

Using Bluetooth Low Energy to Enhance Ad Hoc TaleBlazer Multiplayer Mechanics

by

Carlos Alfredo Henríquez

S.B., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

© Carlos Alfredo Henríquez, MMXIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and to
distribute publicly paper and electronic copies of this thesis document
in whole or in part in any medium now known or hereafter created.

Author
Department of Electrical Engineering and Computer Science
March 11, 2019

Certified by
Eric Klopfer
Director, MIT Scheller Teacher Education Program
Thesis Supervisor

Accepted by
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Using Bluetooth Low Energy to Enhance Ad Hoc TaleBlazer Multiplayer Mechanics

by

Carlos Alfredo Henríquez

Submitted to the Department of Electrical Engineering and Computer Science
on March 11, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

TaleBlazer is an educational mobile game platform designed to build location-based augmented-reality learning games. While most games for TaleBlazer are designed as single-player experiences, there are many benefits to be gained from player-to-player in-game interactions. This thesis introduces a robust and flexible protocol for peer-to-peer communications between TaleBlazer players using Bluetooth Low Energy (BLE), an API for further feature development using BLE, and an interface for game designers to leverage those features. I introduce the concept of teams over BLE in TaleBlazer to facilitate player-to-player communications, which allows members of the same team to communicate with each other without risk of their communications being intercepted by other teams. The protocol and API used for these teams are modular and extensible to simplify future development of player-to-player interactions via BLE.

Thesis Supervisor: Eric Klopfer

Title: Director, MIT Scheller Teacher Education Program

Acknowledgments

I would like to thank Professor Eric Klopfer, Lisa Stump, and Judy Perry for giving me the opportunity to work on TaleBlazer even under unusual circumstances and for being incredibly flexible and understanding as the project evolved. In particular, I'd like to thank Lisa for challenging me to look at problems from a different perspective and for all the technical help and guidance she gave me through this project. I'd also like to thank Judy for making sure I was never down in the weeds for too long and always took a step back to look at the bigger picture. I had an incredible time working on TaleBlazer, and this would not have been possible without your continued guidance, direction, and support.

I'd like to thank my predecessor, William Spitzer, who laid the foundation off of which my thesis is based. His work and insights were invaluable during the course of this endeavor.

To all of the friends I have made during my five and a half years at MIT, especially my community in Next House 2East, thank you for helping me stay positive and supporting me throughout my time here. My MIT experience would not have been the same without each of you.

And last but not least, to my family, for their unwavering support, encouragement, and positivity, and without whose sacrifices none of this would have been possible. Thank you for always believing in me.

THIS PAGE INTENTIONALLY LEFT BLANK.

Contents

1	Introduction	13
1.1	Motivation and Problem Statement	14
1.2	Thesis Summary	17
2	Background on TaleBlazer	19
2.1	TaleBlazer Editor	19
2.2	TaleBlazer Mobile	21
3	Previous Work on Multiplayer Interactions	25
3.1	Server-based Multiplayer	25
3.2	Proximity-based Technologies Considered	26
3.3	BLE-based Sending and Receiving	29
4	BLE in TaleBlazer	33
4.1	A Brief Overview of the BLE Protocol	35
4.2	TaleBlazer’s BLE Infrastructure	36
4.2.1	Android Module	39
4.2.2	JavaScript Layer	41
4.2.3	TaleBlazer Blocks	41
4.2.4	Mobile User Interface	42
5	Using Teams to Accommodate the Field Trip Model	45
5.1	Team Leaders Advertise Teams	46
5.2	Players Send Join Requests to Team Leaders	49

5.3	Team Leader Finalizes Team	52
5.4	In-game Messages	53
6	Bluetooth Blocks for the Web Editor	55
6.1	Creating New BLE Messages	56
6.2	Send Message to Recipient with Value	57
6.3	When I Receive the Message	59
6.4	When I Successfully Send Message	61
6.5	When All Teammates Receive	62
6.6	Value of Last Message Received and Sent	62
7	Future Work	65
7.1	iOS Module	65
7.2	Team Management	66
7.3	On Timeout Block	69
7.4	Dynamically Changing Messages to Scan for	70
7.5	Sending Large BLE Messages	70
7.6	Two-Way Communication	71
7.7	Sending to a Specific Player	72
8	Contributions and Conclusion	75
A	Message Header Structures	77
B	API and Events	81

List of Figures

2-1	The TaleBlazer editor	20
2-2	Blocks-based programming interfaces	21
2-3	The TaleBlazer Mobile map.	22
2-4	A sample agent dashboard.	23
3-1	The previous send and receive blocks	30
4-1	The TaleBlazer BLE architecture.	38
4-2	The new BLE blocks introduced for the game designer	41
4-3	UI updating in response to BLE message receipt	42
5-1	Team Formation Flow	47
5-2	UI for creating a team	48
5-3	UI for joining a team	50
5-4	The final team roster	51
6-1	The new BLE blocks created for the game designer	55
6-2	BLE message creation dropdown and modal	56
6-3	The “Send message to player with value” block	57
6-4	The “When I receive the message” block	60
6-5	Two examples of the “When I receive the message” block	60
6-6	An example of the “When I successfully send message” block.	61
6-7	An example of the “When all teammates receive message” block	62
6-8	Sample usage of the “Value of last message received” and “Value of last message sent” blocks.	62

7-1	The UI for removing a team member during team formation.	68
7-2	The UI seen when a player has been removed from the team	68
7-3	Confirmation dialog for leaving a team	69

List of Tables

A.1	Team formation header structure	78
A.2	Payloads for team, join, and finalize messages	79
A.3	Header structures for in-game messages	79
B.1	The API methods exposed by the native module and invoked by the JavaScript layer	82
B.2	BLE Android Events	83

THIS PAGE INTENTIONALLY LEFT BLANK.

Chapter 1

Introduction

With the advent of augmented reality (AR) into the mainstream thanks to games like Niantic’s Pokémon Go and tools like Apple’s ARKit for developers, many applications have taken to augmenting our real world experiences with visuals and interactive components on our phones. Whereas traditional games immerse players in a virtual world completely separate from reality, augmented reality games combine the rich landscape of the real world with the creative mechanics of a virtual world.

TaleBlazer is a location-based AR game platform in which game designers create mobile games centered around visiting and interacting with objects and places in the real world [1]. Game designers create location-based games on the TaleBlazer web platform using a blocks-based programming language, and players download the TaleBlazer app on their phones to play the games made on the web platform. TaleBlazer games can vary from simple narratives, such as a scavenger hunt around a park, to more complex role-playing games, such as a game simulating life in New England in the 1820s. A number of these games have been made in collaboration with museums, zoos, and aquariums to augment the experience of their visitors with a mobile game they can play alongside enjoying the exhibits.

1.1 Motivation and Problem Statement

Most games made with TaleBlazer are exclusively single-player experiences. However, multiplayer features in TaleBlazer games have the potential to vastly enhance the players' experience, increase the narrative complexity of the games, and create avenues for interesting and engaging player-to-player interactions. For example, multiplayer games in TaleBlazer could be large collaborative endeavors, such as a simulated investigation of the causes and effects of pollution in a local riverbank. A game that is too complex to complete alone encourages players to work together. Such a game may also help players realize that by cooperating they can discover more and progress further in the game than they could on their own, and this, in turn, mirrors the real world benefits of collaboration. Similarly, competition in multiplayer games increases player engagement and adds a layer of challenging fun that is not easily attainable in single-player games as players attempt to outperform their peers. Lastly, by working together students have an opportunity to learn from each other and develop their communication skills. These are only some of the benefits that integrating multiplayer experience into TaleBlazer can have.

One of the goals of my thesis project was to create a way to facilitate player-to-player interactions in TaleBlazer games under the *field trip model* – that is, enhancing TaleBlazer such that small groups of players playing the same TaleBlazer game in the same location can collaborate and share information easily without interfering with other groups of players doing the same. The field trip model is important because many TaleBlazer games operate under this model with small groups working together at the same time as other groups also working together. Because this is one of the main use cases for TaleBlazer, my work had to support games under this model.

TaleBlazer games are usually restricted to one location, such as a zoo, aquarium, or other publicly accessible location, off of which the game is based. With support for these player-to-player interactions, TaleBlazer could go beyond having games be restricted to one location and use location-based mechanics without being limited to one physical location. Easily usable multiplayer functionality could eventually

allow TaleBlazer to support games that are playable anywhere, not just in the region for which the game was made, with the potential to implement a wide variety of interesting mechanics. This could lead to the use of TaleBlazer as an AR-in-a-box platform: a way to create and play multiplayer AR games that can be played virtually anywhere.

But in order to achieve the eventual goal of playing AR games with anyone, anywhere, we must first enhance TaleBlazer to support multiplayer games under the field trip model. Multiplayer mechanics open up the possibility for new types of games within TaleBlazer. These multiplayer mechanics would introduce a new dimension of social interactions that is not readily available within TaleBlazer currently. Some of the first features that multiplayer mechanics might add to TaleBlazer include:

- Passing a limited resource to another player, such as a master key for unlocking doors
- Disseminating discovered in-game information easily with all members of the same team
- Synchronizing in-game events such that some events can only trigger if all players are performing the same action at roughly the same time
- Affecting other nearby players of the same game by either helping or hindering them with certain in-game effects.

These are all examples of mechanics that would enable TaleBlazer game designers to create a shared group experience in which the decisions of each player can affect the game outcome of all the other players. This would open the door for games in which the contributions of all players are necessary and where one player cannot simply rush through the game by themselves but must work together with their peers to accomplish some goal. With these and other similar mechanics in place, players can work together and have their contributions impact the experiences of the other players, thus enabling game designers to construct collaborative outcomes for games where all players working together reach the same concluding narrative.

Many board games and video games have multiplayer mechanics from which we can draw inspiration. There are games that make use of the environment and in particular territory control as a key mechanic. For example, the board game *Go* revolves around strategically placing stones on a grid in order to capture the opponent's stones and eventually control as much of the board as possible. Similarly, the video game *Splatoon* has teams of players vie for control of an enclosed area by coloring the walls and floors with as much of the team's ink as possible.

There are also games in which the game environment is sectioned off into tiles, and players can directly affect other players that are on tiles nearby. For example, in the board game *Risk*, players control small groups of armies that try to take over different parts of Earth. Army units that are on adjacent tiles can provide reinforcements in the event of an enemy invasion, so there is an incentive to keep units together and connected in order to provide support when necessary. Similarly, in the video game series *Fire Emblem* players control an army of unique units laid out on a grid, and these units can provide bonuses for other nearby units as they work to rout the opposing units.

Other games use game tiles as a way to empower players to co-create the game board and construct a new game experience each time they play the game. In the board game *Betrayal at the House on the Hill*, players build the layout of a haunted house by exploring the various rooms, drawing from a deck of room tiles to determine what the room behind a door will be. Players take turns choosing what unexplored areas of the house they wish to venture into, and these room tiles trigger events that can then alter other tiles or affect players on tiles with certain properties.

While there are several other mechanics to draw inspiration from, it is important to keep in mind some of TaleBlazer's design constraints. Once a game is downloaded in TaleBlazer, players do not need a data connection to play the games, which allows TaleBlazer to thrive in locations with intermittent data connectivity and players who might not have large data plans to enjoy TaleBlazer as well. However, there are other technologies that can be used for communicating with nearby devices, like Bluetooth Low Energy (BLE). BLE enables peer-to-peer communications with nearby players in

TaleBlazer games that can directly impact their gameplay without requiring the use of a data connection. Further work beyond the scope of this thesis could leverage BLE to fully implement mechanics like in the games mentioned above or even more complex mechanics that could improve how players collaborate and compete in TaleBlazer games.

With BLE showing promise for enhancing ad hoc multiplayer mechanics in TaleBlazer, one of the questions to answer was what kinds of mechanics would be best suited for TaleBlazer that would be feasible to implement over BLE and how those mechanics should be presented to the game designers. For my Master’s thesis, I took on the task of continuing previous work on multiplayer features using BLE to help game designers easily integrate BLE features into their games. I rewrote the previously written BLE module to provide the building blocks for generic BLE functionality with a clean and non-TaleBlazer specific API. I designed a TaleBlazer-specific protocol that allows TaleBlazer developers to create team-based and non-team-based BLE interactions. Finally, I fully implemented two user-facing features built on this new architecture: a team formation setup phase (described in Chapter 5) and a set of blocks that give game designers access to sending and receiving TaleBlazer messages over BLE during gameplay, described in Chapter 5 and Chapter 6, respectively. The API’s modular design allows future developers to extend its functionality and implement more features that leverage BLE and allow players to collaborate within TaleBlazer games. With this, I hope to empower future developers, game designers, and game players to push the boundaries of what is currently possible within TaleBlazer and one day make TaleBlazer a powerful platform for creating collaborative AR games that players can play with anyone anywhere.

1.2 Thesis Summary

Chapter 2 provides an overview of the two main parts of TaleBlazer: TaleBlazer Editor, the web platform on which game designers create TaleBlazer games, and TaleBlazer Mobile, the mobile app on which users play said games.

Chapter 3 discusses the previous work on TaleBlazer’s multiplayer features and some of the other technologies considered.

Chapter 4 outlines TaleBlazer’s BLE architecture in the context of multiplayer games.

Chapter 5 details the BLE API I created and explores one application of the API in the form of teams of players, in which players must first go through a setup phase to create their teams before they can communicate with their teammates using BLE.

Chapter 6 illustrates how a game designer can include BLE functionality into their games using the specialized blocks on the TaleBlazer Editor.

Chapter 7 suggests ideas for future work on expanding TaleBlazer’s Bluetooth functionality and potential game mechanics.

Chapter 2

Background on TaleBlazer

TaleBlazer is an augmented reality (AR) location-based game platform that allows game designers to create and play their own mobile games in the real world [1]. TaleBlazer games are primarily designed to be played outdoors in parks, zoos, and other public areas. The TaleBlazer Mobile app tracks the player’s movements using the phone’s GPS data, and when a player goes to a certain physical location, they can bump into and interact with characters or items in the game.

TaleBlazer’s platform consists of two main parts: TaleBlazer Editor, the web-based application where game designers use a block-based programming language to create their games, and TaleBlazer Mobile, the Android or iOS application on which TaleBlazer games can be downloaded and played.

2.1 TaleBlazer Editor

The TaleBlazer Editor is where game designers define all aspects of their games, such as the game’s boundaries, what the player sees in the mobile app, and how the player interacts with the game world. A game designer must first select a rectangular region corresponding to a location in the real world where the game will take place. From there, the game designer can then add *agents*—the characters or items that players can interact with—to the game world. Figure 2-1 shows the map view of the editor where game designers can set the boundaries for and see the location of all

agents in a game.

