JFP **28**, e5, 41 pages, 2018. © Cambridge University Press 2018
doi:10.1017/S0956796818000059

Infusing an HtDP-based CS1 with distributed programming using functional video games

MARCO T. MORAZÁN
Department of Computer Science, Seton Hall University, South Orange, NJ, USA
(*e-mail:* `morazanm@shu.edu`)

Abstract

A Computer Science introduction course ought to focus on exciting students about the subject matter and on problem solving through the methodical design of programs. An effective way to achieve both is through the development of functional video games. As most students are interested in video games, their development adds an exciting domain to any introduction to programming. This article advocates that an exciting crowning achievement for students in such a course is the design and implementation of a multiplayer distributed video game. By exploiting a domain that is popular with students, they are taught about design principles, communication protocols, and pitfalls in distributed programming. This article puts forth a successfully used design recipe that places distributed programming well within the reach of beginning students and outlines the use of this design recipe in the classroom. For those teaching beginners, this article presents a model for developing their own distributed programming module. The success of the presented methodology is measured through student feedback on their distributed programming experience. The empirical results suggest that the design and implementation of distributed functional video games is effective and well-received by students. Furthermore, the data suggests that the presented methodology fails to exhibit the gender gap common in Computer Science and is effective regardless of the programming experience CS1 students bring to the classroom.

1 Introduction

The majority of introductory courses to Computer Science (CS1) focus on programming to provide students with training in an essential skill (Joint Task Force on Computing Curricula & Society, 2013). This essential skill is pertinent to a broad audience of students as much as reading, writing, and arithmetic (Felleisen & Krishnamurthi, 2009). Therefore, it is proper for a CS1 course to focus on programming as it is fundamental to all students both in and outside of Computer Science (CS). A CS1 course, however, ought to also excite students about the subject matter. After all, it is nearly impossible to encourage life-long learning of technical material when students find the subject matter dull. To make CS1 palatable and even exciting for students, the instructor can capture student imagination by using popular real-world applications in the classroom (Sung, 2009). One such popular domain for most beginning students is video games. The development of functional video games has proven effective teaching program design principles and exciting students about problem solving (Courtney *et al.*, 2003; Achten, 2008; Felleisen *et al.*, 2009; Morazán, 2011; Bice *et al.*, 2013; Morazán, 2015).

The use of functional video games provides students with the opportunity to learn broad CS principles such as iterative refinement and divide-and-conquer, to learn how to design solutions to problems using structural, generative, and accumulative recursion, and to learn how to design (system) components. The use of functional video games in CS1 to teach broad CS principles and basic program design has been previously discussed (Felleisen *et al.*, 2009; Morazán, 2011; Bice *et al.*, 2013; Morazán, 2015). A natural extension of this work, that is well within the grasp of beginners, is to teach students about the design and implementation of components through the development of a distributed functional multiplayer video game–an exciting crowning achievement for all students in CS1. The key to avoid overwhelming beginners is to expose students to distributed programming without expecting them to become experts–expertise is developed in more advanced courses. Nonetheless, students learn about designing servers and clients, communication protocols, synchronization, marshalling data, and pitfalls in distributed programming. Given the explosion of internet applications (such as social media sites and associated games) and the arrival of multicore processors (including GPUs) to the mass market, it is clear that the use of distributed programming is a trend that is likely to become as common as the use of the light bulb and that beginners need to be exposed to it.

This article describes how distributed programming and component design are made exciting and accessible to beginning students. The work presented builds on previous work presented at Trends in Functional Programming 2013 (Morazán, 2014). In addition to game design and development, this work presents empirical data on student feedback on the distributed programming module. The approach described is implemented at Seton Hall University (SHU) using the *Program by Design* methodology presented in *How to Design Programs* (HtDP) (Felleisen *et al.*, 2001, 2015) and aims to reinforce program design skills and mathematical concepts students have learned in CS1 or elsewhere (e.g., in a Mathematics course). A novel design recipe for the development of distributed applications is presented. This new design recipe is used to illustrate how first-year students are led to develop a multiplayer Space-Invaders-like game called Aliens Attack. Multiplayer Aliens Attack is a natural refinement of the single-player version students develop throughout the course. The development of the multiplayer game builds on allowing a design and implemention that contains subtle distributed programming bugs, stemming from process synchronization and communication overhead, that students are led to discover. These bugs motivate refinements that keep students engaged and enthusiasm high–after all, nothing can be worse than a buggy video game. The article is organized as follows. Section 2 discusses related work. Section 3 presents the SHU student background and the use of the universe teachpack (used to write video games) (Felleisen *et al.*, 2008; Bice *et al.*, 2013). Section 4 presents the design recipe for distributed programming. Section 5 discusses the in-class development of the game using a thin-server that suffers from synchronization problems. Section 6 discusses an in-class refinement that solves the synchronization problem using a thick-server, but that suffers from communication overhead. Section 7 discusses a project in which students develop a thick-server-based refinement that reduces communication overhead. Section 8 presents the overall student assessment through the use of surveys with an analysis based on gender and the programming background students bring to CS1. Section 9 presents some concluding remarks.

2 Related work

Distributed programming has become a core component of the undergraduate curriculum (Joint Task Force on Computing Curricula & Society, 2013). Distributed programming education, however, is still in a state of flux. Some curricula offer distributed programming as an elective course, others pepper a range of undergraduate courses with distributed programming concepts, and yet others may combine both approaches. The approach chosen is made based on whether or not it is possible or responsible to relegate issues concerning distributed programming (e.g., coordination and communication) to an advanced elective (or required) course. Current curriculum initiatives strongly suggest that modern advances in architectures and programming languages require undergraduates, even in the early stages of their CS education, to have distributed programming knowledge and problem solving skills (Joint Task Force on Computing Curricula & Society, 2013; Prasad *et al.*, 2012). Regardless of the approach taken in the curriculum, exposure to distributed programming can start in CS1 to prepare students for latter courses in the curriculum. The work outlined in this article demonstrates an approach to incorporate distributed programming into CS1. This approach is consistent with the goal of introducing undergraduates to distributed programming and with the approach taken by many textbooks that touch upon distributed programming concepts in, for example, computer architecture (Hennessy & Patterson, 2011; Stallings, 2016), programming languages (Friedman & Wand, 2008; Scott, 2000; Tucker & Noonan, 2001), and operating systems (Silberschatz *et al.*, 2010; McHoes & Flynn, 2013; Tanenbaum & Bos, 2014).

Several institutions have chosen to introduce distributed computing modules into several of their courses. At Swarthmore College, for example, six courses are infused with such modules, including computer systems, operating systems, graphics, and compiler courses (Danner & Newhall, 2013). Swarthmore also developed an upper-level undergraduate course on parallel and distributed computing. The work described in this article follows in the spirit of such an approach. Students exposed to distributed programming in CS1 are better prepared to navigate other courses that provide more in-depth exposure. Adams *et al.* (2013) put forth the idea of teaching undergraduates about distributed programming patterns. The goal is to provide students with a framework that they can use to solve problems. The work presented in this article shares this goal. The presented design recipe for distributed programming is a framework that students use to develop distributed applications. This framework can be further specialized to derive common patterns.

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 turn-based game[1] to control a UFO (Felleisen *et al.*, 2009). 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 turn-based games like the UFO game (Felleisen *et al.*, 2009) and Chat Noir (Findler, 2008). 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.

A modest introduction to distributed programming for novices, with some previous exposure to programming, is found in *Realm of Racket* (ROAR) (Bice *et al.*, 2013). This book is intended as a general introduction to programming using video games. It uses Racket (not the student languages) as the programming medium. ROAR presents the development of a distributed video game, *Hungry Henry*, in which players run around the screen eating cupcakes. Like the work presented in this article, ROAR advocates that distributed programming is a natural part of an introduction to programming. In contrast to the work presented in this article, ROAR exposes readers only to a thick-server model and does not discuss the pitfalls of distributed programming. The development outlined in ROAR is in the spirit of the design recipe presented in this article, but ROAR does not explicitly put forth a design recipe for distributed programming nor does it make explicit how to develop a distributed program through a series of verifiable steps.

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 (Achten, 2008). 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 (Courtney *et al.*, 2003). The use of Yampa in the classroom appears to have been mostly discontinued, but work using functional video games in CS1 (Morazán, 2011, 2012) has sparked an interest to reignite the use of Yampa in education. In previous work, the author presents how to use video games to teach programming using primitive data, structures, and structural recursion (Morazán, 2011) and using generative and accumulative recursion (Morazán, 2012, 2015). Functional video games have also been used by Bootstrap to teach programming to high school students with the additional goal of transferring skills to algebra (Schanzer & Fisler, 2015). Similarly to Bootstrap, the work presented in this article aims to reinforce programming and Mathematics lessons that students can then take to future programming and Mathematics courses.

Outside the functional programming world, there have also been efforts to use video games to teach abstract CS concepts (Sung, 2009). Alice, for example, is

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

used to expose students for the first time to object-oriented programming using 3D animation programming (Cooper *et al.*, 2000; Dann *et al.*, 2011). Games have also been used to motivate the need for and the teaching of design patterns (e.g., state and visitor patterns) using Java (Gestwicki, 2007). In contrast to the work presented in this article, these approaches have not been extended to include distributed programming. The same is true for efforts to use video game implementation in a more advanced course (CS 3) using C++ (Pulimood & Wolz, 2008). In addition, this study reports that students were encouraged to work in groups and that game development gave rise to student proposals that were overly ambitious and poorly conceived. In contrast, the approach presented here is carefully designed to motivate and reinforce lessons from earlier in the semester and from Mathematics. A great deal of emphasis is placed on presenting the development in a context in which students feel comfortable and well-focused. Like this study, the efforts reported in this article have students work in groups to develop the different components of the game.

3 Background

3.1 Student's design and programming experience

At SHU, the introductory CS courses focus on problem solving using a computer (Morazán, 2011, 2012). The languages of instruction are the successively richer subsets of Racket known as the student languages that are tightly coupled with HtDP (Felleisen *et al.*, 2001, 2015). No prior experience with programming is assumed. Before introducing students to distributed programming, the course starts by familiarizing students with primitive data (e.g., numbers, strings, booleans, symbols, and images), primitive functions, and library functions to manipulate images (i.e., the image teachpack). During this introduction, students are taught about variables, defining their own functions, and the importance of writing contracts and purpose statements. The next step of the course introduces students to data analysis and programming with compound data of finite size (i.e., structures). At this point, students are introduced to the first design recipe. Students develop experience in developing data definitions, examples for data definitions, function templates, and tests for all the functions they write. A great deal of emphasis is placed on all of these steps as part of the problem-solving design process. Building on this experience, students develop expertise on processing compound data of arbitrary size such as lists, natural numbers, and trees. In this part of the course, students learn to design functions using structural recursion. It is noteworthy to observe that it is useful for students to briefly be shown how this design process applies to object-oriented programming.[2] After structural recursion, students are introduced to functional abstraction and the use of higher order functions such as map and filter.

