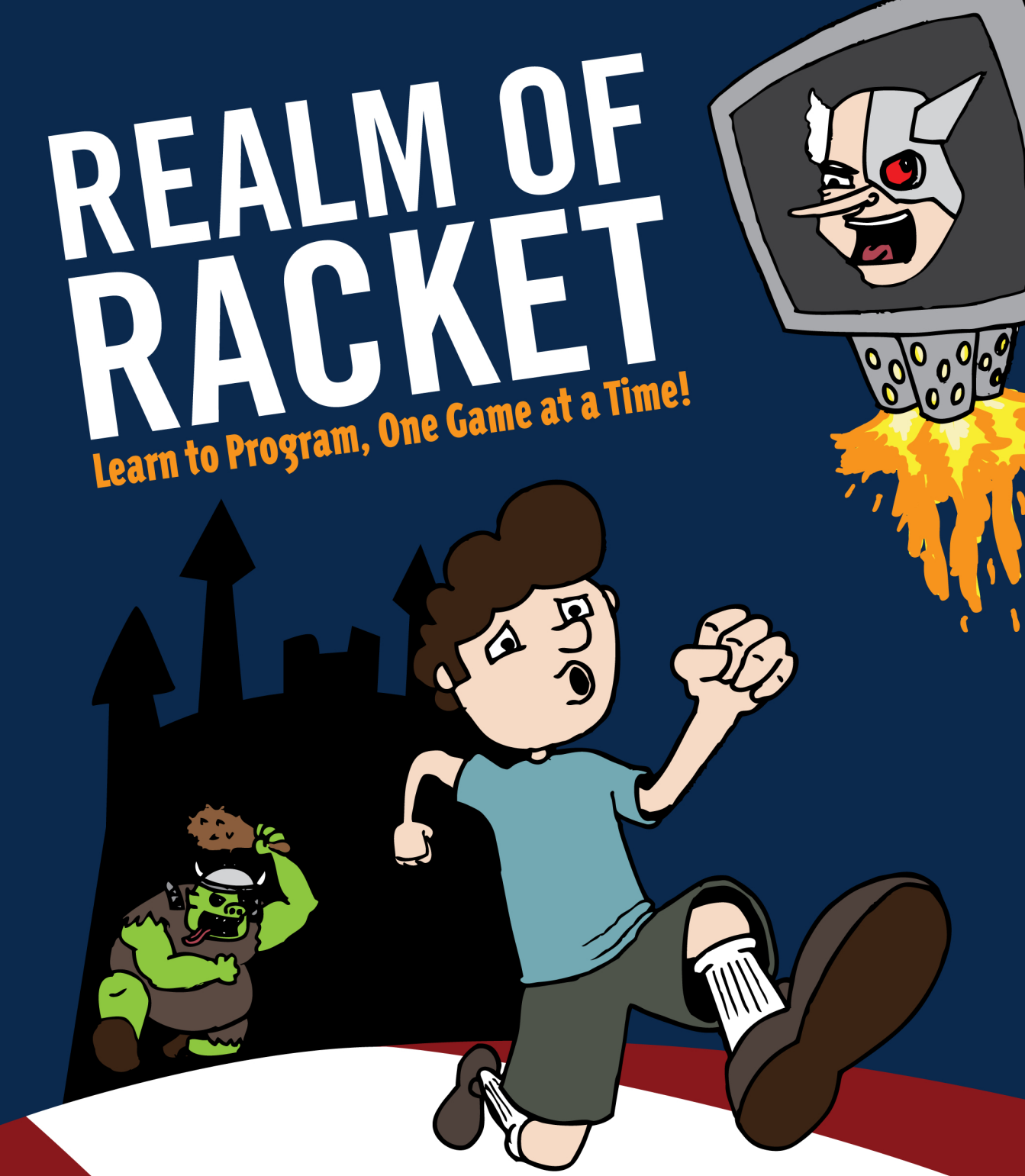
REALM OF RACKET

Learn to Program, One Game at a Time!



Forrest Bice, Rose DeMaio, Spencer Florence, Feng-Yun Mimi Lin,
Scott Lindeman, Nicole Nussbaum, Eric Peterson, Ryan Plessner
David Van Horn | Conrad Barski, MD
Matthias Felleisen

no starch press

;; Chapter 14
(Hungry Henry)

```
#|
Guess My Number is good and all, but it's getting a little old.
Let's take everything we have shown you and create a distributed
game from scratch. Today, we feed Hungry Henry.
|#
```

14.1 King Henry the Hungry

After fleeing from the squirrels, Chad resumes his search for a way out of the dungeons. While exploring, Chad comes across an enormous man sitting on a golden throne. "Hail, traveler!" the man calls, "I am King Henry the Hungry!" Then he proclaims, "Come hither, I have a proposition for you!" As Chad approaches, the king begins to tell his story. "Long ago, I was just like you, boy, a prisoner in the dungeons of DrRacket. Starving, I created a program to gather food, and what better food to eat than cupcakes? I forgot the `stop-when` clause and have been happily eating cupcakes ever since."

The king continues, "I've had so much practice that I could beat anyone at a cupcake-eating competition, and now is the time to prove it. But I need your help to change my program so that the whole world can participate. If you help me, I will help you find a way out." Can you help Chad run Hungry Henry's tournament?

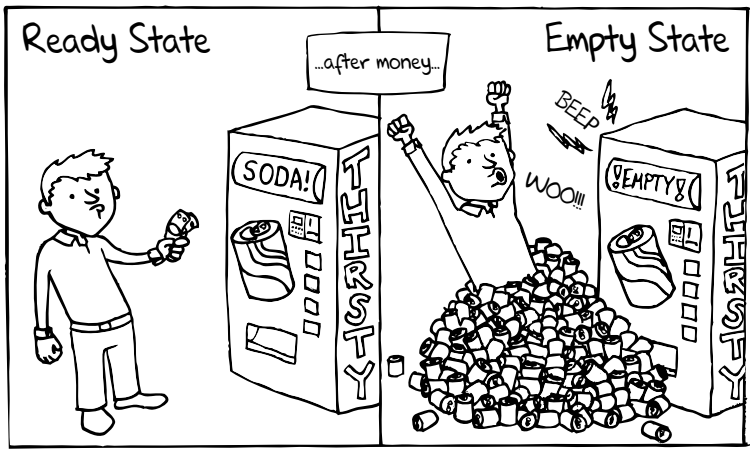
14.2 Hungry Henry, the Game

The goal of this chapter is to turn King Henry's eating competition into a distributed game. A server sets up a field of cupcakes and waits for players to sign up for a round of Hungry Henry. Once a player connects to the server, she is given her own avatar to control. The objective of the game is to navigate this avatar around the screen, getting to cupcakes before other players do. To navigate, the player sets waypoints on the screen by clicking the desired location. The avatar then travels to each of these waypoints in turn. But players must choose these waypoints carefully. Once one has been added, it may not be removed. Each time an avatar collides with a cupcake, the food is removed and the avatar increases in size. Of course, a growth in girth means the avatar slows down.