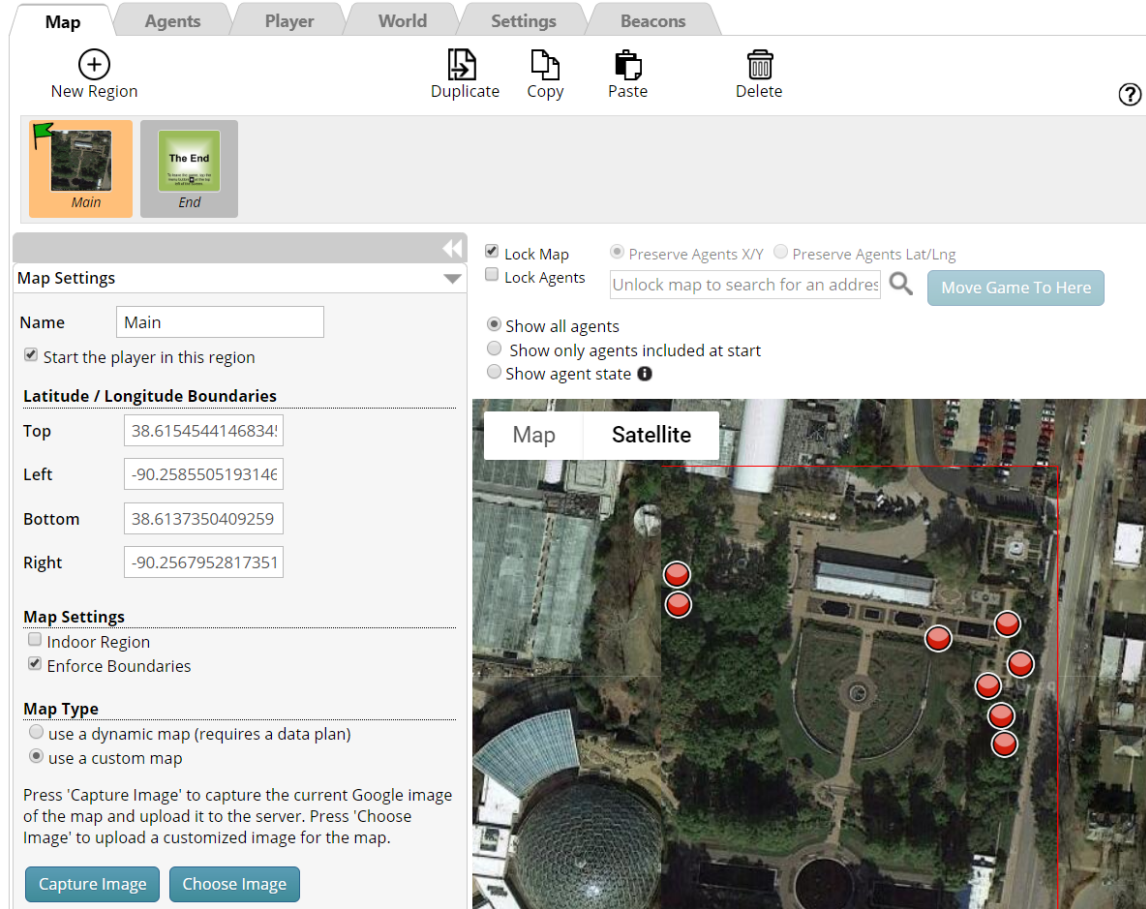


Figure 2-1: The TaleBlazer Editor map. Game designers define the real-world boundaries for their game from the Map tab and see where they have placed the game’s agents.

Agents can also have optional *traits*, such as health or power, that the game designer can add and change in response to in-game actions. The game designer programs the interactions that a player can have with agents using TaleBlazer’s blocks-based programming language, similar to Scratch [2] or App Inventor [3]. Figure 2-2 shows examples of blocks from each of these applications. The blocks in TaleBlazer represent several logical operations, such as modifying game state or displaying and hiding agents and traits from game players. Blocks are associated with an agent, the player, or the game world, and they can be nested (e.g. conditionals and loops) or attached together (e.g. serial code execution). When a player interacts with an agent, the blocks associated with that agent are executed, which can then have varying

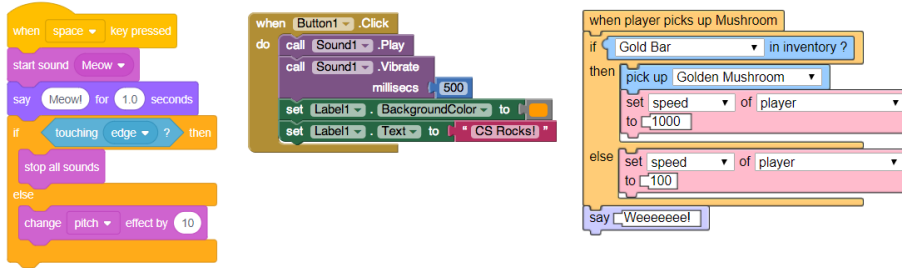


Figure 2-2: Examples of blocks-based programming from Scratch (left), App Inventor (middle), and TaleBlazer (right). Blocks are connected to each other and executed top to bottom.

effects on the game state.

2.2 TaleBlazer Mobile

Players interact with TaleBlazer games via the mobile app. Players can either enter a game code to download a specific game, or they can download games that are built for their specific location if they are visiting an officially partnered TaleBlazer Place. Game designers can create TaleBlazer games that use a data connection to show the player a live map of their current, real-world location, but if data connectivity is a concern game designers can instead opt to use a static image or set of images as a map. A player will need to connect to the Internet initially to download the game, but once the game has been downloaded the player can play their game without a data connection if the game uses static images for its maps. This enables schools and other groups to use devices without data plans and allows players to play games in remote locations that might not have ready or reliable access to the Internet.

Once in the game, the player usually sees a map and at least one agent somewhere on the map that they can interact with by *bumping* it – getting close to where the agent is in the real world. Figure 2-3 shows an example of a game map with one agent on it. Games sometimes begin with an introductory screen that sets the narrative for the game. Some agents are *included* in the game, meaning the player can bump and interact with them to progress the in-game narrative, while other agents can be *excluded*, meaning that the player cannot interact with them until some other in-game

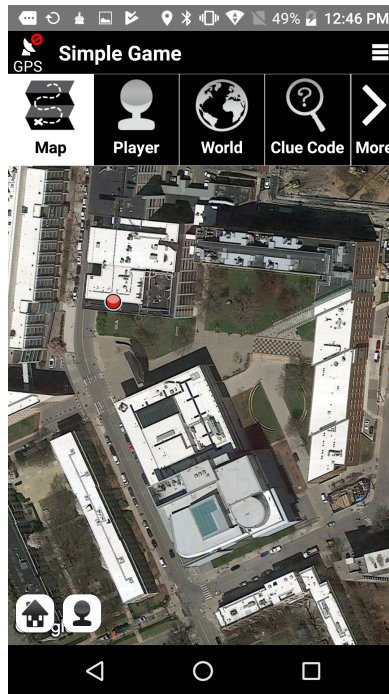


Figure 2-3: The Map as seen in TaleBlazer Mobile with one agent (red circle).

actions includes them. For example, in narrative-based game that requires its agents to be bumped in a particular sequence, the game designer can exclude all but one agent at the beginning and then include the next agent in the sequence after the previous agent is met.

Games can have multiple *regions* that correspond to different map areas. Each region is distinct and can have its own collection of agents, and the game designer can also move agents from one region to the other in response to player actions. The player's region determines what the player sees on the map tab (e.g. different maps for different regions) and which agents are available for the player to bump. The game designer specifies when the player switches from one region to another using a specific block in TaleBlazer.

When a player bumps an agent, the agent *dashboard* appears showing a picture of the agent, its name and description, and optionally traits associated with the agent and buttons corresponding to actions that the player can take with that agent. Two pre-programmed actions for agents are picking up and dropping, which will add an agent to the player's inventory and remove it from the inventory, respectively. Game

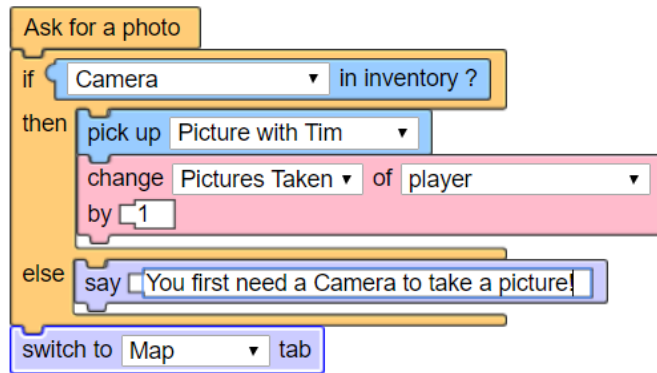
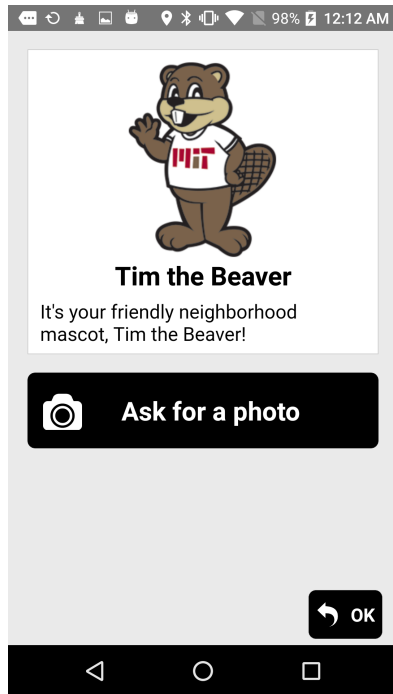


Figure 2-4: A sample dashboard for an agent with the agent's picture, description, and an action.

designers can also create custom actions for agents by using the TaleBlazer editor and creating a *script*. Figure 2-4 shows an example of an agent dashboard with a button for a custom action and its associated script.

Players can also have different *roles* that they can choose from at the beginning of the game. A game designer can choose to have multiple roles for a game that can change how a player interacts with agents, or there can be just one role, in which case the player does not have the option to select a role before the game starts. For example, a water agent might turn to steam or snow depending on whether the player has the fire mage or ice mage role. Roles allow players to see different information and make their experiences more distinct based on their role choice, which highlights the nuances of some TaleBlazer games.

THIS PAGE INTENTIONALLY LEFT BLANK.

Chapter 3

Previous Work on Multiplayer Interactions

Multiplayer functionality is not a new concept for TaleBlazer. In the next sections I outline some of the previous work in a server-based model, the beginnings of a Bluetooth implementation, and some of the other technologies considered for use in a peer-to-peer communication protocol for TaleBlazer, which ultimately informed my decision to leverage BLE as the technology of choice.

3.1 Server-based Multiplayer

A previous version of TaleBlazer implemented multiplayer mechanics using a server to maintain and propagate a consistent shared game world across multiple players[4][5]. Games were made in the same editor as before, but when players wanted to play a multiplayer game together, one player created an *instance* of the game in the multiplayer server that was tied to a game code. The instance creator then shared this game code with other players who wanted to join that same instance, and as the players progressed through the game, the server kept track of the game state and mediated conflicts such as two players attempting to pick up the same object simultaneously.

Being connected to this server during gameplay, however, required a strong, consis-

tent data connection, lest users experience issues with game lag and synchronization due to intermittent connectivity. Many of TaleBlazer’s games are played in locations where a strong Wi-Fi connection might not be available, and some TaleBlazer users might not have budget in their data plan to play these multiplayer games. The issue of intermittent connectivity also made it difficult to create a robust system that synchronized the game state across all players when players lost and regained connectivity as they played a TaleBlazer game. Furthermore, the most glaring problem with the shared, consistent game world model is that it is at odds with many typical TaleBlazer games that require players to visit agents in a specific sequence to unfold the game narrative. This made story-based games, in which players visit agents one by one in a mostly linear fashion according to the plot, difficult to faithfully port into a multiplayer world where the state of the game world (e.g. whether an agent has been visited or picked up) is shared across multiple players who might be progressing through the story at different speeds. The multiplayer games have since been discontinued, but the mechanics offered by the multiplayer server inspired some of the TaleBlazer BLE mechanics, which I explain in Chapter 4.

3.2 Proximity-based Technologies Considered

There are other ways that players could share information in a TaleBlazer game without the use of a server. I evaluated Near Field Communication (NFC), Wi-Fi Direct, QR codes, and Bluetooth as possible alternative methods communication methods, and each has its advantages and disadvantages in the context of TaleBlazer.

NFC is a set of communication protocols that allows two devices to exchange data by bringing them physically close to each other, usually within a few centimeters. The requirement for the devices to be next to each other could be something desirable for facilitating player-to-player interactions in TaleBlazer as it requires players to be in close proximity when exchanging data. Using NFC, users could send small strings of text or even large files like photos. The biggest drawback to NFC for TaleBlazer, however, is its limited functionality in iOS devices. Whereas Android devices can use

NFC to read static messages (such as those on an NFC beacon) and send arbitrary messages by acting like an NFC beacon, iOS devices are only able to read NFC messages. This was a deal breaker for TaleBlazer as we did not want to allow game designers to create features that would only work for Android devices. While Core NFC, the iOS NFC API, has been around for a few years, Apple deliberately did not include the ability for iOS devices to write their own arbitrary NFC messages and mock beacons due to security concerns [6]. Because Apple is unlikely to change its mind regarding NFC, NFC will not be viable as a multiplayer communication mechanism for TaleBlazer.

Wi-Fi Direct is a Wi-Fi standard that allows two devices to establish a direct Wi-Fi connection between them without requiring a wireless access point [7]. Wi-Fi Direct allows devices to send arbitrarily large media directly to each other. However, similar to NFC, Apple has its own proprietary version of Wi-Fi Direct whose API does not allow for easy connection with Android devices. Although there exist workarounds to connect Android and iOS devices to each other by having one device act as a virtual hotspot, this introduces a higher technical barrier and more complex setup for the game players, many of whom are children. This would also disrupt the player's experience with TaleBlazer as it would require them to navigate to their settings to connect to the device acting as a Wi-Fi hotspot. Because of the higher technical floor of Wi-Fi Direct and complicated cross-platform support, I opted against using it for TaleBlazer multiplayer.

QR codes allow for messages to be compressed into a small grid of black and white squares that can be generated on demand thanks to several libraries that exist for creating and scanning QR codes. With the largest QR code and the lowest amount of error correction, one can encode up to 2953 bytes or 4296 alphanumeric characters in a single QR code[8]. Similar to NFC, the use of QR codes requires that the phones be within close range and that the code be in line of sight for the camera reading the code, which is desirable for facilitating player-to-player interactions. QR codes would also eliminate the possibility of any confusion as to who is receiving the data. However, navigating away from the map and into a separate camera screen to scan the QR code

would break the fourth wall in TaleBlazer and would still be complicated for both the game designer to implement and for the game player to use within TaleBlazer. If an agent in a game requires the player to scan a QR code once they find a hidden artifact, the player might resort to simply searching for the visible QR code rather than the artifact itself. Lastly, because of the requirement of line of sight for the camera and close proximity, QR codes would not allow players to send stealthy messages, such as sending the coordinates of a hidden trap that one player planted for another.

