AVENTURA

Situational Awareness

Aventura: Bluetooth LE Beacon Reader

Loki Hondorf
Conner Taylor
Marcus Tonsmann

20 June 2016

Table of Contents

Introduction

Aventura is a software company that provides healthcare providers with a secure, personalized virtual desktop. Aventura's software allows clinicians to log into endpoint devices by simply tapping their badge on a sensor. The company uses what it calls "awareness" to serve different personalized desktops depending on the user, patient, and location within the hospital.

To streamline the login process further, Aventura has tasked our team with developing a system which uses Bluetooth Low Energy (LE) beacons to begin the login process when the user approaches an endpoint device. Clinicians using this system would be able to log into their personalized desktop simply by walking into a patient room, reducing their time spent using a computer and allowing them to spend more time providing care.

Because Aventura wishes to support additional login systems in the future, we were tasked with implementing the login system as a generic token provider. This allows Aventura to extend their client to interface with a variety of token providers, such as Near Field Communication (NFC) or biometric systems.

Requirements

The Bluetooth login system must meet the following functional requirements:

1. The program must read and store Bluetooth LE signals from beacons carried by hospital staff.
2. The program must determine whether a beacon is close enough to an endpoint device to begin the login process.
3. The program must communicate the user's unique identifier to the Aventura software when a beacon comes in range of an endpoint device.

In addition to our functional requirements, the project must meet the following non-functional requirements:

1. The program must be developed using Microsoft Visual Studio 2015.
    a. This was done to support the Universal Windows Beacon Library, which targeted the (asp).NET core framework.
    b. Although the Library was eventually ported to the (asp).NET Framework 4.5 support was continued for VS2015.
2. The program must be compatible with projects developed in Visual Studio 2013.
    a. This was done in order for our project to integrate with the existing Aventura architectura.
3. The program must be a Windows application (targeting the .NET Framework) written in C#.

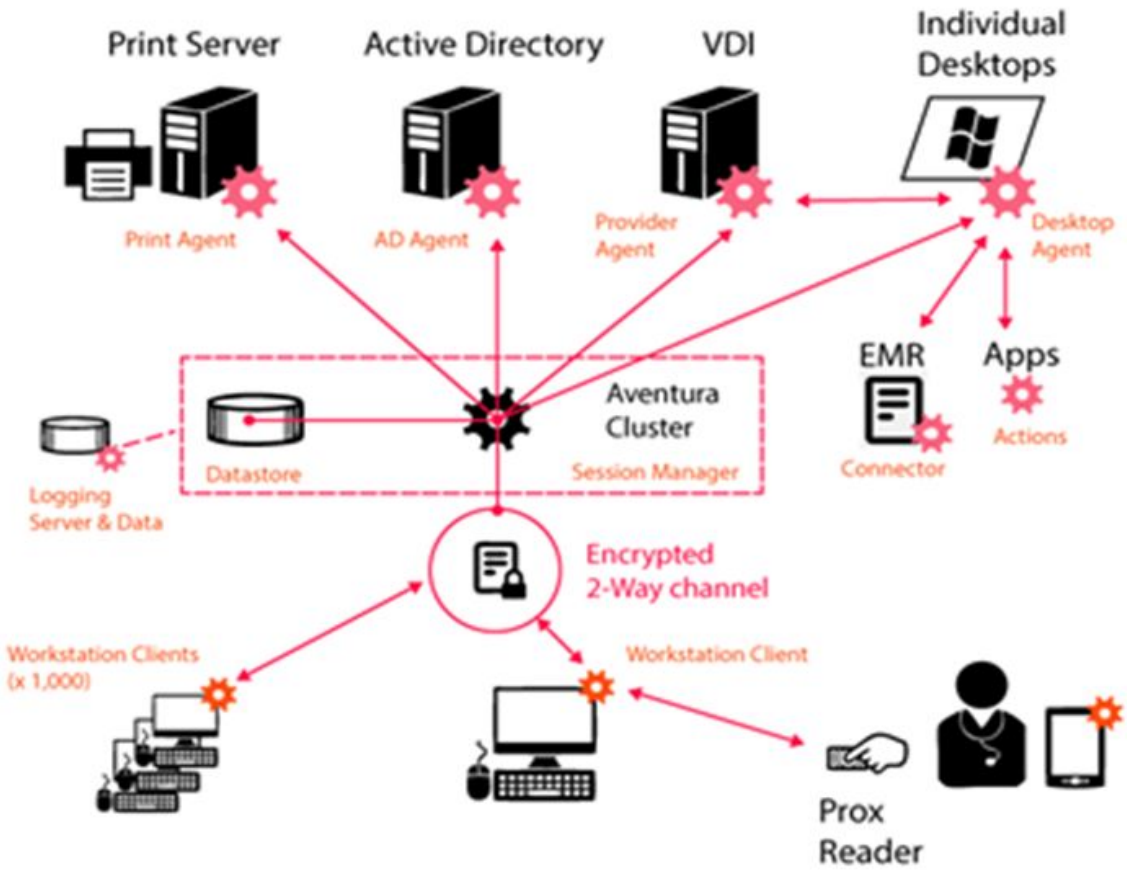
    a. This program needed to integrate with the existing UWP Beacon Library, which was written in C#, with minimal interop code.
    b. The Windows app was also designed to target .NET Framework 4.5 in order to be supported on a wider range of platforms.
4. The program must be integrated with unmanaged code written in C++.
    a. This was done in order to be compliant with the existing code base provided by Aventura.
5. The program must be implemented as a generic token provider.
    a. In order to provide functionality to future hardware token providers, a token provider interface was created.
    b. This interface is in turn implemented by our project.

