Manatee: Using Google Nearby Messages to Build a Cross-Platform, Proximity-Based Mobile Client for Cards Against Humanity-Style Party Games

Manatee: Using Google Nearby Messages to Build a Cross-Platform, Proximity-Based Mobile Client for Cards Against Humanity-Style Party Games

A Thesis in the Field of Information Technology

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

November 2019

Abstract

Party games like Apples to Apples and Cards Against Humanity are, in many ways, better suited to digital decks than to physical decks: digital decks can be easily created, remixed, combined, and shared, and a single thumb drive can easily store a deck that would fill an entire room if printed. The rise of multiple online Cards Against Humanity clones since the game's original publication demonstrates that there is a demand for a digital version of the game that provides those advantages. However, while the permissive Creative Commons license under which Cards Against Humanity is distributed allows for remixing and sharing, it does not permit the decks to be used for commercial purposes, and past attempts to produce a web version of the game have struggled to pay for servers and hosting fees.

Manatee is a cross-platform mobile game designed specifically for in-person gameplay without the need for a centralized game server or for special hardware. It is built with Xamarin.Forms and Google's Nearby Messages, and designed to work within Nearby Messages' inherent latency and bandwidth limitations. Via the Cardcast API, it allows users to choose from both existing published decks and thousands of user-created decks, and lets them start and join games without needing a preexisting shared network or relying on a centralized server.

Dedication

To my mom, who thought this was a totally reasonable idea.

Acknowledgments

I would like to acknowledge the invaluable help of my advisor, David Platt, for sharing his wealth of knowledge and experience and from firmly guiding me away from some very unfortunate potential design choices. I'd like to thank my colleagues, and especially my boss, Ka Kau Chan, for their support (and, occasionally, sympathy) in this process. I'd also like to thank Barbara Liepmann, Alison Hale, Nora Carr, Schuyler Shepherd, Andrea Schuchardt, and everyone else whose thoughtful suggestions and feedback helped shape this project.

I would also like to acknowledge Cards Against Humanity, for distributing their game under such a permissive license, and the unsung heroes behind the Cardcast API, without whom this project wouldn't exist.

Table of Contents

List of Figures

Chapter I.

Introduction

There are several fill-in-the-blank or word-association party games in distribution, the most prominent being the award-winning Apples to Apples (American Mensa, n.d.) and the wildly popular Cards Against Humanity (Deutsch, 2015). The same app could be used to play variations of any of those games, but because of the permissive Creative Commons license under which Cards Against Humanity is distributed (Made By CC, 2017), we'll focus on it primarily. The game consists of two decks: a prompt deck with fill-in-the-blank sentences or phrases (e.g., "TSA guidelines now prohibit ___ on airplanes"), and a response deck with words or phrases to fill in those blanks (e.g., "Goblins", "Poor people", "BATMAN!!!"). All players are dealt a certain number of response cards, and then each player takes turns acting as judge. The judge draws a prompt card from the deck, and the other players must pick the card (or cards) from their hand they feel best fits the sentence or phrase printed on it, or is the funniest. The judge evaluates the submissions and awards a point to the winner, at which point everyone draws a card or cards to replace those played and the role of judge moves on to the next player. The game mechanic is simple, and the players' experience therefore depends heavily on both the contents of the deck and the other people playing the game.

The most obvious benefit of a CAH app over the physical game is in the actual size of the deck. Both CAH and Apples to Apples decks are huge; the most recent edition of the CAH base deck includes 600 cards. Even so, Cards Against Humanity LLC comes out with multiple expansion packs every year, since card combinations you've seen many

times before grow stale rather quickly (Cards Against Humanity, n.d.) With a physical deck, organizing a game of CAH anywhere but the owner's home requires advance planning – and, if the combined deck is large enough, a car. With a mobile app, one could carry a functionally infinite deck in one's pocket.

A less obvious, but perhaps more significant, problem an app can solve is that of the deck's contents. Cards Against Humanity has released several subject-specific expansion packs, but fan-created packs like Cards Against Downtime have sprung up in areas where Cards Against Humanity has failed to meet the demand. Producing a digital deck and integrating it into a game is vastly easier than printing a physical deck that matches existing cards, and it makes it far easier to introduce a temporary expansion pack for one game, and replace it with a different expansion pack for another game, without having to sort through the entire deck after each game.

As discussed in the next chapter, several mobile and web apps have been created to provide just this service, with games generally hosted by a centralized server to which clients connect.

Almost all past digital implementations of Cards Against Humanity have focused on enabling games between remote players, with all the technical overhead that entails. But all of the advantages above remain for in-person games as well. Cardcast, discussed in the next chapter, has in fact developed an app for in-person games, but requires a shared network connection and a Google Chromecast device to serve as a hub, and the reviews show that there is a demand for an in-person app without special hardware requirements. (Cardcast - Apps on Google Play, n.d.) And, as the next chapter will show,

an app which does not require a dedicated, centralized server has economic and stability benefits.

Chapter II.

Prior Work

In an ideal world, this section would include a discussion of existing open-source or community-developed mobile games using peer-to-peer local networks, and the many different strategies they employed. Unfortunately, while projects like that may exist, I couldn't find any. My hope is that this project may someday be a useful reference to someone who wants to build one.

When looking for electronic Cards Against Humanity implementations, on the other hand, we have a rich field to choose from. The most relevant for our purposes are the three major web clones: Cardcast, Cards Against Originality, and Pretend You're Xyzzy.

Cardcast

The Cardcast app enables in-person gameplay via the use of a Chromecast device or Android TV. It launched in 2014 and allows for custom deck creation, mixing multiple decks, and saved special-purpose decks. It's available as an app with a clean, uncomplicated interface on both iOS and Android devices. In fact, it already fits the specification of this project in every respect, except that it relies on a Google Chromecast device and an associated television screen to act as a hub between players. While this decision limits their audience, the Cardcast developers have publicly stated that they are not interested in removing the requirement, both for UX design reasons and because they wound down development "years ago" and "have long had very little interest in growing

this project" (Cardcast, 2017). While distribution numbers are not available, the Google Play App Store shows 2,645 reviews, many of them simply complaints about the Chromecast requirement. (Cardcast - Apps on Google Play, n.d.)

The Cardcast website is also a hub for creating and sharing custom decks; as of 2017, they had over million custom cards in their database. Most importantly for our purposes, Cardcast also makes both the official CAH decks and public user-created decks available via a free API, allowing this project to support custom decks without the need to support in-app deck creation or editing. This project is profoundly indebted to the creators of Cardcast, and might be best described as a Cardcast client.

At first glance, Cardcast's continued success might argue against the need for a low-margin distributed system. However, while Cardcast hosts many thousands of decks, they provide no gameplay mechanism whatsoever via their website. All traffic associated with in-game communications is offloaded onto the Chromecast device. Though Cardcast generates revenue by selling printed versions of the cards users create, the costs of hosting only decks are apparently low enough that the developers are currently donating all proceeds to charity. (Cardcast, 2017)

Cards Against Originality

Information about Cards Against Originality is scarce, since it survives now only in legend. It came online in March of 2015 as a free web app with a slick, mobile-friendly interface, and, unlike Cardcast, the site hosted games itself. (Deutsch, 2015) It shut down before the year was out. (Whitfield, 2015). The url now leads to a static page which reads, in part:

> Three million people spent a combined 47 years playing CAO since March 2015. I feel a deep happiness knowing that people connected and laughed because of something I made as a fan - even though I had nothing to do with the actual CAH game.
>
> Unfortunately, I can't afford the hosting fees any more. (Whitfield, 2015)

While Whitfield's comment about hosting fees speaks for itself, it's also worth noting that, according to Whitfield, this site gained three million users in less than eight months. While we can't know how accurate those numbers are, or how many of those users overlapped with Cardcast's user base, players were clearly eager for a mobile-friendly web app which was truly mobile, rather than requiring a television with a specialty attachment.

Pretend You're Xyzzy

Pretend You're Xyzzy is perhaps the most prominent of the remaining CAH clones. It integrates with Cardcast, in that it fetches user-requested decks using the previously mentioned API, though the Cardcast team is not involved in its development. Like Cards Against Originality, Pretend You're Xyzzy hosts its own games, but unlike Cards Against Originality, which was overwhelmed by demand, it has survived for over three years.

It may, however, have survived due to its user interface (see fig. 1 on the next page). Pretend You're Xyzzy is unapologetically designed to be played on a full-size monitor. On a computer screen the interface is slightly confusing, but on a mobile device it becomes unusable. In my extremely informal early user tests, no one was willing to play more than one hand on a mobile device. Even after transitioning to laptops, everyone

preferred Cardcast — even to the point of preferring home-printed DIY Cardcast cards. Pretend You're Xyzzy also lacks some features, like a discard option, that users agreed were important.

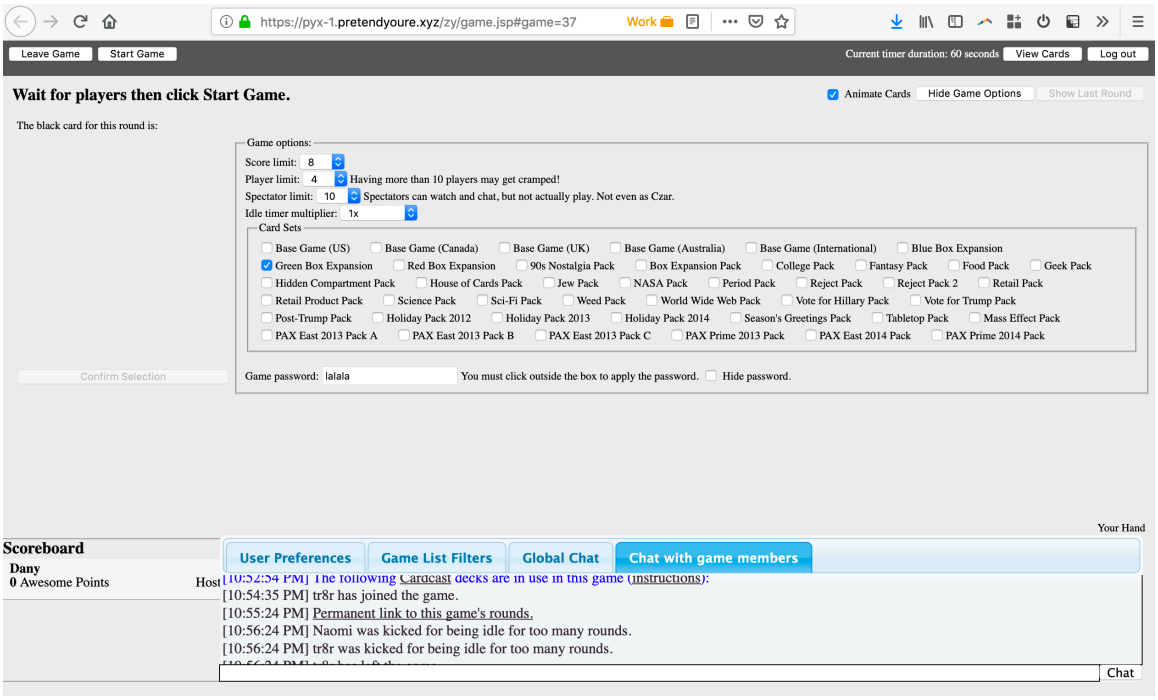

Figure 1. Pretend You're Xyzzy game creation page

None of the elements on the page above resize to accommodate different window sizes or display resolutions, making the site difficult to use on anything but a full desktop monitor. (Pretend You're Xyzzy, 2018)

While it's impossible to compare their unknown user base with a hypothetical surviving mobile-friendly web app, it seems likely that Pretend You're Xyzzy has survived by limiting its audience to a minority of players who enjoy remote games and are willing to stay tethered to a full-size monitor to play them. Even so, Pretend You're Xyzzy runs on three servers for load-balancing purposes, and those servers often reach their posted capacity. (Pretend You're XYZ, 2018)

As previously discussed, CAH operates under a Creative Commons license that allows third parties to share and modify, but not profit from, the original work. History shows that the centralized architecture of web apps like Cards Against Originality and Pretend You're Xyzzy is not sustainable at high traffic volumes (or without significant financial resources available to sink into maintenance costs). A successful and sustainable mobile app would need to offload hosting and traffic to the end users' devices, much as Cardcast has done, but using only the hardware capabilities most mobile phones already possess.

Chapter III.

Requirements

In order to be a worthwhile upgrade from the physical game, this project needed, at a bare minimum, to:

- Provide at least the same game functionality as Pretend You're Xyzzy (text chat aside).
- Run on, and support seamless connections between, Android and iOS devices.
- Allow players to download, create, and remix decks, either by incorporating a deck management subsystem or integrating with a service like Cardcast that provides those services.
- Provide an intuitive interface which will be easy to use on a small-screened mobile device.

In order for it to be economically sustainable, it had to do all that on the players' devices and using peer-to-peer transmissions, without relying on a centralized server.

Additionally, after some initial test games with some potential users (i.e., the people I usually play Cards Against Humanity with), I came away with these "reach goal" feature requests:

- An option to "let the deck play," by adding one or more random response cards to each round.
- A mechanism for discarding cards from one's hand, and for rejecting prompt cards that are confusing or inappropriate for that particular game.

- A mechanism to change player turn order — for instance, so that turns pass in order clockwise around the room. This is obviously not a concern for online games, which generally assign turn order randomly or by the order in which players join the game.
- An undo button, so that players who change their mind after submitting a card can substitute a new one.
- The ability to reorder cards within one's hand.
- A mechanism to gracefully handle a temporarily missing or disconnected player (in that specific test instance, allowing the game to continue with the remaining players if one of the players has to go pick up takeout). Pretend You're Xyzzy handles this dilemma with a hard idle timeout, which my test group didn't love.

Chapter IV.

Design and Technology Choices

The two key decisions in planning this project were the choice of networking protocol and the framework for developing the app frontend. All other design choices flowed from these.

Networking

There are a wide range of ad-hoc peer-to-peer networking protocols available, but far fewer that can interoperate with both Android and iOS devices. Both Android and Apple have gaming platforms of their own, but neither are cross-compatible with the other's OS. (Apple Inc., 2018; Google, 2018) Apple's otherwise promising Multipeer Connectivity framework is likewise compatible only with other Apple devices, whereas Android's implementation of Wi-Fi Direct works only on Android (eskimo, 2018).