These topics are covered following much of the structure of HtDP (Felleisen *et al.*, 2001). For 15 weeks, there are two 75-minute lectures that students are required to

[2] This is very useful to keep students engaged, especially those arriving in the classroom with programming experience in Java, Python, or C++.

attend. The typical classroom has students between 20 and 25 (although sometimes fewer than 20). In addition to the lectures, the instructor is available to students during office hours (3 h/week) and there are 20–30 h of tutoring each week that the students may voluntarily attend. The tutoring hours are conducted by undergraduate students handpicked and trained by the lecturer. These tutors focus on making sure students develop answers for each step of the design recipe (from writing contracts to running tests). Students must attempt to follow the steps of the design recipe prior to attending tutoring. Based on a student's work, the tutors provide guidance but do not solve problems. Students are still responsible for successfully completing all steps of the design recipe. In addition, tutors attend lectures to assist students when they get stuck with, for example, syntax errors. This type of team-teaching with undergraduate tutors has proven to be extremely well-received by students and to be an effective means to enhance the learning experience.

The curriculum above also varies significantly from HtDP in several ways which includes the distributed programming module described in this article. Although distributed programming can likely be introduced after students learn to program with structures, it is introduced at the end of the semester after structural recursion and higher order functions. This is done for two primary reasons. The first is that our experience suggests that most students that have developed some programming expertise do not find distributed programming intimidating. The second is that 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 varies from HtDP by placing a great deal of emphasis on iterative refinement. A video game (e.g., Aliens Attack) goes through several versions that grow in complexity as the course advances. Each refinement motivates the need to learn new syntax (i.e., language constructs) or design strategies. These versions at first focus on developing a single player game (Morazán, 2011) culminating in the multiplayer distributed version described in this article.

3.2 The universe teachpack

The development of video games as part of the course exploits the universe teachpack (Felleisen *et al*., 2009), which provides the functionality to develop distributed games. The clients/players/worlds in a universe exchange messages with a server. All communication occurs through the 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) game state 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 clients, and a list of worlds to be disconnected from the universe. The constructor for a mail, make-mail, requires the recipient client 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. This entire set-up forces students to program using a specific application programming interface (API), which is a useful skill to have them develop.

The syntax required to create a player/client/world specifies the handlers that update the game or render the game to the screen. For example, version 1 of the distributed Aliens Attacks requires the following syntax[3]:

```
(big-bang
   INIT-WORLD                       ;; initial world
   (on-draw draw-world)             ;; drawing handler
   (on-key process-key)             ;; key events handler
   (on-tick update-world)           ;; clock ticks handler
   (stop-when game-over?)           ;; game's end handler
   (register LOCALHOST)             ;; server registration
   (on-receive process-message)  ;; message handler
   (name MY-ID))                    ;; client name
```

This syntax provides the opportunity to guide students through top-down development following the steps of the design recipe for each handler. Students are told that when they develop the handlers to start by writing the functions specified in their big-bang statement and then write any auxiliary functions. This syntax is familiar to students that have developed single player games that brings distributed programming into a familiar domain. There is nothing here that is really foreign to students although there are new elements. Most notably, students realize that they need a function to process messages. Although new, it makes sense to students that messages are required for communication and that these messages must be processed.

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)       ;; world joins handler
   (on-msg receive-message)     ;; message handler
   (on-disconnect rm-world))  ;; client disconnects handler
```

This syntactical set-up provides a framework for students to design and develop the server, once again, in a top-down manner. Readers interested in further details about the universe teachpack are referred to the help pages in DrRacket (Findler *et al.*, 2002) and the modest guide on how to design worlds (Felleisen *et al.*, 2008).

4 A design recipe for distributed computing

After developing a single player Aliens Attack (Morazán, 2011), students are ready for the next refinement to include multiple players. The desire for this refinement is usually born from students and has served as motivation to develop the work described here. The students are generally excited about playing together which

[3] 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.

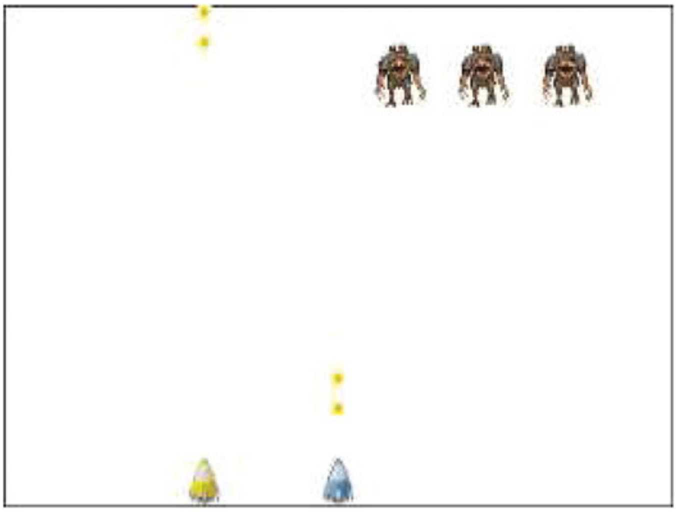

Fig. 1. A snapshot illustrating multiplayer Aliens Attack.

sets the stage to discuss distributed programming. Figure 1 displays a snapshot of a multiplayer version of Aliens Attack. It is easy for students to imagine several ally rockets working together to destroy an invading army of aliens. The question, however, is how this is done. It is noteworthy that, as the single player version, the multiple player version is customizable by students. Students are encouraged to be creative. This can be as simple as using their own images to adding features to the game such as shooting-aliens or a score. For example, some students have added a bomb-shooting capability for the rocket with which the player can destroy several aliens at a time instead of one by one with every shot.

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: clients and server.
2. Draft data definitions for the different components.
3. Design a communication protocol.
4. Create data definitions for messages and develop corresponding marshalling and unmarshalling functions for data that cannot be directly sent.
5. Design and implement the client components (starting with the handlers).
6. Design and implement the server component (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 and yet be broad enough to apply to many problems.[4]

[4] In fact, students have also used the presented design recipe to design and implement a two-player concentration game and a messaging tool.

Students are instructed that, as any other design recipe seen earlier in the course, each step has a specific outcome. This new design recipe, however, is more akin to the design recipe for generative recursion in HtDP (which provides less guidance on how to complete the steps) than to the design recipes for structural recursion. Like generative recursion, distributed programming requires the development of insight into a problem in order to identify components and to understand how to integrate components. The first four steps are intended to help students develop such insight that guides the actual development of code for the clients and the server components using the design recipes in HtDP.

In Step 1, students abstractly define what each component does and include their description as a comment in the software developed. In Step 2, data definitions for the components are drafted. In Step 3, a communication protocol, specifying when a client sends a message to the server and when the server sends mail to a client, is defined. This definition is described at a high-level of abstraction. An efficient way to achieve this is by using *protocol diagrams* that illustrate when communication occurs. In step 4, marshalling and unmarshalling functions are developed for structured data that cannot be directly transmitted using the underlying language. This step provides an excellent opportunity to help students make a connection with a topic they have studied in their Mathematics courses as marshalling and unmarshalling functions are inverses of each other.[5] These functions are developed in tandem with data definitions for the different kinds of messages: *To-Server* and *To-Client* messages. In step 5, the handlers, as well as any necessary auxiliary functions, for each client are designed and implemented. In step 6, the server is designed and implemented. An important component of steps 5 and 6 is the testing of the message processing handlers. These tests must illustrate that messages are having the correct effect on the state of the server/client. In step 7, programs are tested (and redesigned if necessary).

In the last step, students run unit tests written using syntax such as check-expect for all of their functions. This, however, is not enough given that students must also consider the subtle problems that arise in distributed programming such as: messaging errors, process synchronization, communication overhead, and deadlock.

These are difficult to test for using syntax such as check-expect. Messaging errors, for example, arise when messages do not have the proper structure according to a message data definition. The sender of an improperly formatted message causes the recipient to crash as the message received is of an unknown type. This can be avoided if a guarded message processing function were developed by students, but this adds complexity that distracts students from designing functions based on the proper input being received. Another, simpler, solution would be for the Universe teachpack to allow for the specified message-processing handler to be associated with an optional function to test if a message received is valid. Such a predicate would be written by students. In this manner, when a component receives

[5] It is very important to make such connections throughout the course to encourage students to embrace Mathematics as an integral part of programming.

an improperly formatted message a useful error message can be generated by the Universe teachpack. Such a facility is not yet available for students.

The problems of synchronization, communication overhead, and deadlock are difficult for experienced programmers to test, not to mention CS1 students. For example, there are graph-based algorithms to detect the possibility of a deadlock (Silberschatz *et al.*, 2010), but these are beyond the scope of a CS1 student. For these reasons, this type of necessary testing for distributed programming is done by observing behavior instead of writing tests or guarded functions.

5 Multiplayer Aliens Attack version 1

Students are asked to think about how to refine their single player Aliens Attack into a multiplayer game. By an overwhelming margin, the most common answer is to a have each player run their own game. The details of how to do this are, of course, fuzzy at best and they are invited to use the new design recipe. This section outlines the steps of the design recipe as followed in class with a student-driven discussion.

5.1 Divide the problem into components

Students identify each player as a component that is responsible for drawing the state of the game, moving a single rocket, moving the aliens, changing the direction the aliens are moving in, and moving the shots. That is, each player component is responsible for all the work done by the single player version. In addition, each component must provide support for a list of allies (i.e., the other players) 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 game. In essence, only a single player component needs to be refined which simplifies their design.[6] This turns out to be important to keep frustration low with what some students may view as a Herculean task at the beginning. Nonetheless, changing the data definition of the world structure (that represents the state of the game) is familiar to students as a refinement step. In addition to having a list of allies, the player component requires other refinements like the development of a message processing handler.

Student-guided class discussion also 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 is the intuitive choice for (most) students. Further analysis leads to the need for tracking the players in the game (in order to broadcast messages) and for sending the initial army of aliens when the first player joins the game. Figure 2 displays an abstract view of the component decomposition. Observe that in this version of the game each player is responsible for tracking the state of the game.

[6] There is one minor exception stemming from each player having a unique name.

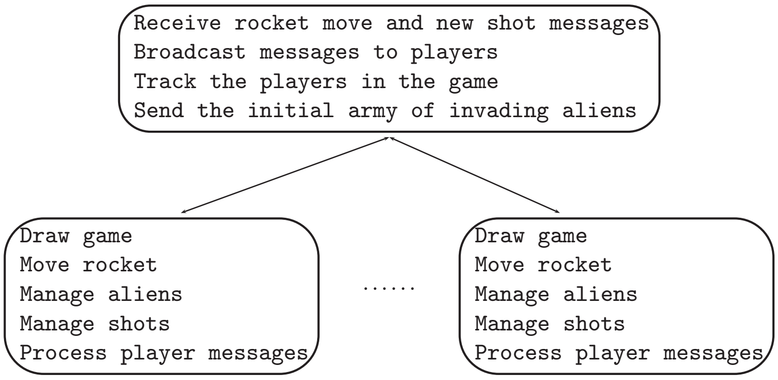

Fig. 2. Components for thin-server version of multiplayer Aliens Attack.

5.2 New data definitions

For a player, students start the refinement using the following data definitions:

