Functional Video Games in CS1 III
Distributed Programming for Beginners

Marco T. Morazán

Seton Hall University, South Orange, NJ, USA
`morazanm@shu.edu`

Abstract. This article puts forth the thesis that developing distributed multiplayer video games using functional programming should be a new trend in the CS1 classroom. A design recipe for the development of distributed applications is presented which has successfully been used at Seton Hall University over the past few semesters. The primary goal is to expose and get students thinking about some of the problems programmers face when writing distributed applications. To the CS1 instructor, this article presents a model for developing their own distributed programming module.

1 Introduction

The explosion in development of internet applications (such as social media sites and associated games) and the arrival of multicore processors to the mass market make it clear that the use of distributed programming is a trend that is likely to become as common as the use of the light bulb. Therefore, it is desirable for a CS1 course to introduce students to distributed programming. The key in CS1 is to expose students without expecting them to become experts–expertise is developed in a more advanced course. To be successful, however, distributed programming must be made appealing to students.

This article puts forth the thesis that developing distributed multiplayer video games using functional programming should be a new trend in the CS1 classroom. The article describes the approach implemented at Seton Hall University (SHU) using the *Program by Design* methodology presented in *How to Design Programs* (HtDP). A design recipe for the development of distributed applications is presented. This new design recipe is used to illustrate how first-year students can be led to develop a multiplayer Space-Invaders-like game called Aliens Attack. The development of the game builds on letting students develop code that contains subtle distributed programming bugs, like process synchronization and communication overhead, which motivate refinements.

2 Background

2.1 Student's Design and Programming Experience

At SHU, the introductory Computer Science courses focus on problem solving using a computer [8, 9]. The languages of instruction are the successively richer subsets of Racket known as the student languages which are tightly-coupled with HtDP [4]. Before being introduced to distributed programming, students have studied topics such as: primitive data, primitive functions, programmer defined functions and variables, programmer defined data, processing finite compound data, processing arbitrarily large compound data and structural recursion, and abstraction with higher-order functions. These topics are covered following much of the structure of HtDP [4].

The curriculum, however, also varies in significant ways from HtDP by including a module for distributed programming. Distributed programming is introduced after structural recursion for two reasons: our experience suggests that students that have developed some programming experience tend to struggle a lot less with distributed programming and from a student's perspective much more interesting video games can be developed after knowing how to design programs that process data of arbitrary size. The curriculum also places a great deal of emphasis on iterative refinement with a video game going through versions that grow in complexity as the course advances culminating in a multiplayer distributed version.

2.2 The Universe Teachpack

The course uses the universe teachpack [5] for video game development which provides the functionality to develop distributed games. The clients/players/worlds in a universe exchange messages with a server. The universe teachpack provides two functions to create messages: make-package and make-bundle. The first is used by a client to create a pair that contains a (possibly new) world and a message to the server. The second is used by the universe server to create a structure that contains a (possibly new) server state, a list of mails to any of the worlds, and a list of worlds to be disconnected from the universe. The constructor for a mail, make-mail, requires the recipient world and the message. Any message transmitted must be an *S-expression*. This means that students must design and implement functions to marshal and unmarshal their data–a topic first-year students can understand and will encounter later in an operating systems course [10]. This set-up also forces students to program using a specific API which is a useful skill to have them develop.

The syntax required to create a player's world specifies handlers that update the game or render the game to the screen. Version 1 of distributed Aliens Attack requires:

```
(big-bang
   INIT-WORLD                   ;; initial world
   (on-draw draw-world)         ;; handler for drawing the world
   (on-key process-key)         ;; handler for key events
   (on-tick update-world)       ;; handler for clock ticks
   (stop-when game-over?)       ;; handler to test for game end
   (register LOCALHOST)         ;; registers with the server
   (on-receive process-message) ;; handler for incoming messages
```