The open-source Bonjour protocol (also known as Zeroconf or, more rarely, Rendezvous), which is the basis of Apple's Multipeer Connectivity, initially seemed ideal. Android has two frameworks compatible with Bonjour, jmDNS and Near Service Discovery, (Van Hoff et al, 2019; Google, n.d.-b), although Near Service Discovery, or NSD, lacked the ability to connect to iOS devices running the same protocol as recently as 2013 (Gwizdz, 2013). While this deficiency in NSD has since been remedied, anecdotes nevertheless indicate that both jmDNS and NSD are unreliable and slow (e.g., (Hacker News, 2016)). More importantly, Bonjour is a protocol for connecting servers on an existing IP network over an existing link layer. In the absence of full root access

and shared proprietary hardware (see Multipeer Connectivity), it would require either that players already be connected to (and reachable on) an existing local network, or that they create a mobile 'hotspot' wireless network, which would involve system-level configuration changes I don't see anyone making for the sake of a party game. The third alternative would be something like IP over Bluetooth, but implementing that successfully would be a thesis unto itself.

Google Nearby Messages was therefore the best available option. In strict terms, Nearby Messages provides not so much a peer-to-peer network as a proximity-based network. Devices broadcast message announcements using Bluetooth and ultrasonic or near-ultrasonic audio, while transmitting the messages themselves to Google's servers via an internet connection. Other devices see those broadcasts and then fetch the messages, again from Google's servers. (Google, 2018)

While Nearby Messages was the best option, there are limitations inherent in that model, and those limitations extend to this project. For one, relying on a third party (Google's servers) taints the ideological purity of this project as a truly peer-to-peer local networking app, even if a free service doesn't add to the marginal cost of operation. For another, Google has imposed a hard 100k per-message and an eight-million-messages-per-day limit. This project will likely never hit that limit under any circumstances, but that's still not quite infinitely scalable.

Finally, and most importantly, this tripod model introduces message latency at multiple points. To pass a message from one device to another, a device must upload that message to a remote server, announce the new message locally, wait for the second device to register that announcement, and then wait for that second device to fetch the

message remotely. Based on my user tests and personal experience, the result is a low-bandwidth connection with high (and highly variable) latency.

Cross-Platform App Development

Having no previous Android development experience, I had planned to use Flutter, Google's newly introduced cross-platform app development framework, in the hopes that it would provide an easier road to incorporating Nearby Messages. However, despite the fact that both projects were produced by the same company, no Flutter plugin for Nearby existed at the time, nor did one became available until March of 2019. (Cross, 2019) In the meantime, my advisor suggested Xamarin.Forms, a more established cross-platform framework with a far more robust mechanism for incorporating platform-specific code. With Flutter's Nearby Messages support lagging behind Xamarin's, and with Flutter still in Beta at the time (Sneath, 2018), there was no good reason not to switch to a more mature product.

Chapter IV.

Implementation

The Manatee app consists, loosely, of four subsystems: the frontend UI; the DeckLibrary singleton and its associated classes; the PostOffice system, which manages communications between devices; and the GameController, which manages the other three and handles transitions between game states.

Backend

The GameController's job is to act as a go-between between the UI and the PostOffice and DeckLibrary class. While I haven't been especially careful about maintaining formal separation between the subsystems, most traffic between the user and one of the backend systems should go through the GameController. The GameController, for example, is the class that generates outgoing messages, acts upon incoming messages from the PostOffice, or fetches information from the DeckLibrary.

The GameController singleton also maintains a Game singleton member that represents the state of — no surprise here — the current game. There's no structural reason for the two to be separated, and in fact, the two classes were initially born out of an attempt to troubleshoot what I thought, at the time, was a dependency issue. However, I've maintained the separation because I hope, at some distant point in the future, to allow users to save and resume games, and on that day having a preexisting self-contained representation of the game state that doesn't include any references to

unserializable classes will be useful. In the meantime, the cost in coding time of maintaining the Game class is virtually nil.

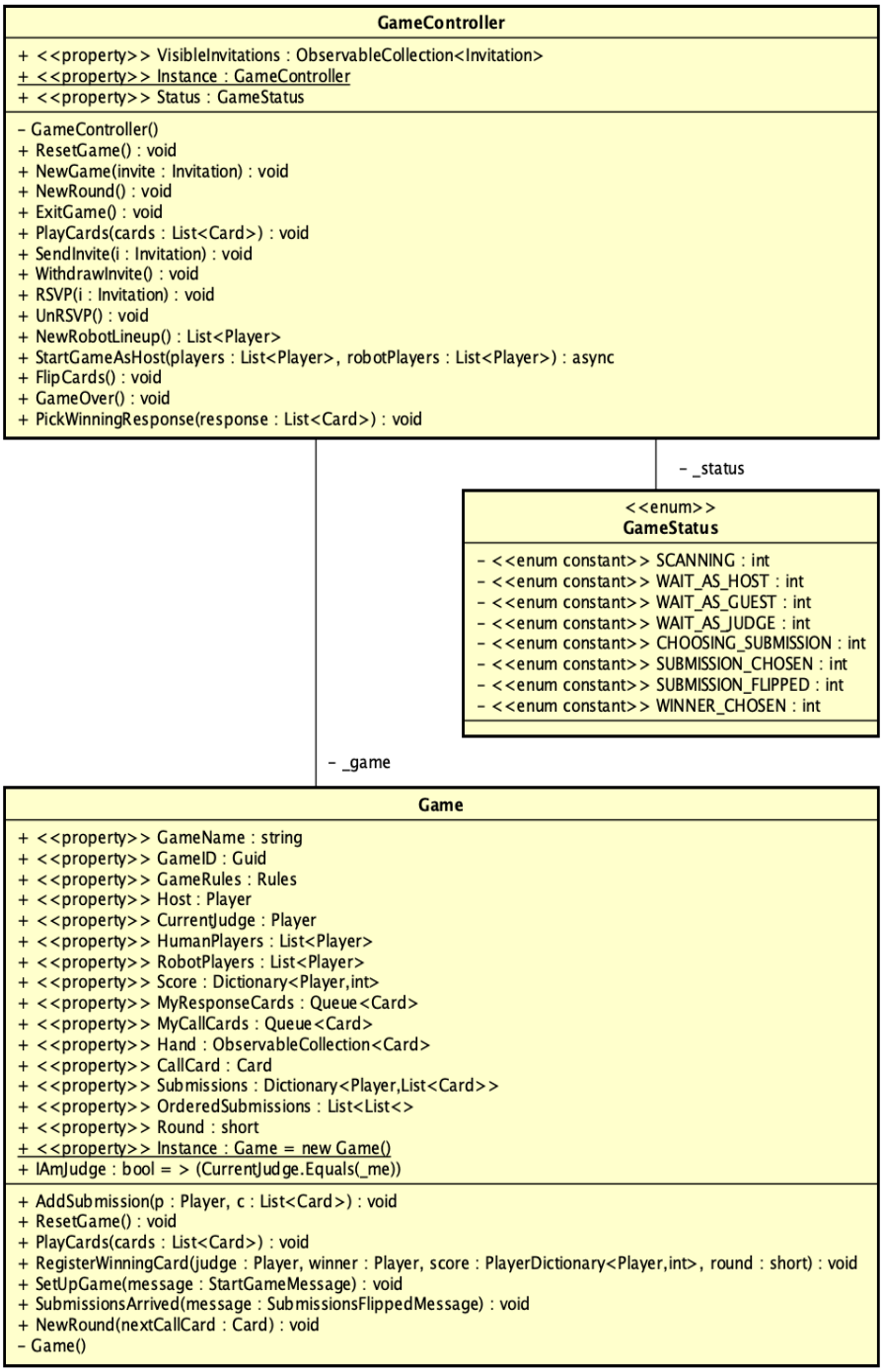

Figure 2. The GameController and Game singleton classes

Deck Library

Manatee uses a DeckLibrary singleton class, with the following representations for cards and decks:

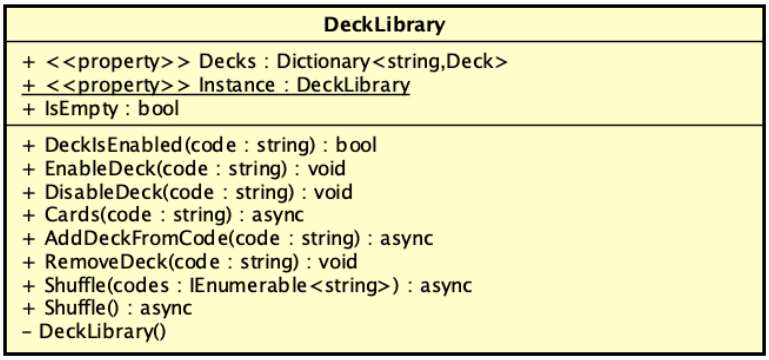

Figure 3. DeckLibrary Class Diagram

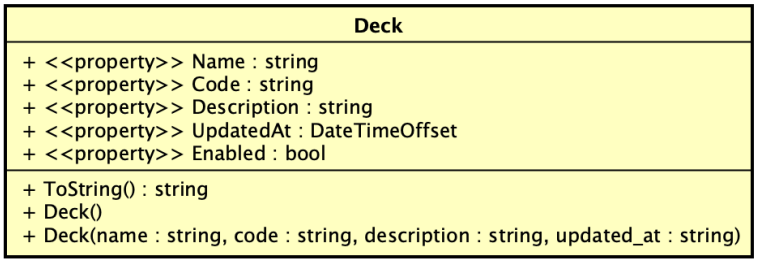

Figure 4. Deck Class Diagram

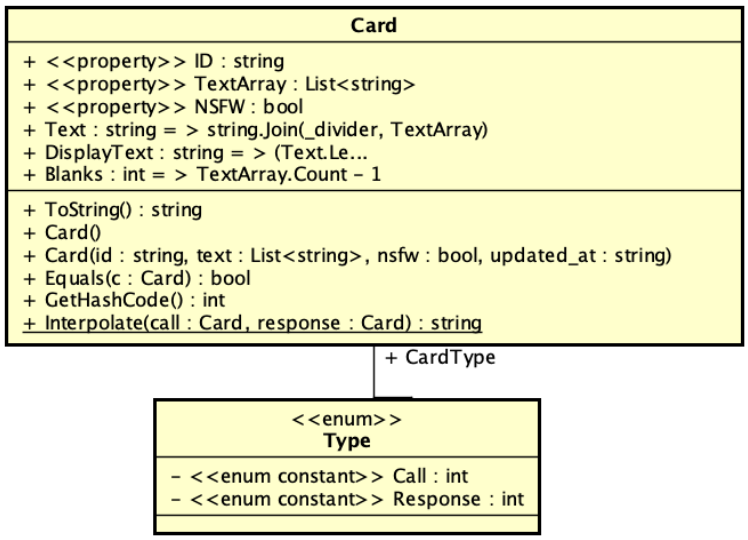

Figure 5. Card Class Diagram

Manatee ships with no decks of its own, since Cards Against Humanity owns the most popular decks and the terms of their license do not allow their decks to be distributed via an app store. Instead, when first launched, Manatee automatically fetches JSON representations of a pre-selected set of decks from Cardcast using Cardcast's web API, then parses and converts them into the above formats.

In the early stages of this project, when it seemed possible that it might include in-app deck editing, the Deck and Card classes contained reciprocal references and were stored in a Realm database for ease of remixing. The Realm object class, unfortunately, is unserializable, which ultimately meant I had to create secondary serializable representations in order to include deck or card descriptions in messages, with all the infrastructure that implied. The resulting system was ridiculously complex. Since it seemed increasingly unlikely that any user would voluntarily use a phone screen to create a new deck or edit an existing one when a web interface was available, I turned to the

most low-tech DeckLibrary implementation possible: the Xamarin.Forms Application.Properties dictionary.

Application.Properties provides persistent storage, but it cannot store complex objects, only built-in types like int or string. It can't even store lists. Any object stored in Properties must first be serialized into a compatible form, generally an XML or JSON string representation. As a result, if a large collection is stored in Properties, accessing one of its members requires reading the entire string from disk, and modifying one of its members means re-writing the entire string. That's less of a concern if we're outsourcing deck remixing to Cardcast, but frequently accessed or modified attributes (like the name and size of a deck, when you don't need the cards yet) need to be stored separately for performance reasons.

Rather than create more dictionary objects to represent relationships or frequently changing values, I instead added those values to the dictionary directly and used the dictionary keys to represent their relationships to each other. The DeckLibrary produces a full list of decks when needed by checking Application.Properties for keys with an appropriate suffix. It's kludgy, but it's lightweight and it doesn't require any additional JSON parsing for basic queries. Parsing a full deck does require some I/O and processing work on the part of the device, but that happens at the start of the game when players are generally willing to accept a 'set up' period.

An example deck representation is given in the table 1. Some unused elements have been removed to make the output easier to read, but readers will still note that the JSON still contains elements not present in the deck and card objects used in actual games (author, external copyright, et cetera). While Cardcast provides some information

about these decks that the app doesn't use, incorporating features that might use that information — maybe an update date, to make sure you have the latest version of your friend's deck? — would be much easier if it were already present in persistent storage, without having to reformat all existing decks, and the cost of retaining that data is minimal.