When all the food is gone, the game displays a table that lists the players and the number of cupcakes they ate. After a delay, the game restarts. If a player attempts to join while a game is in progress, she will be forced to watch. Once the game restarts, these spectators are assigned avatars, too.

14.3 Two United States

Before we show you how to implement the Hungry Henry game, we need to drill down on one more concept: the **state machine**. While all world games are state machine games, Hungry Henry makes explicit use of two distinct states in two different state machines.



Consider the ubiquitous soda machine. Initially, it is full. You put in a dollar bill, and the machine responds by dispensing a can of soda. The machine is now in a different state; it is partially empty. Eventually, it will dispense its last can and enter its final state, the empty state. Think of Hungry Henry's state machines like this, but with less liquid and more cupcakes.

In this chapter, the universe and all of its worlds are in one of two different kinds of states: **waiting** and **playing**. The universe starts in a waiting state. When it's time for the game to start, the universe switches to a playing state. Our program will thus need to understand how to handle events depending on the current state of the universe. Distinguishing these two states will significantly simplify our data and message protocol.

14.4 Henry's Universe

Our next step is to divvy up responsibilities between the server and the clients, and these responsibilities need to be carefully assigned. For example, if we let the client handle the movement of her avatar or the eating of cupcakes, who will enforce the rules of the game? You could easily imagine a player changing the client code to her own advantage. So our choices must prevent players from acting in this malicious manner.

The client will be responsible only for reporting a player's mouse clicks to the server and rendering the current state of the game. Essentially, the client will implement the `on-mouse` and `on-draw` specs for `big-bang`. The server, in contrast, does a lot more work. It is in charge of handling movement, collisions, eating, and ending the game. And, of course, whenever something changes, the server sends the new state of the universe to all the clients.

Message Data and Structures

So what kind of information is actually sent from client to server and vice versa? While the client needs to send only information about where the player clicked on the screen, the server needs to send back four kinds of messages: an ID to inform the player of her avatar's name, a fraction of the waiting period that has passed, the current state of the avatars and cupcakes, and the final scores.

To keep the protocol simple, the multipart messages are lists tagged with a symbol that identifies the type of message. Here is the only type of message the clients can send:

```
(list GOTO Number Number)
```

It is just a three-part list: a constant to identify the message type and the *x*-coordinate and *y*-coordinate of a mouse click.

Two of the server's messages are even simpler than that. When a client registers, the server responds with a new unique ID for the avatar. By defining an ID as a string, the server can make use of `iworld-name` to generate an easily recognizable ID.

The time message is just a number between 0 and 1, representing the percentage of wait time completed. While the server waits for enough players to sign up, the number gets closer to 1 but is never equal to 1.

The most complex message describes the state of the game:

```
(list SERIALIZE [Listof Player] [Listof Cupcake])
```

This message is a three-element list containing the constant `SERIALIZE`, a list of players, and a list of cupcakes.

Players are represented with prefab structures:

shared.rkt

```
#lang racket
(provide ...
         (struct-out player)
         (struct-out body)
         ...)

(struct player (id body waypoints) #:prefab)
```

Players need their own ID, which clients use to differentiate themselves from other players. The next field, `body`—think an astronomical body—is used to describe the location and size of a player. The last field, `waypoints`, is a list of waypoints from oldest to newest, the order in which the avatar will travel. Ordering points in this way is efficient because the program must look at the head of the list many times as the avatar moves incrementally along its path.

Both avatar bodies and cupcakes can be described as physical bodies:

shared.rkt

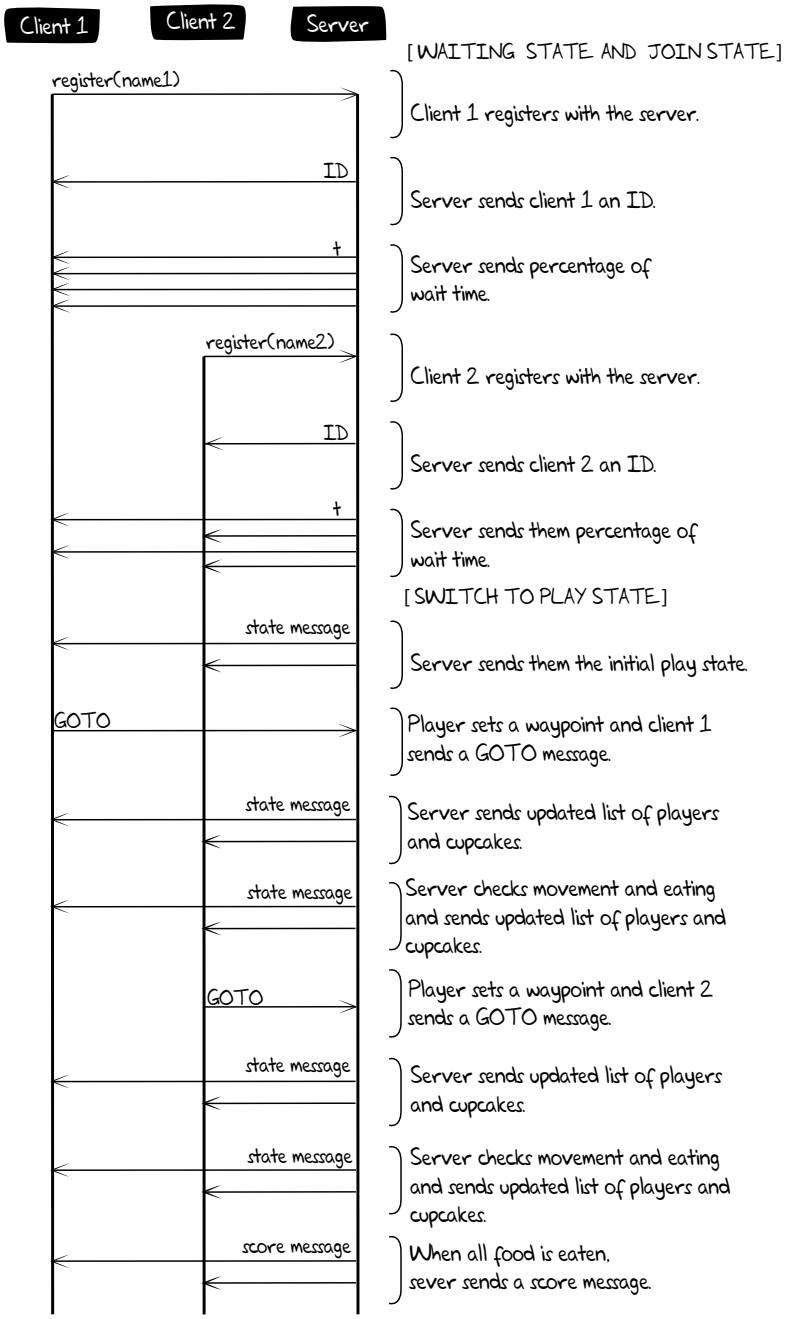
```
(struct body (size loc) #:prefab #:mutable)
```