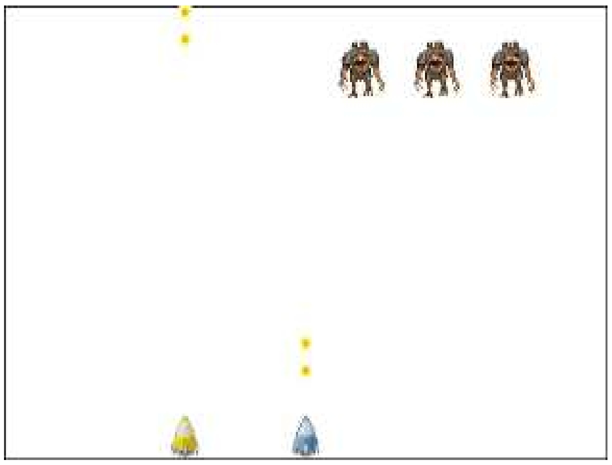

Fig. 1: A Snapshot Illustrating Multiplayer Aliens Attack.

```
   (name MY-ID))                ;; name of this world
```

During development students use LOCALHOST as the address of the server, but at play time they may also use an internet address to specify where the server is running.

The syntax for the universe server is similar and specifies the event handlers. For version 1 of distributed Aliens Attack the following syntax is required:

```
(universe
   initU                       ;; the initial universe
   (on-new add-new-world)      ;; handler for new worlds joining
   (on-msg receive-message)    ;; handler for incoming messages
   (on-disconnect rm-world))   ;; handler for worlds disconnecting
```

This syntactical set-up provides a framework to get students started. Specifically, students identify the handlers which are needed and must write each handler along with any auxiliary functions that may be needed. Readers interested in further details about the universe teachpack are referred to the help pages in DrRacket [7] and the modest guide on how to design worlds [3].

3 A Design Recipe for Distributed Computing

After developing a single player Aliens Attack [8], students ask if it is possible to have multiple players. Figure 1 displays a snapshot of a multiplayer version of Aliens Attack that they have in mind. Students are generally excited about the possibility of playing together which sets the stage to discuss distributed programming.

Students are led through an informal discussion of what is needed to write a multiplayer Aliens Attack. The idea of the game being distributed naturally comes to our students given their experience with internet games. They realize the need to send and to receive messages as well as the need for a server that provides support to coordinate all the players/clients. The following design recipe for distributed programming is presented:

1. Divide the problem into components.
2. Draft data definitions for the different components and the server.
3. Design a communication protocol.
4. Design marshalling and unmarshalling functions and create data definitions for messages.
5. Design and implement the components (starting with handlers)
6. Design and implement a server (starting with the handlers).
7. Test your program.

One of the main goals of the above design recipe is to gently introduce students to distributed programming. Students are explained that, as any other design recipe seen earlier in the course, each step has a specific outcome. In Step 1, they must define what each component/client as well as the server does. In Step 2, they must draft data definitions for the data that each component/server is to manipulate. In Step 3, they must define a communication protocol specifying when a client sends a message to the server and when the server sends mail to a client. An efficient way to achieve this is by using protocol diagrams that illustrate when communication occurs. In step 4, students must develop marshalling and unmarshalling functions. This step provides an excellent opportunity to help students make a connection with a topic they have studied in their Mathematics courses given that a marshalling and the corresponding unmarshalling function are inverses of each other[1]. Another important result of this step is data definitions for different kinds of messages. In step 5, students must design and implement the handlers as well as any necessary auxiliary functions for each client. In step 6, students design and implement the server. In step 7, students must test their programs and redesign/reimplement if necessary. In this step, students must consider the subtle problems that arise in distributed programming such as process synchronization, communication overhead, and speed.

4 Multiplayer Aliens Attack Version 1

Students are asked to think about how to make a multiplayer game from their single player Aliens Attack [8]. By an overwhelming margin, the most common answer is to add to each single player the other players. That is, each player runs their game and others can join. The details of how to do this are, of course, fuzzy at best and they are invited to use the new design recipe.

[1] It has been my experience that making such connections throughout programming courses in a CS undergraduate curriculum is very important, because most student programmers dislike Mathematics and consider it irrelevant to programming.

```
;; A rocket is a non-negative number.
;; An ally rocket, (make-ar x n), is a structure where x is a number
;; and n is a string for the name of the player that controls it.
(define-struct ar (x name))
;; A list of ally rockets (loar) is a (listof ar).
;; An alien is a posn.
;; A list of aliens (loa) is a (listof alien).
;; An alien army (aa) is  either 'uninitialized or a loa.
;; A world is a structure, (make-world r l a d s), where r is a rocket,
;;   l is an loar, a is an aa, d is a string, and s is a los.
(define-struct world (rocket allies aliens dir shots))
```

Fig. 2: Player Data Definitions for Multiplayer Aliens Attack Version 1.