System Architecture

Our project is contained entirely within Aventura's workstation client (or "soft client"), which runs on every endpoint device in the hospital. The client interfaces with available token providers, including the existing Radio-frequency identification (RFID) scanner and our Bluetooth LE listener, to determine whether a user should be logged in or out. When a login or logout decision is made, the workstation client displays a dialog which prompts the user for their passphrase. The client then sends a signal to the Aventura session manager to begin the login or logout process (Figure 1).

The signal passed to the session manager contains the unique ID (UID) of the user to be logged in or out. Upon receiving a signal, the session manager queries the user database to find the session associated with the given ID. If the user is not logged onto any endpoints, the session manager attempts to authenticate the user, initiate the login process, and serve the user's personalized desktop. If instead the user is currently logged into the endpoint that sent the signal, the session manager suspends their virtual desktop and logs them out.

If the user attempts to log into a new endpoint before logging out of their previous machine, the session manager automatically logs them out of the first endpoint before logging them into the second. This is called a "tap-away". The session manager also supports "tap-over", which is when a user tries to log into an endpoint which is already being used. In this case, the new user will override the first user's session provided the new user has sufficient privileges.



(Figure 1)

Technical Design

A Bluetooth LE beacon can transmit using one of many available protocols. As we will discuss later, the Aventura beacons use Google's Eddystone UID protocol. An Eddystone UID frame takes up 20-bytes of a 31-byte "advertisement" packet. As shown in Figure 2, the first byte indicates the frame type (Eddystone UID in our case), the second byte indicates the beacon's transmit power, and the next 16 bytes comprise the beacon ID. The first 10 bytes of the beacon ID corresponds to the beacon's namespace. Namespaces are used to identify groups of beacons, so in our context all beacons in a given hospital will have the same namespace. The remaining 6 bytes form the beacon's unique instance ID, which is used to identify specific users.