The first field, `size`, is a positive integer that represents the radius of the `body`. The second field, `loc`, is a complex number that represents the actual location of the `body`. The structure is mutable to make it easy for the server to grow or move any object.

Finally, the score message contains just the expected table:

```
(list SCORE [Listof (list ID Number)])
```

At the end of the game, the server will send a list of two-element lists. The first element is a player's ID, and the second is the player's score.



Complex Numbers Are Good Positions

Complex numbers, such as `3+2i`, are convenient for representing coordinates, so we'll use them for waypoints. To model movements, we can use normal math operators, such as `*` and `+`. To access the *x*-coordinate of a complex number, we use the built-in function `real-part`, which gives us the real portion of the complex number. To access the *y*-coordinate, we use `imag-part`, which extracts the imaginary portion. If there is a need to construct an imaginary number from two real numbers, we use the `make-rectangular` function, which takes two real numbers and returns a complex number, where the real part is the first number and the imaginary part is the second number.

A Day in the Life of a Server

In the beginning, our server just ticks along, waiting for a client to join. When a client connects to the server, it is sent an ID. During this phase, the server continually keeps its clients up-to-date with the approximate time left until the game starts. Once time has run out and enough players have signed up, the server switches to a play state. From this point on, any new clients are considered spectators, and the server tells the players and spectators the location of all the food and avatars in the game. Once the players have eaten all of the food, the server tallies the scores and sends a `SCORE` message to the clients. Afterward, it resets to its `join` state, joining together all the players and spectators as the players for the next round.

A Day in the Life of a Client

The first action that any good client takes is to register with the server. In response, it gets an ID back and begins to receive time messages. Using these time messages, the client draws a progress bar and waits. When the client gets a message from the server that describes the state of the game, it means the game has started. The client switches to a play state and renders the state messages, which contain the location of all the food and players. Whenever the player clicks the mouse, the client must send a message to the server describing the location of the player's chosen waypoint. Eventually, the client gets a message that contains all score tallies, signaling the end of the current round of play.

14.5 State of the Union

Now that we know what our message protocol looks like and how the clients and server interact, let's turn to the data that is used within the client and the server. As we go along, keep in mind that our server and client are interacting state machines.

State of Henry

The client can be in one of two states: waiting or playing. Clearly, the client is in a waiting state until the server sends the first state message. Then it transitions to playing. Once a game ends, the client transitions back to waiting, giving that state a second purpose—to display the scores from the game.

With that in mind, let's call the first state an `app`etizer:

client.rkt

```
(struct app (id img countdown))
```

The `app` structure has three fields, which the client uses to render a waiting screen. The first field, `id`, names the player's assigned ID, which is later used in gameplay. During the waiting phase, it is `#f`. The next field, `img`, is the base image that displays messages to the screen. These messages include waiting text or the score table from the last game, if there was one. The last field, `countdown`, holds the time left until the game starts, and it is used to render a progress bar.

The second state, the playing state, is called `entree` because it follows an appetizer:

client.rkt

```
(struct entree (id players food))
```

As before, the first field is this player's `id`. The next two fields are lists of the current players and the available cupcakes.

State of the House

Like the client, the server has two states. The first state is a `join` state, representing the period during which players are allowed to join the server:

server.rkt

```
(struct join (clients [time #:mutable]))
```

The first field of `join` is a list of players who have joined. The second field holds the time that remains until the server intends to start the game.

The second server state is called `play`, and it represents the server while the game is in progress:

server.rkt

```
(struct play (players food spectators) #:mutable)
```

Like `join`, the first field of `play` stores a list of the players who are in the game. The second field, `food`, is a list of `body` structures that represent the remaining cupcakes. The

last field keeps track of the current spectators. When the server transfers back to the `join` state, these spectators are appended to the list of current players.

You may have noticed that neither `join` states nor `play` states include information about the `iworld`s that represent clients and allow the server to communicate with clients. Well, the preceding data representation is a bit of a lie. Instead of plain prefab players, the server uses lists of *internal* player representations:

server.rkt

```
(define-values
  (ip ip? ip-id ip-iw ip-body ip-waypoints ip-player)
  (let ()
    (struct ip (id iw body waypoints player))
    (define (create iw id body waypoints)
      (ip id iw body waypoints (player id body waypoints)))
    (values
      create ip? ip-id ip-iw ip-body ip-waypoints ip-player)))
```

This definition is a mouthful, but look closely, and you'll see that it's similar to how we handled forcing moves in the lazy version of Dice of Doom. With this structure definition, we can construct internal players the same way we construct ordinary struct instances. But what really happens is that an `ip` adds a field, which contains the prefab player that would result from the arguments given to `ip`.