```
;; A rocket is a number, r, such that 0 <= r <= SCREENWIDTH-1.
;; An alien is a posn.        ;; A loa is a (listof alien).
;; A shot is a posn.          ;; A los is a (listof shot)
;; A world is a structure, (make-world r a d s), where r is a
;; rocket, l is a loa, d is a string direction, and s is a los.
(define-struct world (r l d s))
```

Students quickly observe the need to represent the allied rockets. Upon discussion, it becomes clear that every allied rocket needs to be associated with the name of the world that controls it. This name is used to move the correct ally when another player moves her rocket. It also becomes clear to students that when a player joins the game she must get the initial army of invading aliens from the server. The initial army of aliens a player receives depends on the state of the game when they join. Thus, when a player runs the game her army of aliens is uninitialized until it receives a list of aliens from the server. Finally, a list of allied rockets is added to the world structure to represent the state of the game. The new data definitions developed are as follows:

```
;; An ally rocket, (make-ar x n),   ;; A list of ally rockets (loar) is
;; is a structure where x is        ;; a (listof ar)
;; a rocket and n is a string.
(define-struct ar (x name))         ;; An army of aliens (aoa) is either
                                    ;; 1. 'uninitialized
                                    ;; 2. A loa
;; A world is a structure,
;; (make-world r ars a d s), where
;; r is a rocket, ars is an loar,
;; a is an aoa, d is a direction
;; and s is a los.
(define-struct world (r rs as dir s))
```

All these data definitions are in familiar territory for the students and require the development of examples and function templates.

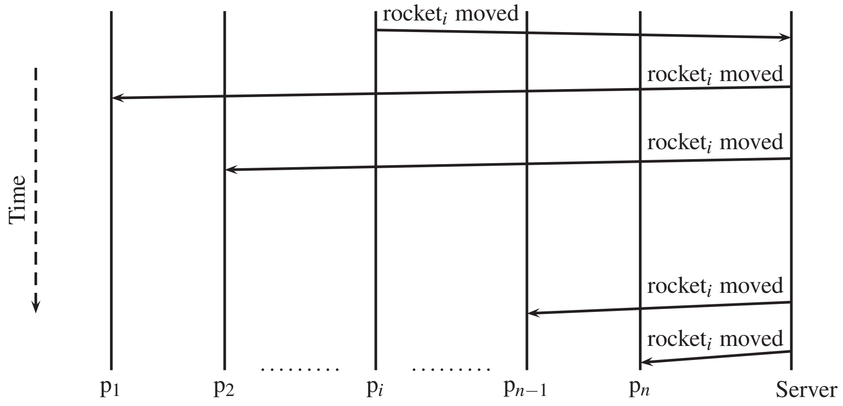

Fig. 3. Communication protocol diagram for a rocket move.

As the server only needs to track the worlds that are part of the game, the only data definition needed is for what we call a universe:

```
;; 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.

5.3 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 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. The horizontal axis represents the components and the vertical access represents time (which grows from the top to the bottom). Messages are represented by solid arrows from source to destination at a slight angle. The angle is used to emphasize that communication is not instantaneous. Dashed arrows are used to represent communication that is implemented by the API (e.g., when a player registers with the server). 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 the rocket is moved or a shot is made by, p_i, the *ith* player. For example, when p_i moves the rocket, a `rocket-moved` message is sent to the server. The server forwards the message to all the other players completing the communication chain. Figure 3 displays the protocol diagram for a rocket move. A similar protocol diagram is developed for shot creation. At this point, no concrete design decisions are made to precisely define messages. The protocol diagrams are purposely left abstract to highlight that message implementation is different from protocol design. In addition, these bite-size steps in development allow students to fully follow the design process.

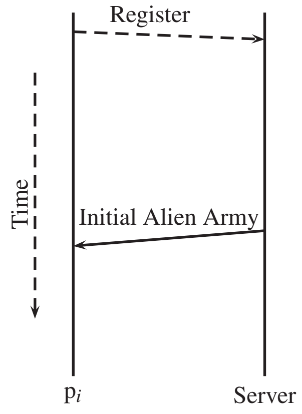

Fig. 4. Joining an empty universe.

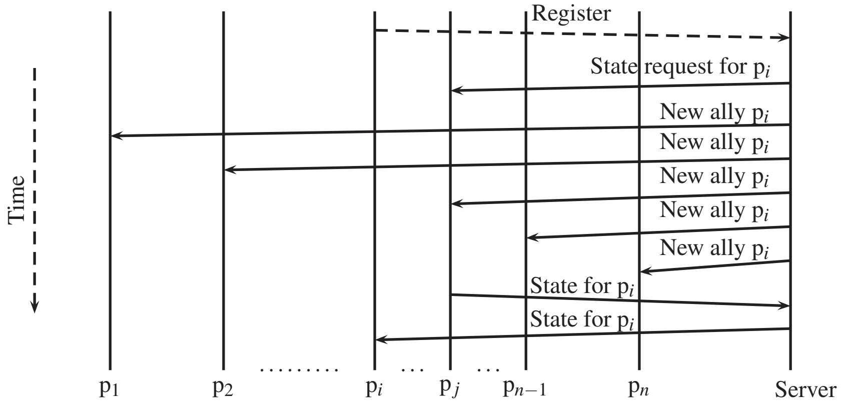

Fig. 5. Joining a non-empty universe.

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, p_i joins an empty universe (i.e., the first player to join the game). 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 first requests on behalf of p_i the state of the game from an existing player, p_j such that $i \neq j$. Then, the server sends a new-ally message to all the worlds already in the universe announcing p_i as a new ally.[7] After the server receives a message from p_j that includes the state of the game for p_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.

[7] The order here matters given that p_i cannot be an ally of itself.

5.4 Message data definitions and marshalling functions

Marshalling is done by converting structured data into an S-expression and appropriately tagging the message. The observant reader will note that tagging messages may not be absolutely necessary, but tagging makes conceptual understanding easier for beginning students. Designing and implementing tagless messages, when appropriate, is a future refinement exercise for students. 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 solid arrows into the server in the protocol diagrams. Likewise, To-Client messages are identified by solid arrows into 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 message data definition is required which is ideal for pedagogy in CS1.

The protocol diagrams are used to develop concrete data definitions for messages and the code required for marshalling. Consider the communication chain in Figure 3. The protocol requires that a player, p_i, send a rocket-move message to the server indicating the ally has changed position. The message must contain enough information to identify which ally has moved. The question posed to students is how this should be done. Class discussion usually leads to two strategies. The first, marshal a new ally rocket to the server that contains the name of the player that made the move and the player's new rocket. Under this design, other players receiving this message from the server unmarshall the allied rocket and make the appropriate substitution in their list of allied rockets. The second, marshal the player's name and the direction in which to move this player's rocket. Under this design, other players receiving this message from the server unmarshall the player's name and the direction, create a new ally rocket, and substitute the appropriate ally in the list of allied rockets.

For illustration purposes, we can arbitrarily choose the first design. Given that an ally rocket is a structure with a number, n, and a string, s, a variety of To-Server messages, a rocket-move message, is a list containing the symbol rocket-move (i.e., the tag), n, and s. The corresponding marshalling and unmarshalling functions are as follows:

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

 ; message --> rocket
 (define (unmarsh-rckt-mv m)
   (make-ar (second m) (third m)))
```

Marshalling and unmarshalling functions require two types of tests. The first is testing that each function works individually as is done in HtDP. The second is

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

Fig. 6. To-server message data definition.

testing to validate that these functions are inverses of each other. For the above functions, sample tests look as follows:

```
(check-expect (marsh-rckt-mv (make-ar 50 "Jen"))
              (list 'rocket-move 50 "Jen"))
(check-expect (marsh-rckt-mv
               (unmarsh-rckt-mv (list 'rocket-move 50 "Jen")))
              (list 'rocket-move 50 "Jen"))
(check-expect (unmarsh-rckt-mv
               (marsh-rckt-mv (make-ar 50 "Jen")))
              (make-ar 50 "Jen"))
(check-expect (marsh-rckt-mv
               (unmarsh-rckt-mv (list 'rocket-move 50 "Jen")))
              (list 'rocket-move 50 "Jen"))
```

Repeating this process for every incoming solid 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. There are three types of To-Server messages. The first is a rocket-move as discussed above. The second is a new-shot message containing the two coordinates of a new shot. The third is a world message containing the name of the receiving world (i.e., a world that just joined the game), the marshalled list of ally rockets, the list of marshalled aliens, the direction the aliens are moving in, and the marshalled list of shots. For each type of message, corresponding marshalling and unmarshalling functions in the manner outlined above are developed in class. Finally, a function template for To-Server messages is also developed in class.

The To-Client message data definition is displayed in Figure 7. To develop this data definition, observe that in the protocol diagrams To-Server messages are broadcasted to the players. Therefore, a To-Client message can be a To-Server message (i.e., varieties 1-3). The protocol diagrams in Figures 4 and 5 inform us that the server sends a player a message to communicate the initial alien army (variety 4), to request the world (variety 5), and to inform a player of a new ally (variety 6). The final variety of a To-Client message is to inform a player to remove an ally (variety 7 is derived from a protocol diagram not displayed). As with To-Server messages, marshalling and unmarshalling functions are developed in class for varieties 4–7 and a function template to process To-Client messages.

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

Fig. 7. To-client message data definition.

5.5 Client 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 now see that their task is to refine their single player game using their new data definitions. This requires updating functions that process data whose definition has been refined, adding communication code to functions that process key events, and the creation of a message processing function. In addition, test must be updated and new tests must be created. 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 may find it tedious and time-consuming. Most students, however, are always enthralled to develop a better game.

The addition of communication code is a new element for students. 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 the key-event handler, displayed in Figure 8, must be updated. The function signature is updated to return a package and the function body is updated to create a package by marshalling the rocket move. The result is displayed in Figure 9. The reader can observe that 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 the player's role in 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 that contains a conditional statement to distinguish among the variety of messages. This handler takes as input a world and a message and it returns a (new) world or package. For

```
; process-key: world key --> world
; 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)))]
        new-world)]))
```

Fig. 8. Key event handler for new data definitions.