4.1 Problem Components

Students identify each player as a component that is responsible for rendering the state of the game, moving a single rocket, moving the aliens, changing the direction the aliens are moving in, and moving the shots. In addition, each component must provide support for a list of allies and must receive messages to reproduce the actions taken by other players. This component decomposition is very attractive to students, because it means that they can re-use the code they have written for a single player Aliens Attack by making a small number of changes and additions (e.g., the development of a message processing handler). This turns out to be important to keep frustration low with what some students view as a Herculean task at the beginning.

Student-guided class discussion leads to the server being responsible for receiving messages from the players indicating their rocket moving and shooting actions and for broadcasting said messages to all the other players. That is, a thin-server appears to be the intuitive choice for (most) students. In addition, the server sends the initial army of invading aliens to the first player that joins the game.

4.2 New Data Definitions

For a player, the new data definitions are displayed in Figure 2. For the server, the only new data definition is for a universe (of players/worlds/clients):

```
;; A universe is a (listof iw), where iw is an iworld.
```

An iworld is the internal representation used by the `universe` teachpack for the clients that join the server. All these data definitions are in familiar territory for the students and require the development of examples and function templates.

4.3 Communication Protocol Design

A communication protocol is described to beginning students as a collection of communication chains. A communication chain is defined as a series of messages

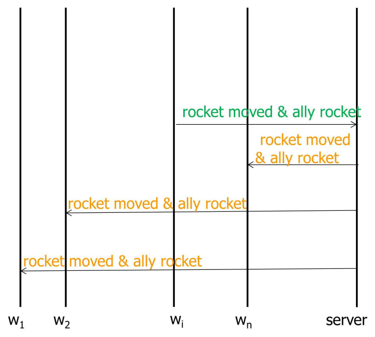
rocket moved & ally rocket
rocket moved & ally rocket
rocket moved & ally rocket
rocket moved & ally rocket
w1
w2
wi
wn
server

Fig. 3: Communication Protocol for a Rocket Move.

that are exchanged between the server and the clients. These chains are sparked by either an action taken by a client or an action taken by the server. A communication chain is visualized using a *protocol diagram*–a diagram illustrating the messages in a chain. This abstraction is understood by students and allows for a well-focused discussion during classroom development.

In Aliens Attack, a player sparks a communication chain when a key event occurs. That is, when a rocket move or shot is made by, p_i, the i^{th} player. For example, when p_i moves the rocket, a rocket-moved message is sent to the server that includes the new ally rocket[2]. The server forwards the message to all the other players. Figure 3 displays the protocol diagram for a rocket move. A similar protocol diagram is developed for shot creation.

The server sparks a communication chain when its state changes. Classroom analysis reveals that this occurs two times: when a new player joins the game and when a player disconnects from the game. Two cases are distinguished when a player, p_i, joins the universe. In the first case, the new player is the first in the universe and the server only needs to send the initial alien army. This communication chain is captured in the protocol diagram in Figure 4. In the second case, p_i, joins a non-empty universe. In this case, the server requests the state of the game from an existing player, p_j such that $i \neq j$, with a mail that includes i. The server also sends a new-ally message to all the worlds already in the universe. After the server receives a message from p_j that includes the state of the game and the destination for said state (i.e., i), the server forwards the game state to p_i. This communication chain is captured in the protocol diagram in Figure 5. A similar analysis leads to the communication chain required when a player leaves the game.

[2] To all other players a move made by p_i is a move made by a ally rocket.

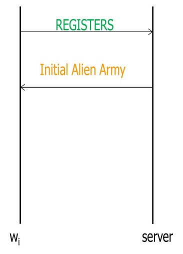

Fig. 4: World Joins Empty Universe

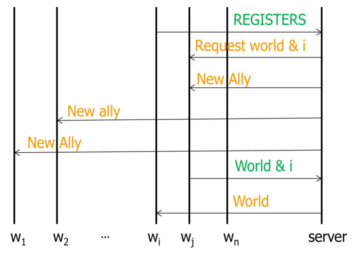

Fig. 5: World Joins Non-Empty Universe

4.4 Design Marshalling and Unmarshalling Functions and Data Definitions for Messages

Based on their data definitions and their communication protocol, students are now ready to design and implement marshalling and unmarshalling functions as well as to develop data definitions for messages. Concepts learned in high school algebra are reinforced by noting that marshalling and unmarshalling functions are inverses of each other. Marshalling is done by converting data into an S-expression and appropriately tagging the message. Unmarshalling is done by removing the tag and reconstructing the original data.