Now we can send a representation of players to clients simply by calling the `ip-player` function and sending that result to the `iworld` in `ip`. To clarify, an instance of `ip` is never sent across a network. It is used only by the server to hold all of its knowledge about a player.

14.6 Main, Take Client

Since the client is clearly simpler than the server, we will deal with the main function of the client first. It handles three actions: drawing, messaging waypoints to the server, and receiving messages in return.

client.rkt

```
(define (lets-eat label server)
  (big-bang INITIAL
            (to-draw render-the-meal)
            (on-mouse set-waypoint)
            (on-receive handle-server-messages)
            (register server)
            (name label)))
```

This function takes a name that the client wishes to use as her ID and the address of the server that the client wishes to join. These two pieces of data are used by the last two clauses of `big-bang`.

The functions in the other clauses all follow the same pattern; they dispatch the event to a different function based on the client's present state. For example, the drawing and message-handling functions look like this:

client.rkt

```
(define (render-the-meal meal)
  (cond [(app? meal)    (render-appetizer meal)]
        [(entree? meal) (render-entree meal)]))

(define (handle-server-messages meal msg)
  (cond [(app? meal)    (handle-appetizer-message meal msg)]
        [(entree? meal) (handle-entree-message meal msg)]))
```

Very little thought is required in writing these. Before anything can be done, we need to understand the context of our situation—that is, the current state—and then pass the current state to the appropriate helper function.

Handling mouse events proceeds in a similar fashion, but the handler must take into consideration another condition:

client.rkt

```
(define (set-waypoint meal x y me)
  (if (and (entree? meal) (mouse=? me "button-down"))
      (make-package meal (list GOTO x y))
      meal))
```

Because the rules state that a player's only action is to click the mouse during the game, the mouse event handler checks these two conditions first. If true, `set-waypoint` sends the appropriate message to the server. If not, the state is returned unchanged.

The Appetizer State

While the player is waiting for the server to start a game, the client is in the so-called `app` state. In this state, it displays any message that the server sends. The top-level function for rendering appetizer states draws a progress bar:

client.rkt

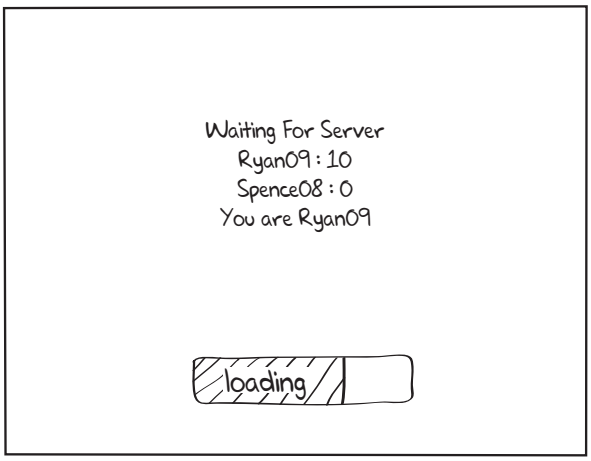
```
(define (render-appetizer app)
  (add-progress-bar (render-id+image app) (app-countdown app)))
```

The `render-appetizer` function adds the most recent message to the image in `app` by making a call to `render-id+image`. It then adds a progress bar on top of that image with a call to `add-progress-bar`. The `render-id+image` function takes the `app` structure and generates an image from the latest message the client received:

client.rkt

```
(define (render-id+image app)
  (define id (app-id app))
  (define base-image (app-img app))
  (overlay
   (cond
     [(boolean? id) base-image]
     [else (define s (string-append LOADING-OPEN-TEXT id))
           (above base-image (text s TEXT-SIZE TEXT-COLOR))])
   BASE))
```

This function renders the image as well as the player's ID if the server has sent it already. Adding a progress bar is so easy that we won't show it here.



The remaining events concern messages from the server:

client.rkt

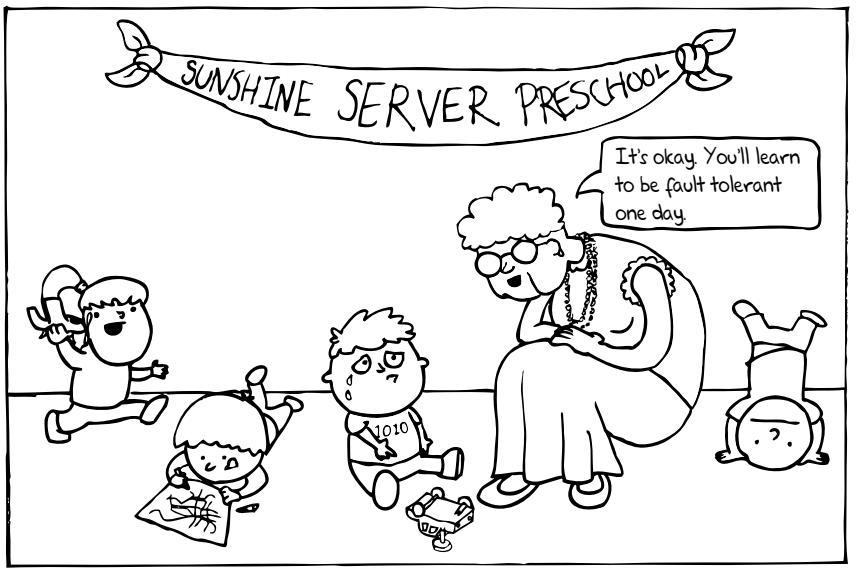
```
(define (handle-appetizer-message s msg)
  (cond [(id? msg)    (app msg (app-img s) (app-countdown s))]
        [(time? msg)  (app (app-id s) (app-img s) msg)]
        [(state? msg) (switch-to-entree s msg)]
        [else s]))
```

As the protocol specifies, this function handles three messages. The first two `cond` clauses should be obvious. They just switch out one field with the newly arrived message. The third `cond` clause switches the client to the `entree` state. The last clause makes our message handling fault tolerant; it handles any violation of the agreed-upon protocol by ignoring the message and returning the current state. To make the program fault tolerant, the predicates that check our messages need to be programmed defensively. For example, here is the definition of `time?`:

client.rkt

```
(define (time? msg)
  (and (real? msg) (<= 0 msg 1)))
```

It doesn't just check whether the message is a number but also ensures that number is real and between `0` and `1`.