```
; 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. 9. Refined key event handler with communication code.

example, this snippet illustrates the idea

```
[(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))]
```

When a rocket-move message arrives, the list of allies is updated and a new world is produced. Students know, from previous experience in the course, that an auxiliary function is needed to process data of arbitrary size (i.e., the list of ally rockets). They also quickly realize how nifty their unmarshalling functions are when processing messages.

The proper design of a function that processes messages requires students to test that a message is having the proper effect, if any, on the state of the client. That is, the client correctly updates its state. To test the snippet above students are presented tests that look like this

```
(check-expect
  (process-message (marsh-rckt-mv (make-ar 50 "Jen"))
                   a-world)
  (make-world (world-r a-world)
              (cons (unmarsh-rckt-mv mess)
                    (filter (lamdba (r)
                              (not (string=? "Jen" (ar-name r))))
                            (world-as a-world)))
              (world-as a-world)
              (world-dir a-world)
              (world-s a-world)))
```

Clearly, this test assumes that update-allies places the moved allied at the front of the list and keeps the relative order of the rest of the allies unchanged. Therefore, the precise tests used by students in their code are dependent on their implementation of auxiliary functions. If students write such tests they demonstrate that they understand that only the list of allied rockets changes when a rocket-moved message is received as required by the design. Furthermore, it shows that they understand how this list changes: substituting the old version of the moved rocket with the new version. Similar tests must be written for every variety of To-Client message. The important point is that students demonstrate by writing such tests that they understand how the client's state changes for every variety of message.

5.6 Server implementation

The server is also implemented 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 in either case. The resulting handler is displayed in Figure 10. The handler for a world disconnecting from the game is developed in the same fashion.

```
; 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. 10. The server's new client handler.

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 code snippet is written:

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

This snippet keeps the universe, u, unchanged, forwards the world message to the player indicated in the message, and removes no players from the universe. The other conditional stanzas for this handler are developed in a similar manner.

Similarly to testing a function to process To-Client messages, testing the server's message processing function requires students to demonstrate how the new state of the server is correct. To test the snippet of code above, for example, students are shown tests that look as follows:

```
(check-expect
  (receive-message
    (list iworld1 iworld2)
    iworld2
    (list 'world "iworld1" empty empty "right" empty))
  (make-bundle
    (list iworld1 iworld2)
    (list (make-mail
            iworld1
            (list 'world "iworld1" empty empty "right" empty)
            empty))))
```

In this test, iworld1 and iworld2 are predefined iworlds in the universe teachpack. The test illustrates that a world message does not change the state of the universe and that the message is relayed to the proper destination world (whose name is embedded in the message) as required by our design. Testing server functions turns out to be difficult for students, because they must use predefined iworlds. That is, they must use an instance of a structure that they did not define and that they are unable to construct. This is why the author has baked into the design that every world has a unique name. To test these functions students must understand that all they need is the name of an iworld and as such it does not matter what the rest of the iworld is. In Aliens Attack, this name is part of every world that joins the game and, therefore, of every iworld in the universe.

5.7 Testing

Students are advised that testing is two-fold: testing function output and testing distributed programming bugs. Function output testing is done using Racket's check-expect (built into the student languages). For client code, this poses little conceptual problems for students. For server code, the situation is slightly more complex. As described above, the student languages do not provide a constructor for iworlds. This has proven counter-intuitive for many students. Tutors report spending a great deal of time helping students to understand using iworlds that they did not create.

Distributed programming bugs refer to, for example, synchronization, communication overhead, and deadlock. This testing is done by running the game. For version 1 of Multiplayer Aliens Attack, students see on their screens a working game with allies. 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 same game state. This approach makes process synchronization a real concern for students and with some class discussion they realize the importance of messages taking time to travel from the source through the server to the destinations. While the messages travel, the players continue changing the state of their copy of the game. Thus, different players have different states. For example, consider player i, p_i, sending a new-shot message. As the message travels to p_j, both p_i and p_j move the shots with each clock tick. When p_i's message arrives at p_j, the new shot starts at the bottom of the screen for p_j, but that same shot is no longer at the bottom of the screen for p_i. In turn, this leads to aliens being hit by some players and not hit by other players. Players are no longer sharing the same game state.

On occasion, a student points out at the beginning of this development that clients may not remain synchronized. Invariably, however, such a student is not confident and cannot clearly explain why clients may become unsynchronized. At this point, encouraging the student to view the development of the thin-server as an exploratory experiment is helpful. That is, students are encouraged to keep those thoughts in mind and to see what happens. The idea of experimentation has proven an effective technique to keep all students engaged.

6 Multiplayer Aliens Attack version 2

Through class discussion, students quickly realize that having multiple copies of the state of the game is problematic. 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 discussions from the results obtained from version 1 of the multiplayer game. This section outlines the steps of the design recipe as done in class for a game that has only one copy of the game state.

Before proceeding with the outline, it is opportune to observe that an instructor can skip the thin-server version and start directly with a thick-server design. Such an approach reduces the amount of time that is needed for a distributed programming module. Given that time is a precious commodity, this is a practical compromise for courses that have little flexibility to introduce new material. The downside to such an approach is that beginning students are less exposed to distributed programming concepts of design and implementation. A balance must be struck in the CS curriculum at large of an institution that allows students to be exposed to all the concepts outlined in the present article. In the author's experience, students benefit from more exposure in CS1 as they are better prepared to face distributed programming modules in courses where it is inevitable such as programming languages or operating systems.

6.1 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 by sending the server requests to move their rocket and to create shots. 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. The syntax required for a player now 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. Figure 11 displays an abstract view of the components. In essence, the students are defining a thick-server that, solely, is responsible for maintaining the state of the game. Students commonly suggest that the server ought to be responsible for detecting the end of the game. Thus, their

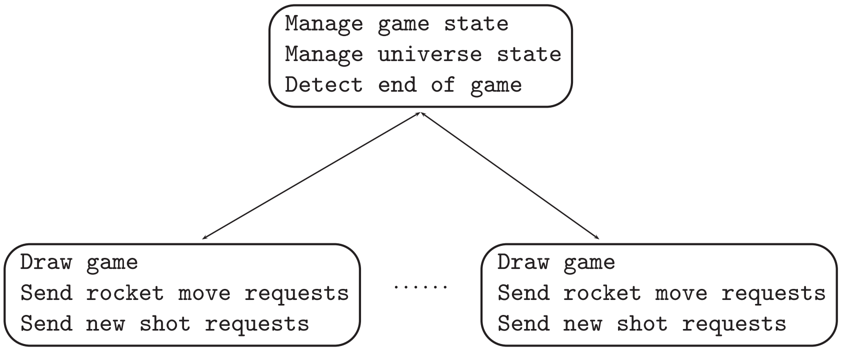

Fig. 11. Components for thick-server version of multiplayer Aliens Attack.

design requires the server to inform the players when the game is over. 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))
```

6.2 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. Setting this boolean to true indicates to the players that the game is over. 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 an aoa, d is a direction, 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))
```

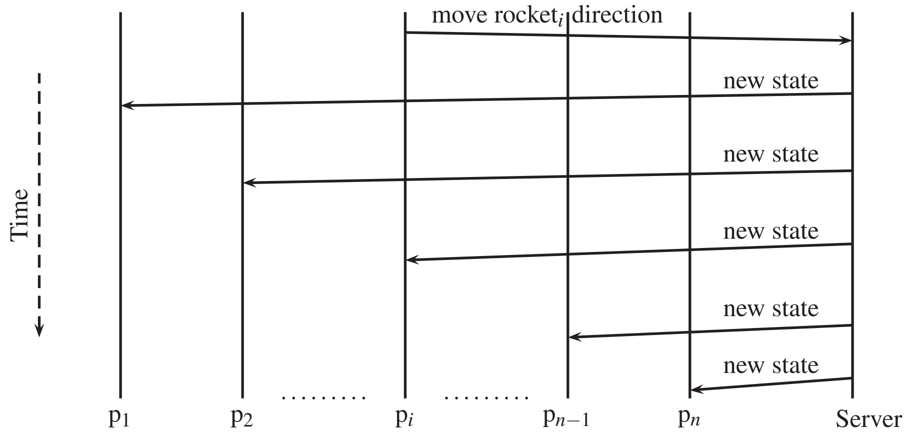

Fig. 12. Version 2 protocol diagram for a rocket move.

6.3 Communication protocol design

As in version 1, players spark a communication chain through key events to move the rocket or to shoot. Figure 12 displays the protocol diagram developed for a rocket move. A player sends the server a message to request their rocket be moved in a given direction. The server changes the state of the game by performing the requested move and sends all players an updated state. A similar protocol diagram is developed for new shots.

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

6.4 Message data definitions and marshalling functions

The protocol diagrams reveal that there is only one variety for a To-Client message and only two varieties for To-Server messages 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 'rckt-move string)
    2. (list 'new-shot number number)
```

Students are usually surprised by the much simpler communication protocol and feel very encouraged.

```
; 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-rckt-move key))]
        [else a-world]))
```

Fig. 13. Player key-event handler for version 2.

For this version, only three pairs of marshalling–unmarshalling functions are required. For example, for a rocket move we have

```
; string --> message
(define (marsh-rckt-move direction) (list 'rckt-move direction))
; message --> string
(define (unmarsh-rckt-move m) (first (rest m)))
```

Testing these functions is done by students as before. There must be at least two tests for output and two tests for the functions being inverses. For example, this pair of functions can be tested as follows:

```
(check-expect (marsh-rckt-move "right") (list 'rckt-move "right"))
(check-expect (unmarsh-rckt-move (list 'rckt-move "right")) "right")
(check-expect (unmarsh-rckt-move (marsh-rckt-move "right")) "right")
(check-expect (marsh-rckt-move
                (unmarsh-rckt-move (list 'rckt-move "right")))
              (list 'rckt-move "right"))
```

The most complex pair is the one for a world message that provides the opportunity to reinforce lessons using lambda expressions and higher order functions like map.