Byte offset	Field	Description
0	Frame Type	Value = 0x00
1	Ranging Data	Calibrated Tx power at 0m
2	NID[0]	10-byte Namespace
3	NID[1]	
4	NID[2]	
5	NID[3]	
6	NID[4]	
7	NID[5]	
8	NID[6]	
9	NID[7]	
10	NID[8]	
11	NID[9]	
12	BID[0]	6-byte Instance
13	BID[1]	
14	BID[2]	
15	BID[3]	
16	BID[4]	
17	BID[5]	
18	Reserved for Future Use	Value = 0x00
19	Reserved for Future Use	Value = 0x00

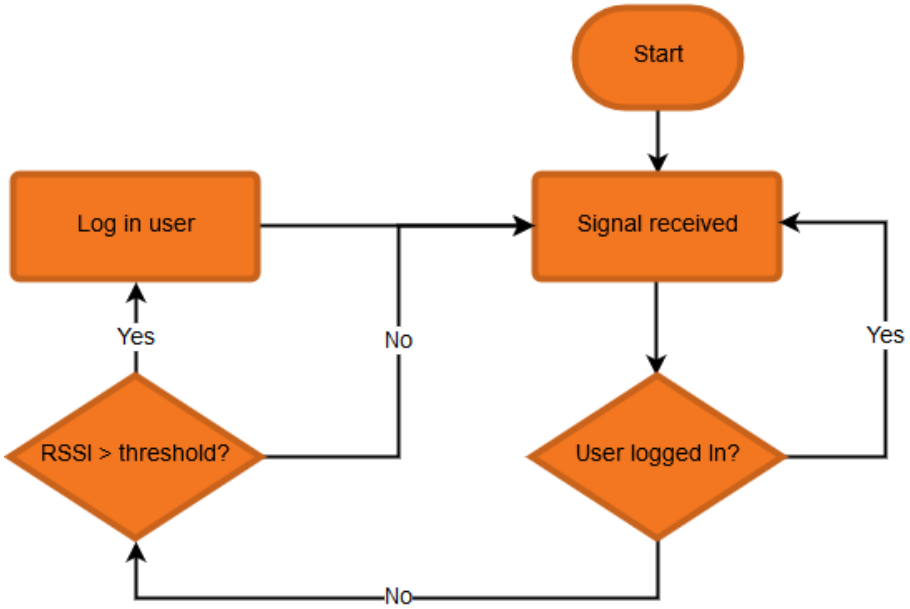
(Figure 2)

The login system relies on the Universal Beacon Library to receive and store beacon signals. The library's Beacon object contains a list of beacon frames broadcast by a specific beacon. The library also provides a BeaconManager class to create new

beacon objects and store the list of recently detected beacons. Finally, the BluetoothLEAdvertisementWatcher class provided by the .NET Framework listens for Bluetooth LE signals and passes them to the BeaconManager. The BeaconManager parses the signal payload and populates the beacon list as necessary.