There are two types of messages: To-Server messages and To-Client messages. To-Server messages are identified by incoming arrows to the server in the protocol diagrams. Likewise, To-Client messages are identified by incoming arrows to the clients. Each set of clients that can receive different kinds of messages must have their own To-Client message data definition. In Aliens Attack this task is simplified since all clients are the same. Thus, only one To-Client data definition is required which is ideal for pedagogy in CS1.

Consider the communication chain in Figure 3. The protocol requires that a rocket-move message be sent to the server that includes the new ally rocket created by the move. This means that an ally rocket must be marshalled and unmarshalled. Since an ally rocket is a structure with a number, n, and a string, s, a To-Server rocket-move message is defined as a list containing the symbol rocket-move, n, and s. The corresponding marshalling and unmarshalling functions are:

```
; ally-rocket --> message
(define (marsh-rckt-mv an-ar)
  (list 'rocket-move (ar-x an-ar) (ar-name an-ar)))
```

```
A To-Server Message is either:
  1. (list 'rocket-move rocket string)
  2. (list 'new-shot number number)
  3. (list 'world
           string
           (list-of (list-of number string))
           (list-of (list-of number number))
           string
           (list-of (list-of number number)))
```

Fig. 6: To-Server Message Data Definition.

```
A To-Client Message is either:
  1. To-Server Message
  2. (cons 'init-army (listof (listof number number)))
  3. (list 'rm-ally string)
  4. (list 'req-world string)
  5. (list 'new-ally number string)
```

Fig. 7: To-Client Message Data Definition.

```
; message --> rocket
(define (unmarsh-rckt-mv m)
  (make-ar (first (rest m)) (first (rest (rest m)))))
```

Repeating this process for every incoming arrow to the server labeled differently in the protocol diagrams leads students to a complete data definition for a To-Server message as displayed in Figure 6, to the development of marshalling and unmarshalling functions, and to a function template for functions that process To-Server messages.

To develop the data definition for a message to clients, observe in the protocol diagrams that any To-Server messages is echoed to the clients. Therefore, a To-Client message can be a To-Server message. The protocol diagrams also inform us that that the server can send a player a message to request the world, to send the initial alien army, and to inform a player of a new ally or of an ally lost. The To-Client data definition is displayed in Figure 7 from which a corresponding function template is developed.

4.5 Component Implementation

Each different component is independently implemented. For Aliens Attack all clients are the same except for their identifying name (a string). This simplifies the task for students given that only one component needs to be developed. Furthermore, students can see that their task now is to refine their single player code into multiplayer code. This requires updating functions that process data whose definition has been refined, adding communication code to functions that

```
; process-key: world key --> package
; Purpose: Handler to process key events.
(define (process-key a-world key)
  (cond
    [(key=? "up" key) (process-up-key a-world)]
    [else (local [(define new-world
                    (make-world
                       (move-rocket (world-rocket a-world) key)
                       (world-allies a-world)
                       (world-aliens a-world)
                       (world-dir a-world)
                       (world-shots a-world)))]
            (make-package new-world
                         (marsh-rckt-mv
                           (make-ar (world-rocket new-world)
                                    MY-ID))))]))
```

Fig. 8: Refined Key-Processing Handler for Multiplayer Aliens Attack.

make changes to the state of the game, and the creation of a message processing function. For the students, the updates are not hard nor intellectually obscure. Previously in the course, students have had to refine their code when a refinement has been made to a data definition. This step is not surprising to them, but some do find it tedious and time-consuming.

The addition of communication code is, however, a new element for them. For any arrow in the protocol diagrams that goes from a player to the server, communication code must be added. For example, the protocol diagram in Figure 3 tells us that a rocket-move message must be sent to the server when the rocket is moved. This means that their original key event handler requires small updates: updating the function signature to return a package and updating the function body to create a package by marshalling the rocket move. The updated code is displayed in Figure 8. As the reader can observe, adding communication code to the client is not complex for students after a communication protocol has been designed. Performing the same work for all out-going arrows from a player to the server yields the refined functions for player-sparked communication chains.