6.5 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 13 displays the handler for key events developed 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 request message is sent to the server containing the direction to move the player's rocket. 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 straightforward to design:

```
; 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)]))
```

Technically, the conditional is not needed as there is only one variety of To-Client message. Classroom experience, however, has proven that it is sometimes useful to have a conditional to guard for invalid messages. When students work in groups, for example, it is common for a subset of the group to design and implement the server and another subset to do the same for the player code. It is not uncommon for errors in the tagging convention to exist between the subgroups. In these cases, the above conditional has proven effective in reducing debugging time and, more importantly, frustration in the students.

As before testing this function requires tests that demonstrate that the state of the client is correctly updated. For this version of Aliens Attack, the world embedded in the message becomes the state of the client. For example, this function may be tested as follows:

```
(check-expect
 (process-message INIT-WORLD
                  (list 'world
                        (list (50 "Jen") (120 "Beth"))
                        (list (list 10 20))
                        "left"
                        empty
                        false))
 (make-world (list (make-ar 50 "Jen")
                   (make-ar 120 "Beth"))
             (list (make-posn 10 20))
             "left"
             empty
             false))
```

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))
```

Recall that, according to this design, setting this boolean is the responsibility of the server.

```
; 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 (iw)
                                (make-mail iw (marsh-world new-w)))
                              (univ-worlds u))
                         empty))]))
```

Fig. 14. Clock tick handler for version 2.

6.6 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, as before, dispatches on whether the state of the universe has an empty list of iworlds or not. The message handler dispatches on the two varieties of To-Server messages and is tested in the same manner as version 1. 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 no longer an ally.

The clock tick handler is the most complex. It dispatches on whether or not the game has come to an 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 is displayed in Figure 14. Testing this function requires at least two tests: one in which the game is over and one in which the game is not over. Figure 15 displays two such tests presented to students. In the first, the game is not over and the state of the server is correctly updated to have a new game state. Students are told that as long as update-world is properly tested, then it may be used to test update-univ. In the second test, the game is over (i.e., the list of aliens is empty) and the protocol requires that all worlds in the universe be sent a state where the over flag in the world is set. This is precisely what is done by the function.

6.7 Testing

Testing reveals that the synchronization problem appears resolved. We say *appears*, because we do not prove that it is resolved.[8] An instructor can, indeed, leave it

[8] Program correctness is not yet woven into CS1 at SHU.

