Software Engineering Work Term Report
Creating an Adapter for the Nymi API

Anthony Weston

Relevant Courses: CS 137, CS 138, ECE 222

Request to be graded by a recent Software Engineering graduate.

Contents

1 Introduction

The Nymi band is a wearable hardware device which uses biometric data for authentication. Applications can send and receive messages from Nymi bands over the Bluetooth Low Energy protocol using the Nymi API, which is an interface to a runtime service connecting the computer to Nymi bands. A brand new version of that runtime service is in development, so during the past work term, I was responsible for re-designing the Nymi credential provider to use the new Nymi API. Credential provider is an API exposed by Windows to allow alternate forms of authentication for login. A series of functions, which the credential provider must implement, are called by the Windows Login Subsystem at login time. Because the WinLogon process runs just above the kernel, any runtime exceptions or memory issues can cause LoginUI.exe to crash and block the user from logging in to their computer. Therefore, the interface must have robust error handling and strong safety checks in order to avoid these potential issues. This report will cover the design and implementation of the components that make up the completed NapiAdapter class.

2 Nymi API Service Access

In designing the NapiAdapter class, I followed the Object Adapter design pattern to produce a robust, maintainable interface for the credential provider to consume. The Object Adapter design pattern "encapsulates the functions and data provided by existing non-object-oriented APIs within more concise, robust, portable, maintainable, and cohesive object-oriented class interfaces" (Service Access). This design pattern, among others, was a key part of our curriculum in CS138 and I applied the knowledge I gained from that course in order to design a lean and efficient interface.

I made the decision to follow the Object Adapter design pattern due to the dynamic nature of the Nymi API. During my work term, the Nymi API was evolving in parallel to my development, with new features being introduced and message structures changing. From the credential provider perspective however, the data it required from the band did not change. The credential provider interacts with the band to establish that the band is authenticated and within close proximity of the computer, then retrieves the private key from the band. Given that the credential provider's interactions with the band were well defined, I was able to design the interface with three functions exposed. The three functions were to initialize the NapiAdapter and load the Nymi API dynamic-link library, retrieve the symmetric key from the Nymi band, and get the most recent presence state of the Nymi band.

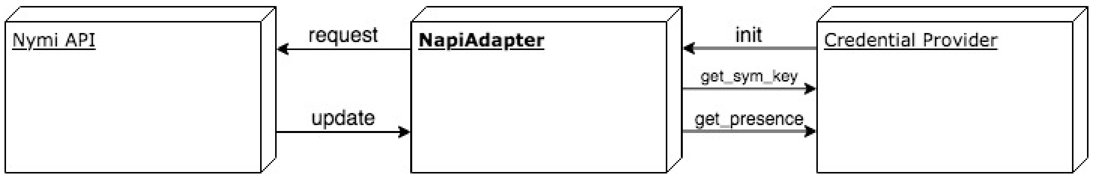

Figure 1: The NapiAdapter interface calls the two functions from the Nymi API, each of which sends data over its communication channel and exposes three functions to the credential provider. The arrows indicate the direction of information flow.

A good confirmation that the interface was well designed was that despite many changes in the Nymi API over the course of the term, and corresponding changes to the private functions of NapiAdapter responsible for band interaction, the interface exposed by the NapiAdapter did not have to change. This satisfies the design constraint of robustness and maintainability, since it was straightforward to make updates to NapiAdapter as the Nymi API changed and no changes had to be made in the credential provider code using the interface. This design also satisfies the constraint of conciseness by exposing the minimum number of functions while fulfilling its functionality requirements. Additionally, it eliminates accidental complexity by wrapping the low-level asynchronous function pointers of the Nymi API, which require handling, null checks and multiple calls, into synchronous functions so that the calling code can be as simple as possible.

3 Event Handling

Developing the event handler for Nymi API messages was a very interesting problem because the messages come both as responses to a specific request, and as advertisements with no associated request. The Reactor design pattern is relevant to this problem; it uses an event loop to "wait for indication events synchronously, demultiplex them to associated event handlers that are responsible for processing these events, and then dispatch the appropriate hook method on the event handler" (Event Handling Patterns). In my NapiAdapter implementation I created a new thread which executes the listener loop and the event handlers, while requests are made on the main thread of the application. I applied my knowledge of flow control and data structures from CS137 to efficiently process messages.

When I was first thinking of how to design the event handler, I considered that since only one operation would be done at a time, we could simply make the request and then call update from the same thread. This would be a simpler solution since it would eliminate the need for a new thread to listen for messages, and there would be no concurrency problems.

There are several reasons why I decided not to use the single thread approach. First, without a separate thread receiving messages as they come in, all of the advertisements from the surrounding bands would be queued and would need be processed before the request response. The issue is that these presence advertisements directly correspond to the band state in NapiAdapter. Because all of these messages would get queued and processed in order in a single thread system and messages do not have timestamps, the state of the band from the view of NapiAdapter could become misaligned with the actual state of the band. This misalignment has the potential to cause issues with the state machine of the credential provider at runtime, and therefore should be avoided.