There is only one more function to deal with the `app` state. The function `switch-to-entree` is called when the first state message arrives. It consumes the current state and the state message, and it returns an `entree`:

client.rkt

```
(define (switch-to-entree s m)
  (apply entree (app-id s) (rest m)))
```

Do you remember `apply`, the higher-order function introduced in chapter 7? Go back and reread that section if you don't. All `apply` does here is call the `entree` constructor on the current `id` and the remaining two elements of the state message, which just happen to make up the fields of an `entree`. Isn't that easy? Time to eat.

The Entree State

The `entree` state represents the client's playing state. In this state, the player may interact with the world by clicking the screen to direct the avatar. The client displays the player's avatar, all of the other avatars, and some information about each one in the rendering function `render-entree`:

client.rkt

```
(define (render-entree entree)
  (define id (entree-id entree))
  (define pl (entree-players entree))
  (define fd (entree-food entree))
  (add-path id pl (add-players id pl (add-food fd BASE))))
```

This function starts by drawing all of the food and players onto the base scene. It then draws the path of this client's player. So let's look at how players are drawn:

client.rkt

```
(define (add-players id lof base-scene)
  (for/fold ([scn base-scene]) ([feaster lof])
    (place-image (render-avatar id feaster)
                 (feaster-x feaster) (feaster-y feaster)
                 scn)))
```

The `add-players` function consumes this client's `id`, a list of all the players in the game, and a scene to add images on. The function uses `for/fold` to create a single image from the list of players. The given scene is used as the base case, and the function iterates over the given list of players. The loop creates an image with `render-avatar` and places it on the scene.

The `render-avatar` function creates an image of an avatar based on the client's `id` and the `feaster` that represents the player to be drawn:

client.rkt

```
(define (render-avatar id player)
  (define size (body-size (player-body player)))
  (define color
    (if (id=? id (player-id player)) MY-COLOR PLAYER-COLOR))
  (above
   (render-text (player-id player))
   (overlay (render-player-score player)
            PLAYER-IMG
            (circle size 'outline color))))
```

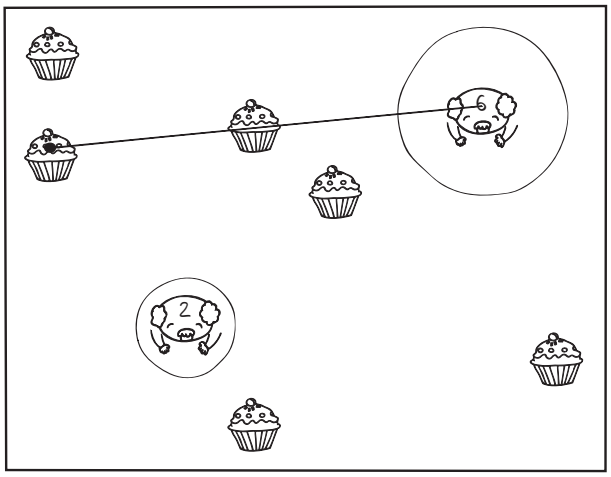
This function decides what color the avatar's bounding circle should be, based on the given `id`. It then draws the bounding circle that has a radius based on the size of the given player. The actual avatar is placed at the center of the circle. By drawing the player's avatar in this way, we have an accurate view of how large the avatar is, but we avoid pixelating the image by stretching it. We will leave it to you to write the rendering function that displays scores. The list of food is rendered in a similar fashion.

The last rendering step in the client concerns the path for this client's avatar:

client.rkt

```
(define (add-path id players base-scene)
  (define player
    (findf (lambda (x) (id=? id (player-id x))) players))
  (if (boolean? player)
      base-scene
      (add-waypoint* player base-scene)))
```

This function takes an `id`, a list of players, and an image to which the path is added. The function checks if this client's `id` exists in the list of players. If it does not exist, then the `id` came from a spectator and no waypoints are drawn. Otherwise, it draws all the points for this client. This function is straightforward, so we leave it as an exercise for you.



Finally, we deal with message handling for the `entree` state. The function for handling the messages in the `entree` state is similar to the message handler for the `app` state:

client.rkt

```
(define (handle-entree-message s msg)
  (cond [(state? msg) (update-entree s msg)]
        [(score? msg) (restart s msg)]
        [else s]))
```

In the case of a state message, the handler calls `update-entree`, which works like `switch-to-entree`. In the case of a score message, the game restarts:

client.rkt

```
(define (restart s end-msg)
  (define score-image (render-scores end-msg))
  (app (entree-id s) (above LOADING score-image) ZERO%))
```

This function builds an image that contains a table of all the scores and uses it as the base image for a new `app` structure. Building the table image is really easy as well:

client.rkt

```
(define (render-scores msg)
  (define scores (sort (second msg) < #:key second))
  (for/fold ([img empty-image]) ([name-score scores])
    (define txt (get-text name-score))
    (above (render-text txt) img)))
```

Here, we take the list of scores and sort it in ascending order based on the second value of each list. If you want more detail about `#:key`, look up "keyword arguments" in the documentation. The function then iterates across this sorted list with `for/fold` and builds an image with the name and score, sticking it above the previous rows. And there you go. That's the entire client.

14.7 Main, Take Server

Whenever we get to modules, we start with a main function, which is a `universe` function in the case of a game server:

server.rkt

```
(define (bon-appetit)
  (universe JOIN0
            (on-new connect)
            (on-tick tick-tock TICK)
            (on-msg handle-goto-message)
            (on-disconnect disconnect)))
```

The initial state of the universe is a `join` state with no clients and some initial time.

The `on-new` clause deals with new connections:

server.rkt

```
(define (connect s iw)
  (cond [(join? s) (add-player s iw)]
        [(play? s) (add-spectator s iw)]))
```

Like all `on-new` handlers, this function takes the current universe and an `iworld`, a piece of data representing the new connection. Once again, this event handler dispatches to auxiliary functions depending on the current state `s` of the universe. All of the `universe` handlers employ a similar strategy.

The function `disconnect` does exactly what it sounds like. It handles dispatch for disconnections:

server.rkt

```
(define (disconnect s iw)
  (cond [(join? s) (drop-client s iw)]
        [(play? s) (drop-player s iw)]))
```

Just as before, this function determines the current state of the server and passes along its argument to the appropriate function.