```
(check-expect (update-univ initU)
              (make-bundle (make-univ (univ-worlds initU)
                                      (update-world (univ-state initU)))
                           empty
                           empty))
(check-expect
 (update-univ
  (make-univ (list iworld1)
             (make-world (list (make-ar 50 'iworld1))
                         empty
                         "right"
                         empty
                         false)))
 (make-bundle (make-univ (list iworld1)
                         (make-world (list (make-ar 50 'iworld1))
                                     empty
                                     "right"
                                     empty
                                     true))
              (list (make-mail iworld1
                               (marsh-world
                                (make-world (list (make-ar 50 'iworld1))
                                            empty
                                            "right"
                                            empty
                                            true))))
              empty))
```

Fig. 15. Tests for the clock tick handler.

at that and move on. Students have done enough to get them started thinking about synchronization. There is, of course, an additional issue that can be pointed out to 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 that in such cases mutual exclusion must be guaranteed. This topic is not thoroughly discussed, but students are told that solutions will be studied, for example, in an operating systems course.

More importantly for our purpose, testing also reveals a most annoying characteristic for students: the game is much slower. The bottleneck and communication overhead issues are brought forward during class discussion. This motivates the development of a third version of the game.

7 Multiplayer Aliens Attack version 3

The development of version 2 marks the end of lecturing in 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 client 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. If necessary, of course, they must redesign to fix bugs.

The refinements developed by students are 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 game state every time. Instead, they only transmit the part of the state that is changed. This type of communication protocol has been implemented in practice by, for example, Quake 3 (Sanglard, 2012). On an unequivocally personal note, to have CS1 students write distributed applications on their own is nothing short of amazing and students feel extremely proud of this achievement.

8 Student Assessment

The distributed programming module has been part of CS1 at SHU for four years (Fall 2012 to Fall 2016 semesters). At the end of each semester, students are asked to fill out a survey to evaluate the distributed programming module. It should be noted that the intent here is to probe into students' perceptions/attitudes and not to define objective measures that beginning students can make judgments against. This means that the subject matter of any given question was not formally defined. Each respondent is being asked about their personal and subjective opinion.

The students were asked to rank their opinion on the following questions:

How intellectually stimulating is distributed programming? Scale: 1 (low stimulation) to five (high stimulation).

How much more difficult is distributed programming? Scale: 1 (not at all more difficult than non-distributed programming) to 5 (a lot more difficult than non-distributed programming). For this question, students are comparing distributed programming with the type of programming done earlier in the semester in which protocol diagrams, communication code, and marshalling/unmarshalling functions are not required.

Do you feel empowered by designing and implementing distributed programs? Scale: 1 (not at all empowered) to 5 (very empowered).

Does developing distributed programs make you a better problem solver? Scale is from 1 (Not at all) to 5 (Very much so). Developing in this context means designing and implementing.

What is your level of excitement after developing a multiplayer video game? Scale: 1 (not at all excited) to 5 (extremely excited).

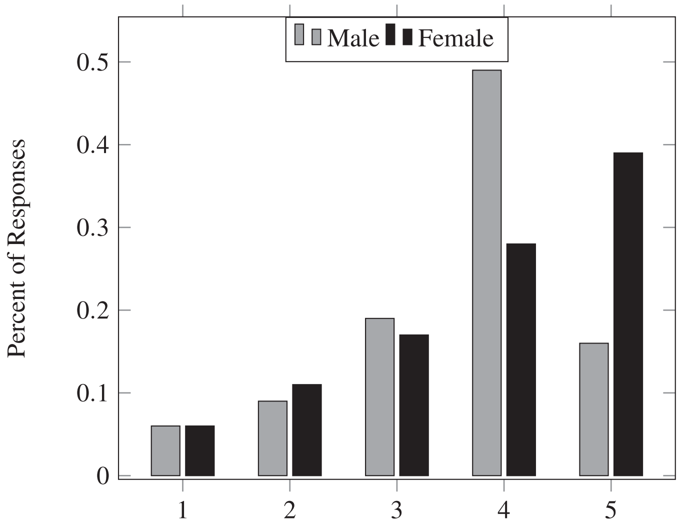

Fig. 16. Distributed programming is intellectually stimulating by gender.

Has learning about distributed programming been helpful? Scale: 1 (not at all helpful in developing your problem solving and critical thinking skills) to 5 (very helpful in developing your problem solving and critical thinking skills).

For each question, the data is analyzed through the control variables of gender and previous programming experience. A total of 91 students were surveyed of which 19 were females (20.9%) and 72 were males (79.1%). For previous programming experience students picked one of the following categories: None, High School, and Other (College or Self-taught). If they had programming experience, they were asked what programming languages they have used. Among the respondents there is 59 students (64.8%) with no prior programming experience, 23 students (25.3%) with programming experience from high school (Java or C++), and 9 students (9.9%) with other[9] programming experience. None of the respondents had any experience with program design or HtDP's design recipe.

During the final examination period, but after exams are submitted by the students, the instructor along with the tutors have an open discussion with the students about the distributed programming module. This discussion provides us with the opportunity to ask follow-up questions that are reported on later in this section. The students have been very open and eager to provide feedback in this manner.

8.1 How intellectually stimulating is distributed programming?

Figure 16 displays the distribution of responses classified by gender. The data suggests that both female and male students find distributed programming intellectually stimulating. For both groups the distribution is slightly negatively

[9] Six students (6.6%) with self-taught programming experience (Java, Python, or C++), and three students (3.3%) with college programming experience (Maple or Mathematica).

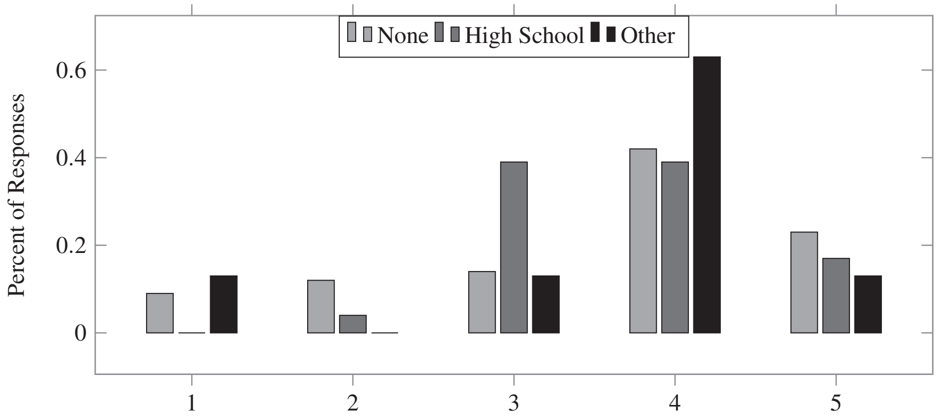

Fig. 17. Distributed programming is intellectually stimulating by programming experience.

skewed meaning that most responses provided are above the mean ($\mu = 3.8/3.6$, respectively, for female and male students). In fact, an overwhelming majority in both groups (67% of females and 66% of males) gave a response in [4..5]. These numbers clearly indicate that overall CS1 students find distributed programming intellectually stimulating. This is important, because material that students find stimulating fosters enthusiasm for learning. This success is attributable to meeting the idiosyncratic needs of students (through tutoring), encouraging students to be creative (allowing students to discover bugs instead of prescribing bugs), and helping students to accomplish more than they expect or believe possible (designing a multiplayer game) as is done in transformational leadership (Bolkan *et al.*, 2011). The data also suggests that the approach described in this article does not exhibit a gender gap and, therefore, may be an effective way to increase gender diversity in CS.

Figure 17 displays the distribution of responses classified by prior programming experience. The distributions of all three categories are slightly negatively skewed indicating that most responses are above the mean ($\mu = 3.57/4/4$, respectively, for none, high school, and other). In fact, for all three categories the majority of respondents gave an answer in [4..5] suggesting that regardless of programming experience students find distributed programming very intellectually stimulating. It is noteworthy that students with prior high school programming experience find distributed programming only slightly less intellectually stimulating. Usually, at Seton Hall at least, these students are the most resistant to the program by design methodology, because the course is not taught in a language they believe they know (e.g., Java or C++). This is an indication of the success the methodology presented has in overcoming the students' bias and in getting them engaged in program by design.

8.2 How much more difficult is distributed programming?

Figure 18 displays the distribution by gender. A visual inspection of the bar plot suggests that for both males and females few students feel distributed programming

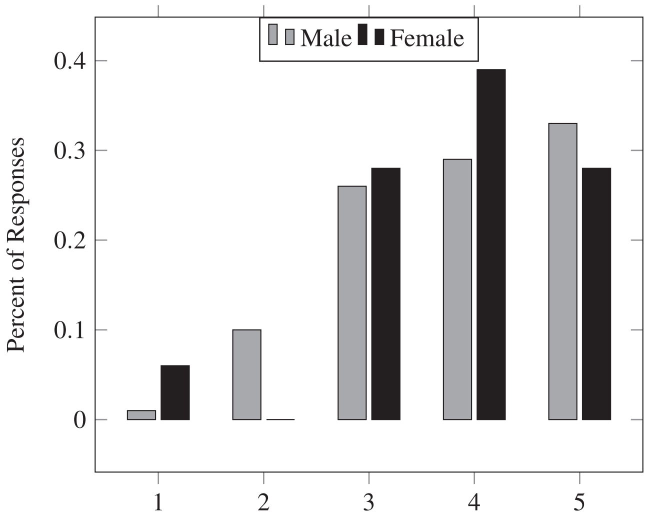

Fig. 18. Distributed programming is more difficult by gender.

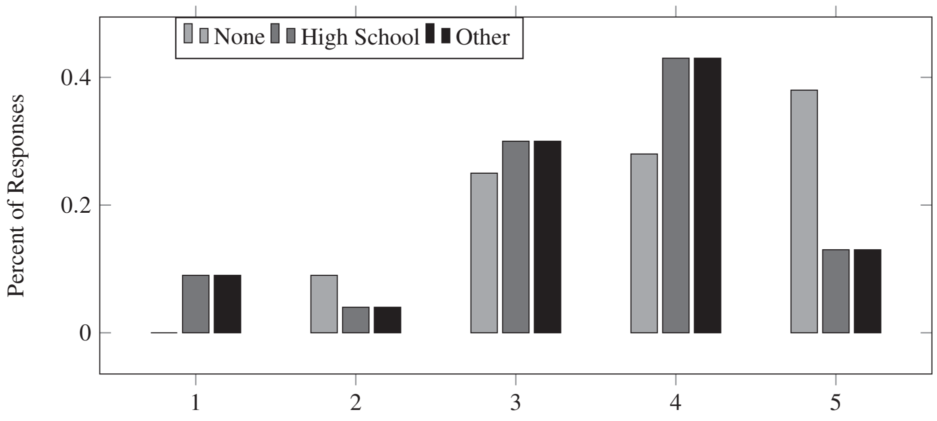

Fig. 19. Distributed programming is more difficult by programming experience.

is equally as hard as non-distributed programming and most feel distributed programming is harder. For both male and female students the mean is 3.8, suggesting that the difficulty of the material presented is gender neutral. There is, however, a slight variation in the percentage of responses in the [4..5] range. The percentage for males is 62% versus 67% for females. This differences suggest that instructors ought to make sure female students seek feedback as they progress in the development of their distributed applications. An effective way to achieve this is to instruct the tutors to reach out to female students during tutoring hours.

Figure 19 displays the distribution of responses by programming experience. The bar plot clearly reveals that respondents in all categories find distributive programming harder than non-distributive programming. The data suggests that students with prior programming experience find distributed programming slightly less difficult. The primary reason for this is that these students are more skilled at chasing down bugs. That is, they are less frustrated by having to fix errors, because they can find the reason for a bug faster. To overcome the increased difficulty encountered by students with distributed programming, a well-trained team of undergraduate tutors is very helpful. Students identify with the tutors, because the tutors are also undergraduate students. Therefore, encouragement by tutors accompanied by guidance through the steps of the design recipe for distributed programming tends to be very effective and results in the overwhelming majority of students completing their projects.