The final step implements a handler to process To-Client messages. This function is written by specializing the function template for To-Client messages which contains a conditional statement to distinguish among the variety of messages.This handler takes as input a world (i.e., game state) and a message and it returns a (new) world. For example, when a rocket moved message arrives the list of allies is updated and a new world is produced. This snippet illustrates the idea:

```
; add-new-world: universe iworld --> bundle
(define (add-new-world u w)
  (make-bundle
   (cons w u)
   (cond [(not (empty? u))
          (cons (make-mail (first u) (marsh-req-world (iworld-name w)))
                (map (lambda (iw)
                       (make-mail iw  (marsh-new-ally (iworld-name w))))
                     u))]
         [else (list (make-mail w (marsh-loa INIT-ALIEN-ARMY)))])
   empty))
```

Fig. 9: The Server's New Player/World Handler.

```
        [(symbol=? 'rocket-move (first mess))
         (make-world (world-rocket w)
                     (update-allies (unmarsh-rckt-mv mess)
                                    (world-allies w))
                     (world-aliens w)
                     (world-dir w)
                     (world-shots w))]
```

4.6 Server Implementation

The server is implemented in a top-down manner starting with the handlers and consulting the protocol diagrams. For example, the handler used when a player joins the game is based on the protocol diagrams of Figures 4 and 5. This function takes as input a universe and a joining iworld and produces a bundle. To create the new universe, the joining world is added to the list of current worlds. The mails that must be generated depend on the state of the universe. According to Figure 4, if the universe is empty the server sends the joining world the initial alien army. According to Figure 5, if the universe is not empty the server requests the game state from an existing world and sends a new ally message to all the current players in the universe. There are no worlds that need to be disconnected from the universe. The resulting handler is displayed in Figure 9. The handler for a world disconnecting from the game is developed in the same fashion.

The server's message processing handler is developed using the template for functions on a To-Server message. For example, for the communication chain in Figure 5 the following snippet of code is written:

```
[(symbol=? (car msg) 'world)
 (make-bundle
   u
   (list (make-mail (get-world (first (rest msg)) u) msg))
   empty)]
```

This snippet keeps the universe unchanged, forwards the world message to the player indicated in the message, and removes no players from the universe.

4.7 Testing

Students are advised that testing is two-fold: the testing they are familiar with checking that functions produce the correct output (using Racket's `check-expect` library) and testing for bugs that only arise in distributed programming such as synchronization, communication overhead, and deadlock.

The distributed programming bugs are tested for by running the game. Students see on their screens a working game with allies, but unlike the experienced reader they do not realize there is a synchronization bug. The instructor ought to let the students discover the bug by joining the game and projecting the instructor's screen to the class. It does not take long for students to realize that not all players have the game in the same state. This approach makes process synchronization a real concern to students and with some class discussion they realize that messages take time to travel from the source to the destinations. While the messages travel, the source player continues changing the state of their game. Thus, different players have different states.

5 Multiplayer Aliens Attack Version 2

Students quickly realize that a possible solution is for the game state to reside in one location and this strongly motivates the next refinement. It is important to note that this refinement is not prescribed by the instructor based on knowledge that students do not have. Instead, this refinement has its genesis in the students based on their results from version 1 of the multiplayer game.

5.1 Problem Components

Students identify each player as a component that is responsible for rendering the state of the game to the screen and for processing key events. Players do not update the state of the game and, therefore, do not need a handler to update the world every time the clock ticks. When a key event occurs, a message is sent to the server requiring a new key event handler. The syntax required for a player is:

```
(big-bang INIT-WORLD
          (on-draw draw-world)
          (on-key process-key)
          (on-receive process-message)
          (register LOCALHOST)
          (name MY-ID)
          (stop-when game-over?))
```

As in version 1, the server needs handlers to add new players, to remove players, and to process messages. The server is now also responsible for maintaining the state of the game, thus, requiring a handler for clock ticks. When the state of the game changes, the players are sent the new state. In essence, the students are defining a thick-server that is solely responsible for all the necessary computing. The required syntax for the server is:

```
(universe initU
          (on-new add-new-world)
          (on-msg receive-message)
          (on-disconnect rm-this-world)
          (on-tick update-univ))
```