There are two different forms of Bluetooth that devices can use: Bluetooth Classic and Bluetooth Low Energy (BLE). Bluetooth Classic is used for sending large amounts of data between a variety of devices such as computer mice, printers, headphones, laptops, and other phones. Bluetooth Classic requires that devices be *paired* with each other, a handshake process that has the devices identify themselves to each other such that they can quickly re-establish a connection between the two if one device is out of range but goes into range again. Pairing devices enables a high throughput, low latency connection between the devices that are within range, such as streaming audio from a computer to Bluetooth headset.

Bluetooth Low Energy, on the other hand, is a technology that uses the same Bluetooth hardware to transmit data between devices at lower speeds but with reduced power consumption. Unlike with Bluetooth Classic, devices do not need to be paired a priori to transmit data. However, due to the lower power consumption, BLE has lower throughput and higher latency than Bluetooth Classic. Instead of pairing, the sending device sends out advertisements signaling that it has data it can send. The receiving device scans for these advertisements, and upon finding an advertisement of interest, forms an ephemeral connection to the sending device. Once connected, the sender sends the data packets to the receiving device, and the connection is terminated. BLE is used in devices that have stricter power requirements, such as heart rate monitors, proximity sensors, and fitness devices.

Although Bluetooth Classic enables connections with higher data throughput with lower latency, it faces the same cross-platform compatibility issues that NFC and Wi-Fi Direct have. iOS devices cannot be paired with Android devices, but even if they

could be paired, this might lead to some uncomfortable situations for TaleBlazer players. Many of TaleBlazer’s users are children, and the TaleBlazer team deemed asking users to pair their devices an inappropriate requirement for an application for children, particularly in a school setting.

BLE, however, does not have as many restrictions regarding the types of devices that can advertise, scan, or connect as Bluetooth Classic does, and as a result BLE supports cross-platform communication. This allows a TaleBlazer player playing on an iOS device to interact with a TaleBlazer player playing on an Android device. BLE has a smaller impact on the device’s battery life, and although the data throughput with BLE is lower, it satisfies TaleBlazer’s crucial requirement of cross-platform compatibility. I ultimately chose BLE for the work of this thesis.

3.3 BLE-based Sending and Receiving

Although NFC, Wi-Fi Direct, QR codes, and Bluetooth Classic have their advantages over BLE, such as the ability to send larger amounts of data with each of them than with BLE, the constraints these technologies have with regards to cross-platform compatibility and proximity make them less viable than using BLE. BLE allows for Android-iOS communication, which is essential for TaleBlazer. With BLE, users can communicate at distances of up to 30 to 50 feet (depending on factors like signal strength and the presence of walls or other obstacles), which creates fewer restrictions for game mechanics that NFC or QR codes would. Lastly, using BLE requires virtually no setup from the user apart from enabling Bluetooth on their phone, whereas Wi-Fi Direct and Bluetooth Classic would require users to navigate away from the TaleBlazer app to properly configure their settings and connect to other devices.

Returning to the field trip model mentioned in Chapter 2, one of the goals for multiplayer interactions is the ability for two players to interact with each other in-game in isolation from other nearby players. Using BLE, players in a TaleBlazer game can bump an agent, choose to send some data to another player, and then resume playing their TaleBlazer game without ever having to leave the TaleBlazer

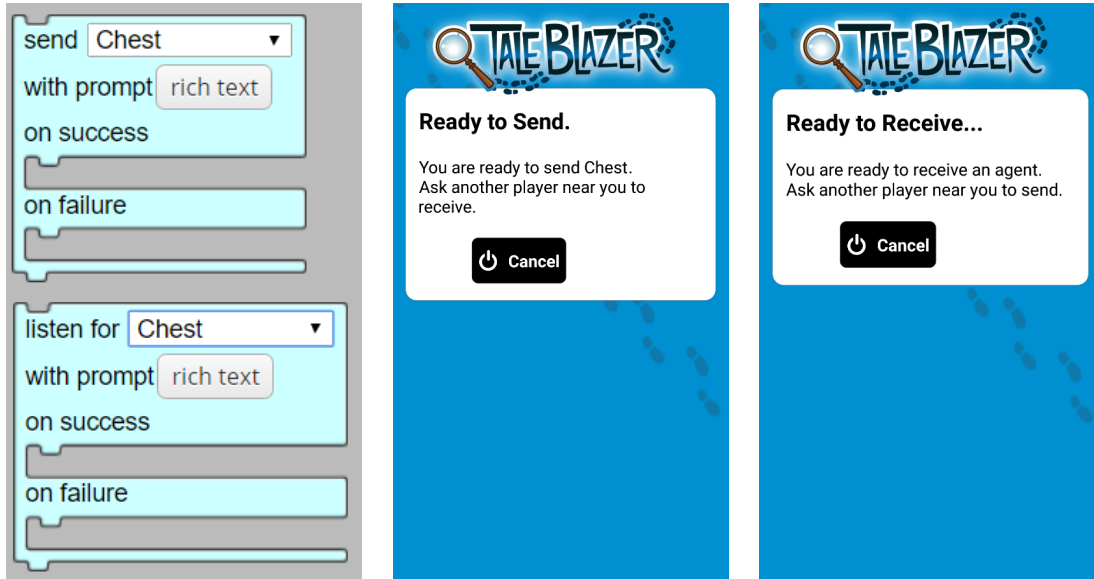


Figure 3-1: The previous send and receive blocks (left) created by Spitzer[9] and their corresponding UI screens that a game designer can use for synchronous sending over BLE. When the send block was executed, the player saw the “Ready to Send” screen (center), and when the receive block was executed, the player saw the “Ready to Receive” screen (right).

app. William Spitzer, a previous Master’s student who worked on TaleBlazer, created the first modules for TaleBlazer that integrated BLE functionality[9].

The first implementation of a TaleBlazer feature that used BLE to send information with other players involved players synchronizing to send or receive agents or text strings by explicitly performing two matching actions at the same time. To implement this functionality, the game designer added a pair of blocks, one for the sending and one for receiving, that when executed prompted players to either prepare to send data or prepare to receive data, as shown in Figure 3-1. If the sender and receiver perform the same action at roughly the same time, such as pressing a button on the screen, then the data was sent over. If the players do not perform their actions at the same time, such as if one player tries to send but no other player presses the button to receive, then the action will time out after 30 seconds. The game designer could also add blocks scripts to be executed depending on whether the transaction was successful or timed out.

Spitzer’s work with the synchronous send and receive blocks was a fundamental

first step for enabling multiplayer interactions over BLE in TaleBlazer. The blocks allowed players to send agents or messages to one another, but there was no way to send an agent or message to a specific player. If two sets of players were playing the same game and chose to send the same agents, there was nothing to prevent one player from accidentally intercepting a message intended for another player if they happened to try to send and receive at the same time. This situation was possible under the field trip model in which different groups of players might need to send and receive agents to another at around the same time. Further, having players synchronize by waiting to send and waiting to receive with the screens shown in Figure 3-1 broke the fourth wall for the players and the flow of the game by forcing them to stop for up to 30 seconds to continue. Despite this, having players synchronize by preparing to send and receive together was an important step toward a solution to the problem of intercepted messages and for enabling multiplayer interactions.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

BLE in TaleBlazer

In order to better accommodate the field trip model for TaleBlazer, we first need a way to restrict the number of possible recipients for any given TaleBlazer message sent over BLE. If TaleBlazer players play a game together as part of a team, members of the same team should be able to send data to one another without risk of their messages being accidentally intercepted by players not on their team. Using this goal as a starting point, we want players to be able to send data to each of the following recipients:

- **A specific player**, where one player sends data to another player specifically.
- **Any one teammate**, where one player sends data to one other teammate when it does not matter who receives the data so long as they are a teammate.
- **Any one non-teammate**, where, conversely, one player sends data to another nearby player when it does not matter who receives the data so long as the recipient is explicitly not a teammate.
- **Any one player playing the same game**, where one player sends data to another nearby player playing the same game, regardless of whether the recipient is a teammate.
- **All teammates**, where one player sends data to all nearby teammates.

- **All players not on team**, where one player sends data to all nearby players who are not teammates.
- **All players in game**, where one player sends data to all nearby players, regardless of whether they are teammates.

There are some subtle nuances for the options to send to more than one player. All of the options above are restricted to sending to one player that is within range for BLE (30 to 50 feet). When sending to any one player (a specific player, any one teammate, any one non-teammate, any one player playing the same game), the sender will attempt to send for 30 seconds before sending times out. The sender will stop advertising when either one player receives the sender's data or after 30 seconds have passed, whichever happens first. For the options for sending to multiple players (all teammates, all players not on team, all players in game), the sender will keep advertising until 30 seconds have passed. During those 30 seconds, the sender will send the data to any appropriate recipient that connects to the sender during that window. Because receiving players must be within BLE range and within the 30 second timeout window, there is a chance that no one will receive the message. If this happens, the sender must attempt to send their data again.

The options enumerated above were inspired by some sample use cases. The option to send to any non-teammate is useful for games where teams have access to different types of information, and in order to progress the game narrative, teams need to share what they discover with a representative from another team. The option to send to any player in the game could be used when the specific recipient is irrelevant and the message just needs to be passed from one player to another, such as in a game of hot potato. In order to apply a power-up in some TaleBlazer game to all nearby teammates, a player could send said power-up with the all players on team option. Likewise, if there is an agent that can damage or hinder all non-teammates, a player can send said agent with the all players not on team option. Lastly, the option to send to all players in game can be used in a game where a player might be sick and unknowingly spreading some contagious disease to all other players.

To add the above functionality to TaleBlazer Mobile, I first needed to rebuild parts of the underlying TaleBlazer BLE architecture to make them more modular and easily extensible. One of the features missing from the previous implementation of BLE in TaleBlazer was the ability to send a message while scanning for messages to receive at the same time. Previously, players had to choose to either exclusively send or receive (see Figure 3-1), and there was no way for a player to receive a message at an arbitrary time during gameplay.

To address the above, I designed and implemented a generalized BLE module on Android for TaleBlazer that allows the device to scan for advertisements and receive BLE messages at any time during the game. I built a TaleBlazer specific advertising protocol and API on top of this module that enables TaleBlazer Mobile to send messages to a specific player or set of players, which allows TaleBlazer to better accommodate the field trip model. Lastly, the protocol and API allow for future developers to add other BLE features to TaleBlazer without needing to worry about any lower level formatting of messages.

4.1 A Brief Overview of the BLE Protocol

Bluetooth Low Energy allows devices to exchange information via Bluetooth with a predefined protocol[10]. Given two devices, a *peripheral* device that wishes to send information and a *central* device that wishes to receive that information, the exchange occurs as follows:

1. The peripheral device creates a Generic Attributes (GATT) server that responds to connection requests from other central devices.
2. The peripheral device begins sending out short, 20-byte advertisements signaling that other devices can connect to it.
3. The central device begins scanning for BLE advertisements sent out by other BLE devices, including other mobile phones, headphones, televisions, and speakers.

4. If an advertisement is of interest to the central device (e.g. the central is scanning for advertisements with a certain prefix that the advertisement contains), the central device connects to the peripheral device using the MAC address contained in a reserved section of the advertisement.
5. The central device sends a read request to the peripheral's GATT server for a particular characteristic containing the information that the central device is interested in.
6. The peripheral's GATT server handles the request and sends over a maximum of 512 UTF-8 encoded bytes of data (the limit on BLE GATT characteristics).
7. The central device disconnects from the peripheral device once the transaction is over.

In TaleBlazer, a *BLE message* is composed of two parts: the *header* and the *payload*. The *header* is the BLE advertisement that a peripheral, or sender, sends out and that a central, or receiver, scans and filters for. Previously, these headers were hard-coded, but with the new API a TaleBlazer developer can customize what goes in the 20-byte advertisement. Central devices can also filter for a specific header or set of headers and then request to connect to a peripheral that is advertising a header that the central is scanning for. The *payload* is the 512-byte message sent by the peripheral to the central upon a successful BLE connection and can be any arbitrary data that does not fit into the header.

4.2 TaleBlazer's BLE Infrastructure

The TaleBlazer mobile app is built using the Titanium framework by Axway Appcelerator¹. Titanium allows developers to create cross-platform mobile apps by writing JavaScript code that is then packaged specifically for those devices. There are certain conventions regarding file tree structure that the framework dictates to allow

¹<https://www.appcelerator.com>

the devices to interpret the same JavaScript code properly. Some functionality, such as access to a phone's Bluetooth hardware, is not available via the Titanium SDK. Developers must instead write their own Titanium modules in Java for Android and Objective-C for iOS to expose an API to provide that functionality. This module mediates the transfer of data between the usually platform-agnostic JavaScript code and the platform-specific module APIs. Figure 4-1 shows the four parts of TaleBlazer's BLE infrastructure:

- the Android or iOS module that interfaces directly with the device's hardware
- the JavaScript layer that interfaces between the native Android or iOS module and the mobile user interface
- the blocks on TaleBlazer Editor that a game designer can include in their game to add Bluetooth functionality which interface with the JavaScript layer
- the mobile user interface that reflects the state of the game and responds to user actions.

Due to resource constraints, I focused mostly on restructuring the Android module and the JavaScript interface to the module. I had to first decouple the code that interacted with the module from the user interface code and stabilize the Android module such that future developers should not need to recompile the native module and can simply add functionality from the JavaScript layer upward. By decoupling the native module, the JavaScript API to that module, and the user interface, TaleBlazer's BLE structure now follows a modular design that is much easier to build and improve upon thanks to the layer of abstraction the JavaScript API provides. Because the JavaScript API that I wrote provides control over what one can include in BLE message headers and payloads, what headers to filter for, and when to start advertising or start scanning, future TaleBlazer developers can build other multiplayer features on top of the JavaScript layer without needing to worry about writing native Android code.