Follow-up questions reveal three major observations. The first is that some students find it confusing to have more than one file for a program (one for the server and one for each of the players). This difficulty is well-addressed during tutoring hours. Tutors explain that code is written separately for each distinct component. The second is that the design recipe for distributed programming does not feel to students as easy to follow as other design recipes studied during the semester. In essence, the difficulty for students is that the design recipe for distributed programming does not outline a template that tells a programmer how to process data. Up to this point in the course, students have developed templates to process data by exploiting structural recursion (e.g., lists, natural numbers, and trees). These templates, in essence, tell a programmer how data ought to be processed. Since distributed programming is based on components, not structure, this is a rather encouraging observation by students. It demonstrates that they are becoming aware that there are design strategies beyond structural recursion. It is interesting to note that students in the CS2 course taught by the author (Morazán, 2015) make the same observation about the design recipe for generative recursion. The third is the resolution of bugs in the marshalling and unmarshalling functions that led to "unknown message" errors or errors trying to access parts of a message that do not exist. The difficulty lies in that a message that, for example, causes the server to crash is not always fixed in the server's code. Instead, it may have to be fixed in the client's code. Students, however, tend to only search for the bug in the code that signals the error (i.e., the server's code in this example). Clearly, this issue is intertwined with having more than one file associated with a program as the bug may be on either side of the communication (i.e., in one of two files). This obstacle is successfully addressed by emphasizing to students that communication is always between a client and a server. Such emphasis is done with an in-lecture example and by attentive tutors that remind students during discussions about bugs. Instructors that may consider these issues unsurmountable in their institutions, at first, should not be discouraged and ought to keep in mind that at SHU virtually all the students successfully complete their distributed programming project.

8.3 Do you feel empowered by designing and implementing distributed programs?

Figure 20 displays the distribution of responses by gender. The proportion of responses in the range [4..5] is 56% for female students and 35% for male students, indicating that female students feel more empowered than male students by learning to design and implement distributed programs. This suggests two conclusions. First,

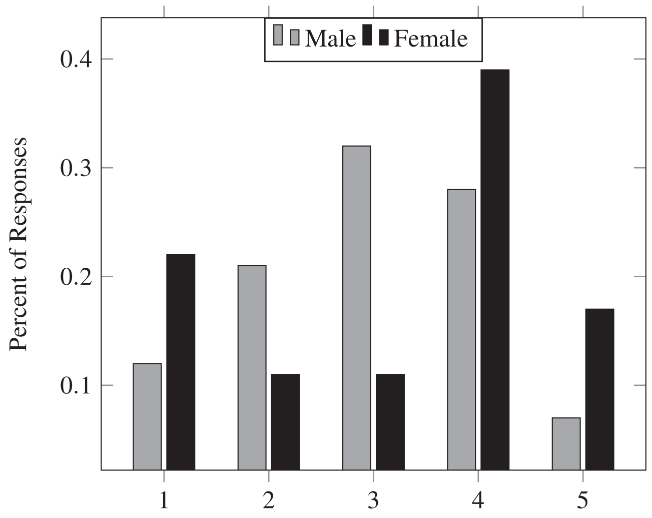

Fig. 20. Distributed programming is empowering by gender.

that using the design and implementation of distributed video games may be an effective tool to engage female students in programming. Second, that male students also feel empowered by learning about distributed programming, but their feeling of empowerment is lower.

The distribution of female respondents suggests a bimodal distribution caused by some third variable. Co-variance tests were computed using variables that are independent of the course (e.g., major, age, college year, calendar year, and semester). Nothing indicates there is a characteristic that separates female respondents into two separate groups. Therefore, the only conclusion that can be reached is that a larger sample is needed to see a bell-shaped distribution or to confirm that a bimodal distribution for this variable actually exists. Given that none of the other measured variables in this study exhibit a bimodal distribution, it is unlikely that a larger sample will exhibit one.

Figure 21 displays the distribution of responses by programming experience. A visual inspection of the bar plot reveals that students with programming experience acquired in college or by self-study tend to feel more empowered. The range [4..5] contains 38% of respondents with no prior programming experience, 35% of respondents with prior high school programming experience, and 63% of respondents with other prior programming experience. For all three categories, however, the overwhelming number of respondents are in the range [3..5]: 59% for no prior programming experience, 83% for prior high school programming experience, and 75% for other prior programming experience. The conclusion is that the methodology described in this article is empowering for all students regardless of prior programming experience.

Follow-up questions reveal that students would feel even more empowered if they knew how to do distributed programming using other programming languages (e.g., Java). Students commented, for example, that "*it would be cool to now do this in*

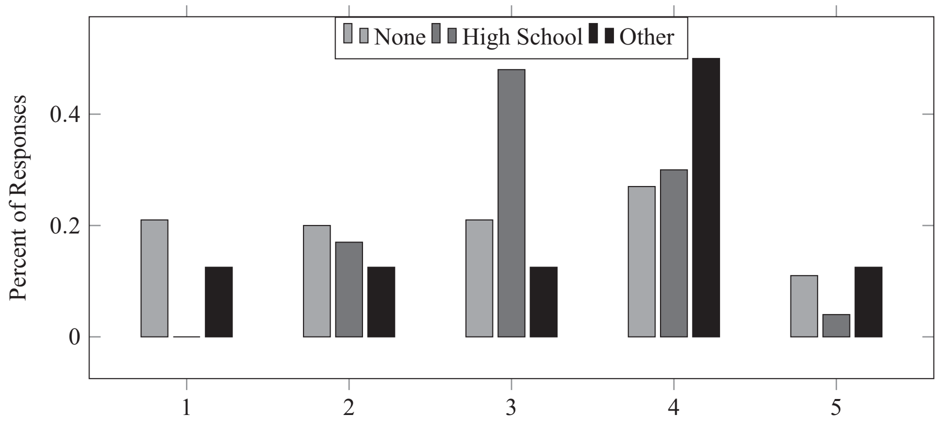

Fig. 21. Distributed programming is empowering by programming experience.

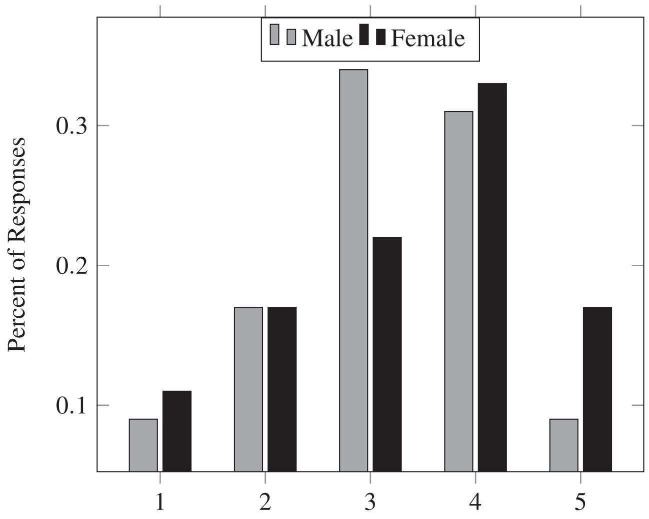

Fig. 22. Distributed programming has made you a better problem solver by gender.

Java" and that "*I want to learn how to write this as an app.*" This desire by students to transfer the knowledge gained to other programming environments is a clear indication of the success of the approach described in this article.

8.4 Does developing distributed programs make you a better problem solver?

Figures 22 displays the distribution of responses by gender. There is a significant number of students that, subjectively, feel strongly about being better problem solvers. We can observe that female respondents do feel slightly stronger about being better problem solvers suggesting that an emphasis on problem-solving using distributed video games can be an effective tool to close the gender gap in CS. This is not offered as a conclusion, but rather as an avenue that is worth exploring more in practice.

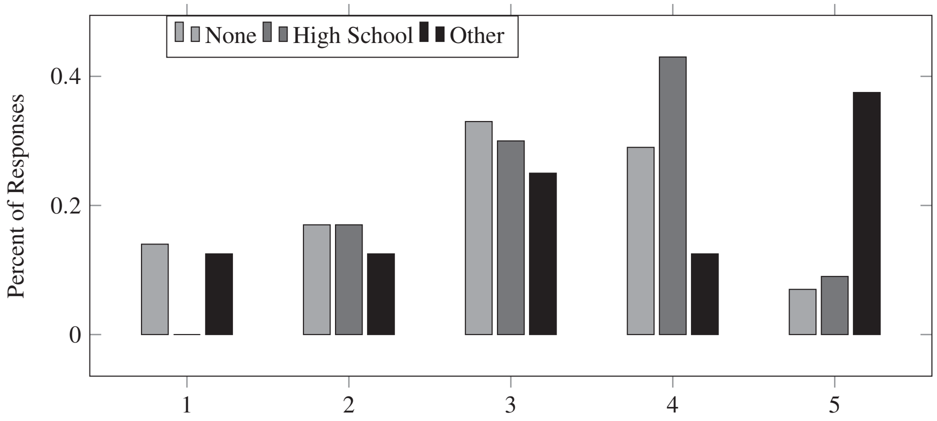

Fig. 23. Distributed programming has made you a better problem solver by programming experience.

Figures 23 displays the distribution of responses by prior programming experience. A visual inspection of the bar plot reveals that for all categories respondents feel that they are, subjectively, better problem solvers as we can observe the overwhelming number of respondents in all categories in the range [3..5]. Respondents with prior high school programming experience feel the strongest about being better problem solvers, while those with no prior programming experience feel the least strong about being better problem solvers. The fact that most students feel that they have become better problem solvers is particularly encouraging considering that none of the students has had prior exposure to distributed programming.

Follow-up questions reveal that many students feel they understand the need to divide large problems in order to solve them. They also feel that there is a connection with structural recursion, but unlike structural recursion the programmer must figure out how to divide a large problem into smaller problems. In fact, students also equate distributed programming with running a large company: different offices are responsible for different things much like each component in a program. Many students do state that they need more practice to become good problem solvers using distributed programming. Unfortunately, there is not enough time to provide students with more practice in CS1. Nonetheless, this attitude among students clearly indicates that they feel distributed programming is within their reach and that they want to further develop their skills. The lesson is that the introduction to distributed programming described in this article is not overwhelming students and they are aware of the growth in their skills thanks to this module.

8.5 What is your level of excitement after developing a multiplayer video game?

Figure 24 displays the distribution by gender. A visual inspection of the bar plot reveals that both genders are excited after developing a distributed multiplayer video game. Females students, however, are more excited as evidenced by a higher proportion of respondents, 56% versus 41%, in the range [4..5]. This is further

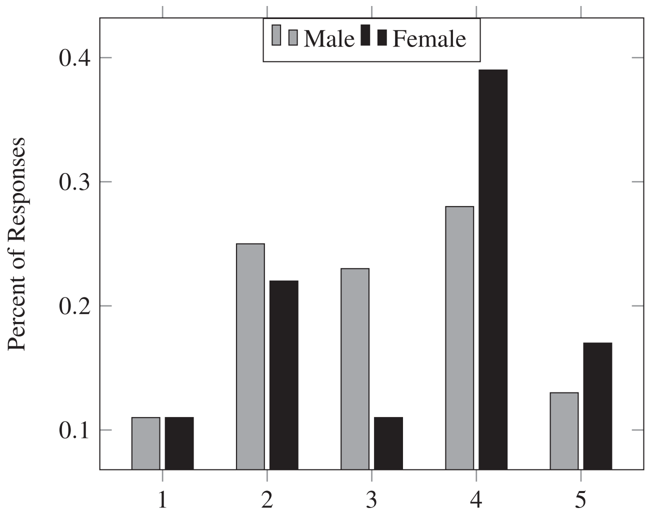

Fig. 24. Level of excitement by gender.

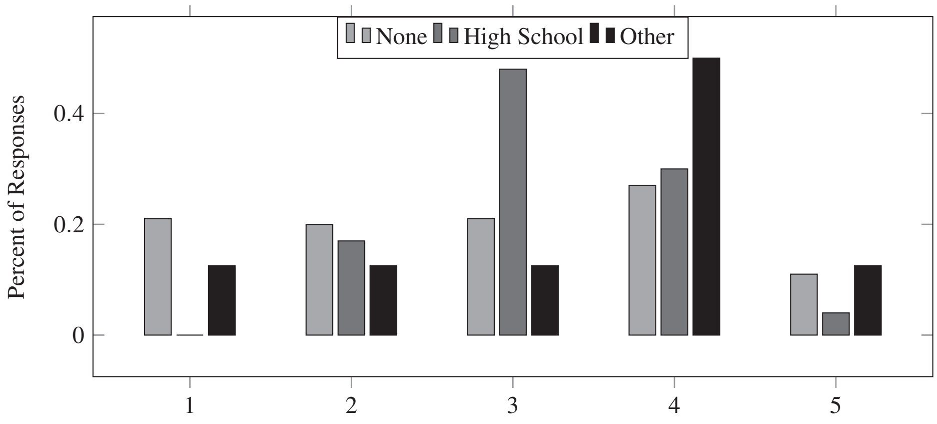

Fig. 25. Level of excitement by programming experience.

confirmation that developing distributed video games in CS1 may be an effective means to close the gender gap in CS.

Figure 25 displays the distribution by prior programming experience. A visual inspection of the bar plot reveals that across all three categories students feel excited. A significant number of respondents in each category gave a response in [4..5]: 47% of the respondents with no prior programming experience, 39% of the respondents with prior high school programming experience, and 50% of the respondents with other prior programming experience. The conclusion to reach is that designing and implementing distributed video games excites students in general regardless of prior programming experience.

There are, however, some students that do not share in the excitement. Follow-up questions revealed that those that are not excited (range [1..2]) never imagined "*how much work goes into creating a multiplayer distributed video game*." It is the amount

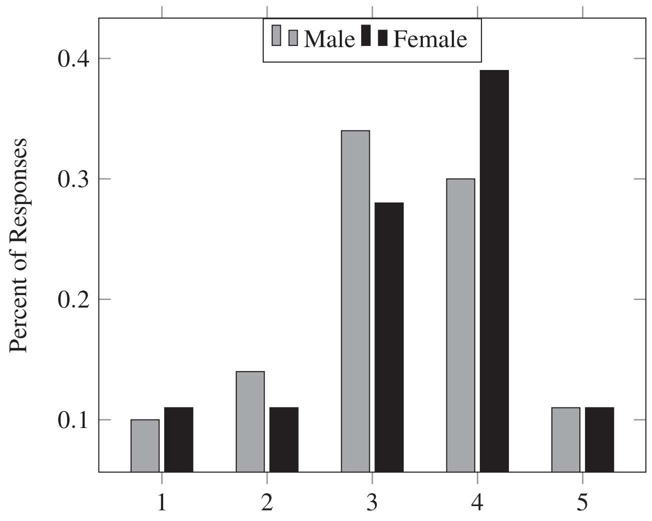

Fig. 26. Distributed programming helpful by gender.

of work, not the complexity of the material, that tapers excitement among some students. This is a significant difference as no one can now argue that distributed programming is too hard for CS1 students. Being surprised by the amount of work is to be expected as the majority of students are in their first year of university and consider CS1 the hardest course they have ever taken. Nonetheless, the data signals that the curriculum presented in this article is appropriate for CS1. Instructors worried about the amount of work being discouraging for students can choose to implement a game that is simpler than the one described in this article.

8.6 Has learning about distributed programming been helpful?

The distribution of responses by gender is displayed in Figure 26. In general, students found the material valuable in developing their problem-solving and critical-thinking skills. This strongly suggests that distributed programming has a proper place in CS1. Females students overall, however, feel stronger that the distributed programming module is helpful as 50% of females respondents are in [4..5] versus 41% of male respondents. This is an important characteristic of the distribution, because it signals that female students are completing the module with a bigger sense of accomplishment that goes beyond programming than their male counterparts. Once again, the data suggests that it is reasonable to expect that the approach described in this article serves as strong motivation to study CS and programming for female students.