Table 1. Sample representation of a deck as Application.Properties entries

```
"BQWH4": {
      "name": "Esperanto for Humanity",
      "code": "BQWH4",
      "description":
"This is an English set with (generally positive) themes about Esperanto.\n I'm making it up as I
go along, so it won't be very polished yet. 4 responses per call.\n Maybe I'll make an Esperanto-
language version later on.",
      "created_at": "2018-02-21T04:24:11+00:00",
      "updated_at": "2018-02-22T01:18:49+00:00",
      "external_copyright": false,
      "call_count": "17",
      "response_count": "68",
      "author": { "id": "c6e6ab89-b2d7-40d8-a284-fcd09084e82d",
                  "username": "Scrambled_Egg" }
    },
  "BQWH4-enabled": true,
  "BQWH4-cards": {
    "calls": [
      { "id": "07ac2307-f526-4115-b04e-54401964d0fd",
        "text": [ "It's not Spanish, it's ", "." ],
        "created_at": "2018-02-21T05:24:57+00:00" },
...
      { "id": "1d9505a3-79df-44f6-b91c-8fdc6e62fadb",
        "text": [ "", " should be banned from all Esperanto meetings." ],
        "created_at": "2018-02-21T05:32:57+00:00" }
      ],
    "responses": [
      { "id": "004a8078-08f4-4af8-b0f5-236480aa3d10",
        "text": [ "a neutral common language" ],
        "created_at": "2018-02-21T04:54:00+00:00" },
...
      { "id": "03a97704-df9f-47ab-bf45-89577552300f",
        "text": [ "unnecessary diacritics" ],
        "created_at": "2018-02-21T04:50:58+00:00" }
    ]
}
```

Messaging

When I started this project, a C# wrapper for Nearby Messages on Android was easily available via Nuget. The same was not true of iOS. There is a demonstration project including a Nearby Library for iOS hosted on GitHub (Soto, 2016), but the library is in binary format with no included headers, and produced an immediate segmentation fault when accessed from the sample program on my test devices. I suspect the errors were due to operating system changes since the project's publication, but without access to any source files I was unable to test that theory or work around the issue. I was, however, eventually able to produce a binding project of my own using Objective Sharpie (Microsoft, 2017) and the binaries and header files from Google's Nearby Messages CocoaPod. I won't bore the reader with the many false starts and dead ends involved, but I am posting the full source code for that binding in the GitHub repository for this thesis, so that future generations need not suffer as I suffered.

With a working C# wrapper for both iOS and Android, I could use Xamarin.Forms' DependencyService to produce a PostOffice class that would provide a consistent API across platforms. A diagram of the resulting subsystem is provided in fig. 4.

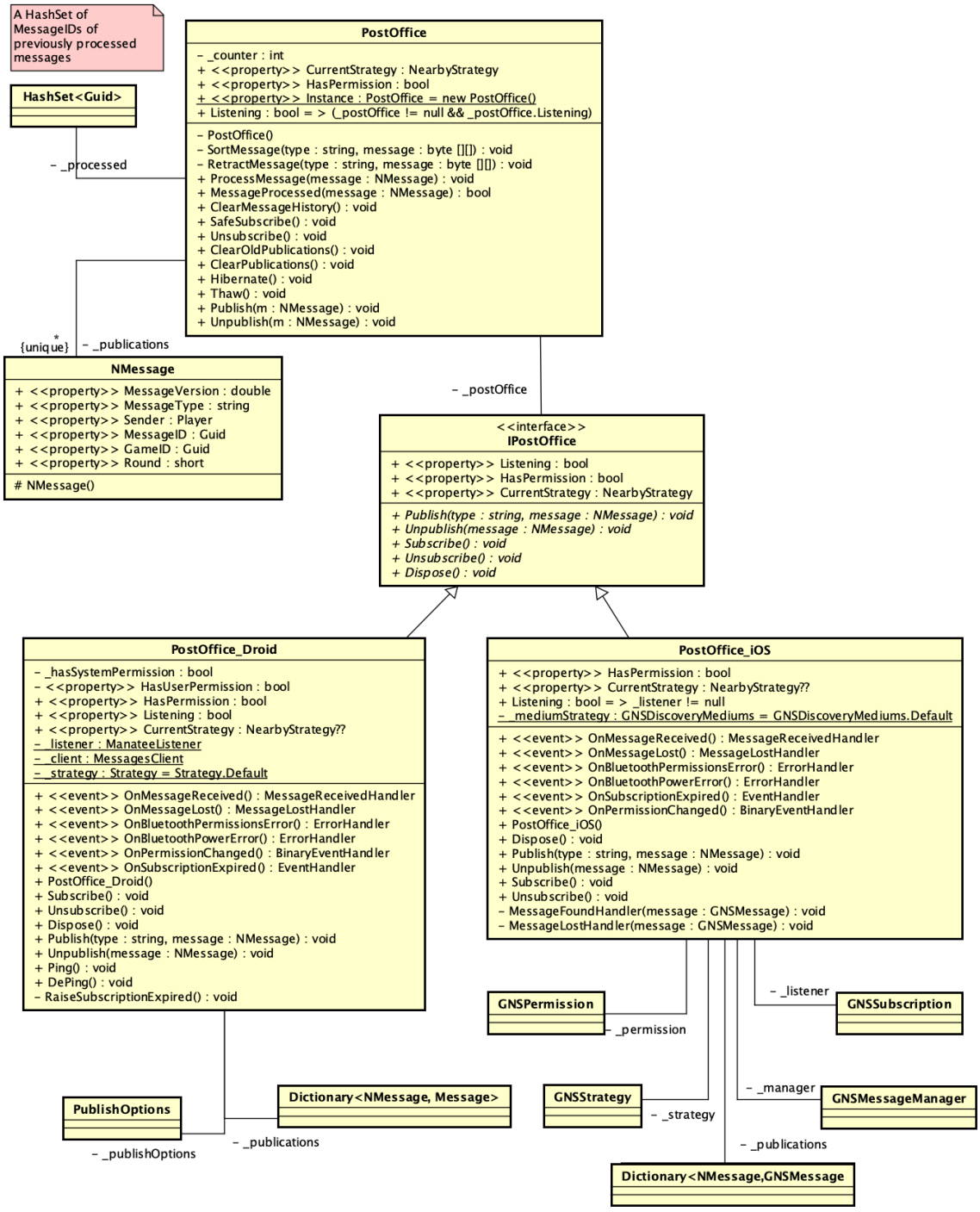

Figure 6. A UML class diagram of the PostOffice subsystem

The IPostOffice interface defines an API for interacting with the different device-specific implementations of Nearby Messages. The PostOffice singleton acts as a wrapper to the appropriate device-specific IPostOffice implementation, chosen by Xamarin.Forms' DependencyService at compile time. It also executes non-platform-specific tasks like deserializing and sorting incoming messages, and high-level management of publications and subscriptions. Since both Android's Message and iOS' native message constructors accept a binary payload (as byte arrays and NSData, respectively) for message contents, I could simply use the standard .Net BinaryFormatter to serialize an arbitrary object (in practice NMessage, above, and its children) for publication and deserialize it when it landed on another device, without any further parsing or formatting. The message format itself was by far the most convenient aspect of Nearby Messages.

All Nearby Messages publications are broadcast messages. Once published, a message is broadcast continuously until the Unpublish() method is called on the message (Android) or the publication object is released (iOS). Nearby provides no assurance of reliable transmission, in-order transmission, or consistent transmission times. Additionally, Nearby's functionality when operating in the background is too limited and too costly to be worthwhile for this project, meaning that in practice messages can be sent and received only when the app is focussed on screen. (citation) A publisher has no reason to think their message has been received until they get an explicit notification otherwise. The built-in latency discussed in Chapter III makes round-trip communication extremely slow, announcing a response messages requires as many resources from the sender as the original message, and if that weren't enough to make TCP-style

acknowledgement packets unattractive, there's Google's hard limit on daily messages. Requiring a long-lived ACK message from each receiver of a message would bump the total number of publications from O(nk) to $O(n^2k)$, with the potential device and network performance costs that implies. The economical route is to continue to publish a given message until it can no longer possibly be useful; that is, until the game has moved on to the next stage.

Due to the lag times involved and the need for responsiveness, it's not feasible for one device to request information from another. In general, if one player has a higher-latency internet connection, or has their screen off at the time the winning card is announced, the other players' devices will continue until the lagging player's input is absolutely needed. Ideally, whenever they're waiting, players should feel that they are waiting for each other, not the app. The latter is frustrating, but the first is an accepted (and even integral) part of any similar party game.

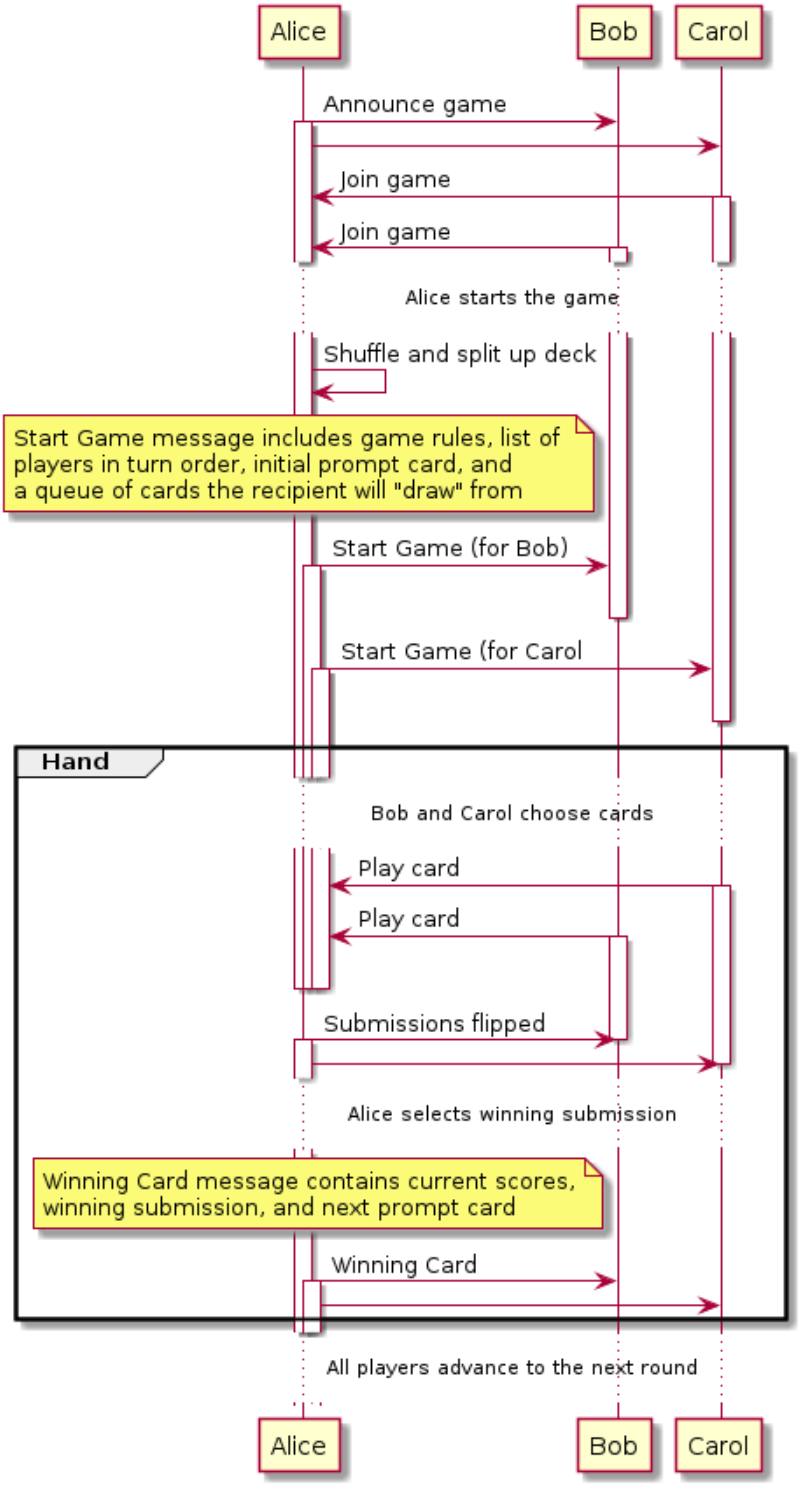

Figure 7. Typical message exchange between players

Bars indicate the lifetime of a given publication. All publications are broadcast, but arrows indicate a publication is both seen and processed; for example, Carol sees the Start Game message addressed to Bob but discards it.

With the glaring exception of the handshake sequence at the beginning of the game (see fig. 5), the app never requests confirmation that other players have seen its messages; it will continue as far as possible until it requires human input. For example, when a player receives a Winning Card message, the app displays the message, transitions to a new round, displays the new prompt card, 'draws' from its saved card queue to replace cards played in the previous round, prompts the user for a selection, and publishes a message declaring that selection. For all that the app knows, at the time it broadcasts the user's choice, the player judging the current round has not even received the original Winning Card message yet, and therefore may not even know what this round's prompt card is yet. All the app knows is that it has the information it needs to progress to that point, and waiting would only cause delays for its user.

In aid of this behavior, the app tries to distribute as much information as possible as quickly as possible. For example, the 'Start Game' message addressed to a given player contains all the response cards they will draw from over the course of the game, and all the prompt cards that will be announced on their turn. Distributing the deck among all players eliminates the host as a chokepoint, splitting the deck reduces the size of the 'Start Game' messages the host needs to publish in the case of large decks and long-running games, and having players 'draw' from their own locally saved decks reduces the maximum size of the messages that need to be transmitted in-game. Similarly, the 'Winning Card' message (as shown in fig. 5) includes not only the winning card from this round, but the prompt card for the next round (rather than waiting for the

next judge to receive the Winning Card message and publish a ‘Drawn Card’ announcement in response).

Message Publication Strategies

As previously discussed, Nearby Messages uses both Bluetooth LE and audio tones near the edge of the human hearing range to announce messages. The library does allow a ‘BLE only’ mode, and since Bluetooth is more familiar to users and also easier, from a permissions standpoint, to access — iOS doesn’t even require user permission to use Bluetooth when an app is foregrounded — I had hoped to implement this project as Bluetooth-only. Unfortunately, after I enabled audio transmissions as a means to use an extremely old smartphone as an additional test device, I found that communications between my two existing Bluetooth-compatible Android and iOS devices became noticeably more responsive, with fewer extremely long message delays. Since Bluetooth LE has more than enough bandwidth for this application, I suspected that result had more to do with doubling the number of message announcements than with anything inherent to either transmission protocol, but since some other players also noted the change in early test games, I made plans to include audio transmissions in the final product. With message transmission times for two phones on a cellular network ranging from half a second to over ten seconds, it seemed important to remove any source of additional latency that I could.

While requiring microphone access would almost certainly lose the app potential users, it seemed better to offer a more responsive app to a smaller audience than a less responsive one to a larger audience. Unfortunately it’s impossible to extrapolate from my

test players' feedback how many people will refuse to use the app because of microphone access. My testers are, by definition, all people who trusted me enough to switch off their safety settings and install a strange and unvetted app on their devices, and they were unlikely to balk at allowing me microphone privileges.

User Interface

I used Xamarin.Forms, with the invaluable help of LiveXAML, to produce a cross-platform GUI based on my initial sketches. I modified the resulting program based on ongoing test games, first by hosting games in which I manually downloaded the app onto my testers' phones, and later also by distributing the app via Microsoft's App Center distribution platform and soliciting feedback from people who had played games on their own. The following is a walkthrough of the final user interface, along with a discussion of some of the user responses that led to its final form.