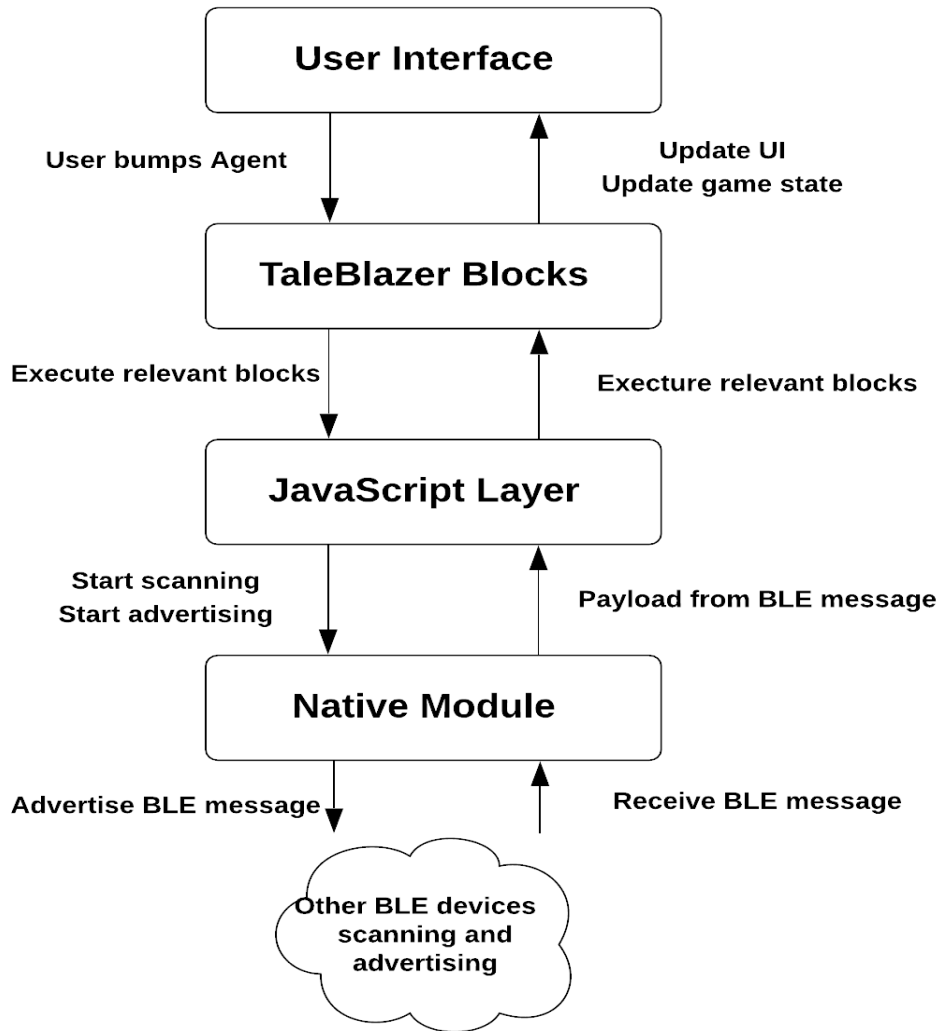


Figure 4-1: The TaleBlazer BLE architecture. Going from top to bottom, the user interface executes blocks as a response to user actions. Blocks then call upon their corresponding functions in the JavaScript layer. These functions then call upon the appropriate methods in the native Android or iOS module to start advertising or scanning for advertisements. Going from the bottom to the top, when the native module receives an advertisement of interest, it fires an event that the JavaScript layer listens for. The JavaScript layer then searches for and executes the appropriate blocks which then update the user interface and the game state.

4.2.1 Android Module

The Android module (and similarly the iOS module) handles any requests made to the device’s Bluetooth hardware. The module can turn the device’s Bluetooth on and off, and it is responsible for using the hardware to advertise and scan for advertisements. Whereas previously the native module could only advertise or scan for advertisements (but not both simultaneously), I restructured to allow for a device to advertise and scan at the same time. This was crucial in the development of team formation, discussed further in Chapter 5.

Rather than have the player initiate a request to scan for advertisements and “prepare to receive”, once the player starts a TaleBlazer game with BLE functionality, their device periodically scans for TaleBlazer advertisements in the background.

Whereas previously, in order for a player to receive a BLE message, they had to press a button to start scanning for advertisements, once the player starts a TaleBlazer game with BLE functionality, their device scans for TaleBlazer advertisements in the background. This lowers the cognitive load on the receiving player as they can now receive BLE messages during gameplay without needing to take action. This also opens the door for sending stealth messages to a player, such as a secret message that a player from another team could send them. Although Google—the company behind Android—recommends that developers not scan on a loop and to turn off scanning as soon as the desired device is found, we are deliberately going against this guideline [11].

Because players might not know a priori who will send them a message, it is impossible to determine what the desired device to connect to over BLE is. Whereas in other BLE applications the mental model is for the central device to read a possibly changing value from a constantly advertising peripheral device (e.g. a phone receiving temperature readings from a sensor), in TaleBlazer the model is reversed: the peripheral initiates the transfer of data, such as one player passing a message to another player, by sending out advertisements as a response to in-game events.

Players in a Bluetooth-enabled TaleBlazer game must remain scanning for in-game

messages for the duration of the game, as they do not know when they might receive a BLE message from another player. Although continually scanning for BLE messages drains the phone’s battery more than stopping a scan after the message has been sent or received, a study by Aislelabs on the impact of several BLE parameters on battery life, such as scanning interval, concluded that “battery use is very sensitive to the scan interval” but that in newer phones “the battery use is minimal even with [an] aggressive scan interval”[12]. Further, because TaleBlazer games are location-based and require use of the phone’s GPS, the battery drain from GPS dwarfs the battery drain from BLE[13]. Lastly, background scanning will only happen when the player is playing the game; if the player leaves the game or the game is paused (either manually by the player or automatically after a few minutes of inaction), the device will stop scanning for advertisements until the player resumes the game.

The Android module can advertise only one header and associated payload at a time, but it can filter for an arbitrary number of headers by keeping a list of the advertisement prefixes of interest. When the module discovers an advertisement with a matching prefix, it fires an event that the JavaScript layer listens for containing the advertisement in question (Table B.2 in Appendix B enumerates these events and the data passed along). Because we wanted the Android module to be agnostic to any TaleBlazer specific design patterns, the module delegates any decision making as to whether to request the payload for an advertisement to the JavaScript layer. There are certain methods in the Android module that are exposed to the JavaScript layer, and these methods should be functionally equivalent in the iOS module as well. Table B.1 in Appendix B details what these methods are.

Although the devices can only connect to one device at a time through BLE, thanks to the reworking of the module, players no longer have to switch between just sending or just receiving. Once the game has started, players are always ready to receive an incoming message by scanning for advertisements of interest in the background, and likewise players can send BLE messages at arbitrary points during the game.

4.2.2 JavaScript Layer

The JavaScript layer acts as the intermediary between the native module and the rest of the TaleBlazer application. This layer is also responsible for formatting the advertisement headers and payloads that are passed down to the Android module. The JavaScript layer listens for different events fired by the native module, and among these is an event fired whenever the module sees an advertisement with a prefix of interest while scanning for advertisements. The JavaScript layer currently determines whether to request the payload by using higher sequence numbers in the advertisement to denote newer messages and prevent redundant connections to the advertising device. If the sequence number is not higher than the previously seen sequence number for a particular device, there is no new information to be gain, and thus no need to spend time connecting to the advertising device. If an advertisement is new (e.g it is the first time seeing the advertisement of interest) or it contains a higher sequence number, the JavaScript layer makes a call to the native module to request the payload from the advertising device. Once the payload is received, the native module fires another event on the receiving device containing the contents of that payload, which the JavaScript layer also listens for. A different event is fired on the sending device when the payload has been successfully delivered.

4.2.3 TaleBlazer Blocks

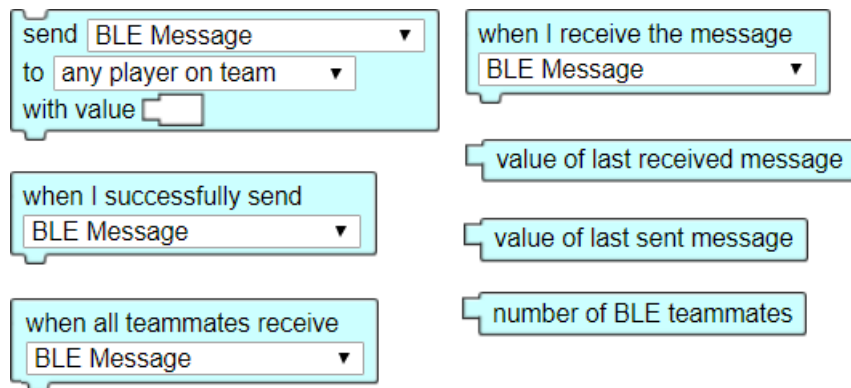


Figure 4-2: The new BLE blocks introduced for the game designer.

In order to give game designers the ability to send and receive messages in a Tale-

Blazer game, we need to provide them with a way to access Bluetooth functionality via blocks. I added seven new blocks to the TaleBlazer Editor that allow game designers to specify when to send BLE messages and what they should contain, as shown in Figure 4-2. There are also blocks that allow the game designer to program behavior in response to players receiving messages or successfully sending messages. For example, if a player has five coins and they send one to another player, we would want the receiving player to perform some actions upon receipt of the coin (such as adding the coin to their inventory and updating a trait that keeps track of their coin count), and likewise we would want the sending player to perform a potentially different set of actions upon successfully sending the coin (such as decreasing the number of coins they have only once the coin has been sent over). The handlers for the events that the JavaScript layer listens for can also execute blocks based on the message received. Chapter 6 describes the BLE blocks for the editor in more detail.

4.2.4 Mobile User Interface

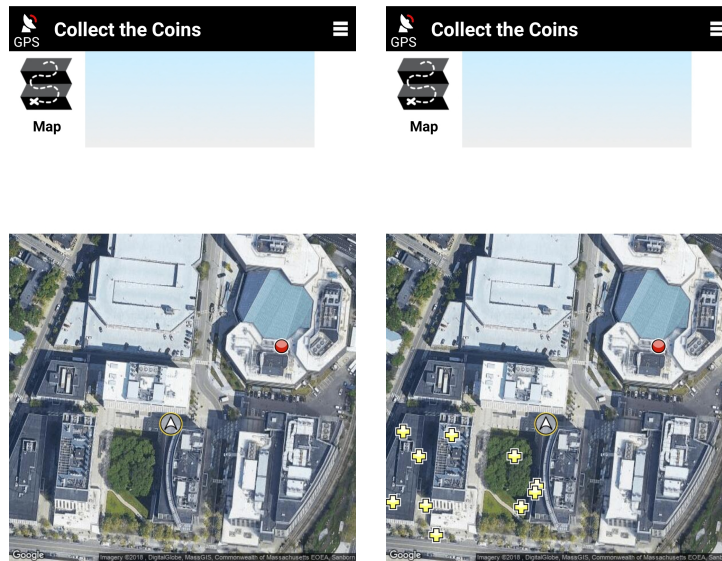


Figure 4-3: The UI before and after a BLE message was received. In this sample game, a teammate has sent this player a BLE message causes agents (yellow pluses) to spawn, and the UI updates the map and agents in response to receiving the message.

The mobile user interface (UI) responds to user input and reflects the state of the game. For example, if a player sends a BLE message to another player and the game designer has programmed the TaleBlazer game such that the receiving player has other agents spawn upon receipt of a BLE message, the UI should update to show that agent, as shown in Figure 4-3. The mobile UI can also invoke the JavaScript API and cause the device to start advertising or scanning for advertisements, such as during team formation, which is described further in Chapter 5.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Using Teams to Accommodate the Field Trip Model

With the given architecture in place for always being ready to receive messages and for being able to send messages at arbitrary times to an arbitrary number of players, we now need a protocol to specify whether we want to send to a specific player, one player on our team, one player not on our team, any one player in the game, all players on our team, or any players not on our team or simply just any players in the game. With this in mind, I designed a protocol and message structure with which a sender can specify whether a message is meant for a specific player or a subset of players playing the same TaleBlazer game.

When scanning for advertisements, Bluetooth devices first see the advertisement header before deciding whether to request the payload. Because I wanted to identify whether an advertisement was for a TaleBlazer message before requesting a payload, I needed to include an identifier in the advertisement. However, to prevent one player from accidentally receiving a message not meant for them, I need to encode within the advertisement a way to let the intended recipient know to request the payload for an advertisement while letting other players know that this message is not meant for them.

BLE advertisements are limited to 20 characters, which was a challenging constraint to work around when designing the advertisement structure. In order to

leverage the simultaneous advertising and scanning that we can now do with BLE, I introduced the concept of creating and joining teams over BLE in TaleBlazer. These teams allow users to listen for any messages that their teammates might be sending to them (which usually should go out to the entire team) while ignoring messages from other teams. Users can choose to either create their own team and have their devices send out advertisements to recruit other players, or they can scan for existing teams and join one of those.

During this team formation phase, the headers for the team formation messages sent out are all of the form:

TBX GAME TID PLID SQN

where the different sections correspond to the TaleBlazer specific prefix, an encoded game ID, a randomly generated team ID, a randomly generated player ID, and a sequence number, respectively. Table A.1 in Appendix A provides further details on each of these components.

There are three different types of team formation messages: *team* messages, *join* messages, and *finalize* messages. Each of these messages has an associated *payload*, which is the data sent by the peripheral to the central upon receiving a connection request. Figure 5-1 outlines the team formation flow, and in the next few sections I go through a typical scenario for players joining and creating teams. In the following walkthrough of team formation, suppose that Alice, Bob, Carol, and Dave all want to play the same TaleBlazer game and wish to form teams to do so.