The distribution of responses by programming experience is displayed in Figure 27. The most noticeable trend is that most respondents in all three categories feel that the distributed programming module is helpful. This feeling is stronger among respondents with prior high school programming experience. The weaker feeling in the other two categories is likely due to the fact that students with no prior academic programming experience are making an effort that they had never had to

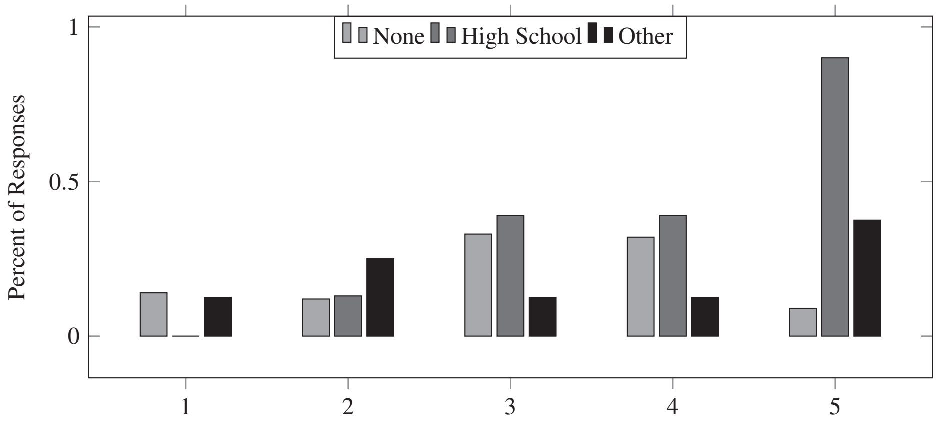

Fig. 27. Distributed programming helpful by programming experience.

undertake before. This means that instructors must be more attentive to this group of students to make sure that they are not stuck to the point of overwhelming frustration when developing a distributed application. A proactive tutoring team that looks over students' shoulders to make sure they are making progress is a great asset to help those with less experience to become successful.

9 Concluding remarks

Distributed programming can be an integral part of CS1 as long as it is put within reach of beginners. The universe teachpack developed as part of HtDP in conjunction with a design recipe makes it possible for distributed programming to be a natural topic in an introductory course. Beginning students are aware that distributed programming has become ubiquitous and most are curious to discover how it works. This article presents an example of how to be successful with distributed programming in CS1 through an illustrative development of a non-trivial functional multiplayer video game. Not a single function needed for the presented multiplayer 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 the development of a distributed functional video game in CS1 is that students become excited about programming, problem solving, intellectual stimulation, and CS. Another advantage is that students think about distributed programming issues early in their undergraduate years, thus, providing a solid foundation for advanced courses.

The empirical data presented strongly suggests that CS1 students are receptive and appreciate an introduction to distributed programming. Students feel empowered and feel they are better problem solvers by writing a distributed application. Undoubtedly, part of the success stems from the application being functional. That is, students are not burdened with the task of sequencing assignment statements. Part of the success is also attributable to the framework provided by a design recipe for distributed programming. The design recipe provides students with a road map they can follow producing a tangible result at each step. This facilitates the development for students, the ability for tutors to offer guidance, and the grading for instructors.

The empirical data also indicates that female students are very receptive to the approach described. For the sample studied, female students feel, more so than their male counterparts, that distributive programming is intellectually stimulating, is empowering, makes them better problem solvers, is exciting, and is helpful in the development of problem solving and critical thinking skills. They also feel that distributed programming is equally as hard as their male counterparts feel. Finally, the empirical data suggests that the approach described is successful with students regardless of their programming background. Students that have prior high school programming experience seem to struggle a little less with the material, but students with other prior programming experience feel more empowered.

The example used, Aliens Attack, is one of many suitable for asynchronous distributed programming in CS1. Turn-based games have also been successfully used by others and provide the opportunity for shorter discussions in class. The caveat, of course, is that they also make it more difficult for students to experience some of the pitfalls encountered in distributed programming. Either way, the inescapable conclusion is that distributed programming is well within the reach of beginners that are given the proper framework and guidance in class. The design recipe and approach described in this article can easily be adapted to other student programming languages and readers are encouraged to develop their own distributed programming modules for CS1.

Acknowledgments

The author is very grateful to Lindsey M. Reams, Nicholas R. Olson, Shamil Dzhatdoyev, Josephine Des Rosiers, and Nicholas Nelson for contributing, as outstanding teaching assistants and tutors, to the success of the distributed programming module in CS1 at SHU. The author also thanks the plt-scheme and the plt-edu mailing list community for the many frank and eye-opening discussions about teaching programming, about HtDP, and about interesting programming projects for students. Finally, 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 Seton Hall students that implemented distributed Aliens Attack.

References

Achten, P. (2008) Teaching functional programming with soccer-fun. In Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education. FDPE '08. New York, NY, USA: ACM, pp. 61–72.

Adams, J. C., Brown, R. A. & Shoop, E. (2013) Patterns and exemplars: Compelling strategies for teaching parallel and distributed computing to CS undergraduates. In Proceeding of the 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20–24, pp. 1244–1251.

Bice, F., DeMaio, R., Florence, S., Lin, F.-Y. M., Lindeman, S., Nussbaum, N., Peterson, E., Plessner, R., Horn, D. V., Felleisen, M. & Barski, C. (2013) In Proceeding of the Realm of Racket. No Starch Press.

Bolkan, S., Goodboy, A. K. & Griffin, D. J. (2011) Teacher leadership and intellectual stimulation: Improving students' approaches to studying through intrinsic motivation. *Commun. Res. Rep.* **28**(4), 337–346.

Cooper, S., Dann, W. & Pausch, R. (2000) Alice: A 3-D tool for introductory programming concepts. *J. Comput. Sci. Coll.* **15**(5), 107–116.

Courtney, A., Nilsson, H. & Peterson, J. (2003) The yampa arcade. In Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell. Haskell '03. New York, NY, USA: ACM, pp. 7–18.

Dann, W. P., Cooper, S. & Pausch, R. (2011) *Learning to Program with Alice*. 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall.

Danner, A. & Newhall, T. (2013) Integrating parallel and distributed computing topics into an undergraduate CS curriculum. In Proceedings of IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20–24, pp. 1237–1243.

Felleisen, M., Findler, R., Fisler, K., Flatt, M. & Krishnamurthi, S. (2008) How to Design Worlds. Accessed February 15, 2018. Available at: http://world.cs.brown.edu/1/.

Felleisen, M., Findler, R., Flatt, M. & Krishnamurthi, S. (2001) *How to Design Programs: An Introduction to Programming and Computing*. Cambridge, MA, USA: MIT Press.

Felleisen, M., Findler, R., Flatt, M. & Krishnamurthi, S. (2009) A functional I/O system or, fun for Freshman kids. In Proceedings of the ACM SIGPLAN International Conference on Functional Programming, pp. 47–58.

Felleisen, M. & Krishnamurthi, S. (2009) Viewpoint: Why computer science doesn't matter. *Commun. ACM* **52**(7), 37–40.

Felleisen, M., Findler, R. B., Flatt, M. & Krishnamurthi, S. (2015) (August) How to Design Programs. Accessed February 15, 2018. Available at: http://www.ccs.neu.edu/home/matthias/HtDP2e/.

Findler, R., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P. & Felleisen, M. (2002) DrScheme: A programming environment for scheme. *J. Funct. Program.* **12**(2), 159–182.

Findler, R. B. (2008) *CS 15100 Fall 2008 Project 3: ChatNoir*. Dept. of Electr. Engr. and Comp. Sci., Northwestern University. Accessed February 15, 2018.

Friedman, D. P. & Wand, M. (2008) *Essentials of Programming Languages*. 3rd ed. MIT Press.

Gestwicki, P. V. (2007) Computer games as motivation for design patterns. In Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education. SIGCSE '07. New York, NY, USA: ACM, pp. 233–237.

Hennessy, J. L. & Patterson, D. A. (2011) *Computer Architecture: A Quantitative Approach*. 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM) & IEEE Computer Society. (2013) *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. New York, NY, USA: ACM. 999133.

McHoes, A. & Flynn, I. M. (2013) *Understanding Operating Systems*. Cengage Learning.

Morazán, M. T. (2011) Functional video games in the CS1 classroom. In *Trends in Functional Programming: 11th International Symposium*, TFP 2010, Norman, OK, USA, May 17–19, 2010. Revised Selected Papers, Page, R., Horváth, Z. & Zsók, V. (eds), Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, pp. 166–183.

Morazán, M. T. (2012) Functional video games in CS1 II. In *Trends in Functional Programming: 12th International Symposium, TFP 2011, Madrid, Spain, May 16-18, 2011, Revised Selected Papers*, Peña, R. & Page, R. (eds), Lecture Notes in Computer Science, vol. 7193. Berlin, Heidelberg: Springer, pp. 146–162.

Morazán, M. T. (2014) Functional video games in CS1 III. In *Trends in Functional Programming: 14th International Symposium, TFP 2013, Provo, UT, USA, May 14–16, 2013, Revised Selected Papers*, McCarthy, J. (ed), Lecture Notes in Computer Science, vol. 8322. Berlin, Heidelberg: Springer, pp. 149–167.

Morazán, M. T. (2015) Generative and accumulative recursion made fun for beginners. *Comput. Lang. Syst. Struct.* **44**(PB), 181–197.

Prasad, S. K., Chtchelkanova, A., Dehne, F., Gouda, M., Gupta, A., Jaja, J., Kant, K., Salle, A. L., LeBlanc, R., Lumsdaine, M., Padua, D., Parashar, M., Prasanna, V., Robert, Y., Rosenberg, A., Sahni, S., Shirazi, B., Sussman, A., Weems, C. & Wu, J. (2012) *NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing-Core Topics for Undergraduates*. Tech. rept. Center for Parallel and Distributed Computing Curriculum Development and Educational Resources.

Pulimood, S. M. & Wolz, U. (2008) Problem solving in community: A necessary shift in CS pedagogy. *SIGCSE Bull.* **40**(1), 210–214.

Sanglard, F. (2012 June) *Quake 3 Source Code Review: Network Model*. Accessed February 15, 2018. Available at: http://fabiensanglard.net/quake3/network.php.

Schanzer, E. & Fisler, K. (2015) Teaching algebra and computing through bootstrap and program by design. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education. SIGCSE '15. New York, NY, USA: ACM, pp. 695–695.

Scott, M. L. (2000) *Programming Language Pragmatics*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Silberschatz, A., Galvin, P. B. & Gagne, G. (2010) *Operating Systems Concepts With Java*. 8th ed. John Wiley & Sons, Inc.

Stallings, W. (2016) *Computer Organization and Architecture: Designing for Performance*. 10th ed. Pearson Education Limited.

Sung, K. (2009) Computer games and traditional CS courses. *Commun. ACM* **52**(12), 74–78.

Tanenbaum, A. S. & Bos, H. (2014) *Modern Operating Systems*. 4th ed. Upper Saddle River, NJ, USA: Prentice Hall.

Tucker, A. & Noonan, R. (2001) *Programming Languages: Principles and Paradigms*. 1st ed. McGraw-Hill Higher Education.