Welcome Screen

The first page the user encounters when opening the app is a welcome screen which asks the user to select a screen name. Based on player feedback, this was the minimum user input I could demand up front. I had expected that players wouldn't stop to visit Settings to set a screen name before starting a game. I had not expected that they would also consistently choose to click through an initial name prompt without entering anything if they were given the option to do so. In an early attempt, I gave users a prompt when entering the game but also allowed them to skip the input and use a default value; the end result was a series of games in which every player was named "Three

Raccoons In A Trenchcoat" or "Player #8232", and nobody could tell which human corresponded to which username. Heading that confusion off at the pass was ultimately less frustrating for my test users, and once the requirement was in place and well-communicated new users accepted it as normal.

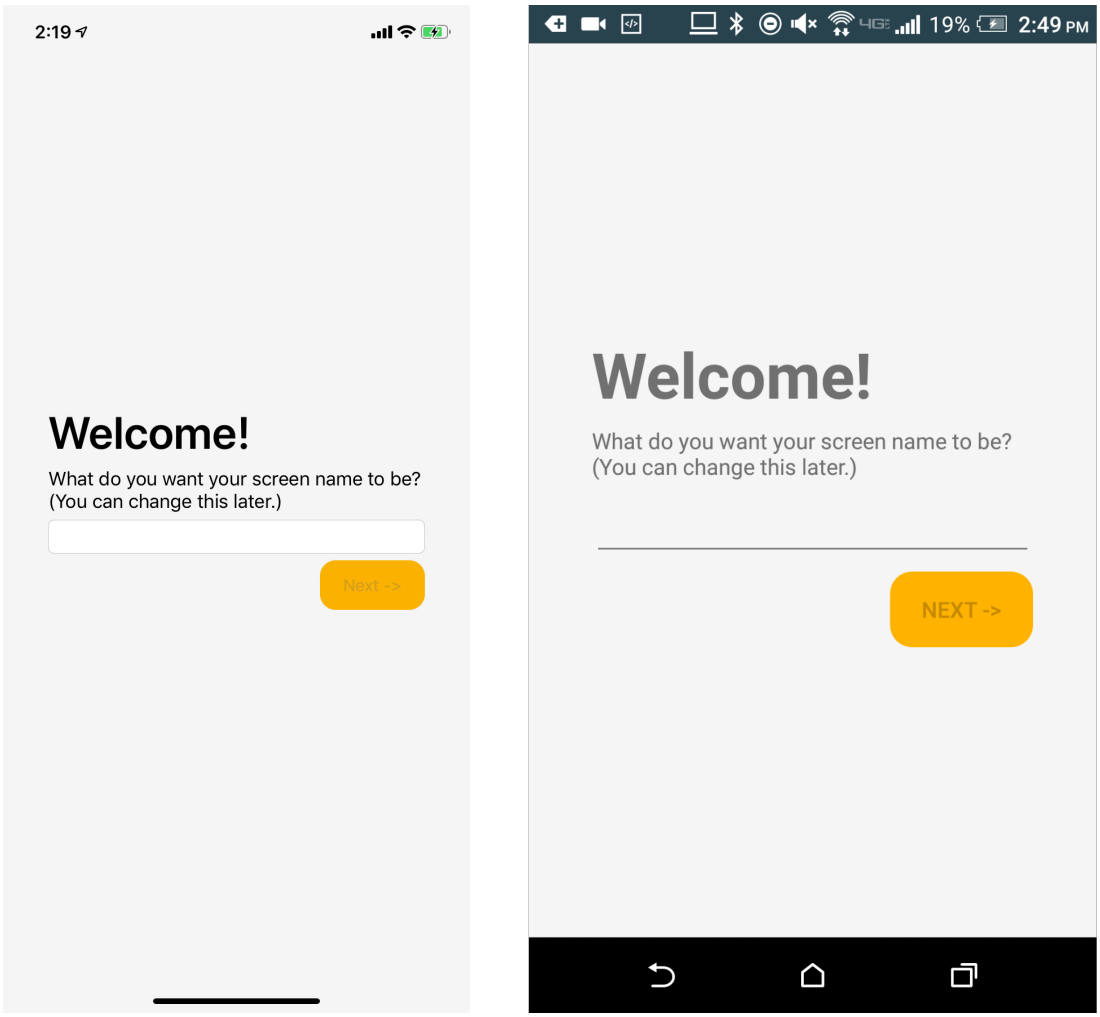

Figure 8. Welcome Page on iOS and Android

The orange buttons used throughout the app were also chosen partially because of user response to this page. After my first user trials, to prevent players from clicking

through the screen prompt and then becoming frustrated with six identical screen names, I configured the button below the text input to remain disabled until the user had entered a screen name. On iOS, the default button appearance is plain blue text on a transparent background, and the appearance for a disabled button is plain gray text. Without any other experience with the app to establish a visual language, several beta testers assumed that gray buttons were, in the words of one tester, a "stylistic choice" and not a semantic one, and spent several seconds trying to press the inactive button without filling a name in first. Black vs. grey on a bright background communicated the button's enabled or disabled state much more effectively, and frustrated people less.

Permissions

As previously discussed, I've chosen to use audio communications, which requires microphone permissions from the user. That was not my initial plan. Even after resigning myself to using ultrasonic audio, I had planned to offer users a choice between a higher-performance dual broadcast option and a less reliable, but more battery- and privacy-friendly, Bluetooth-only option. I had produced a fairly verbose dialog sequence (fig. 7), which was well-received by one of my test groups, and spent a fair amount of time writing and debugging the code to allow users to switch modes back and forth while hiding the platform-specific implementations of different broadcast strategies from my shared code project.

My advisor, however, convinced me that my test players were almost certainly outliers, and that if I wanted to distribute the app I would need to appeal to more than my test group. I don't have broader data to compare, but intuitively, this makes sense; the

people I know and work with are, for example, far more likely than the average human to have two-factor authentication on their online accounts or little EFF stickers covering the front-facing cameras on their phones. It seems reasonable to extrapolate that they would also be more likely than the average user to want detailed background information on and control of their personal devices. It remains to be seen whether the average Cards Against Humanity or Cardcast player is more 'average' or more like the CAH players I know personally, but until I have feedback from wider and more systematic user testing I've chosen to assume that my small circle is not representative.

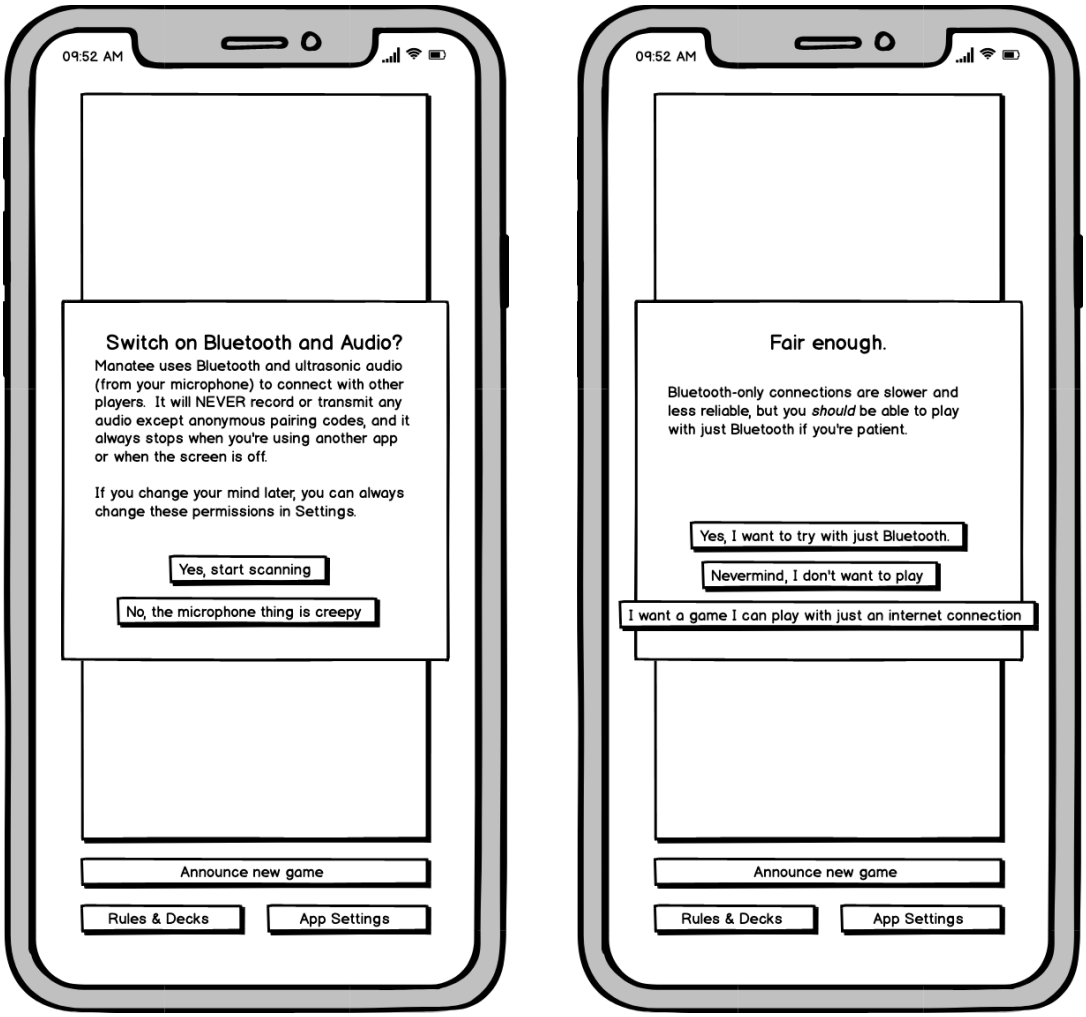

Figure 9. Early mockup permissions sequence

Instead I've gone for a simpler permissions sequence with only one selectable publication strategy, with the working hypothesis that most people are not interested in a detailed discussion of how Google Nearby works, and that furthermore most people are not likely to go back and tweak the settings if the first network option they choose proves not to be performant.

The new permissions sequence isn't necessarily simple, but it is at least simpler.

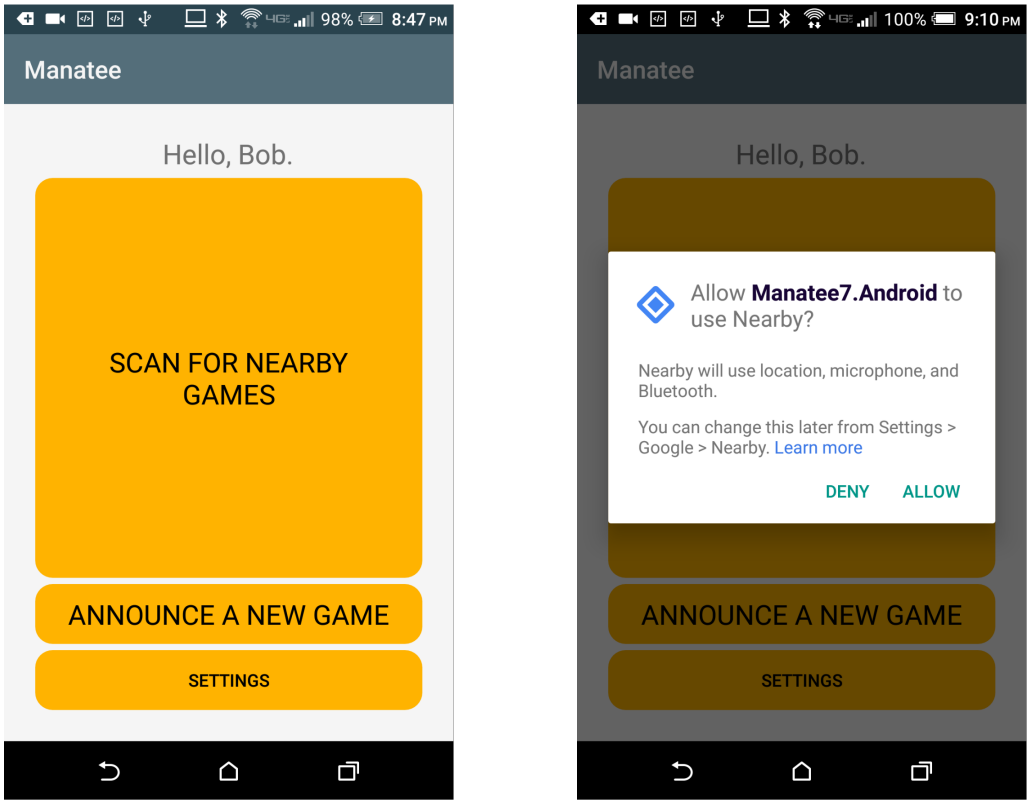

Figure 10. Android Nearby permissions dialog sequence

The left screenshot shows the landing page for the app; the right screenshot shows the popup the system presents once a user selects either 'Start Scanning' or 'Announce A New Game.' Since Android users are more likely to be familiar with Nearby, there is no pre-flight screen.

The current school of thought on permissions UX emphasizes providing context for any request; for example, asking for location information only after a user has asked for directions. (Mulligan, 2014; Babich, 2016) Since Manatee is entirely built around the ability to communicate with other devices, there isn't an obvious way to 'showcase the value proposition' of the app before requesting permission. To try to mitigate this issue, instead of initiating a connection automatically, I've presented users with a large 'start scanning' button once they enter their screen names. Hopefully, this makes the connection between the microphone permissions request and the app's central purpose quicker to grasp, and gives the users more of a sense of control.