5.1 Team Leaders Advertise Teams

During team formation, players have the option of choosing to join a team, create a team, or play the game alone. When players begin the team formation phase of a BLE-enabled TaleBlazer game, TaleBlazer Mobile randomly generates a 4-character player ID using 94 of the 95 printable ASCII characters (with one character reserved as a delimiter). When teams are created, TaleBlazer Mobile generates a random 3-character team ID with the same encoding. This encoding gives us $94^4 = 78,074,896$

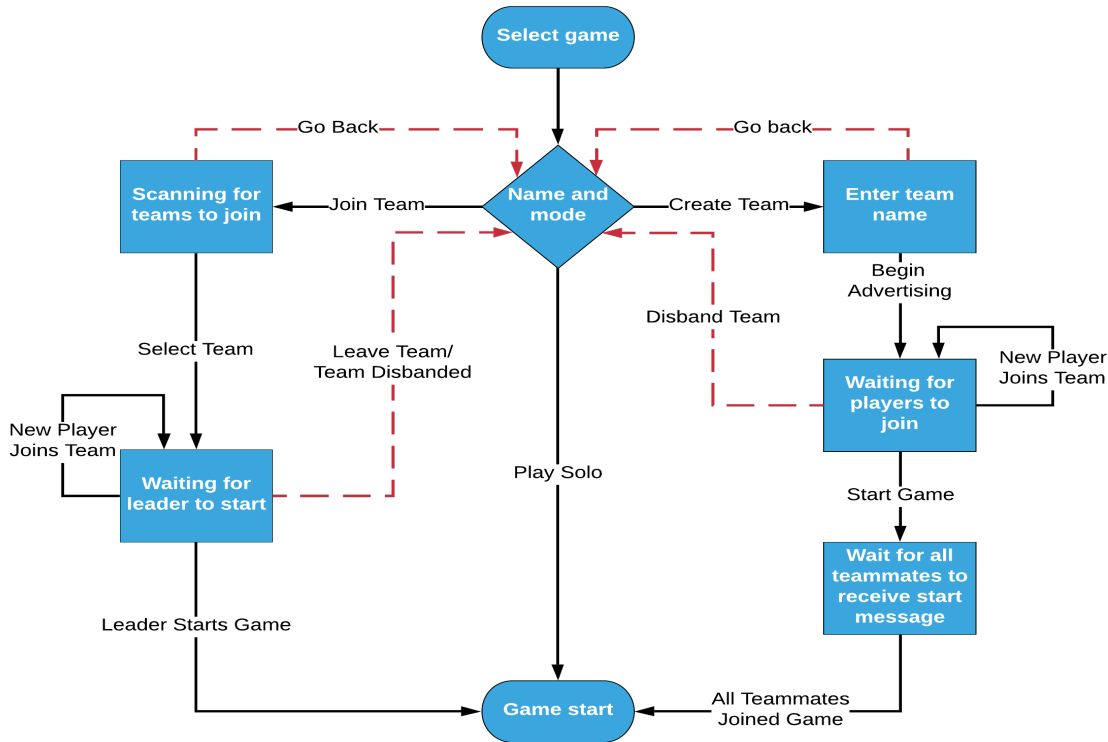


Figure 5-1: The team formation start up flow. A user can choose to create their own team and recruit players, join another team that is recruiting players, or, if the game allows for it, play the game on their own in a team of one.

different player IDs and $94^3 = 830584$ different possible team IDs. With these ranges, even if we had 100 different teams being created for the same TaleBlazer game at the same place at the same time, the chance of two teams out of those 100 teams generating the same team ID is 0.59%.¹ Similarly, even if we had 1000 players playing the same TaleBlazer game in the same location at the same time, the chance of two players generating the same player ID is less than 0.64%.²

To illustrate the team formation flow for players, let's take an example where Alice

¹This is an instance of the Birthday Paradox. To get the probability of any 2 IDs colliding, we first compute the probability of all the IDs being unique, and then subtract that result from 1. In this case, we have $94^3 = 830,584$ possible unique values, 100 different teams leading to $\frac{100 \cdot 99}{2} = 4950$ possible pairings, the chance of 1 unique pairing being $\frac{830583}{830584}$ which implies the chance of 4950 unique pairs is $\frac{830583^{4950}}{830584^{4950}} \approx 99.41\%$. Subtracting this result from 1 gives us our chance of a collision being roughly 0.59%.

²Similar to the previous birthday paradox, this time we use $94^4 = 78,074,896$ possible unique values, $\frac{1000 \cdot 999}{2} = 499,500$ possible pairings, $\frac{78074895}{78074896}$ as the chance of 1 unique pair which leads to a $\frac{78074895^{499500}}{78074896^{499500}} \approx 99.36\%$ chance of all the pairs being unique. Subtracting that from 1 gives us the chance of a player ID collision among 1000 players as 0.64%.

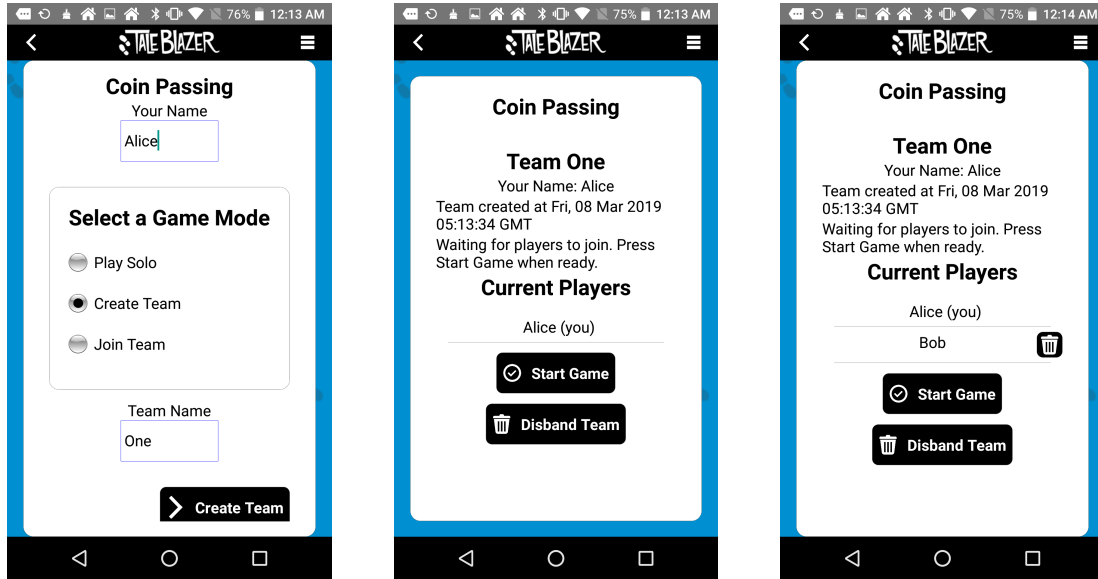


Figure 5-2: The team formation starting UI for creating a team. After choosing to play the game Coin Passing, Alice chooses to create her own team (left) and begin advertising in an effort to recruit other players. After creating a team and giving it a name, the Alice as the team leader then scans for join requests (center). Once Bob joins her team, Alice sees his name on her team roster (right).

and Carol will create their own teams and where Bob and Dave will join Alice’s team. Alice and Carol each begin by selecting the “Create Team” option and giving their respective teams names. Alice and Carol’s devices then each advertise the existence of their teams by sending out *team messages*. Initially, each team contains just the team leader—in this case Alice and Carol separately—and the payload for each team message contains just the team leader’s player ID and name. Bob and Dave, looking to join teams for the same game Alice and Carol want to play, begin to scan for team message advertisements containing a special prefix and the encoded game ID. Figure 5-2 shows what the UI for Alice and Carol looks like when creating a team.

While Bob is searching for a team to join, suppose his device is the first to receive a team advertisement, in this case from Alice’s device. Upon receiving the advertisement, Bob’s device checks that the header contains the prefix Bob is looking for, in this case the prefix denoting a team advertisement. Bob’s device then requests to connect to Alice’s device. Alice’s device sends Bob the team payload with just her player ID and name, and Bob now sees Alice’s team as a potential to join. When Bob

receives Carol’s advertisement, the same process occurs, and Bob then sees Carol’s team as another potential team to join.

Although Alice and Carol’s devices are continuously sending out advertisements, the sequence numbers on their advertisements denote whether there is any new information. The JavaScript BLE layer for TaleBlazer Mobile keeps track of an ID for each player the device has connected to and the most recently received sequence number from that player. The team leader’s device increments the sequence number in the header when new information is available to send in the payload. Upon seeing an advertisement, the team joiners’ devices checks the relevant sequence number. If the sequence number of an advertisement is not higher than that of the most recently received advertisement for a particular player ID, the scanning device does not request the payload from the advertising device and does not request the payload when there is no new information to be gained. In this case, when Bob receives the same advertisements from Alice and Carol again, if there are no updates to the team, the sequence number will not have changed, and Bob will not re-request the team payload. This is an optimization that prevents unnecessary connections and speeds up how quickly other devices receive the most up-to-date information.

5.2 Players Send Join Requests to Team Leaders

Following from the example where Bob has now discovered both Alice’s team and Carol’s team, suppose Bob chooses to join Alice’s team. Bob’s device now begins advertising *join messages* specifically for Alice. While Alice’s device is advertising her team, it is simultaneously listening for join messages meant specifically for her. Alice differentiates between the messages meant for her and those meant for Carol by looking for advertisements containing the join prefix, the encoded ID for the game she wants to play, and the team ID for her team, which with high probability should be different from Carol’s team ID. Upon receiving a join request from Bob, Alice’s device requests to connect to Bob’s device and receives Bob’s name and player ID as the payload from the join message. The JavaScript layer automatically adds Bob

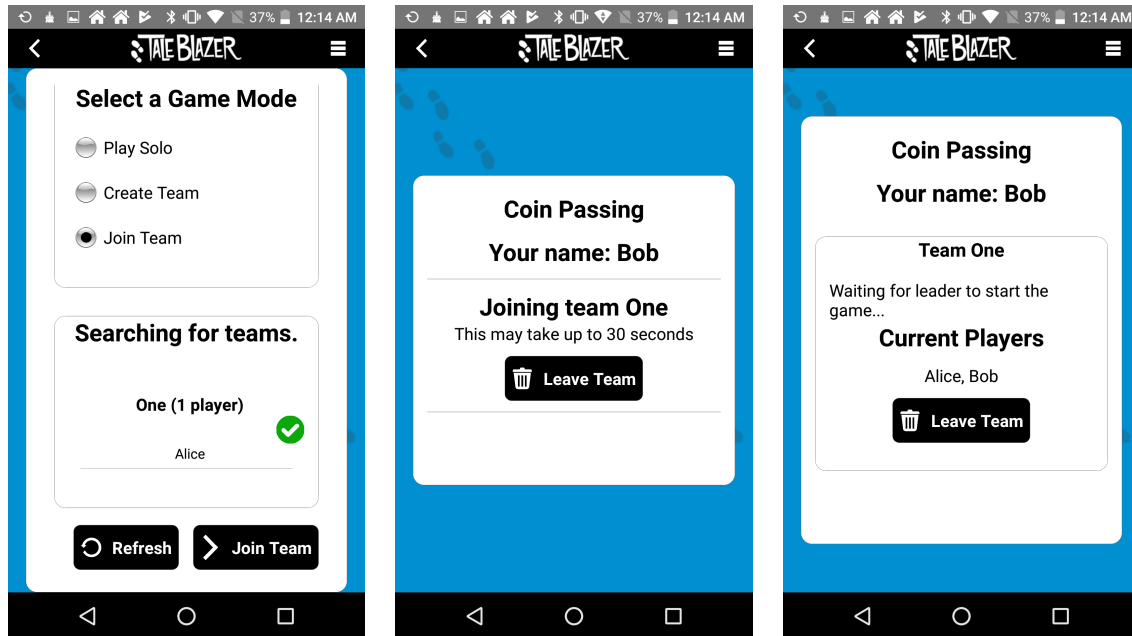


Figure 5-3: The screens that Bob sees when he is joining a team. He first scans for teams, and upon finding a team that he wants to join, selects that team (left). Once Bob presses “Join Team”, he sees a temporary screen while he advertises his join request to Alice (center). Once Bob has joined, he sees the current roster for the team, which updates if team members are added or removed (right).

to Alice’s team, updates her team message payload to include Bob’s player ID and name, and finally updates the sequence number on her advertisement header. Alice then sees that Bob has joined her team on her screen, and Bob sees a different screen signaling that he has joined Alice’s team and is now waiting for Alice to start the game. Figure 5-3 shows examples what Bob sees as he joins Alice’s team..

Having now joined the team by sending Alice his join request, Bob waits and listens for team messages from Alice with updates, such as a new member, or for the finalize message once Alice is ready to start the game. Bob’s device updates the prefixes it is searching for to include the team and finalize prefixes, the game ID, and also the team ID. By scanning for two more specific prefixes, Bob can now ignore any team or finalize messages sent out by other team leaders, like Carol.

Although Bob has joined Alice’s team, Dave has yet to decide which team to join. Because Alice’s device has updated the sequence number on the advertisement header, the next time Dave’s device receives an advertisement from Alice, it will

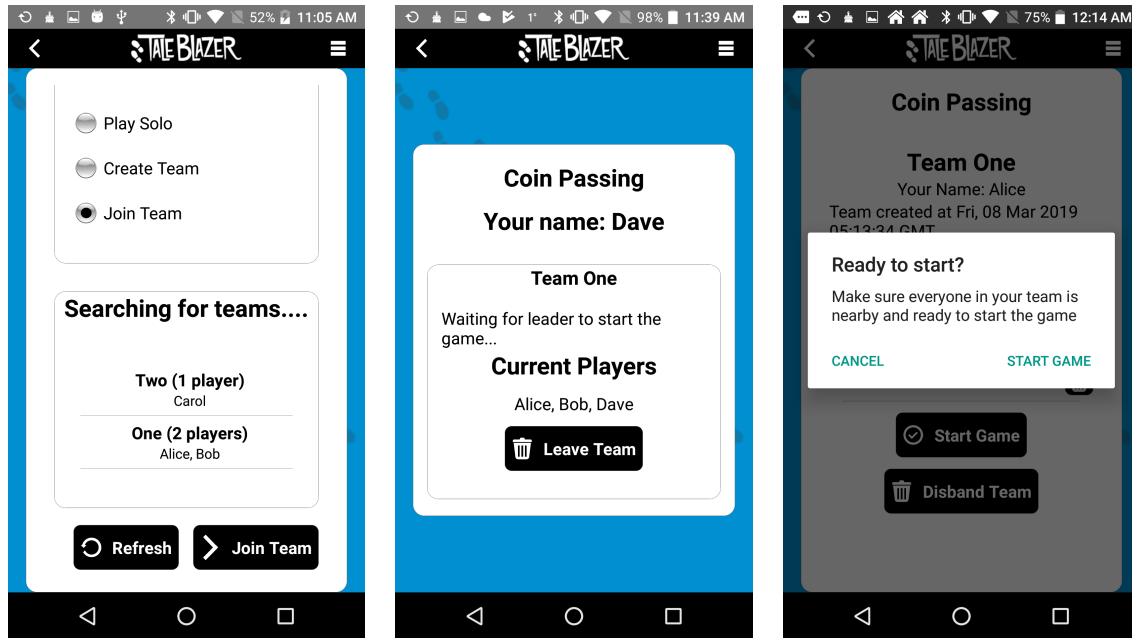


Figure 5-4: The updated list of joinable teams that Dave sees after Bob has joined Alice’s team (left). Once Dave also Joins, Bob and Dave see the roster for the team (center). Once Alice decides to start the game, she sees a confirmation message before her device begins advertising the finalize message (right).

register the higher sequence number and request the payload from Alice’s device, this time receiving Bob’s name and player ID as part of the payload from Alice. Dave decides to join Alice’s team and goes through the same steps that Bob did. Once Dave joins the team, Alice device once again increases the sequence number on her advertisement header such that Bob can also request the updates to the team, in this case Dave’s information. Carol continues to advertise her team messages, but as both Bob and Dave are on Alice’s team, her messages are ignored. Figure 5-4 shows the updated team list that Dave sees and the final team roster. With this finalized roster, TaleBlazer Mobile now knows the names and IDs of all the players on the players team and can send messages specifically to teammates, non-teammates, or other players in the game without worrying about accidentally intercepting messages from another team or having messages meant for the player’s team be sent to another team.

5.3 Team Leader Finalizes Team

Lastly, Alice decides to start the game. After confirming that all players are ready to start the game (see Figure 5-4), Alice's device sends out *finalize messages*, the payload for which is the finalized team that will be playing the game together. This is redundant if all team members have received the most recent team update before the game has started, but in the event that not all team members are up to date, including the finalized team information in the finalize message guarantees that every player will have the same information before starting the game. As Alice's device advertises her finalize message, Alice's teammates connect to Alice's device one by one and begin the game one after the other.