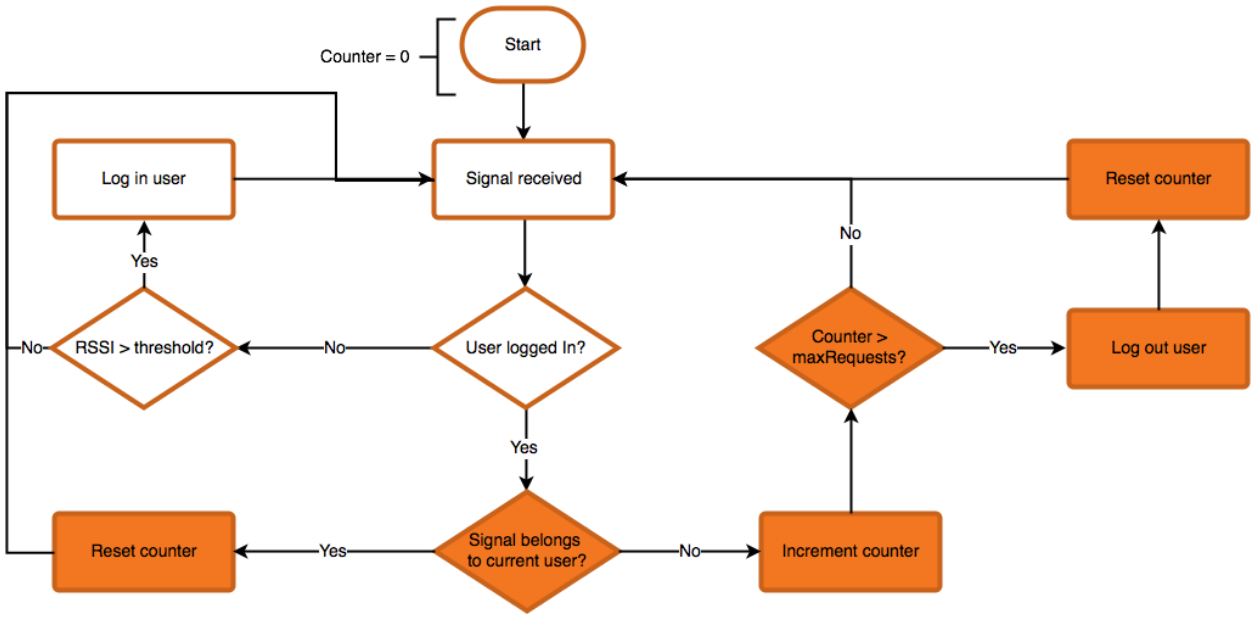
The information required to make the login or logout decision is contained within the BluetoothListener class, which is the main component of our BluetoothAPI DLL. The listener stores information about the currently logged-in user including their unique ID and the number of GetActiveID() calls since the user's beacon was last detected. Using this information, the listener determines when users should be logged in or out.

Before we describe our full decision-making algorithm, we will examine a simplified version of our login protocol (Figure 3). When the listener receives a signal, it first checks whether a user is already logged in. If this is the case, the signal is simply ignored. Otherwise, we check that the signal strength is greater than our login threshold. If the signal is strong enough, the new user is logged in and the process repeats.



(Figure 3)

The algorithm described above must be modified to support our logout protocol. The orange-colored boxes in Figure 4 represent the changes that must be made to our algorithm above. We start by initializing a request counter to 0. When a signal is received, the listener once again checks whether a user is already logged in. If no user is logged in, the algorithm is identical. Otherwise, the program checks whether the signal belongs to the current user. If so, the counter is reset and the signal is ignored. Otherwise, we increment the counter and check it against maxRequests. If the current user is not seen after maxRequests calls to GetActiveID, the counter is reset and the user is logged out.



(Figure 4)

Design Decisions

Language Choice

After researching the available Bluetooth Low Energy protocols and libraries, we decided to write the login system in C#. The .NET Framework provides the BluetoothLEAdvertisementWatcher class which is central to our program's functionality. Additionally, the only complete library supporting Bluetooth LE beacons in Windows is written in C#, so we decided to avoid interoperability issues by using the same language.

Libraries Used

The Universal Beacon Library is an unofficial library which supports Bluetooth LE beacons in Windows 10. We chose this library to interface with the beacons because it is well documented, uses the Apache license, and supports our targeted beacon protocol. This was a clear choice because it is the only complete beacon library we found. However, it makes use of .NET calls available only in Windows 10, so we are not able to support client machines running earlier versions of Windows.

We examined a number of beacon specifications in our research, including Google's Eddystone UID and URL protocols, Apple's iBeacon protocol, and Radius Networks' AltBeacon protocol. Ultimately we decided to use Eddystone UID because it is

open-source and allows us to assign a unique identifier to each beacon. We could not have used the iBeacon protocol, as it is closed-source and only works on iOS devices. The AltBeacon protocol was another open-source option, but we elected to use Eddystone UID because it is supported on more devices than AltBeacon.

Implementation

To meet our definition of done, we were required to integrate our login system into Aventura's workstation client. However, to prevent invasive modification of their existing client code, our program is implemented as a DLL which is referenced by the client (Figure 1). Because Aventura's workstation client is written in C++, we were also required to write a wrapper to call our managed C# code from native C++. This wrapper exposes a single method, acquireToken, to the workstation client, which is called every 250 milliseconds.