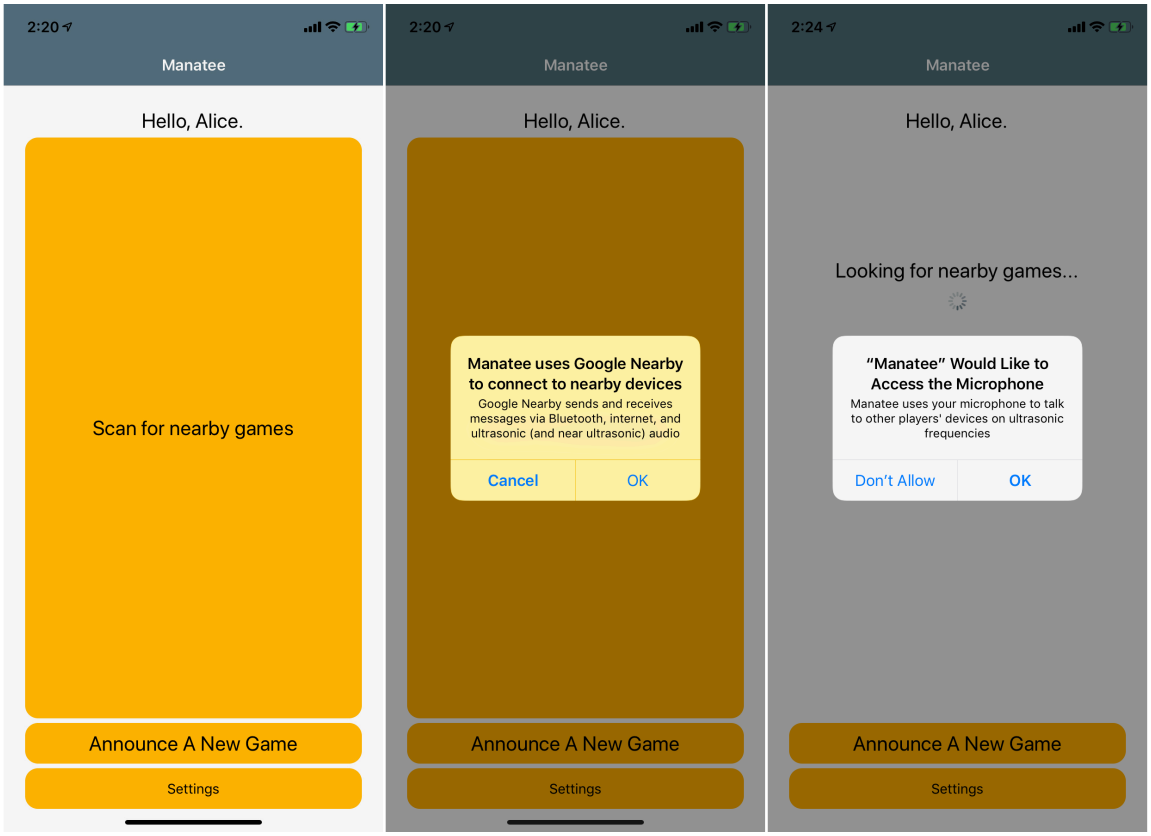

Figure 11. iOS pre-flighting permission sequence

The center screenshot shows the 'pre-flight' dialog shown by the app, which replaces the Nearby default dialog. In the right screenshot, the user has selected 'OK', the app has requested microphone permissions (by subscribing to audio and Bluetooth messages), and the operating system is asking the user to approve the request.

Since iOS only allows an app to ask for user permissions via system dialog in-app once — if the user decides to grant permission later, they have to do so in system settings — providing a 'primer' dialog also gives the app a way to screen people who initially deny the permissions request, so they can still try again in-app later.

Settings

App settings and deck management can be accessed in the same location.

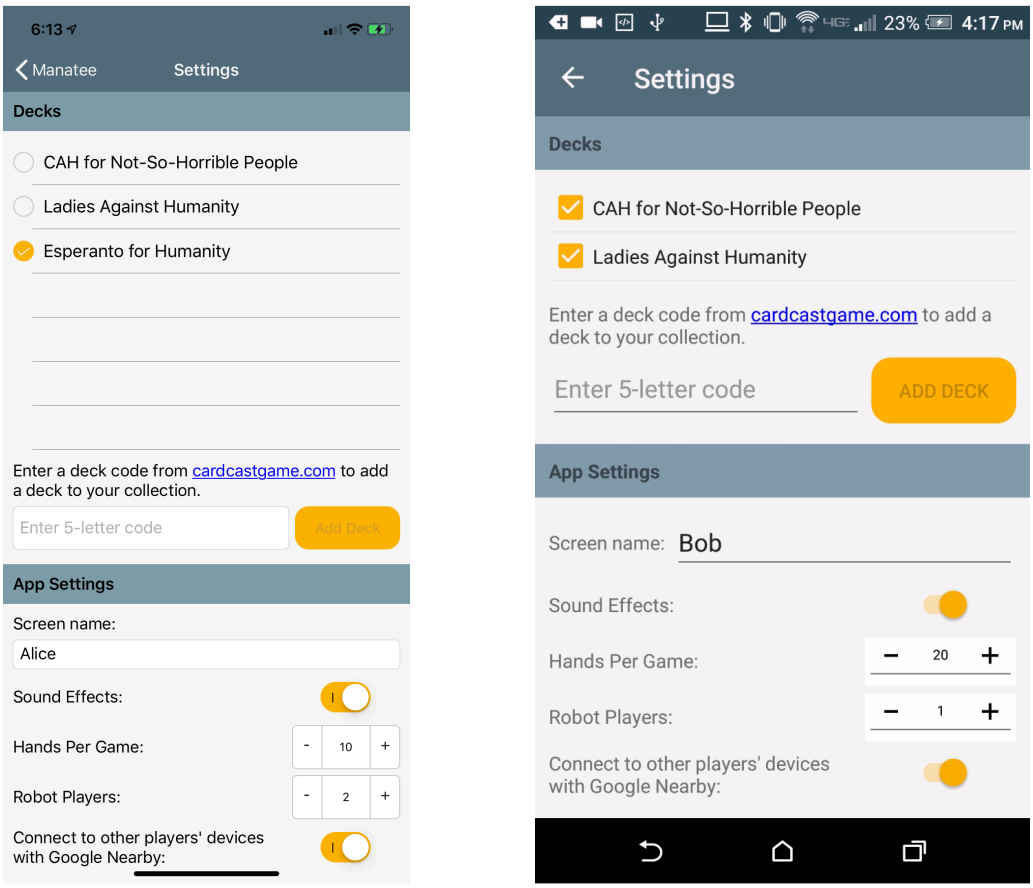

Figure 12. Settings on iOS (left) and Android (right)

The Android screen shows the default app values. On the iOS screen, a new deck has been added and the original decks disabled.

Decks

Early iterations of the app presented users with a long list of popular decks to choose from immediately after the welcome page. In test games, however, I never saw anyone carefully read that list and pick the decks that interested them. At best, they would select whatever was at the top, and come back to add more decks later.

In the current iteration, that list is gone. Instead, when the app starts, it checks persistent storage to see if it has any decks saved. If not, it automatically checks for an

internet connection and then attempts to download two decks: “CAH for Not-So-Horrible People”, a pruned-down version of the classic Cards Against Humanity pack; and “Ladies Against Humanity”, a deck meant to remedy some of the obvious gaps in the official Cards Against Humanity decks. I don’t specifically endorse these decks, and they are certainly not the most popular (that honor goes to the original, uncensored CAH decks), but their purpose is to allow new users to start a game immediately and see what the app has to offer them, and those decks are chosen for having broad appeal and for being relatively inoffensive. Ideally, I’d go with one of the Apples to Apples decks, but those are owned by Mattel, and Mattel, unlike CAH, sues people who use their intellectual property.

Decks can be added and permanently removed, as well as temporarily selected or deselected. If a user hosts a game, the cards for that game will be drawn from whichever decks they have selected at the time. In earlier versions of the game, the decks were presented in the operating system’s standard list form, with tinted/non-tinted cells to indicate selection and deselection. Multiple people found that presentation confusing; either it was not obvious that the decks were selectable or de-selectable, or the act of selecting decks left the user looking for a missing ‘next step’. Once I replaced the standard item selection with checkboxes (again courtesy of SyncFusion’s community license), no new users reported those issues.

App Settings

The sound effect setting can be switched on and off both in settings and in the in-game sidebar. Sound effects include ‘whooshing’ noise when the user sends a message,

such as a card submission or a winning selection, and a jingle played on the device of the winning player. I plan to replace the SimpleAudio Nuget plugin I used to implement the sound effects, since it ignored the mute setting on at least one Android user's device, but nearly every user said they appreciated the audio confirmation when making a selection.

The 'Hands Per Game' setting was added at the request of multiple testers, and accepts a range of 1-100. In the initial implementation, as in the physical card game, the game continued until the deck ran out or the players got bored. Since there is no maximum limit on the size of a digital deck, at least three testers felt the game needed an artificially imposed stopping point.

Game Initiation Sequence

Before starting, players need to agree on the parameters of the game: what are the rules (i.e., how many hands), and what decks are we using? The easiest way to answer those questions is to apply the settings chosen by the host, without supporting in-app negotiation. In our case, when a round-trip message exchange over the network can be slower than two players simply talking to each other, that's also the most user-friendly option. Manatee, therefore, relies on the players to communicate with each other and on the host to choose an acceptable set of rules, and in-app only offers non-hosting players the option to join or not join a game. The next few screens demonstrate that process.

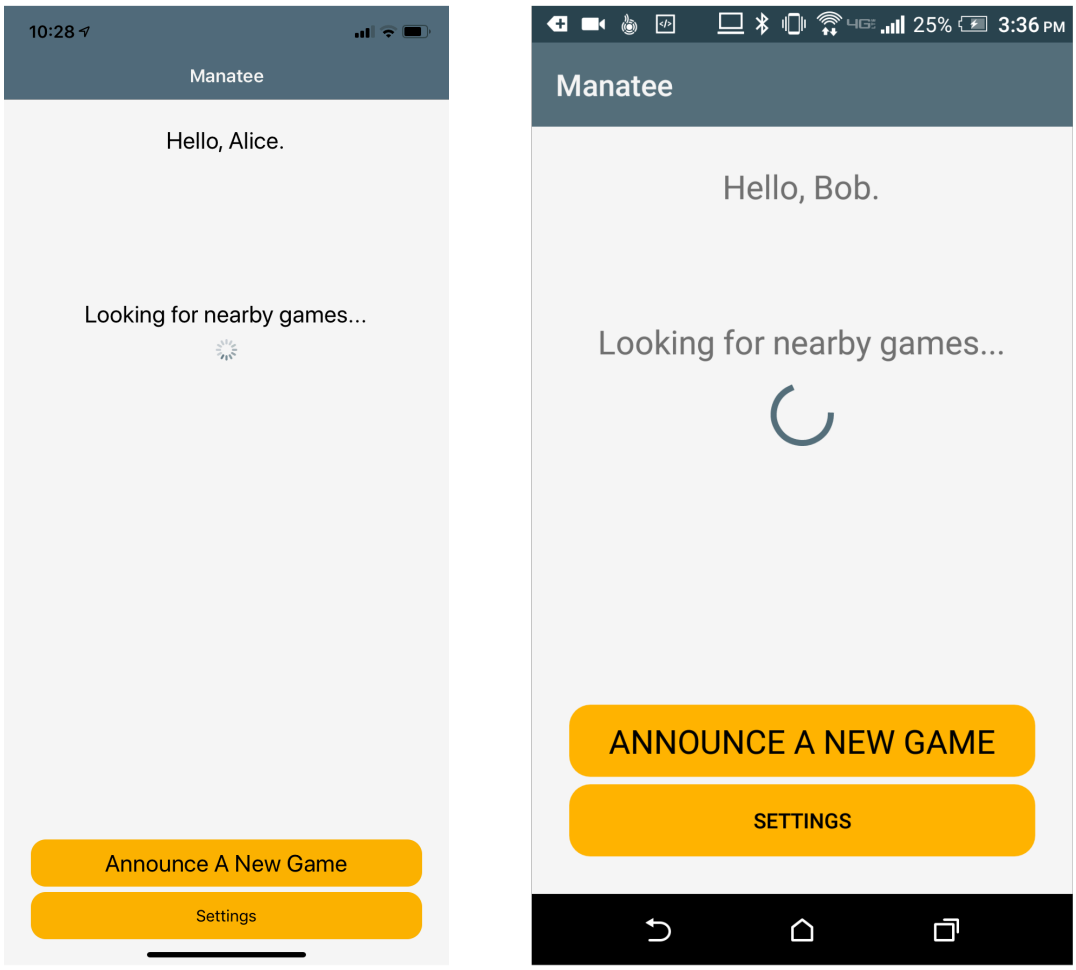

Figure 13. Main page

Once nearby permissions are granted, this will be the first page users see when opening the app.

Once permissions have been granted, players are presented with a 'scanning' screen that also includes 'Announce A New Game' and 'Settings' buttons at the bottom. Initially both buttons were listed at the top of the screen, and 'Announce A New Game' was simply titled 'New Game'. Predictably, this led to extreme confusion as most players opened the app and immediately tried to start a game, inadvertently creating competing announcements in the process. Rewording the button made it clearer that

players could join a game without creating a new one, and after moving it to the bottom of the screen players were less likely to click it reflexively without necessarily meaning to.