Suppose Alice chooses to finalize the team and start the game before Bob receives the update message that includes Dave's information. Alice advertises her finalize message, and when Bob connects, he requests the payload from Alice. Bob accepts this most recent payload as the final team roster, and with it he receives the team information that includes Dave on his team. Once Bob receives the payload from Alice and updates his team, he can finally begin playing the TaleBlazer game. Once he enters the game, Bob stops listening for team formation messages and begins listening for in-game messages, described further in the next section.

Alice, however, cannot begin her game until all team members have received the finalize message. As Alice is the team leader, it is her responsibility to ensure that all team members have the same information before they enter the game. TaleBlazer Mobile knows how many players are on Alice's team and increments a counter each time a teammate successfully receives the finalize message and begins the game. Once all teammates have received the finalize message, Alice's device can stop advertising the finalize message, begin listening for in-game messages as well, and start the game. Carol, depending on the game, could start the game with a team of 1, or she could wait until someone new joins the game in the hopes of creating a larger team. Table A.2 in Appendix A describes the payloads for the team, join, and finalize messages used for the team formation phase.

5.4 In-game Messages

Once Alice, Bob, and Dave have started playing a game together as members of the same team, they can send each other in-game messages in response to in-game actions as programmed by the game designer. Because all players on the same team now share the same team ID, they can send messages during the game that are exclusively meant for their teammates only without worrying about other players, such as Carol, receiving those messages unintentionally. Using this team ID players can also send messages that are meant specifically for players not on their team. Table A.3 outlines how the advertisement messages are structured depending on the recipient of the in-game message. One difference of note is that in-game messages contain 2 characters used for the ID for a TaleBlazer BLE message (discussed further in Chapter 6). Future TaleBlazer developers can customize the payloads for these in-game messages to be whatever data they define so long as it is under the 512-byte limit. Game designers use TaleBlazer’s BLE blocks to set a part of the payload for these messages, as well as program the what happens when players send or receive in-game message, which I discuss in the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Bluetooth Blocks for the Web Editor

TaleBlazer games are made in the TaleBlazer Editor and played on the TaleBlazer mobile app. In order to give the game designer control over when players can send and receive messages, I created blocks that the game designer can use to trigger sending or to respond to the receipt of a BLE message, shown in Figure 6-1. We wanted the game designers to control when players sent BLE messages, who could receive those messages, what happens when a message is successfully sent, and what happens when a message is received. These blocks all invoke the JavaScript API to perform their different sending and receiving functions, outlines in the sections below.

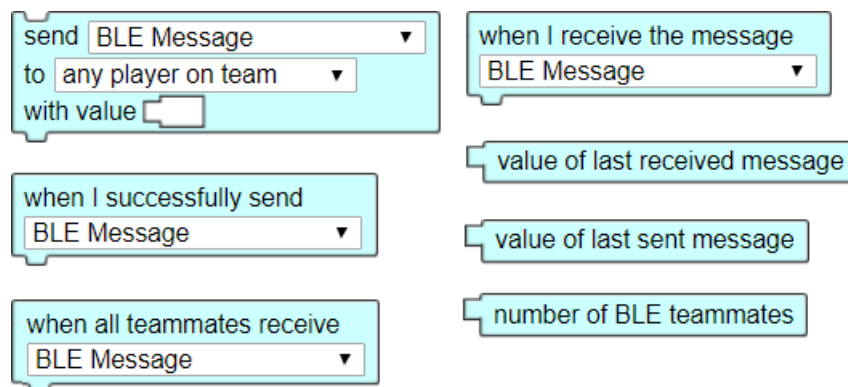


Figure 6-1: The new BLE blocks created for the game designer.

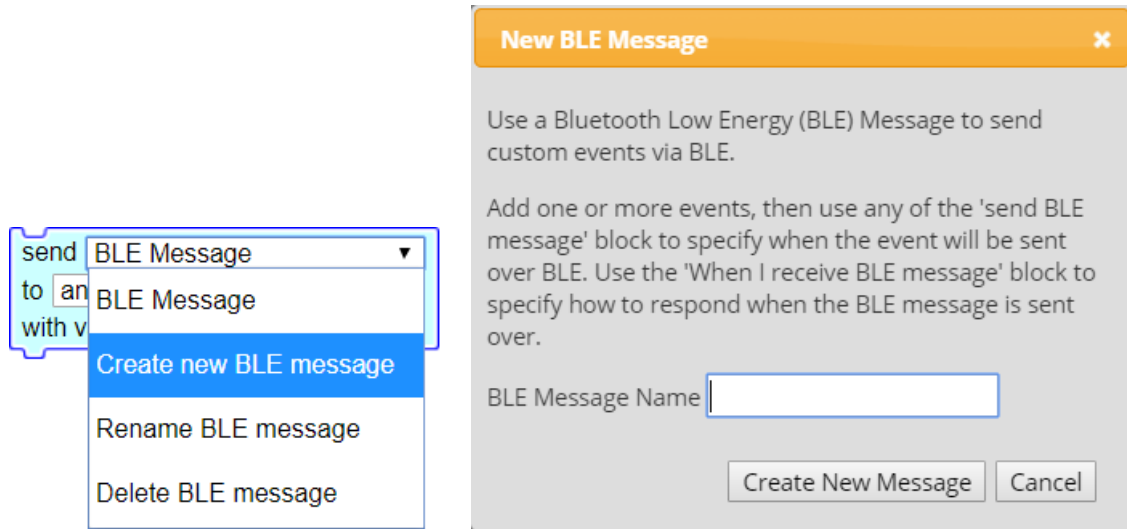


Figure 6-2: The BLE message creation dropdown and modal. Game designers can select the “Create new BLE message” on a Bluetooth block (left) to bring up the BLE message creation modal (right) and create a new BLE message. The modal also checks to see that new message names do not clash with existing messages. The rename and delete options operate on the BLE message that was selected in the before the dropdown was clicked.

6.1 Creating New BLE Messages

TaleBlazer BLE messages are a game designer-facing construct that represent the data sent over BLE. Game designers can configure unique messages for arbitrary in-game events to allow the game to listen for and react differently to those in-game messages. These BLE messages can be sent to a specific recipient or set of recipients with an associated value. The game designer chooses who the recipient for a particular message will be and determines what the associated value for that message will be, and this value is sent as part of the BLE message payload.

Figure 6-2 shows the dropdown and modal used for BLE message creation. We wanted game designers to create new BLE messages without having to navigate away from the editor for the particular agent that they are on. The game starts out containing one default BLE message, but the game designer can create new BLE messages. On the blocks used for sending or receiving, there is a dropdown that determines which BLE message to send or listen for. This dropdown also contains

three non-message options: “Create new BLE message”, “Rename BLE message”, and “Delete BLE message”. Creating or renaming a BLE message displays a modal that gives some information on BLE messages and prompts the game designer for a name for their BLE message.

6.2 Send Message to Recipient with Value

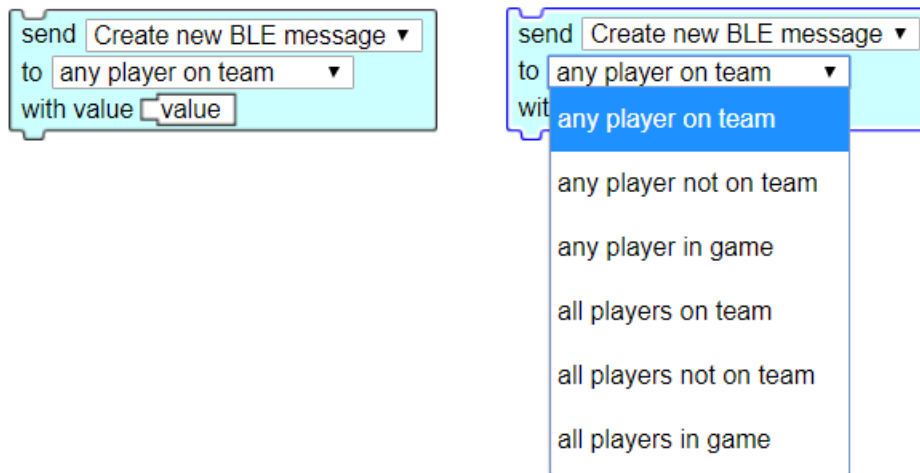


Figure 6-3: The “Send message to player with value” block (left) with the available recipient options (right).

The main block used for sending BLE messages is the “Send message to player with value” block, shown in Figure 6-3. With this block, game designers can select a particular BLE message to send and assign a value with it. For example, a game designer can create a “Heal teammate” message with value 50 to have a player heal a teammate for 50 health points when this block is executed. We wanted to simplify the experience of sending and receiving BLE messages for the game player, so rather than having these blocks block execution of other blocks in game, the blocks will attempt to send the associate BLE message asynchronously to the intended recipients. This way, the player can perform other in-game actions while the sending happens in the background.

Game designers have six options for recipients of a given BLE message: *any player*

on team, any player not on team, any player in game, all players on team, all players not on team, all players in game:

- **Any player on team:** This option lets the player send a BLE message and its associated value to any one player on their team. The teammate who receives the message is arbitrary but guaranteed to be one of the sender's teammates. If a game designer wants to send a BLE message and associated value to all of their teammates, they should use the "all players on team" option.
- **Any player not on team:** This option lets the player send the BLE message and value to any one player not on their team. Similar to the "any player on team" option, the player who receives this message is arbitrary but guaranteed to be someone *not* on the player's team. If a game designer wants to send the message to more than one non-teammate, they should use the "all players not on team" option.
- **Any player in game:** This option lets the player send the BLE message to any player in the game regardless of whether they are a teammate.
- **All players on team:** Game designers can use this option when they want a player to send a message to all of their teammates. With this option, the sending player will continue to send the message after the first teammate has received it until 30 seconds have passed. Every time the message is successfully sent to a teammate, the 30 second timeout is reset. Teammates do not run the risk of receiving a duplicate message as the sequence numbers in the advertisement will prevent teammates from requesting the payload from this message more than once. An event is fired each time a teammate receives the message which can trigger the "When I successfully send message" block. When all teammates have received the message, a different event is fired that can trigger the "When all teammates receive message" block.
- **All players not on team:** Game designers can use this option when they want a player to send a message to players not on their team. Because it is

impossible for a player to know how many players are not on their team, rather than advertise indefinitely until an unknown number of players in the game receive the message, the player will attempt to send the message to any non-teammate for 30 seconds. If a non-teammate successfully receives this message, then the 30 second timeout will be reset.

- **All players in game:** Similar to the “all players not on team” option, with this option the player will advertise to any nearby players playing the same game with a 30 second timeout that refreshes when someone successfully receives the message.

Because this block fires asynchronously, and because the underlying BLE architecture can only advertise one message at a time, it is important that this block not be inside of a loop. Game designers should use the “all players on team”, “all players not on team”, or “all players in game” options if they would like to send a message more than once. It is important to note that this block will only attempt to send the message once and will stop sending when either 30 seconds have passed without a successful send or if the player successfully sent the message to a teammate (in the case of the “any” options).

When the message is successfully sent to another player, the underlying Android module will fire an event signaling a successful send. Game designers can use the “When I successfully send message” block to respond to this event (see Section 6.4). For the “send to all teammates” option, the JavaScript layer keeps track of how many teammates have received the message and then fires a different event that can trigger the “When all teammates receive” block (see Section 6.5).

6.3 When I Receive the Message

The “When I receive the message” block, shown in Figure 6-4, allows the game designer to program what should happen when a player receives a BLE message. This block is a *top block*, a block that cannot be connected to from above. The “When I

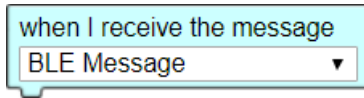


Figure 6-4: The “When I receive the message” block.

receive” message block is instead executed in response to some in-game event, in this case the receipt of a BLE message.

While players do not need to take any action in order to receive BLE messages, the game designer must explicitly use this block in conjunction with the “Send message to recipient with value” block so that TaleBlazer Mobile can scan for the appropriate messages. When a player is attempting to send a message to another player, their device sends an advertisement that encodes the intended recipients and the BLE message being sent (see Table A.3 in Appendix A regarding the format of advertisements sent once the game has started). When this block is included, TaleBlazer Mobile adds a series of advertisement prefixes to request more information from when scanning for in-game messages. When TaleBlazer Mobile receives an advertisement, it checks to see if the message sent out matches any of the messages the device should be listening for. If so, the device requests the payload and receives the value that was associated with the message.

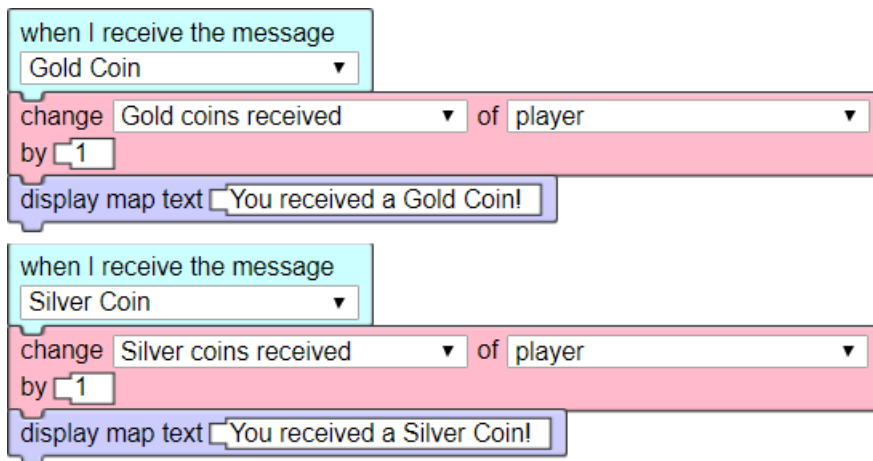


Figure 6-5: Two examples of the “When I receive the message” block.

Once the payload for a particular message has been received, TaleBlazer Mobile then looks for all of the “When I receive the message” blocks in the game that match

the message received and executes those blocks. Figure 6-5 shows an example of two of these blocks. When the player receives a Gold Coin message, TaleBlazer Mobile will execute the blocks for the Gold Coin message but will not execute the blocks for the Silver Coin message. If there are multiple “When I receive blocks” that are associated with the same message, TaleBlazer Mobile will execute all such blocks.