Ticking works like this:

server.rkt

```
(define (tick-tock s)
  (cond [(join? s) (wait-or-play s)]
        [(play? s) (move-and-eat s)]))
```

In the `join` state, the function counts down and possibly transitions to the `play` state. In the `play` state, it moves all the players and lets them eat cupcakes. The `move-and-eat` function may also transition back to the `join` initial state if the last cupcake has been eaten.

The final universe clause we need to explain is `on-msg`. The `on-msg` handler clause is a little different from all of the others. Recall that the server accepts messages only when it is in the `play` state, and even then, there is only one kind of message that it accepts:

server.rkt

```
(define (handle-goto-message s iw msg)
  (cond [(and (play? s) (goto? msg)) (goto s iw msg)]
        [else                        (empty-bundle s)]))
```

If the server is in the `play` state, and the message is a valid `GOTO` message, we add the new waypoint to the given player's path. If not, then we do nothing, which in the case of a server, means we return a bundle with no messages, no dropped clients, and an unchanged state. It's a really short function:

server.rkt

```
(define (empty-bundle s)
  (make-bundle s empty empty))
```

The rest of the section explains how the server reacts to the network events and clock tick events, depending on which state it is in. Altogether, this makes four combinations. We start with a close look at how to handle network events while the server is in the `join` state.

The Join State and Network Events

One network event signals the arrival of a new client. Think about this event and how we might handle it in the `join` state. Now think about how we might handle it in the `play` state. In both states, we must create a new internal player, add it to the state, and send the client its new `id`. The difference is that in the `join` state, we add the new internal player to the list of players, and in the `play` state, we add it to the list of spectators.

So let's take advantage of these similarities with a little abstraction:

server.rkt

```
(define (make-connection adder)
  (lambda (u iw)
    (define player (named-player iw))
    (define mails  (list (make-mail iw (ip-id player))))
    (make-bundle (adder u player) mails empty)))
```

This function consumes `adder`, a function that takes a state and a player and returns a state. Using `adder`, `make-connection` creates another function. This newly created function consumes a universe state and an `iworld`. It builds a new player, and it constructs mail for the new client that contains the guaranteed unique ID created with the new player's name. The function returns this bundle.

This abstract function allows us to define the function to handle new clients for the `join` state in one line:

server.rkt

```
(define (join-add-player j new-p)
  (join (cons new-p (join-clients j)) (join-time j)))

(define add-player (make-connection join-add-player))
```

The other part of the `make-connection` function we need is `named-player`:

server.rkt

```
(define (named-player iw)
  (create-player iw (symbol->string (gensym (iworld-name iw)))))
```

It uses a fancy function, called `gensym`, to create a unique name that starts with the name of the client. Then it creates a player using `create-player`:

server.rkt

```
(define (create-player iw n)
  (ip iw n (create-a-body PLAYER-SIZE) empty))
```

This creates a player with all the specified information, a body with some initial size, and an empty list of waypoints. The body of the player is placed at some random point on the playing field.

```
#|
NOTE: The gensym function comes with all members of the Lisp family.
It's used to generate unique symbols that are not eq? to any other
symbol in the entire program. You will learn more about why this
facility is useful for writing programming languages in the next
chapter. Here, we just use it for making unique names.
|#
```

Other than the arrival of new players, the server must also deal with player disconnections:

server.rkt

```
(define (drop-client j iw)
  (empty-bundle (join-remove j iw)))
```

When a client is dropped, `drop-client` must find the player with the same `iworld` and remove it from the list of players. It does this using `join-remove`:

server.rkt

```
(define (join-remove j iw)
  (join (rip iw (join-clients j)) (join-time j)))
```

The `rip` function finds the player with the given `iworld` and removes it from the list of clients. To do so, it uses the `remove` function, which we know from chapter 6, but with a twist:

server.rkt

```
(define (rip iw players)
  (remove iw players (lambda (iw p) (iworld=? iw (ip-iw p)))))
```

As this definition shows, `remove` is actually a higher-order function that takes an optional third argument for equality testing. By default, this function is `equal?`, but here we define "equality" to mean "having the same `iworld`" instead of "being exactly the same." Why do you think we use `iworld=?` instead of the `equal?` function?

The Join State and Tick Events

Dealing with tick events requires a complex handler. It all starts with `wait-or-play`:

server.rkt

```
(define (wait-or-play j)
  (cond [(keep-waiting? j) (keep-waiting j)]
        [else              (start-game j)]))
```

As the name says, this function either continues waiting or transitions into the `play` state.

Checking whether or not to transition works as follows:

server.rkt

```
(define (keep-waiting? j)
  (or (> PLAYER-LIMIT (length (join-clients j)))
      (> WAIT-TIME (join-time j))))
```

The server waits if there are not enough players for a good game or if the allotted wait time is not up. It's equally easy to define `keep-waiting`:

server.rkt

```
(define (keep-waiting j)
  (set-join-time! j (+ (join-time j) 1))
  (time-broadcast j))
```

All that's necessary is to increment the time and send that new time to all clients, which the following function accomplishes:

server.rkt

```
(define (time-broadcast j)
  (define iworlds (map ip-iw (join-clients j)))
  (define load% (min 1 (/ (join-time j) WAIT-TIME)))
  (make-bundle j (broadcast iworlds load%) empty))
```

The key is the call to `broadcast`. This helper will be used in a number of places to send a message to all clients. It takes a list of `iworld`s and the message to be sent, which in this case is a percentage of the current time. It returns a list of `mail`s, one for each client.

Let's take a look at the `broadcast` function:

server.rkt

```
(define (broadcast iws msg)
  (map (lambda (iw) (make-mail iw msg)) iws))
```

With `map`, this function definition becomes downright trivial. We show it only because it is used in almost every function that sends a message from the server to the clients.

Now it is time to look at how the server starts a game:

server.rkt

```
(define (start-game j)
  (define clients (join-clients j))
  (define cupcakes (bake-cupcakes (length clients)))
  (broadcast-universe (play clients cupcakes empty)))
```

The `start-game` function takes a `join` state and creates a `play` state. This `play` universe starts with the list of players from the `join` universe, a list of food, and an empty list of spectators. Using `broadcast-universe`, we send this initial state to all of the clients. We generate a number of cupcakes that is directly proportional to the number of players:

server.rkt

```
(define (bake-cupcakes player#)
  (for/list ([i (in-range (* player# FOOD*PLAYERS))])
    (create-a-body CUPCAKE)))
```

We chose to generate cupcakes in this fashion, but you can define this function in whatever way you want. Take note that we use the `create-a-body` function.