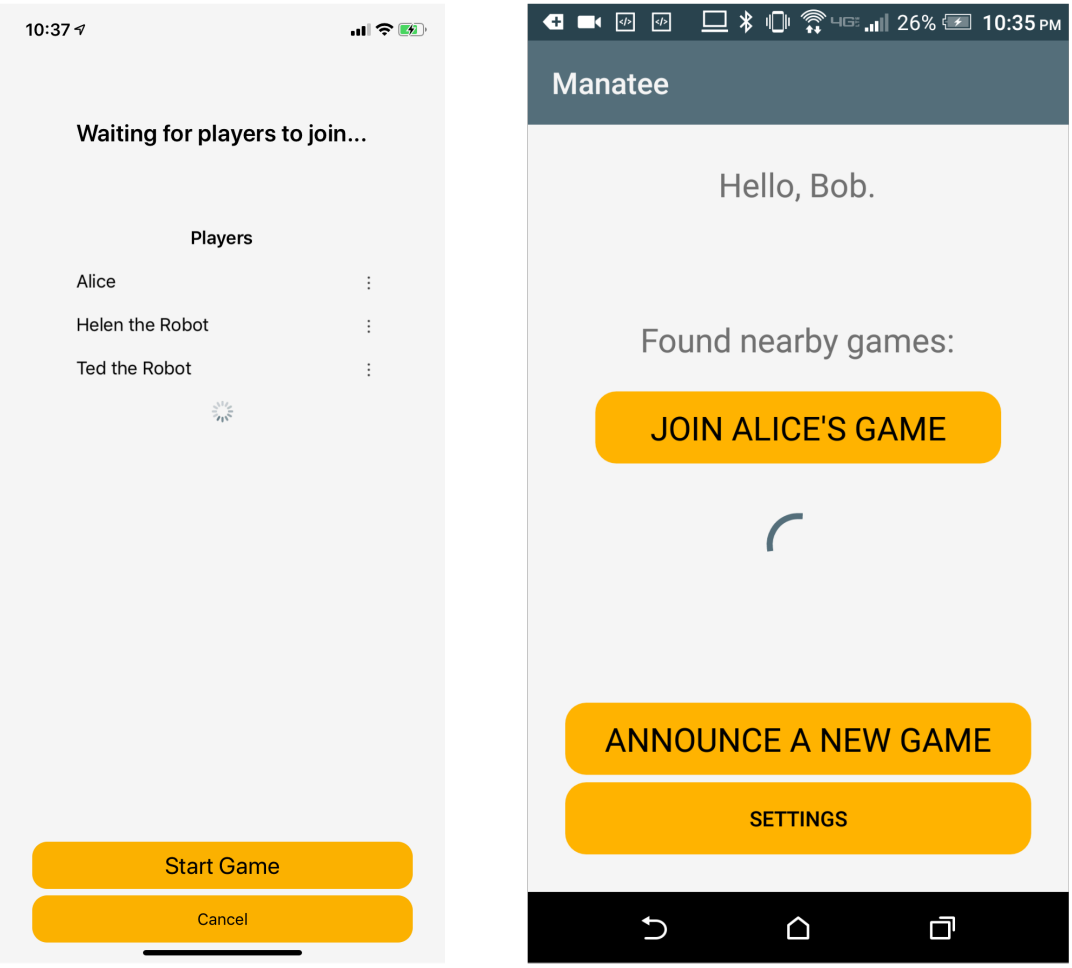

Figure 14. Initiating A New Game

In this screen, Alice has selected 'Announce A New Game' and Bob's device now registers Alice's game as an option.

Once Alice has selected "Announce A New Game", her screen allows her to vet players and modify turn order by reordering the list of player names. The ability to edit

turn order to match players' seating arrangements was one of the initial requested features for this app, but having implemented the option, I have yet to see a single person actually use it. I've retained it nevertheless because it's fairly unobtrusive, and because it may yet prove useful for groups larger than my test games. As with the main page, with the 'Start Game' button at the bottom of the screen, players are less likely to reflexively/ accidentally press before other players have joined.

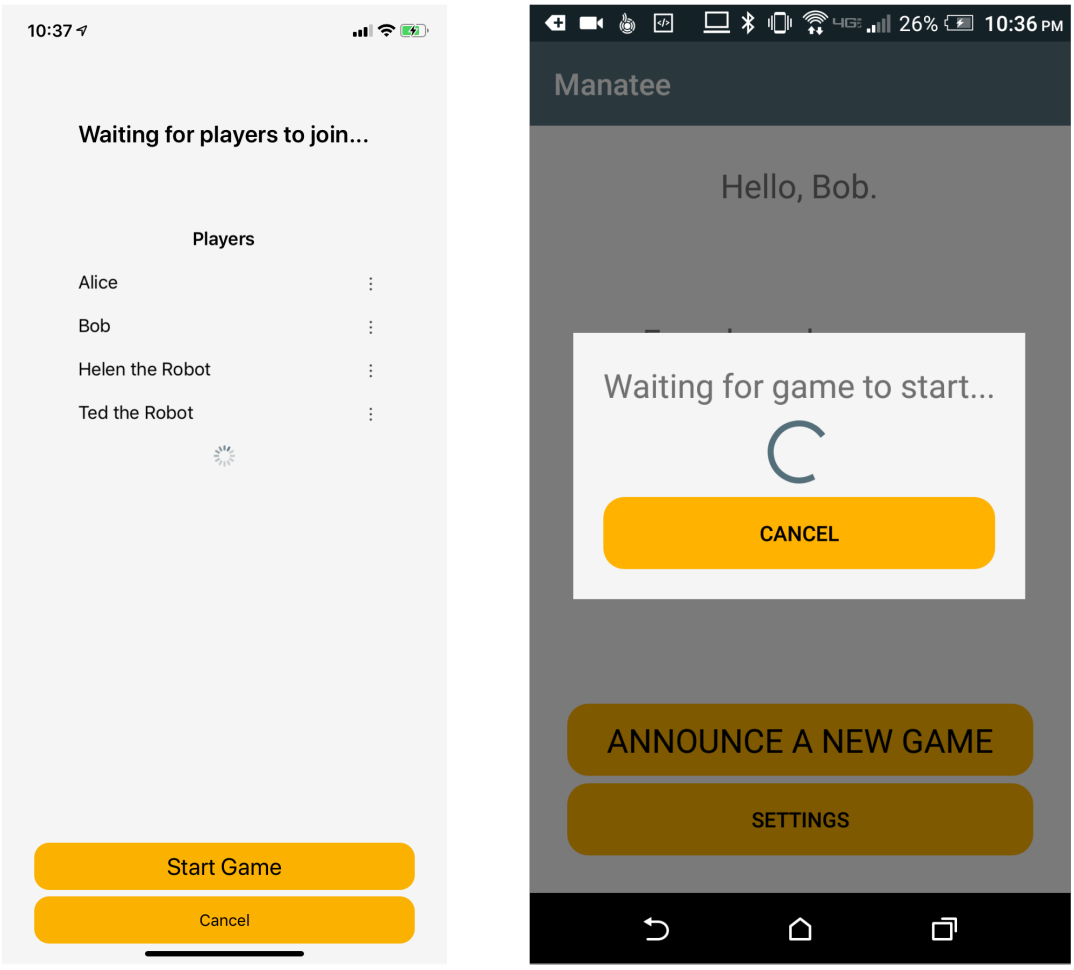

Figure 15. Joining a Game

Bob has chosen to join Alice's game; Alice's screen registers Bob as a new player.

Robots were not initially included on the ‘waiting for players’ page, since the host does not have to ‘wait’ for them to join, but at least two testers found their absence confusing and took it as an indication that the Robots setting wasn’t working. The same testers felt that having entries already in the list when the page was first displayed made its purpose more immediately comprehensible.

The text of this last pair of screens (fig. 13) is hopefully self-explanatory, but the behavior of the ‘Waiting for game to start’ dialog does need some explanation. Initially, when a host cancelled a game announcement, their device would broadcast a ‘Game Cancelled’ message and all joined players’ devices would display a cancellation notice and return to the main screen. In practice, however, it turned out that players rarely clicked ‘Cancel’ because they actually wanted to cancel the game or have another player take over as host. Since the connection lag tends to be most apparent during this handshake phase — it does, after all, involve a round trip communication that doesn’t require players to stop and think very hard in the middle — players were more likely to repeatedly cancel and re-announce a game in an attempt to speed up the connection, which would cause the other players’ connections to reset in a vicious cycle.

Clearly fixing the lag would be ideal, but in the meantime, it helped to stop associating ‘Join Game’ messages (see fig. 5) with the message ID of a specific ‘New Game’ message and to instead associate them with a specific host — and not to close the ‘waiting for game to start’ window or withdraw ‘Join Game’ messages without input from the user. This is also the reason that GameID values are now included in the StartGame message and not the initial game announcement; the game a player finally joins may not

have the same GameID as the announcement they initially responded to, and so the app doesn't register a GameID until the game has actually begun. This behavior does sometimes result in the app continuing to wait on a genuinely cancelled game until the user intervenes, but this outcome is still less problematic than a repeated subscribe/cancel cycle, and it occurs less often.

Waiting for Cards

In my initial mockups, this was just a blank page that read "Waiting for players to submit cards." To my surprise but, I'm sure, nobody else's, when transferred to a working app, players interpreted that page as a frozen screen, or just found the complete lack of interactivity intensely aggravating. In response to those initial test games I added an animated activity indicator, then a 'Wall of Shame' list with the names of players who had yet to submit cards. Finally, per users' requests, I added a full list of players (including robots) from whom submissions were expected and checked them off as submissions arrived.

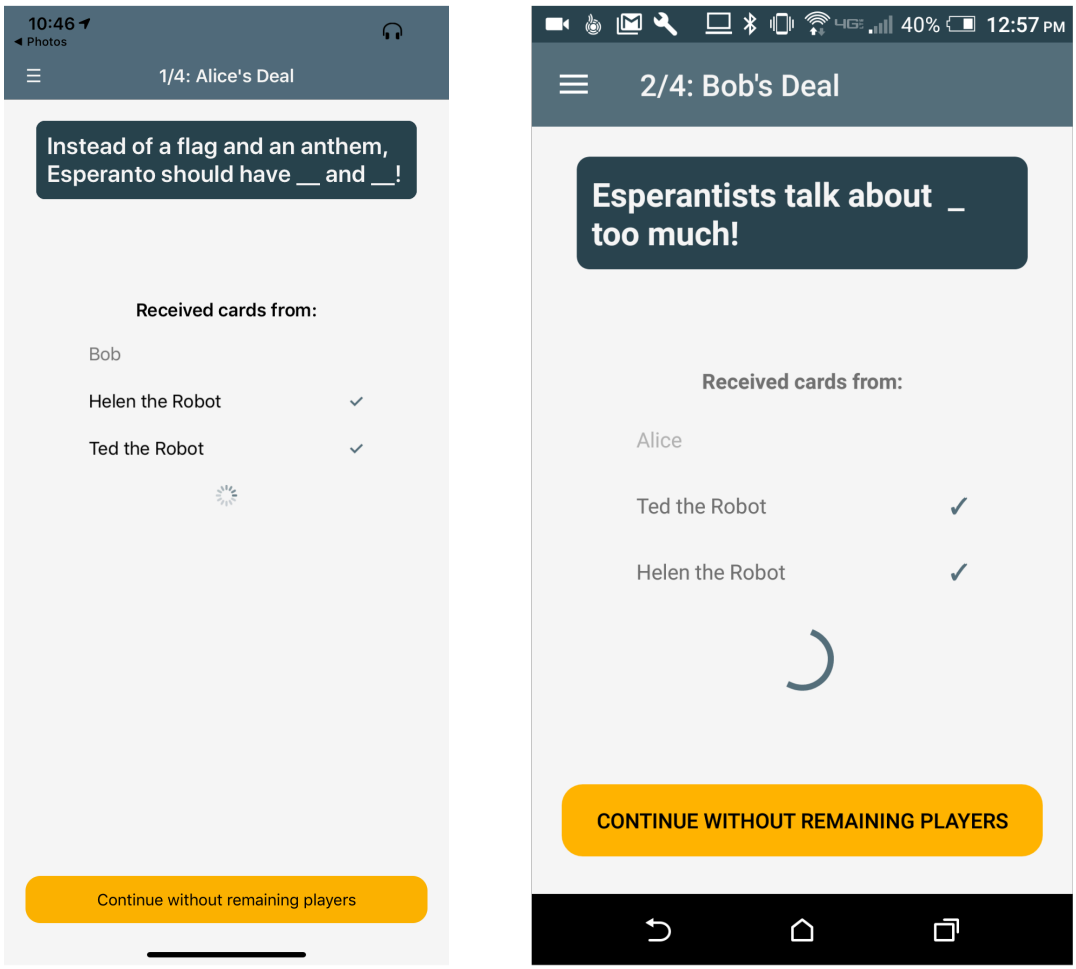

Figure 16. Waiting for cards

A waiting screen on iOS (left) and Android (right), taken from different rounds. The 'Continue without remaining players' button appears after a timeout has been reached and a given percentage of players have submitted cards.

On my first UI attempt, the judge's name was not displayed on any of the game screens. One of the most consistent points of feedback I got from Pretend You're Xyzzy test games was that the interface felt cold and mechanical — not because of its visual design but because of its behavior. For instance, when a winner is selected, PYX displays it on all players' screens very briefly and then immediately deals a new hand, leading one tester to describe it as "an assembly line." In my early plans I tried very hard to replicate

the feel of an actual card game by *not* taking steps the players would ordinarily take for themselves, like announcing the next judge. I had hoped the judge would identify themselves by reading the new card aloud, as usually happens in standard games, and I thought keeping the interface minimal would encourage that.

That decision was poorly received. It turns out that while users may not enjoy being railroaded by a program that advances without their consent, they also don't enjoy being railroaded into the developer's idea of a positive game experience by having information purposely withheld. Additionally, an in-person game of CAH or Apples to Apples, players have the visual cue of only one person in the game holding a single card with their hand face down. I was providing them with *less* information than they'd have in an in-person game. Per user demand, the judge's name and the current round now appear on all in-game screens. Happily, when playing in person, it turned out my test players didn't need to be forced to talk to each other; they read out the cards to each other even when it wasn't strictly necessary.

The 'Continue without remaining players' button was also added in response to user feedback to PYX. My initial list of reach-goal feature requests included a graceful way to deal with temporarily missing players, without using a hard timeout. My solution was to give the judge the option to continue after a certain period and once a given percentage of players had turned in their cards. I was careful to design the 'Winning Card' message so that it included enough information for someone who had skipped a round (or several) to sync back up with the rest of the game.

In all my test games, I have never seen a single person use this feature. That may, however, be the result of a surprise bonus of a mobile game versus a card game: unlike a physical deck, a phone has a 10-meter Bluetooth range. If a player wants to go downstairs to the fridge, they can simply take their phone with them without interrupting the round.

Selecting Cards

Because the ability to reorder cards was on my requested features list, I tracked down a Xamarin.Forms list view implementation that would allow drag-and-drop reordering. I found SyncFusion's SFListView, which offered that feature and had a generous community usage license for small companies and non-profit use, and I constructed a CardList class around it with subclasses to deal with single vs. multi-selection prompt cards, selecting cards to submit vs choosing submissions, et cetera. I did not include a discard option, because SFListView has reordering but, unlike Xamarin.Forms' ListView, no built-in context menus, and I didn't think I could produce a clean UI to mimic that functionality in the time required for this project.

In test games, I have never seen anyone reorder the cards in their hand.