6.4 When I Successfully Send Message

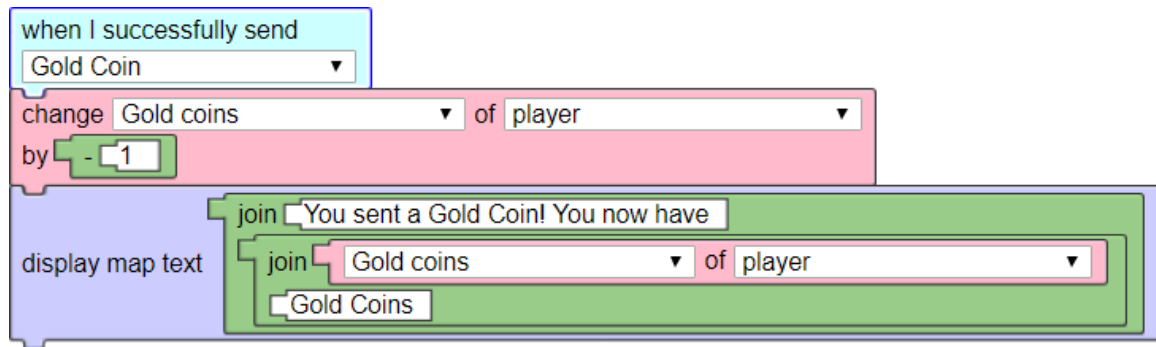


Figure 6-6: An example of the “When I successfully send message” block. In this case, each time the sender receives confirmation that the message was successfully sent, the script above will reduce the amount of gold coins the sender has by 1.

Sometimes a game designer will want a block script to execute only when a message has been successfully sent. For example, if a player is attempting to send coins or another limited resource to a teammate, the game designer might want the coin count on the sender’s side to decrease only when the coin has been successfully sent to the teammate. For a scenario like this, the game designer can use the “When I successfully send message” block, as show in Figure 6-6. When a sender’s device receives confirmation that their BLE message payload has been read, the Android module fires an event that will trigger this block. This event will fire each time that the payload for a given message is read, and as such the blocks associated blocks will be executed each time the event fires. As is the case with the “When I receive message” block, if there are multiple copies of this block listening for the successful sending of the same BLE message, then they will all be executed each time that message is sent.

6.5 When All Teammates Receive

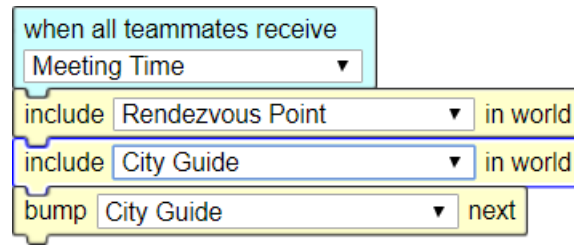


Figure 6-7: An example of the “When all teammates receive” block. Here, the Rendezvous Point will only be included in the game world once all teammates have received the message specifying a meeting time.

The “When all teammates receive message” block is used in conjunction with the “all players on team” option in the “Send message to recipient with value” block. If a player was sending a message to all of their teammates, once TaleBlazer Mobile receives confirmation that all the players on the team received the message the sender was advertising, it fires an event that will execute this block. Figure 6-7 shows an example of this block. This block can be used to receive confirmation that all teammates have received some piece of information, like a meeting time or password, that they might all benefit from.

6.6 Value of Last Message Received and Sent

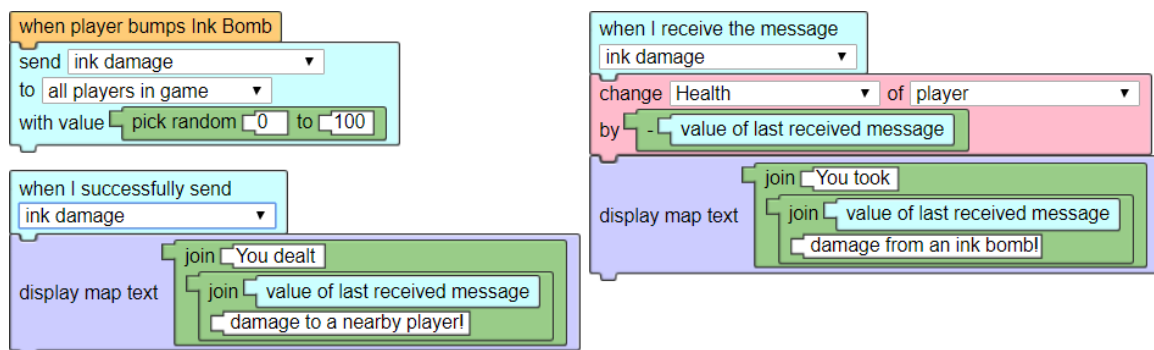


Figure 6-8: Sample usage of the “Value of last message received” and “Value of last message sent” blocks. With this, the receiving player can access the value that was sent along with the BLE message, and the sending player can access the value of the last message sent once they received confirmation of the message being sent.

The “Value of last message received” block allows game designers to write block scripts that use the value associated with a received BLE message. For example, a player might send out a message that deals a random amount of damage to nearby players. Game designers can use the “Value of last message received” block to access the random value and act upon it. Similarly, game designers can also use the “Value of last message sent” to access the value of the last sent message. as Figure 6-8 shows sample usage of both of these blocks.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Future Work

Having a protocol for sending BLE messages to a specific player or set of players that is built on top of a modular infrastructure opens up the possibility for more multiplayer features in TaleBlazer. The following sections discuss some suggestions for future work that TaleBlazer developers can use for extending the current BLE functionality.

7.1 iOS Module

Now that the JavaScript layer handles the logistics for how BLE advertisements and payloads are structured, the underlying native modules no longer need any knowledge of TaleBlazer specific features. The Android module now provides an API for setting the advertisement to send out and its associated payload as well as a way to specify what ads to filter for. In order for the iOS native module to mirror the Android module as closely as possible (and to provide to the JavaScript layer the expected functionality), an iOS module needs to be written to implement the same interface and follow the same event flow that the Android module uses:

- On the sender, set the advertisement and payload to send out, and fire the appropriate event to let the JavaScript layer know that sending was successful.
- On the receiver, set the list of prefixes to scan for. Whenever the module sees

an advertisement, iterate through the array of prefixes to scan for and check for any prefix matches. If a prefix match is found, fire the appropriate event and let the JavaScript layer decide whether to connect to the advertising device.

- On the receiver, if it successfully receives the payload from the advertiser, fire the appropriate event that contains the payload so that the JavaScript layer can execute the necessary blocks.
- On the receiver, if there is some error when attempting to receive the payload, fire the appropriate event such that the JavaScript layer can retry the connection.

The appropriate events and their contents can be found in Table B.2 in Appendix B.

7.2 Team Management

There is currently no way for a team leader to manage their team once the game has started. However, the messaging protocol supports the addition of a new player during the game. By providing a way for the team leader to advertise team messages like during the team formation phase, other players who are just starting the team formation phase will be able to see and join the team leader's team. This would require that team members also scan for update messages during the game, something that can easily be done by adding another prefix to scan for to the list of advertisements of interest.

Another way to possibly send team updates once the game has started could be to use a different GATT service for team updates. Currently, whenever a device requests a payload from an advertising BLE device, the receiver iterates through a list of GATT services that the advertiser exposes. Each of these services has a GATT characteristic, and there is one service that contains the BLE message payload as its GATT characteristic. A future implementation of TaleBlazer could use separate services or characteristics to advertise different data like updates to the team or the payload for a BLE message. Keeping team update information in a communication

channel separate from the one used for sending in-game messages would allow us to handle these two types of messages differently.

The team leader is mostly needed during team formation, but once the team has been made, it could be possible for any team member to play the role of the leader and add another teammate as all team members should have the same information about their team once the game has started. However, one critical concern is that if teams are allowed to change mid-game, there is no way to guarantee that all players on the team are fully up-to-date with the new team roster. Another concern is that if there are certain pieces of information that all players must have when the new player joins, such as meeting time or rendezvous point or other agent providing similar information, the new player should also receive those pieces of information, though it is unclear how this would affect the game narrative for the joining player. This becomes a bit trickier if having a player join mid-game would break the narrative of the game, such as if a team of adventurers exploring an underground cave suddenly gain another team member. The game designer should ideally have control over whether to allow scenarios like this.

Currently, players can leave a team freely during the team formation phase, but once the game has started they cannot leave the team unless they restart the game. If a player leaves the team during team formation, there is no way for a team leader to know that that player has left. The team leader can, however, manually remove the player from the team, as shown in Figure 7-1. If a player receives a team update message from their team leader that no longer contains that player, TaleBlazer Mobile assumes that the player has been removed from the team and brings that player back to the beginning of the team formation UI, as shown in Figure 7-2.

Conversely, a team leader can disband their team if they want to create a different team or join someone else's team entirely. Figure 7-3 shows the confirmation screens players will see when attempting to leave a team or disband the team. However, if the team leader chooses to disband their team, there is no way for the team members to automatically know that the team has been disbanded, and they must each manually leave the team.

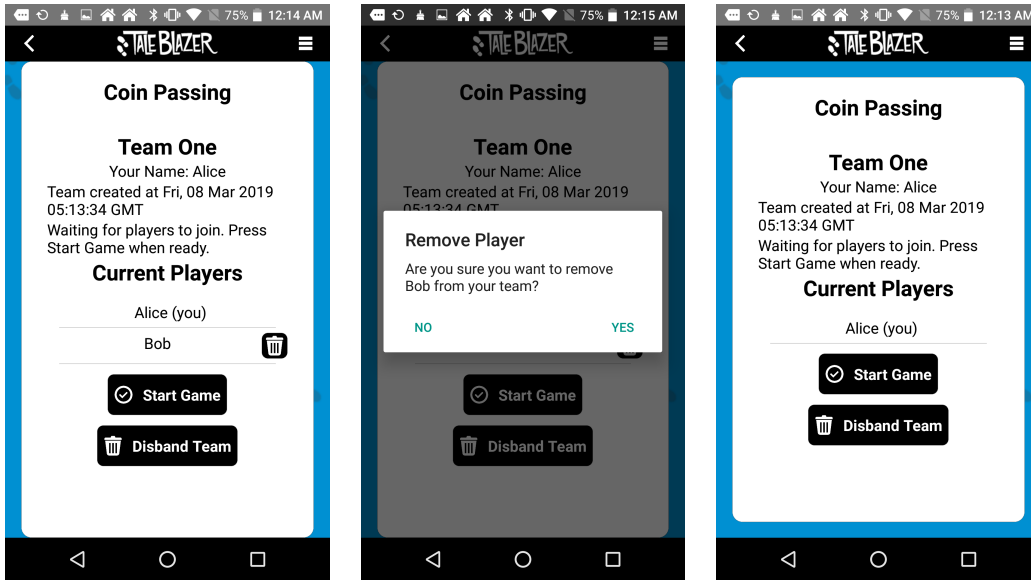


Figure 7-1: The UI Alice sees for removing a team member during team formation. When Alice clicks on the button next to Bob’s name, she sees a message asking her to confirm Bob’s removal, after which Bob has been removed from the team and Alice begins advertising the updated team roster.

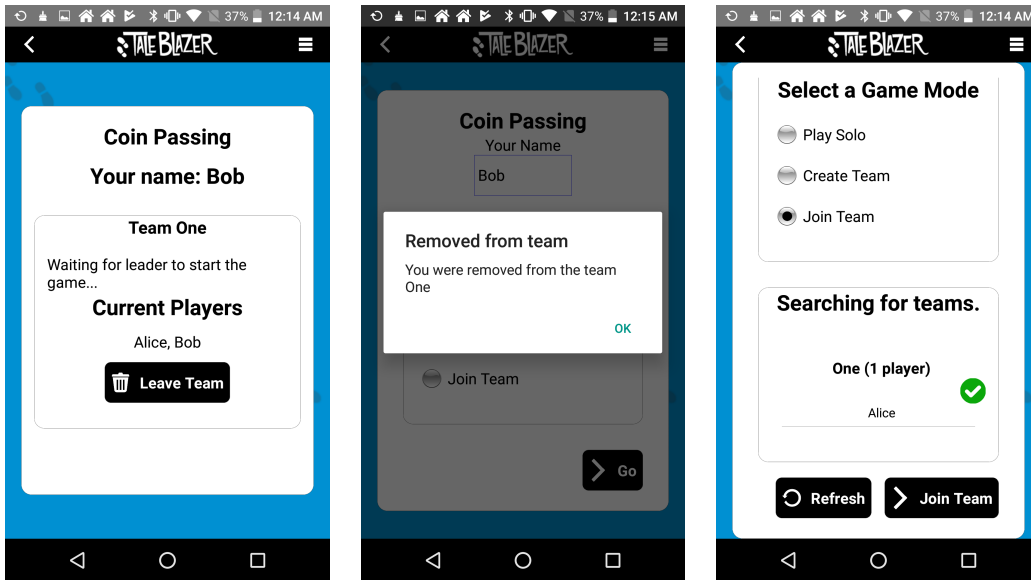


Figure 7-2: The UI Bob sees when he has been removed from the team. When scanning for teams to join, Bob can still see Alice’s team and join again.

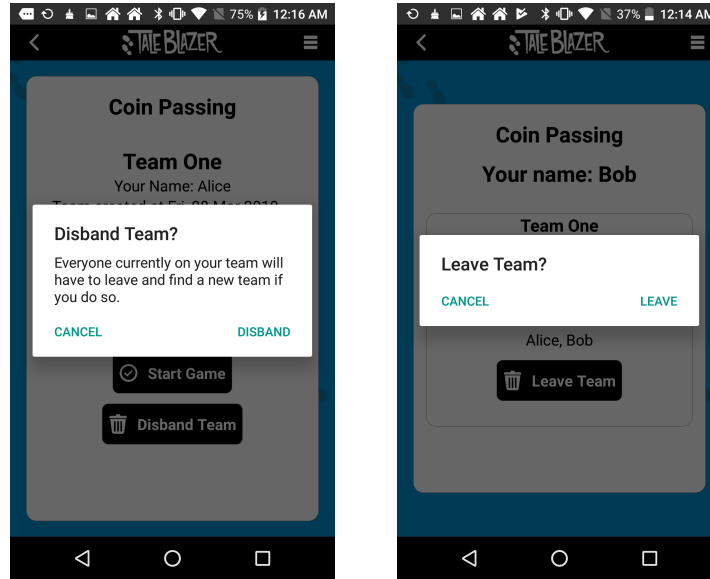


Figure 7-3: The confirmation dialog the team leader sees if they wish to disband a team (left) and the dialog that a player sees if they wish to leave a team (right).

7.3 On Timeout Block

Currently, my implementation does not allow for a way to notify the game designer properly if a message fails to send at all. One type of block that could be added in the future could be a “If no one receives the message” block that would allow the game designer to alert the player if message sending was unsuccessful for some reason (e.g. no more recipients nearby who haven’t received the message, no eligible recipients). This could be implemented similar to the “When I receive the message” blocks in that the blocks can be searched for and executed by TaleBlazer Mobile if sending times out. This would function similarly to the “on timeout” socket in Spitzer’s blocks (see Figure 3-1) and would trigger after the 30 seconds sending timeout.