5.2 Draft Data Definitions

Students are led to see that in this refinement there is no need to distinguish between a rocket and the allies. For the server, all the players are allies each of which is still controlled by a single player. In addition, students include a boolean in the game state to indicate if the game has ended. The following is the refined data definition for the game state:

```
;; A world is a structure, (make-world l a d s o), where
;; l is a loar, a is a aa, d is a string, s is a los, and
;;    o is a boolean.
(define-struct world (allies aliens dir shots over))
```

Given the added work done by the server, the representation of the state of the server must also be refined to include both the state of the game and, as before, the players represented as iworlds. The refined data definition for the state of the server is:

```
;; A univ is a structure, (make-univ l w), where l is a
;;    (listof iworld) and w is a world
(define-struct univ (worlds state))
```

5.3 Communication Protocol Design

Students are asked when does a player send initiate a communication chain and are asked to develop protocol diagrams. Figure 10 displays the protocol diagram students develop for a rocket move. A player sends the server a rocket move message containing the direction of the move. The server processes the move and sends all players an updated world. A similar diagram is developed for new shots.

Students realize that the server starts a communication chain when a player joins the game, a player disconnects from the game, and when the game state is updated after a clock tick. The protocol diagrams are easy to visualize with the server always sending the game state to all the players.

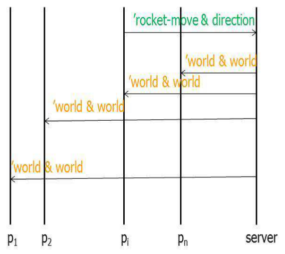

Fig. 10: Version 2 Protocol Diagram for a Rocket Move.

5.4 Design Marshalling and Unmarshalling Functions and Data Definitions for Messages

The protocol diagrams reveal to students that there is only one variety for a To-Client message and only two varieties for a To-Client message in this refinement:

```
A To-Client message is:
    (list 'world
          (listof (listof number string))
          (listof (listof number number))
          string
          (listof (listof number number))
          boolean)

A To-Server message is either:
    1. (list 'rocket-move string)
    2. (list 'new-shot number number)
```

This means only three pairs of marshalling-unmarshalling functions. For example, for a rocket move we have:

```
; string --> message
(define (marsh-rocket-move direction)
  (list 'rocket-move direction))

; message --> string
(define (unmarsh-rocket-move m) (first (rest m)))
```

The most complex pair is the one for a world which provides the opportunity to reinforce lessons using lambda expressions ad higher-order functions like `map`

```
; process-key: world key --> package or world
; Purpose: This function is the handler to process key events.
(define (process-key a-world key)
  (cond [(key=? "up" key)
         (make-package
          a-world
          (marsh-shot (make-posn (get-my-x (world-allies a-world))
                                 ROCKET-Y)))]
        [(or (key=? "left" key) (key=? "right" key))
         (make-package a-world (marsh-rocket-move key))]
        [else a-world]))
```

Fig. 11: Player key-event handler for version 2.

5.5 Component Implementation

Implementing the components means updating the handlers for processing key events and for rendering the game state using the refined data definition for world. In addition and according to the protocol diagrams, communication code must be added for key event handling and message handling. As before, one goal is to reuse as much code as possible.

Figure 11 displays the handler for key events developed by the students during class discussion using the protocol diagrams. If the "up" key is pressed, the state of the game is not changed by the player and a message with a new marshalled shot is sent to the server. Similarly, if the "left" or "right" key are pressed the state of the game is not changed and a marshalled rocket move is sent to the server. If any other key is pressed, the state of the game is unchanged and no message is sent to the server (which means a package is not constructed).

Given that there is only one type of To-Client message the message handler is very straightforward:

```
; process-message: world message --> world
(define (process-message w mess)
  (cond [(symbol=? 'world (first mess)) (unmarsh-world mess)]
        [else (error "World received an unknown message" mess)]))
```

Similarly the handler to check if the game has ended is also straightforward for students at this point in the course:

```
; world --> boolean
(define (game-over? w) (world-over w))
```

5.6 Server Implementation

The four handlers for the server are implemented during class in the same manner as version 1. The handler to add a new world dispatches on whether the state of the universe has an empty list of iworlds or not. The message handler dispatches

```
; univ --> univ
(define (update-univ u)
  (cond [(game-over? (univ-state u))
         (make-bundle
          u
          (map (lambda (iw)
                 (make-mail iw (marsh-world (mk-end-wrld (univ-state u)))))
               (univ-worlds u))
          empty)]
        [else
          (local [(define new-w (update-world (univ-state u)))]
            (make-bundle
               (make-univ (univ-worlds u) new-world)
            (map (lambda (w) (make-mail I (marsh-world new-w)))
                 (univ-worlds u))
            empty))]))
```

Fig. 12: Clock tick handler for version 2

on the two varieties of To-Server messages. The handler used when a player disconnects, creates a bundle with a new list of iworlds that does not contain the disconnected player and a new game state in which the disconnected player is not one of the allies.