Onwards to the `broadcast-universe` function:

server.rkt

```
(define (broadcast-universe p)
  (define mails (broadcast (get-iws p) (serialize-universe p)))
  (make-bundle p mails empty))
```

Here, we reuse the `broadcast` function to create a state message for each of the clients. As usual, this function returns a bundle with the universe and the list of mail that needs

to be sent. While `get-iws` is another easy little function that you should be able to write yourself, `serialize-universe` is a little trickier:

server.rkt

```
(define (serialize-universe p)
  (define serialized-players (map ip-player (play-players p)))
  (list SERIALIZE serialized-players (play-food p)))
```

Remember that `ip`s contain complete `player`s and that these are represented with prefab structures. As planned, we can send those across the network, and that's what we do. With that, we are finished with the `join` state for the server.

The Play State and Network Events

Now our server can reach the `play` state. It is time to think about how the `play` state should handle network events. As with the `join` state, we start with handling new connections, which are added as spectators. Recall that we have already dealt with new connections for `join` states and that we created an abstraction for dealing with them.

Having said that, here is `play-add-spectator`, which `cons`es the new client onto the list of spectators.

server.rkt

```
(define (play-add-spectator pu new-s)
  (define players (play-players pu))
  (define spectators (play-spectators pu))
  (play players (play-food pu) (cons new-s spectators)))

(define add-spectator (make-connection play-add-spectator))
```

The next network event concerns the arrival of `GOTO` messages, which are the only ones that the server deals with:

server.rkt

```
(define (goto p iw msg)
  (define c (make-rectangular (second msg) (third msg)))
  (set-play-players! p (add-waypoint (play-players p) c iw))
  (broadcast-universe p))
```

First, `goto` creates a complex number representing the new waypoint. Second, it modifies the current list of players with that waypoint added to the client who sent the message. Finally, it broadcasts the new state of the universe. This broadcast allows the client to see that its waypoint message was accepted and to draw an appropriate path. Indeed, all clients get to see the new waypoint, and that may concern you.

The only undefined auxiliary function in `goto` is `add-waypoint`:

server.rkt

```
(define (add-waypoint ps c iw)
  (for/list ([p ps])
    (cond [(iworld=? (ip-iw p) iw)
           (ip (ip-iw p)
               (ip-id p)
               (ip-body p)
               (append (ip-waypoints p) (list c)))]
          [else p])))
```

This function traverses the given list of players, and when it finds the player with an `iworld` matching the one that sent the new waypoint, it reconstructs the given player, adding the provided complex number to the end of its list of waypoints.

The last kind of network event to worry about is a client disconnection:

server.rkt

```
(define (drop-player p iw)
  (broadcast-universe (play-remove p iw)))
```

Disconnecting from the `play` universe is like disconnecting from the `join` universe with the difference that we broadcast the new state to all players so that they know some client has dropped out. The `play-remove` function has a straightforward definition:

server.rkt

```
(define (play-remove p iw)
  (define players (play-players p))
  (define spectators (play-spectators p))
  (play (rip iw players) (play-food p) (rip iw spectators)))
```

The only difference between `join-remove` and `play-remove` is that the latter uses `rip` on both `players` and `spectators` because we don't know whether a player or a spectator has dropped out.

The Play State and Tick Events

We have one server handler left to deal with: the clock-tick handler. It is a large one because this function deals with all the game logic. We start with `move-and-eat`:

server.rkt

```
(define (move-and-eat pu)
  (define nplayer (move-player* (play-players pu)))
  (define nfood (feed-em-all nplayer (play-food pu)))
  (progress nplayer nfood (play-spectators pu)))
```

For doing so much, this three-line function definition looks almost trivial, but bear with us—it gets a little complicated. First, `move-and-eat` uses `move-player*` to move the players in the appropriate direction at an appropriate speed. Second, a new list of food is generated that doesn't contain any cupcakes that a player has eaten. The function `feed-em-all` also mutates the bodies of players who are eating; remember that they grow in size. Finally, these two lists, along with the list of spectators, are sent to the `progress` function. It will either progress the game by sending the new `play` state to all clients or transition to a `join` state by sending out the final score message.

Here is how we move all the players:

server.rkt

```
(define (move-player* players)
  (for/list ([p players])
    (define waypoints (ip-waypoints p))
    (cond [(empty? waypoints) p]
          [else (define body (ip-body p))
                (define nwpts
                  (move-toward-waypoint body waypoints))
                (ip (ip-iw p) (ip-id p) body nwpts)])))
```

For every player with waypoints, we mutate the player's body with `move-toward-waypoint`. If the player reaches her first point, `move-toward-waypoint` chops off the first waypoint and returns the new list. If the next waypoint has not been reached, the list remains the same. A player without waypoints is left unchanged.

Moving individual players takes a bit of tricky math, but using complex numbers makes our lives pretty easy:

server.rkt

```
(define (move-toward-waypoint body waypoints)
  (define goal (first waypoints))
  (define bloc (body-loc body))
  (define line (- goal bloc))
  (define dist (magnitude line)) ;;in pixels per clock tick
  (define speed (/ BASE-SPEED (body-size body)))
  (cond [(<= dist speed)
         (set-body-loc! body goal)
         (rest waypoints)]
        [else ; (> dist speed 0)
         (define velocity (/ (* speed line) dist))
         (set-body-loc! body (+ bloc velocity))
         waypoints]))
```

The main part of the function is the conditional block. It checks the distance the player moves in this tick. If this condition is satisfied, the function sets the location to the goal to avoid overshooting it. Then, the function returns the rest of the waypoints. Otherwise, we add `velocity` to the current location, where `velocity` is the fraction of the complex number by which the body should move toward the next waypoint. Also notice that `speed` depends on the size of the body, meaning that a player's body slows down as it gets bigger.