Along with this, game designers might want to have a way to configure how long players should attempt to send a BLE message before the “If no one receives the message” block executes. There is a customizable parameter for the BLE timeouts in TaleBlazer Mobile, but there is currently no interface through which the game designer can manipulate said timeout.

7.4 Dynamically Changing Messages to Scan for

Currently, TaleBlazer devices scan for all messages associated with the “When I receive message” block. If there are multiple “When I receive message” blocks, the game will be listening for all of those messages simultaneously. Game designers might want to dynamically switch when players are listening for particular message or set of message rather than have them listen for all game messages at once. Allowing a game designer to change what messages a player is scanning for as a response to in-game events might reduce the risk that a player receives and acts upon a BLE message that they were not supposed to receive until later in the game.

7.5 Sending Large BLE Messages

Although we now have the ability to send BLE messages to specific subsets of players, the current blocks do not offer to ability to send agents through TaleBlazer, arguably one of the more important functions of BLE for TaleBlazer. The protocol, however, still supports sending agents in the same way that BLE messages can be sent. Rather than send a message with a value, a player could send an agent and send its traits and description as the payload. In fact, this is what Spitzer had done before, and could be done still, except sending no longer pauses other functions of the game as the previous blocks did. The “When I successfully send” top block would serve the same purpose as the “on success” socket in the old blocks, and the “If no one receives the message” block mentioned above would serve the same purpose as the “on timeout” socket.

The issue of sending large amounts of data over BLE, be they messages or agents, is still present with the current protocol. The maximum length of an attribute value in the Bluetooth Core 4.2 specification is 512 bytes [14]. This means that our message and agent payloads cannot exceed those 512 bytes. It should be possible to split larger payloads over multiple sends by breaking them down it into 512 byte pieces, but even this could be problematic for payloads as small as a few megabytes. A report by

design firm Punch Through showed that theoretically, under the best conditions, the maximum throughput for BLE on Android would be 16 kilobytes per second [15]. Suppose an average photo taken is roughly 8 megabytes in size. Even at 16 kilobytes per second, it would take $8 \text{ MB} \cdot \frac{1 \text{ s}}{16 \text{ KB}} \cdot \frac{1 \text{ minute}}{60 \text{ seconds}} \approx 8.33$ minutes to transfer the photo over BLE. For payloads less than 1 megabyte, which should encompass most agent text descriptions, this should not take much longer than a minute to send, but given the slow throughput even under the best conditions for BLE, it would be worth considering other ways to send very large agents over BLE. Compressing large images would also significantly reduce the time needed to send the image over BLE. Alternatively, future TaleBlazer developers could also investigate functionality to share large media like photos using a combination of BLE and a Wi-Fi connection if the latter is available to upload the media to a server and then pass along to the receiver the ID on the server of the asset to receive.

7.6 Two-Way Communication

Currently, BLE communications in TaleBlazer are one-way interactions: a sender sends to a receiver, but the receiver does not send anything back in the same connection. If the receiver wanted to send a BLE message back to the sender, they would need to advertise a new message themselves. One way in which this problem arises is when there is an error with the receiver receiving the payload. Sometimes, the method the Android module uses to send data from the sender to the receiver will return a success status to the sender even though there might have been an error on the receiver's side with receiving the payload (these errors are unfortunately not well-documented).

However, it might be possible to have a receiver respond to the sender in the same transaction. On Android, there is a method `BluetoothGatt.writeCharacteristic` that the receiver's device can use to write to a characteristic on the sender, which fires the `BluetoothGattServerCallback.onCharacteristicWriteRequest` that the sender's device can use to respond to the characteristic write. Using these two meth-

ods, it could be possible to use a separate characteristic in BLE as an acknowledgment characteristic that the receiver’s device can write to if it successfully reads the payload from the sender’s device. If so, the sender can fire the appropriate event to let the JavaScript layer that sending was successful. If the receiver encounters an error during the receipt of the payload, they can write an error status to that characteristic and request the payload in the same transaction without the sender triggering the “When I successfully send” blocks until there is confirmation that the payload was successfully sent.

This paradigm of writing to the sender’s device while receiving could also be used to implement a trade mechanic in TaleBlazer. Rather than simply sending and receiving messages and agents, players might also want to trade agents during the same interaction rather than have the trade be split over two separate interactions. Although the exact interface for the players would need some thought as would the way that the game designer would program a trade interaction, it might be possible to send data in both directions during the same transaction. Further research would need to be done to determine whether the same methods are available and called at the same time on iOS.

7.7 Sending to a Specific Player

The BLE protocol for TaleBlazer supports designating a specific player as the recipient of a specific message, which is crucial for implementing the aforementioned trade mechanic. There could either be a block that the game designer can use such that a player can choose who to send a particular agent to, a separate game tab in which the player can see choose a player from their team to begin a targeted send to, or something different entirely that would allow a player to first choose an agent to send and then the recipient (similar to how on mobile phones users first choose the media they would like to send, like a YouTube video, and then choose the recipient).

In order to choose a specific player to send data to, the sending player must have interacted with that player so that they can know the recipient’s player ID. For players

on the same team, TaleBlazer Mobile already keeps track of the current teammates, their names, and player IDs, which were all sent by the team leader to all teammates before the game started. There would need to be a way for players to receive maintain information about other players in the game not on their team. Although TaleBlazer Mobile can send to players specifically not on the sender's team, it does not keep a record of who these players are the same way it does for teammates. However, the protocol can be modified such that the sender's name is sent as part of all the payloads and TaleBlazer Mobile can keep a record of all the players one has previously interacted with, regardless of whether they are on the same team. Alternatively, there could also be another message type introduced in which the senders of that message advertise themselves as available for a trade, similar to how team leaders advertise themselves as accepting players during team formation.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 8

Contributions and Conclusion

TaleBlazer is a location-based augmented-reality game platform in which game designers create mobile games centered around visiting and interacting with objects and places in the real world. Many of these games were designed primarily as single-player experiences, and there were many games that would have benefited from having a way to communicate with other players in game. Previous usage of BLE in TaleBlazer to send agents synchronously provided a first step in using BLE as a way to communicate with other players in the game. We wanted to determine what types of mechanics would be feasible to implement using BLE, and we also wanted to provide game designers with a way to utilize those functions in their games.

In the work for this thesis I:

- introduced a robust and flexible protocol for sending data to a specific player or subset of players over BLE
- leveraged teams to enable communication over BLE that work under the field trip model, in which several groups of players playing the same game in the same location can communicate with their teammates without worrying about messages being accidentally intercepted
- updated the previous Android module to be more flexible and agnostic to TaleBlazer-specific data

- modularized the existing BLE architecture to simplify future development
- provided an API in TaleBlazer for using these new BLE features
- created a set of blocks to give game designers access to these new BLE features.

The new BLE protocol paves the way for many new possible multiplayer mechanics in TaleBlazer, which can encourage players to work together, interact with each other through the game, and provide a richer experience for the player that cannot be easily obtained in a single-player game. These features will open the door for other game mechanics in TaleBlazer that do not have to be restricted to a particular location, thus taking a step closer to the dream of building a platform that enables game designers to create augmented reality games that can be played with anyone anywhere.

Appendix A

Message Header Structures

Table A.1: The different parts of a team formation header. The headers each contain the TaleBlazer prefix, a message type, the encoded game ID, the team ID, the player ID, and the encoded sequence number, and are all of the form **TBX GAME TID PLID SQN**

TBT	The prefix denoting a team message. Team leaders send advertisements with this prefix when recruiting new team members, and players seeking to join a team scan for this advertisement.
TBJ	The prefix denoting a join message. Team leaders scan for this advertisement, and team members, after discovering a team, send an advertisement to the team leader with this prefix to signal their intent to join the team.
TBF	The prefix denoting a finalize message. The team leader sends an advertisement with this prefix when they are starting the game.
GAME	4-character ASCII encoding of the game ID. This allows TaleBlazer players to ignore advertisements from other players who might be playing a different Bluetooth-enabled TaleBlazer game.
TID	3 ASCII characters representing the team ID. This ID is randomly generated when a player creates a new team.
PLID	4 ASCII characters representing the player ID. This ID is randomly generated each time a player starts the game.
SQN	3 character ASCII encoding of the monotonically increasing sequence number for an advertisement. Higher sequence numbers denote newer messages.

Table A.2: The payloads for the team, join, and finalize messages used during team formation. TBT, TBJ, and TBF are the prefixes that identify each of the messages types in the header, with TB being used as a TaleBlazer-specific prefix. The rest of the advertisement header retains the same structure of TBX GAME TID PLID SQN.

Message Type	Payload	Example
TBT	The <i>team message</i> , which includes the current team, denoted as the 4-character player ID, followed by the variable-length player name. Player entries are delimited by the left brace [.	#2EPALice[-+_ *Bob
TBJ	The <i>join message</i> , which includes the ID and name of the player requesting to join the team.	-+_ *Bob
TBF	The <i>finalize message</i> , which includes the finalized team and the signal to start the game. The team members are delimited the same way the as in payload for the team message.	#2EPALice[-+_ *Bob

Table A.3: The header structures for in-game messages based on the recipient. The X in the prefix can be either an M or an A depending on whether the user sends an in-game message or agent, respectively. TID is the 3-character team ID, EI is the 2-character entity ID that identifies the in-game message or agent being sent, GAME is the 4-character ASCII encoding of the game ID, FROM is the 4-character player ID of the message sender, and SQN is the 3-character ASCII encoding of the sequence number.

Recipient	Header Structure
Any/all teammate(s)	TBX TID EI FROM SQN
Any/all non-teammate(s)	TBX GAME TID EI FROM SQN
Any/all players in game	TBX GAME EI FROM SQN
Specific player on team	TBX TID TOID EI FROM SQN

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B

API and Events

Table B.1: The API methods exposed by the native module and invoked by the JavaScript layer

Method	Description
<code>getBLESupported()</code>	Returns whether BLE is supported on the device.
<code>getBLEOn()</code>	Returns whether BLE is turned on for the device.
<code>enableBluetooth()</code>	Attempts to turn on device BLE managers if possible.
<code>startTransmit(Object)</code>	Takes in an Object of the form <code>{header: "header", payload: "payload"}</code> . Starts advertising the given header and sends the associated payload when a central device connects. header must be at most 20 bytes, and payload must be at most 512 bytes.
<code>stopAdvertise()</code>	Stops advertising the current BLE message. Does not stop scanning for advertisements.
<code>setWantedAds(Object)</code>	Takes in an object of the form <code>{ads: [ads]}</code> . Clears the current list of ad prefixes to filter and replaces it with <code>[ads]</code> , which is a list of strings containing the prefixes to filter for.
<code>startReceive()</code>	Begins scanning for advertisements. Should be called after <code>setWantedAds</code> such that there are specific prefixes to filter for.
<code>stopDiscover()</code>	Stops the current scan for advertisements, if any.
<code>connectBLE(deviceMac, currentAd)</code>	Takes in two strings: <code>deviceMac</code> which is the MAC address of the BLE device to connect to, and <code>currentAd</code> , which is the advertisement whose payload to attempt to receive.
<code>stopAll()</code>	Stops both advertising and discovery, and attempts to stop any ongoing BLE connection, if applicable.

Table B.2: The events the Android module can fire in response to different BLE connection events.

Event	Description
onBLEAdReceived	Fired when an advertisement matches a prefix of interest. The JavaScript layer listens for this event, then decides whether to request the payload from the device that sent the advertisement that caused this event to fire. Contains {deviceMac: “address”, ad: “advertisement”}, the MAC address of the device to connect to and the advertisement itself.
onBLEPayloadReceived	Fired after the JavaScript layer decides that the central should connect to the peripheral and when the central reads the payload from the peripheral for a given advertisement. Contains {ad: “advertisement”, payload: “payload”}, the advertisement acted upon and the corresponding payload.
onBLEDeviceDisconnected	Fired on the peripheral device when a central device has successfully read from the peripheral. Returns {deviceMac: “address”, payload: “payload”}, the MAC address of the central that read from the peripheral and the payload read.
onBLEDisconnectError	Fired when an error occurred when attempting to read from a peripheral. Returns {errorCode: “code”, ad: “advertisement”}, the error code[16] received and the advertisement itself. The ad is used to update the cache of advertisements in the JavaScript layer so that the central can retry to receive the same payload from the peripheral.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

- [1] MIT Scheller Teacher Education Program. TaleBlazer. <https://taleblazer.org>. Accessed: 2018-07-11.
- [2] MIT Lifelong Kindergarten Group. Scratch. <https://scratch.mit.edu>. Accessed: 2018-07-11.
- [3] MIT App Inventor. <https://appinventor.mit.edu/>. Accessed: 2018-07-11.
- [4] Tanya X. Liu. TaleBlazer multiplayer: Expanding multiplayer functionality for meaningful location-based AR games. Master's thesis, Massachusetts Institute of Technology, 2013.
- [5] Arjun V. Narayanan. Improving the TaleBlazer multiplayer platform. Master's thesis, Massachusetts Institute of Technology, 2016.
- [6] Apple. Core NFC. <https://developer.apple.com/documentation/corenfc/>. Accessed 02-01-2019.
- [7] Wi-Fi Alliance. Wi-Fi Direct. <https://www.wi-fi.org/discover-wi-fi/wi-fi-direct>. Accessed 02-01-2019.
- [8] DENSO WAVE INCORPORATED. Information capacity and versions of QR code. <https://www.qrcode.com/en/about/version.html>. Accessed 02-01-2019.
- [9] William Spitzer. Using bluetooth to enable multi-user communications in TaleBlazer. Master's thesis, Massachusetts Institute of Technology, 2017.
- [10] Bluetooth SIG. GATT overview. <https://www.bluetooth.com/specifications/gatt/generic-attributes-overview>. Accessed: 2018-07-11.
- [11] Google. Bluetooth low energy overview. <https://developer.android.com/guide/topics/connectivity/bluetooth-le>. Accessed: 2019-02-01.
- [12] Nick Koudas. iBeacon and battery drain on phones: A technical report. Technical report, Aislelabs, 2014.

- [13] Kaisen Lin, Aman Kansal, Dimitrios Lymberopoulos, and Feng Zhao. Energy-accuracy trade-off for continuous mobile device location. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services, MobiSys '10*, pages 285–298, New York, NY, USA, 2010. ACM.
- [14] Bluetooth SIG. Selecting a Bluetooth Microcontroller – A Developer’s Perspective. <https://blog.bluetooth.com/selecting-bluetooth-microcontroller>. Accessed: 2018-12-01.
- [15] Punch Through Design, LLC. Maximizing BLE Throughput on iOS and Android. <https://punchthrough.com/pt-blog-post/maximizing-ble-throughput-on-ios-and-android/>. Accessed 2019-01-01.
- [16] Paul Gavrikov. Android BLE error/status codes explained. <https://allmydroids.blogspot.com/2015/06/android-ble-error-status-codes-explained.html>. Accessed: 2018-07-11.