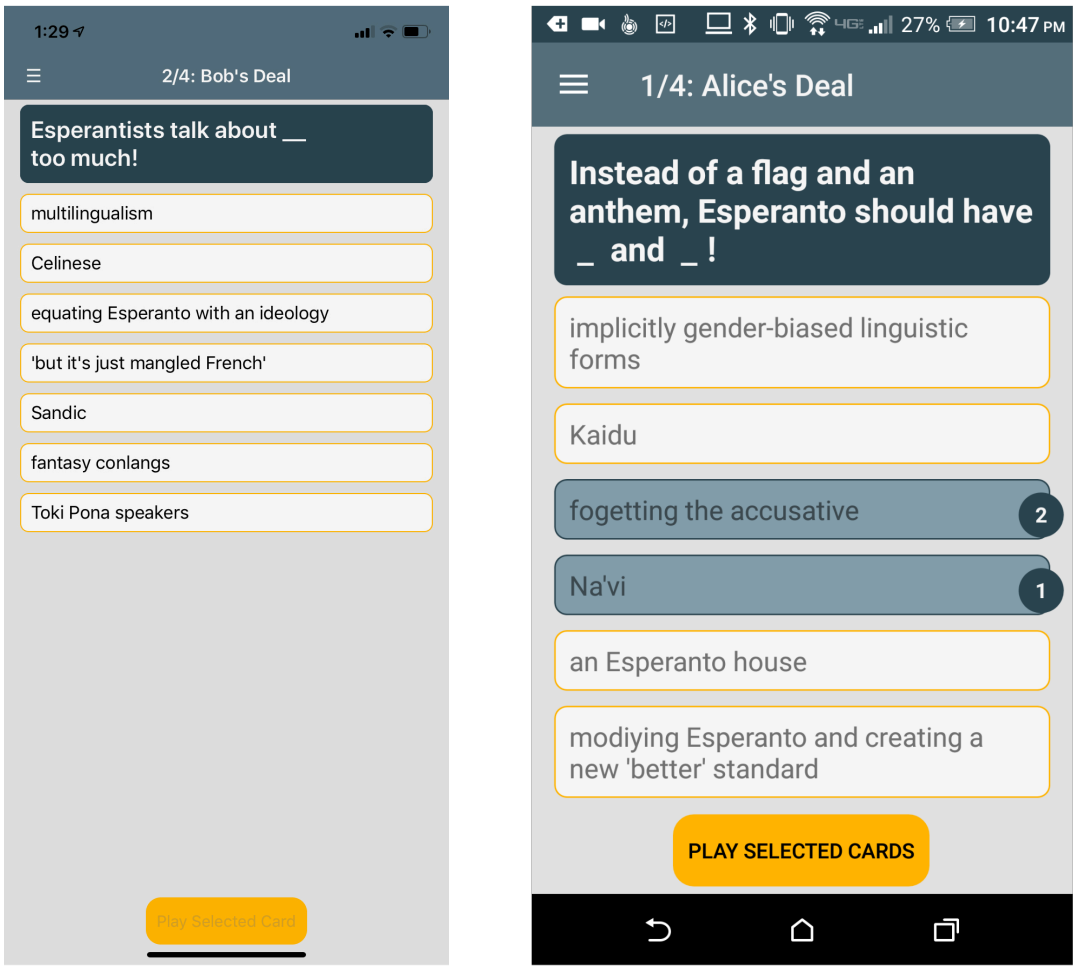

Figure 17. Choosing response cards to submit

The example screenshots in this section use two decks: ConLangs (a deck that focusses on constructed languages) and Esperanto for Humanity. These are examples of niche decks that would never be commercially viable and would be expensive to print as one-offs, but can be easily distributed via Cardcast.

The multi-card prompt screen (fig. 16, right, Android) demonstrates another surprise bonus of a mobile deck over a physical one. I've never played a game of Cards Against Humanity that didn't include at least some confusion about what order multi-card submissions should be read in. Multiple testers commented that the app's number

displays made rounds with multi-card prompts significantly easier than playing with a physical deck.

Picking the Winning Submission

The 'Flipped Cards' screen (fig. 16) is nearly identical to the response card selection screen, with the occasional exception of multi-card entries (fig. 14, right). In order to visually distinguish the two, the background is set to the same color as the prompt card, to associate the screen with the judge's (rather than the player's) hand. Testers said that the buttons and text at the bottom of the screen made the two pages easy to differentiate, but a few also absentmindedly tried to select cards on the 'Flipped Cards' page while they were not the judge. I've seen that less since adding a different background color, but it's impossible to say if that's a result of the UI change or simply the result of people growing more familiar with the app.

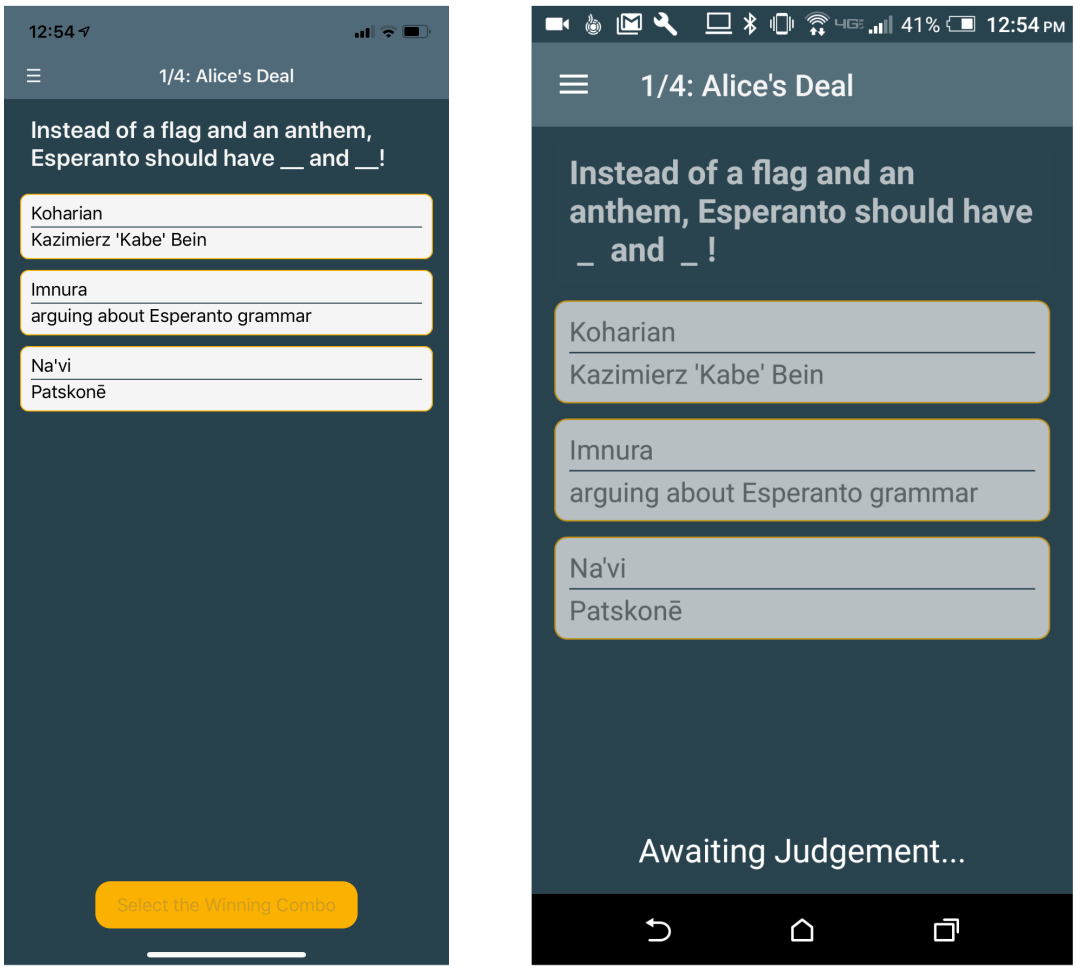

Figure 18. Submissions

Left, Alice's screen presents a list of submissions to choose from; right, Bob's device displays the same cards, greyed out to indicate that they aren't selectable, while waiting for Alice's decision.

Winning Card Alert

Winning cards are announced with a custom alert dialog (generated using rotorgames' Rg.Plugins.Popup). I did ask users if they wanted the app to display the source of all submissions, not just the winner, but everyone unanimously agreed it was more fun to guess at the results and interrogate each other after the round.

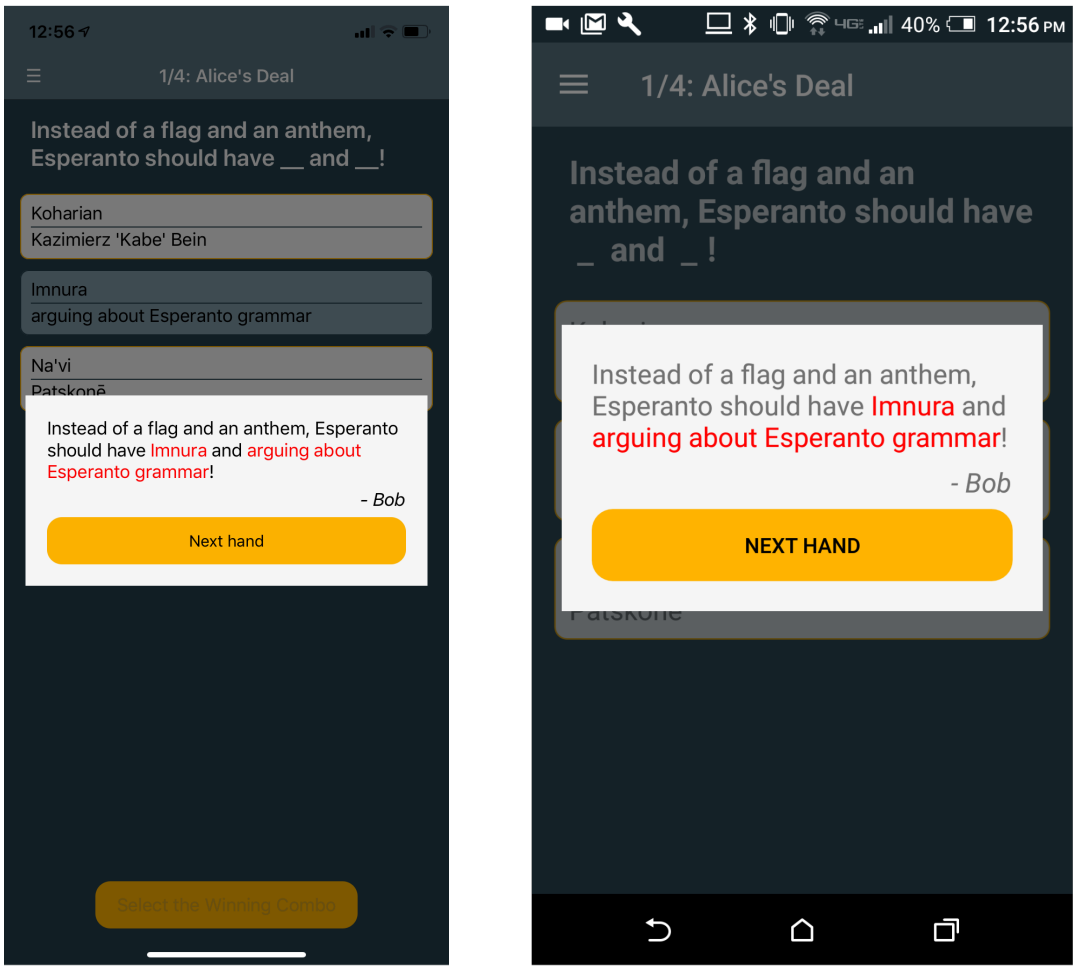

Figure 19. Winning Card Alert

Left, Alice's screen presents a list of submissions to choose from; right, Bob's device displays the same cards, greyed out to indicate that they aren't selectable, while waiting for Alice's decision.

In terms of game state, the winning card announcement acts as a sync point for both the backend and the UI. As discussed in the section on messaging protocols, Google Nearby doesn't guarantee in-order transmission of messages, and Manatee does not expect that. If your internet connection is extremely poor or you've had a different app pulled up on your device while other messages were exchanged, your device may not receive a list of submitted cards for the round before seeing the WinningCardSelected

message. If the host elected to progress to the next round without you (the feature is there, even if nobody ever uses it) you may get an alert for a winning card for a prompt you never even saw. At any point in the game, if the app sees a WinningCardSelected message for a round number equal to or greater than what it believes to be the current round, the app will immediately display this alert and update the game state to reflect the state information contained in that message, regardless of how far ahead of the current state that information is.

On that note, technically, the app doesn't need to wait for the user before moving forward. As noted in fig. 5, at this point, the app already has all the information needed to progress to the next round. However, Pretend You're Xyzzy made the choice to automatically clear the winning card and progress to the next round without user input, and it was one of the behaviors my testers found most annoying. When asked, testers unanimously agreed that it was more important to wait to advance until a player had had the opportunity to view the winning submission and confirmed they were ready.

Sidebar and Final Score Dialog

As a bit of UX trivia, the robots were originally not included in either scoreboard, and were given numbers instead of names. My testers, however, not only quickly anthropomorphized the nameless robots, but developed intense rivalries with specific robots, despite the fact that they were and are all manifestations of the same random number generator. They unanimously demanded that the robots both be added to the scoreboard and given identifiable human names. Thus Helen the Robot, Eliza the Robot, Harold the Robot, Ted the Robot, Karen the Robot, and Chad the Robot were born.