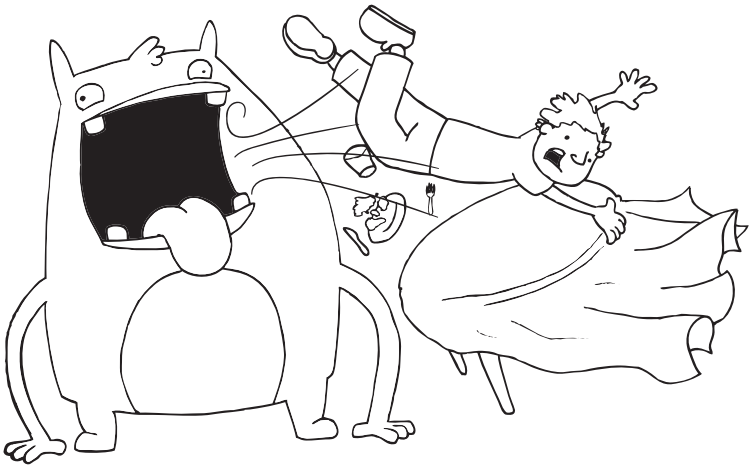
If this looks to you like a bunch of complicated vector math, hidden behind complex arithmetic, then you're right. We recommend that you browse the Web for more information on vector-based movement. You should admire, however, how this avoids all uses of `sin`, `cos`, and other trigonometry.

Now think about the `ip` structure for a moment. Don't we have an internal `player` that needs updating? No, we don't. We got away with this because `body` in the `ip` and `player` structure is the *same* body, so mutating one mutates the other. Because they are really `eq?`, no extra modifications are needed.

It is time to feed the players. We start with `feed-em-all`:

server.rkt

```
(define (feed-em-all players foods)
  (for/fold ([foods foods]) ([p players])
    (eat-all-the-things p foods)))
```



This function folds across the players, accumulating a new list of food by removing any food that a player collides with. The function `eat-all-the-things` is the workhorse here, filtering out the eaten cupcakes and face-stuffing players:

server.rkt

```
(define (eat-all-the-things player foods)
  (define b (ip-body player))
  (for/fold ([foods '()]) ([f foods])
    (cond
      [(body-collide? f b)
       (set-body-size! b (+ PLAYER-FATTEN-DELTA (body-size b)))
       foods]
      [else (cons f foods)])))
```

This function also uses `for/fold`, but this time, it folds across the list of cupcakes. If the current food collides with the given player, it is not accumulated; instead, the player is fattened. Otherwise, the food is put back in the list.

The collisions themselves are straightforward:

server.rkt

```
(define (body-collide? s1 s2)
  (<= (magnitude (- (body-loc s1) (body-loc s2)))
      (+ (body-size s1) (body-size s2))))
```

This function compares the distance between the centers of the two bodies with the sum of their sizes. If the distance is less than the combined radius, then it must be a collision. And that's it.

Finally, we must discuss the `progress` function:

server.rkt

```
(define (progress pls foods spectators)
  (define p (play pls foods spectators))
  (cond [(empty? foods) (end-game-broadcast p)]
        [else (broadcast-universe p)]))
```

If all cupcakes have been eaten, `progress` ends the game by transitioning to a `join` state and sending the score list to all clients. If not, it just uses `broadcast-universe` to send out the current state.

Ending the game is simple:

server.rkt

```
(define (end-game-broadcast p)
  (define iws (get-iws p))
  (define msg (list SCORE (score (play-players p))))
  (define mls (broadcast iws msg))
  (make-bundle (remake-join p) mls empty))
```

All that happens here is that we build a score message, broadcast it to all players, and create a new `join` state from the current players and spectators. To accomplish this, we need the `score` function, which builds an association list linking each player's `id` to her score:

server.rkt

```
(define (score ps)
  (for/list ([p ps])
    (list (ip-id p) (get-score (body-size (ip-body p))))))
```

In truth, this function builds the second half of the score message. In order for the client to recognize this message, it needs to be placed in a list with `SCORE` as the first element. But the only part of this function that might be a little tricky is getting a player's score from its weight. Recall that a player's weight is increased by some amount whenever it eats. Hence, its weight is directly proportional to the number of cupcakes eaten:

shared.rkt

```
(define (get-score f)
  (/ (- f PLAYER-SIZE) PLAYER-FATTEN-DELTA))
```

We leave the explanation of this math up to your grade school math teacher.

Last, but not least, we turn our eyes to the `remake-join` function. Given a `play` state, this function constructs a new `join` state:

server.rkt

```
(define (remake-join p)
  (define players (refresh (play-players p)))
  (define spectators (play-spectators p))
  (join (append players spectators) START-TIME))
```

To do this, we refresh the current list of players. This moves them to a new location with an unfattened body. This list is joined with the spectators list to make the list of clients for the new state. Add `START-TIME` to the new `join` state to get the countdown going, and you have a complete new `join` state.

14.8 See Henry Run

The game is complete, but we need to see it run so that we can correct problems that unit tests don't uncover. While it is possible to run the game with the existing code, doing so is cumbersome—politely put. If we write a bit of extra code, life becomes easy.

As in the preceding chapter, let's create a new file to run the game and require all necessary pieces into this fourth file:

run.rkt

```
#lang racket
(require (only-in "server.rkt" bon-appetit)
         (only-in "client.rkt" lets-eat)
         2htdp/universe)
```

The `require` specification tells Racket to include one function from **server.rkt** and another one from **client.rkt**, plus everything that **2htdp/universe** provides.

With these functions, we define `serve-dinner`, launching a server and two clients:

run.rkt

```
(define (serve-dinner)
  (launch-many-worlds (bon-appetit)
                      (lets-eat "Matthias" LOCALHOST)
                      (lets-eat "David"    LOCALHOST)))
```

And bam! We can play the game just by running one simple function. It doesn't even take any arguments. With this last file, the game is complete. Go play.

`On-disconnect`—Chapter Checkpoint

In this chapter, we created an interesting distributed multiplayer game:

- Distributed programming requires a planned-out communication protocol. Ours fits into a one-page diagram.
- Designing a good protocol can save you a lot of work. Spend time on it.

Chapter Challenges

● **Easy** Modify the game so that new cupcakes appear on the screen every other time one is eaten.

■ **Medium** Right now, a clever player could write a client that takes advantage of the other player's waypoints. Modify the data protocol so that this security breach cannot happen.

♦ **Difficult** Write an AI that follows the modified protocol.