The clock tick handler is the most complex. It dispatches on whether or not the game has come to a end. If the game is over, then a world in which the over flag is set is mailed to all the players. Otherwise, the state of the server is updated by updating the state of the game. This updated game state is mailed to all the worlds. A sample implementation developed by students is displayed in Figure 12.

5.7 Testing

Testing reveals that the synchronization problem appears resolved. We say *appears*, because we do not prove that it is resolved[3]. An instructor can, indeed, leave it at that and move on. Students have done enough to get them started thinking about synchronization. There is, of course, an issue with the list of shots that can be pointed out to the students. In the case of Aliens Attack, the order in which shots are added to the game state does not matter. In a different distributed application, however, order may very well matter and students are made aware of this.

Testing also reveals a most annoying bug for students: the game is much slower. The issues of bottleneck and communication overhead are brought forward during class discussion. This bug motivates the development of a third version of the game.

[3] Program correctness is not yet woven into CS1.

6 Multiplayer Aliens Attack Version 3

The development of version 2 marks the end of the distributed-programming module in CS1 at SHU. Students now have some experience with a complex communication protocol (version 1), with a simple communication protocol (version 2), and with some important bugs that arise in distributed programming. It is time for them to test their skills and their understanding on their own.

The next refinement of the game is assigned as a group project. Students are divided into groups of 2 or 4 students. Each group is further divided into two subgroups. One subgroup is responsible for developing the components (i.e., the players) and the other is responsible for developing the server. The subgroups must work together to agree on the data definitions, the communication protocol, and the marshalling functions. Then each subgroup develops their own code. When both subgroups are ready, they get together to test their program and, hopefully, enjoy the game and/or fix bugs.

The programs developed by students have been extremely encouraging. Students submit working games that employ a communication protocol that can be described as middle of the road between version 1 and version 2. That is, they keep the components of version 2, but do not transmit the whole state of the game every time a server makes an update. Instead, they only transmit the part of the state that is changed. Having CS1 students writing distributed applications on their own is nothing short of amazing.

7 Student Assessment

After each semester of CS1 at SHU, students are asked to fill out a short survey to evaluate the distributed-programming module. On a scale from 1 (low) to five (high), students are asked if distributed programming is intellectually stimulating. The average of the distribution to date is 3.35 and 76% of the students answering 3-5. The middle 50% of the students are in the range 3-4. Surprisingly (to the author), a follow-up question reveals that students felt that in terms of intellectual stimulus distributed programming was much like what they have been doing all semester. From the student's perspective, the module contained new interesting material, but the transition to distributed programming required mostly tasks they had done before. This can only be interpreted as a success for the described methodology. The introduction to distributed programming is gentle enough that students feel it is a natural progression that builds on what they have learned.

Students are also asked to rank how much more difficult distributed programming is to non-distributed programming on a scale from 1 (not more difficult at all) to 5 (a lot more difficult). The average of the distribution to date is 3.9 with the middle 50% in the range 3-5. A follow-up question revealed that the top reason distributed programming is harder is error messages that are not very informative. This type of problem occurred mostly when their were bugs in the marshalling and unmarshalling functions that led to "unknown message" errors or errors trying access parts of a message that did not exist. The difficulty lies in that a message that, for example, causes the server to crash is not fixed in the server's code. Instead, it must be fixed in the client code. If there are several copies of the client code, this means fixing the bug in several different files. It may be possible to lessen this problem by introducing students to modules. In this manner, the server and clients can import code (e.g., marshalling and unmarshalling functions) and fix such bugs in a single file. It is unknown at this time how difficult it may be to introduce CS1 students to modules. Another reason cited as to why distributed programming is harder by some students is that they felt that keeping track of a communication protocol was a lot of work. That is, they had to add communication code to "a lot" of functions and had to write message processing functions.