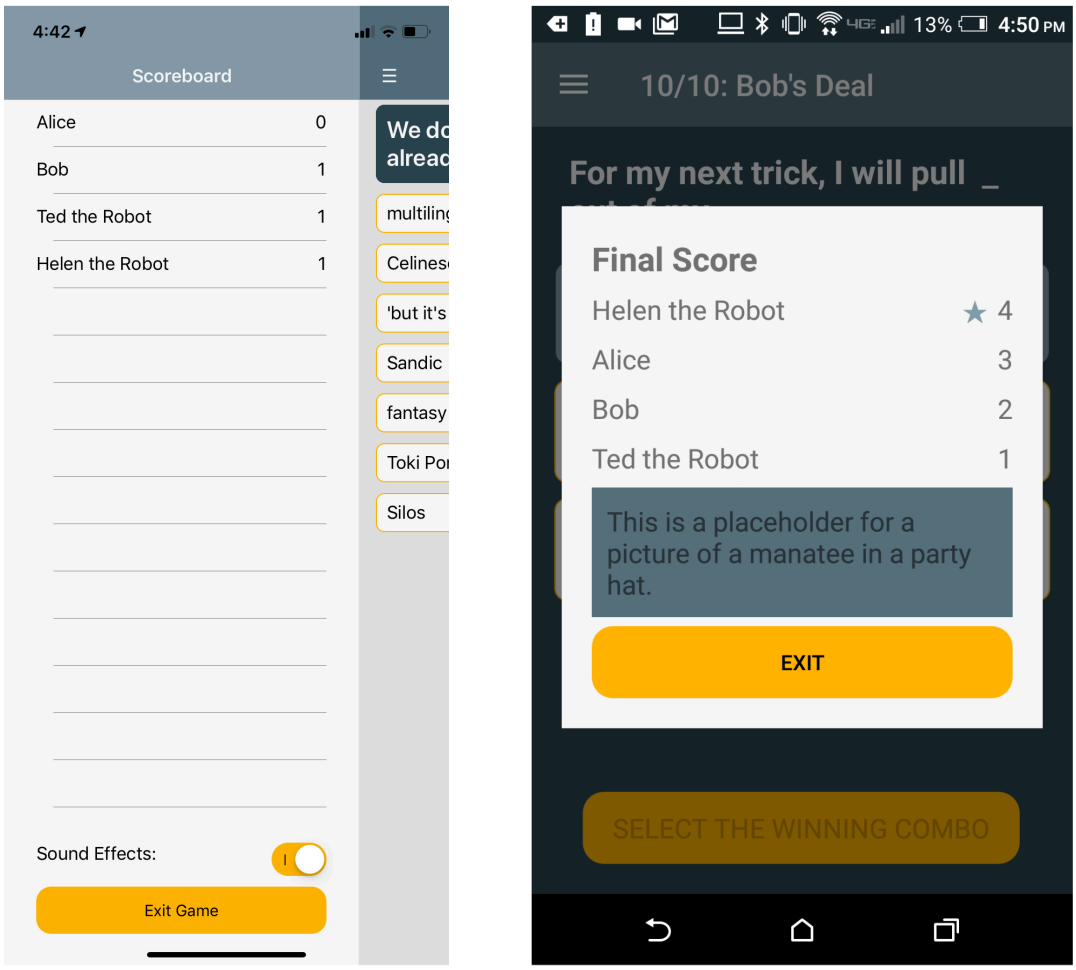

Figure 20. Scoreboards

Left, the game sidebar as displayed on Alice's iOS device with a scoreboard and an exit button; right, a final score dialog showing Helen the Robot as the game winner.

Chapter V.

Results and Evaluation

From a technical standpoint, I firmly believe that Xamarin.Forms in combination with Google Nearby was the best possible choice of tools when I began this thesis. As discussed in chapter III, Nearby Messages was (and remains) the only cross-platform messaging protocol feasible for this project in the time given. And while developing a C# binding for the Nearby Messages Objective C library was grueling, that's largely because the Nearby Messages CocoaPod was distributed in a non-standard format, and I was therefore forcing non-standard input into conversion tools I was unfamiliar with and then trying to turn the output into something functional. It's not clear to me that integrating an iOS library into Flutter would have been any easier, especially since Flutter's Channels are so much less powerful than Xamarin.Forms' DependencyService. (Google, n.d.-b) Regardless, anyone starting this project now would at least have my Nearby Messages C# binding available to use as a baseline. And the Xamarin development community is, in my experience, superior in both the vast range of plugins available and in its absolutely stellar written documentation.

Now that Flutter has a Nearby Messages plugin available, though, I'm not sure Xamarin.Forms would be the best choice for someone trying to develop an iOS compatible app today (i.e., for someone forced to work on a Mac.) I went in knowing Visual Studio for Mac had its detractors, and I still believe that was the right choice, but it was not a pleasant experience.

Visual Studio for Mac — the stable branch — would regularly decide that my cursor was ten pixels below and eight pixels to the left of where my operating system said it was, and would only reset that value after the application was uninstalled and reinstalled. It would, if run long enough without restarting, start deleting alphanumeric characters as well as whitespace when auto-formatting. It would somehow open alert dialogs offscreen, so that the only clue to their existence was the fact all other Visual Studio windows became suddenly un-selectable. I've never had a more baffling experience with an IDE. But I needed Visual Studio, because LiveXAML, the only 'hot refresh' tool I found for Xamarin that would work on a physical iOS device, only runs as a Visual Studio plugin. Towards the end of the project period, I finally landed on a system where I kept a VS instance running in the background while doing all my editing in JetBrains Rider, so that VS could pick up on the code changes and propagate them to my device using the LiveXAML plugin. This strategy rendered most of the above issues moot, but it took me a long time to land on, and the resulting setup was ridiculously byzantine.

So when I say that despite all that, Xamarin.Forms may still be the best option for a similar project, I hope it's clear just how highly I am speaking of Xamarin.Forms. As someone with almost no UI coding background, I found it extremely developer-friendly, and I'm not sure I could have successfully completed this project with a different framework.

Probably my most important takeaway from this project, however, was not "connections that involve at least two TCP handshakes per message are slow" or "Microsoft sometimes invests more in the Windows version of a product than the Mac

version" but "I need to rethink my strategy for setting project parameters." Going in, I made a concerted effort not to make assumptions about what features users wanted, and not to invest time and effort into aspects of the project users wouldn't care about. And yet, I broke that rule many times without even realizing it.

A perfect example is the graphical depictions of cards. I invested a significant amount of time at a point when I was still struggling with the basics of Xamarin UI coding trying to make my card displays resemble physical cards, because I assumed my users would want that. This was an assumption so profound I didn't even realize I was making it. All the similar online games had skeuomorphic UIs; of course mine should too. And yet, when I finally gave users the option of a non-skeuomorphic UI, it turned out they had no strong feelings one way or another.

Another possible example of an unnecessary feature is audio transmission support, which may not be necessary, or even especially helpful, after all. To produce some more specific numbers on the difference in latency distribution between Bluetooth-only and Bluetooth-plus-audio transmission for the final version of this paper, I went back and ran some formal tests with a newer device (an iPhone 6s and an HTC U11, vs. the same iPhone and a now-dead Samsung HTC One m7 for the original testing), and was completely unable to reproduce the original issue. Since the HTC One from the first test was also my only Bluetooth-compatible Android device for the first several months of development, during which I was transitioning fully towards a dual broadcast mode, it's entirely possible that I put a huge amount of unnecessary work into this project just because an elderly Android phone on its last legs had some Bluetooth issues.

I also made this mistake in cases where users specifically asked for features they later didn't use (which readers may have noticed as a recurring theme in this paper). Having preexisting web games to run initial test games with was a boon, but it probably also backfired a bit. Almost everyone understands that other people are uncomfortable criticizing someone's work to their face, and that friends are especially unlikely to be willing to point out issues. What I didn't take into account when running my initial play tests with existing CAH-style web games is that friends, or even neutral-to-friendly acquaintances, may also be more likely to actively try to poke holes in a product if they view it as competing with yours. For example, multiple test users complained about the lack of an intuitive mechanism to edit turn order in preexisting web versions of this game, but now that they have that mechanism nobody uses it. Maybe they're just comforted to know that they could edit turn order if they ever wanted to, but it's much more likely that they emphasized the importance of a minor issue while trying to be polite or helpful.

If I were starting this project from the beginning today, I would be more aware of that bias. Running multiple test games with Pretend You're Xyzzy was difficult, because my testers genuinely didn't enjoy using it, but I would have tried to run some followup test games, to see which complaints people still raised the second time around. I would spend more time watching people's actions to see if, for example, they instinctively tried to use a feature that wasn't there. If I'd had more time for testing, I would have waited longer to roll out features to see if users still asked for them when they were no longer framed as deficiencies in a competing app. Amazingly, the latter versions of this app have been pretty fun to test, but I have no doubt I could have produced something even better if I had used my time and energy more efficiently.

Chapter VI.

Conclusions and Further Work

I originally conceived of this project because I wanted to add a bunch of rock climbing jokes to a Cards Against Humanity deck without waiting ten business days for them to be printed and shipped. Nine months later, I am proud to have achieved that goal.

While I've implemented the minimum requirements, and some of the stretch goals, laid out in Chapter II, I consider this project far from complete. The most immediate task is to try to run Bluetooth speed tests on a wider array of devices, but the next step is to try to put the app before a wider audience. There are several features left to implement, but answering questions like the optimal permissions setup for people beyond my small gaming group will require fresh eyes.

Longer term, while we've enjoyed many games with the Nearby Messages protocol as a base, testing has shown that it's not quite ready for prime time in this use case. If clunky UI is what prevents Pretend You're Xyzzy from being overloaded, user frustration at message lag times will be what prevents this app from ever hitting Google's 8-million-a-day message limit. And, indeed, since starting this project, I notice that 'multiplayer gaming' has disappeared from Google's list of potential uses for Nearby Messages. (Google, 2018a; Google, 2019)

That said, at least one Google employee has reported that iOS support for Nearby Connections is in active development (Kapoor, 2018). Nearby Connections, unlike Messages, is truly peer-to-peer and doesn't go through Google's servers. That not only

eliminates the internet connectivity requirement for devices to talk to each other; it also substantially cuts down on the lag time between devices. And while users may not care about this, as a truly peer-to-peer protocol Nearby Connections has no externally imposed message limit, making it infinitely scalable as this project was originally intended to be. If the Nearby Connections API became available for iOS, transitioning from Messages to Connections would immediately become the next top priority.

There's no official Connections library for iOS out yet, and the project may still fall through. The fact that Google is devoting resources to it, however, does show that at least one major player in the industry is starting to pay attention to true peer-to-peer cross-platform mobile connectivity. Perhaps others will follow their lead.

Appendix A.

Source Code and Binaries

Full source code for this project can be found at https://github.com/mshepher/Manatee6.

Android binaries can be downloaded from https://install.appcenter.ms/users/mshepherd/apps/manatee-android/distribution_groups/global.

iOS binaries can be downloaded from https://install.appcenter.ms/users/mshepherd/apps/manatee-ios-1/distribution_groups/global.

Appendix B.

Selected Third-Party Packages and Resources

MedallionRandom. https://github.com/madelson/MedallionUtilities/tree/master/MedallionRandom

Newtonsoft.Json. https://github.com/JamesNK/Newtonsoft.Json

Serilog. https://github.com/serilog/serilog

serilog-enrichers-thread. https://github.com/serilog/serilog-enrichers-thread

Rg.Plugins.Popup. https://github.com/rotorgames/Rg.Plugins.Popup

Syncfusion.Xamarin.Buttons, Syncfusion.Xamarin.SfListView, Syncfusion.Xamarin.SfNumericUpDown: https://www.syncfusion.com/xamarin-ui-controls

ConnectivityPlugin. https://github.com/jamesmontemagno/ConnectivityPlugin.

SimpleAudioPlayer. https://github.com/adrianstevens/Xamarin-Plugins/tree/master/SimpleAudioPlayer.

Xam.Forms.Plugins.KeyboardOverlap. https://github.com/paulpatarinski/Xamarin.Forms.Plugins/tree/master/KeyboardOverlap.

References

American Mensa. (n.d.). The Mensa Select® Seal. Retrieved from https://mensamindgames.com/about/the-mensa-select-seal/

Babich, N. (2016, May 30). Mobile UX Design: The Right Ways to Ask Users for Permissions. Retrieved from https://uxplanet.org/mobile-ux-design-the-right-ways-to-ask-users-for-permissions-6cdd9ab25c27

Cardcast (2017). About Cardcast. https://www.cardcastgame.com/about.

Cardcast - Apps on Google Play. (n.d.). Retrieved April 20, 2019, from https://play.google.com/store/apps/details?id=com.cardcastgame.android.app

Cards Against Humanity. (n.d.). Cards Against Humanity Store. Retrieved April 20, 2019, from https://store.cardsagainsthumanity.com/

Deutsch, L. (Deutsch, 2015). You can now play Cards Against Humanity on your phone. *USA Today*.

eskimo (Apple Staff). (2018, February 5). iOS and Wi-Fi Direct. Retrieved from https://forums.developer.apple.com/thread/12885

Google. (n.d.-a). Use network service discovery | Android Developers. Retrieved from https://developer.android.com/training/connect-devices-wirelessly/nsd

Google. (n.d.-b). Writing custom platform-specific code. Retrieved April 18, 2019, from https://flutter.dev/docs/development/platform-integration/platform-channels

Google. (2018a). Nearby | Google Developers. Retrieved from https://web.archive.org/web/20180806130739/https://developers.google.com/nearby/

Google. (2018b). Overview | Nearby Messages API | Google Developers. Retrieved from https://developers.google.com/nearby/messages/overview

Google. (2019). Nearby | Google Developers. Retrieved from https://developers.google.com/nearby/

Gwizdz, G. (2013). Zero configuration networking and communication using iOS and Android. Master's thesis, Grand Valley State University.

Hacker News (2016). Ask HN: Bonjour-like service discovery on Android? https://news.ycombinator.com/item?id=16214878.

Kapoor, V. (2018). Nearby Connections support for iOS & delay in sending the message by "Nearby Messages" · Issue #40 · googlesamples/ios-nearby. Retrieved from https://github.com/googlesamples/ios-nearby/issues/40#issuecomment-442151631

Made With CC. (2017). Cards Against Humanity. Retrieved from https://medium.com/made-with-creative-commons/card-against-humanity-8c0cc2c6c299

Microsoft. (2017). Walkthrough: Binding an iOS Objective-C Library - Xamarin. Retrieved from https://docs.microsoft.com/en-us/xamarin/ios/platform/binding-objective-c/walkthrough

Mulligan, B. (2014). The Right Way To Ask Users For iOS Permissions. Retrieved from https://techcrunch.com/2014/04/04/the-right-way-to-ask-users-for-ios-permissions/

Pretend You're XYZ (2018). Pretend you're XYZ. http://pretendyoure.xyz/zy/.

Soto, I. (2016). SotoiGhost/NearbyMonkey. Retrieved April 18, 2019, from https://github.com/SotoiGhost/NearbyMonkey

Sneath, T. (2018). Flutter 1.0: Google's Portable UI Toolkit. Retrieved from https://developers.googleblog.com/2018/12/flutter-10-googles-portable-ui-toolkit.html

Van Hoff, A., Blair, R., & Kreuzer, K. (2019). JmDNS. Retrieved from https://github.com/jmdns/jmdns

Whitfield, D. (2015). That was fun. https://web.archive.org/web/20151202031039/http: // www.cardsagainstoriginality.com/.