As an additional requirement, we were asked to write an interface that would allow for the extensibility of the current identification system. All token providers (eg, BLE Beacons, RFID scanners) would inherit from the token provider interface, and could be abstracted so that the logic of the session manager doesn't need to know what kind of token provider it is using, it just receives the unique token.

Results

The goal of the Bluetooth LE beacon reader project was to provide Aventura with a new possible feature with minimal changes to their existing platform. Our implementation of the Bluetooth listener API meets our functional requirements through its contract with the soft client.

In addition we successfully implemented a generic token provider interface. This interface allows for the future extensibility of the session manager to allow identification with other methods such as with biometric or NFC hardware.

We were also able to make some small changes to the Aventura soft client code to successfully register a beacon and start a session using our DLLs and the provided beacons. This was a first step in getting the BLE Beacons fully integrated into Aventura's product, but a very effective proof of concept.

Our project demonstrates that it is possible to substitute Aventura's RFID card readers with a Bluetooth LE listener. However, more work remains to be done before the project can be successfully integrated into Aventura's code base. Currently, the listener fails to detect beacon signals if the endpoint is not connected to an RFID reader. We believe this may be caused by code that ensures at least one device is connected, as the listener is not yet recognized as a device in the soft client. Additionally, we were unable

to test some edge cases that require more than one endpoint. For example, we do not test the case when a user is close enough to log into two endpoints at the same time.

One possible way of extending our project would be to design a mobile app that would allow the user to tap to authenticate instead of using a passphrase. Most smartphones can transmit BLE beacon signals, so integrating this feature into the app might prove more convenient than using physical beacons.

Appendices

Building and Running

The build order should be set so that the project will automatically build without any errors. However, if the build fails we recommend building the solution in the following order:

1. BeaconLibrary
2. BluetoothAPI
3. TokenProvider
4. BluetoothInterop
5. Soft Client

Expected Output

The program should display the Aventura login dialog when an Aventura beacon comes into range. The user should then be able to either login or register the beacon to a new user. If the dialog does not appear, check the following:

1. A Prox reader must be plugged into the device
2. The beacon must be turned on and must not be covered or blocked
3. The beacon must be configured to use the expected namespace

Configuration

The BluetoothAPI project contains an XML file named ‘app.config’ which contains the configuration settings for this project. This file was automatically generated Visual Studio’s Project->Properties->Settings menu. Currently the file contains data for the following variables:

1. avNamespace - only recognize beacons with this namespace
2. loginThreshold - ignore beacon signals with RSSI < loginThreshold
3. maxRequests - max number of successive calls to GetActiveID where the current user’s beacon is not detected before they are logged out

BeaconLibrary

This is a slightly modified version of the Universal Beacon Library available at https://github.com/andijakl/universal-beacon. Slight modifications were needed to allow us to use the library with projects that target .NET Framework. Specifically, .NET Core references were replaced with equivalent .NET Framework references where possible. The BeaconHelper class, which implements a ToArray method that could not be ported from .NET Core, is our only significant addition to the library.

BluetoothAPI

This contains the BluetoothListener class which makes up the majority of our code. It interfaces with the Aventura beacons according to its contract with the soft client. Public facing methods are GetActiveID, GetDevices, and a small number of public test methods which can be removed without consequence.

TokenProvider

This project contains the abstract classes ITokenProvider and TokenProviderFactory. Any future token providers will inherit from ITokenProvider and implement acquireToken to fulfill the contract with the soft client.

BluetoothInterop

This project contains the interoperability code needed to call our C# code from native C++. It provides the CppBluetoothListener class, which serves as a wrapper for the C# BluetoothListener class. CppBluetoothListener implements the generic ITokenProvider interface and thus defines acquireToken. This project also contains the BluetoothListenerFactory class which extends TokenProviderFactory.