Finally, students were asked about their level of excitement to develop a multiplayer video game on a scale from 1 (not at all excited) to 5 (extremely excited). The average of the distribution to date is 3.5 with the middle 50% in the range 3-4 and with 76% of the students in the range 3-5. The overwhelming majority of students in the top half of the range clearly indicates that the use of multiplayer video games can serve as great motivation for students to explore distributed programming.

8 Related Work

Teaching distributed programming in CS1 was virtually unheard of a few years ago. Now, there is a growing group of academics attempting it. The developers of DrRacket and HtDP have taught distributed programming in CS1 and have briefly described their approach using a step-locked game[4] to control a UFO [5]. In contrast, the work presented in this article aims to expose students to both distributed programming and to some of its pitfalls like synchronization and communication overhead. Exposing students to such pitfalls is difficult to do with step-lock games like the UFO game [5] and Chat Noir [6]. In addition, the work described in this article can be used by educators "in the trenches" focusing on the actual deployment of a distributed functional video game module in the classroom that is tightly coupled with other work developed by students during the semester.

The use of functional video games in CS1 is a little more extensive, but still just beginning to flourish. Soccer-Fun, developed using Clean, aims to motivate students by having them write programs to play soccer games [1]. There have been no reported efforts to make the platform distributed in order to allow players to compete against each other nor has this platform been used in CS1. Yampa is a language embedded in Haskell used to program reactive systems such as video games [2]. Yampa, in fact, has been used to implement a Space-Invaders-like game. The use of Yampa in the classroom appears to have been mostly discontinued, but work using functional video games in CS1 [8, 9] has

[4] A game in which players take discrete turns.

sparked an interest to reignite the use of Yampa in education by some of its developers.

9 Concluding Remarks

This article argues that distributed programming ought to and can be an integral part of CS1. The need for distributed programming in CS1 is based on the undeniable fact that the use of distributed computing is becoming as common as the use of the light bulb. The argument for success with distributed computing in CS1 is based on the illustrative development of a non-trivial functional multiplayer video game in SHU's CS1. Not a single function needed for the presented multiplayer Aliens Attack video game is beyond the ability of students that have studied structural recursion and the associated design recipes in HtDP. One of the major advantages of including distributed functional video game development in CS1 is that students become very excited about programming. There is no doubt that students feel empowered as programmers when they can develop a distributed application in a realm that is of interest to them. Another major advantage is that it gets students thinking about programming issues early in their undergraduate years, thus, providing an excellent foundation for more advanced courses.

10 Acknowledgements

The author thanks Matthias Felleisen for our discussions about distributing programming in CS1, for his feedback on the approach described in this article, and for the "code walk" done with students at Seton Hall University that implemented distributed Aliens Attack.

References

1. P. Achten. Teaching Functional Programming with Soccer-Fun. In *FDPE '08*, pages 61–72, New York, NY, USA, 2008. ACM.
2. A. Courtney, H. Nilsson, and J. Peterson. The Yampa Arcade. In *Haskell '03*, pages 7–18, New York, NY, USA, 2003. ACM.
3. M. Felleisen, R. Findler, K. Fisler, M. Flatt, and S. Krishnamurthi. How to Design Worlds. http://world.cs.brown.edu/1/, 2008.
4. M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing.* MIT Press, Cambridge, MA, USA, 2001.
5. M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. A Functional I/O System or, Fun for Freshman Kids. In *ICFP 2009*, pages 47–58, 2009.
6. R. Findler. CS 15100 Fall 2008 Project 3: ChatNoir. http://www.eecs.northwestern.edu/ robby/uc-courses/15100-2008-fall/proj3.pdf, 2008. Dept. of Electr. Engr. and Comp. Sci., Northwestern University.

7. R. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A Programming Environment for Scheme. *J. of Functional Programming*, 12(2):159–182, 2002.
8. M. Morazán. Functional Video Games in the CS1 Classroom. In R. Page, Z. Horvath, and V. Zsók, editors, *TFP 2011*, volume 6456 of *LNCS*, pages 196–213. Springer, 2011.
9. M. Morazán. Functional Video Games in the CS1 Classroom II. In R. Peña and R. Page, editors, *TFP 2012*, volume 7193 of *LNCS*, pages 146–162. Springer, 2012.
10. A. Silberschatz and P. Galvin. *Operating System Concepts.* Addison-Wesley, Reading, Massachusetts, USA, 1994.