In ECE 222 we learned about different communication protocols such as SPI and USB as well as the differences between simplex and duplex communication. Although communication with the Nymi band occurs at a much higher level of abstraction, I applied the knowledge I gained in that course to help understand the system I was working with. The Nymi API uses full-duplex communication; there are two separate channels, the request function for sending messages from the client to the Nymi API and the update function for the Nymi API to send messages back to the client. Therefore, using two threads, one for each channel, conforms with the intended usage of the API. Additionally, the Nymi API supports multiple open requests at once. If I had chosen a single-thread design then the adapter would have to be significantly altered to support multiple open requests. Using the two-thread listener loop design, minimal modifications would be needed since it is already made to communicate on the two channels independently.

Finally, we only have to create at most one new thread per login attempt. If we were creating and terminating a large number of threads in a short amount of time, that would be very resource intensive and a thread pool would be used to mitigate the cost of continually spawning new threads. However, since we only use one thread which runs until the NapiAdapter destructor is called, there is a small cost associated with choosing the two thread setup. Additionally, as I will discuss later, the synchronization model and thread management were both relatively straightforward to implement. Based on the low cost and the benefits of facilitating future development and tracking the state of the band accurately, I decided to use a two-thread setup with the reactor pattern design for the event handler of the Nymi API Adapter.

4 Error Handling

There are many different errors defined in the Nymi API related to the state of the band as well as the state of the Bluetooth service. Errors are separated into two classes, application and runtime. Only runtime errors were relevant for me to handle, because the application errors are caused by either incorrect code or incorrect data, so there is no user action that can resolve those issues. For example, if the band device identifier is not recognized by the Nymi API, an error will be reported but the user cannot do anything to resolve this. In this case the credential provider can only display an error message and suggest that the user use an alternate form of login. In order to properly record and report failed operations, I used an enumerated type with one code for success and several codes for various errors. I also created a simple struct to encapsulate the operation identifier, the operation type and the operation outcome. The use of abstractions to keep code clean and organized is another concept I learned in my CS137 and CS138 courses that I was able to apply during my work term.

Runtime errors are reported for timeouts and interrupted operations or for loss of connection to some component of the Bluetooth service. Timeouts are expected during normal operation, so in that case we can just set the outcome of the corresponding operation and return. In the case of a component of the Bluetooth service going down, there are two actions that the NapiAdapter takes. First, it updates any open operations so that they do not hang waiting for the service to restart since it could take some time. Second, in the listener thread, it goes into a special handler that waits for a specific reconnection message. There are different ways that this special state could be dealt with, however since no other valid messages can come before a reconnection, there is no need to handle any other messages. All request operations check a state variable to make sure that the listener thread is not waiting for reconnection before starting an operation to avoid producing additional error responses.

5 Thread Synchronization

The Nymi API interface uses a two thread structure which necessitates a synchronization scheme to protect shared data from being simultaneously accessed by multiple threads. The data that is accessed by both threads consists of two types: boolean and string.

Boolean values are used for representing state variables, and I decided to use the std::atomic type to protect these values because it has well-defined behaviour for simultaneous reading and writing from multiple threads. I chose this approach because it is simpler and less resource intensive than acquiring a lock guard, with the same protection guarantees.

Since strings are not a primitive type in C++, the same approach cannot be used for them. Instead, I decided to use a lock guard which takes ownership of a class member mutex used in both the read and write functions. The lock guard constructor will block if the mutex it is trying to acquire is already owned, so this guarantees that string data such as the symmetric key retrieved from the band cannot be written to and read from at the same time. The use of a lock guard rather than a plain mutex follows the Scoped Locking C++ idiom, which ensures that the mutex owned by the lock guard is released when the lock guard goes out of scope.

In order to notify the main thread if and when a response to an operation is received, I used a condition variable that is also a member of the NapiAdapter class. In the main thread, the condition variable will wait for some configurable amount of time. If the listener receives a response before that timeout expires it will notify the condition variable, which will stop waiting and return the value.

This design follows standard industry best practices as well as the resource acquisition is initialization programming idiom. It also satisfies the design constraints of safety, by preventing data races, undefined behaviour or deadlock, and simplicity, by making effective use of core language features instead of a custom implementation.

6 Threading Library

As a result of my decision to use a two thread setup to interact with the Nymi API, I had to choose a threading library to manage the new listener thread. There were two main choices I considered, the Windows native process and thread functions or C++11 std::thread. The requirements for the threading library were simplicity, safety and ability to terminate a child thread from the main thread.

In terms of simplicity and safety, std::thread is better than the Windows thread functions. CreateThread from the Windows library requires a special intermediate function which accepts a void pointer to the class instance we want to pass to the new thread, and uses a static cast to convert it back into the actual object it represents. Then the desired function must be called inside the intermediate function on the casted object. The std::thread constructor on the other hand, can be called directly with the target function of the new thread and the class instance passed as arguments. The C++ Standard Library version is definitely more straightforward and safer to work with since it takes care of any pointer conversions, so it has the advantage on those criteria. Despite this, in thousands of test runs there have been no issues arising from the use of the Windows threading functions, which is strong empirical evidence that although they are more complicated they are not unsafe to use.

On the second requirement, std::thread falls short. The Nymi API interface is needed to report the presence of the band and retrieve the symmetric key. When these operations are finished, no further communication with the band is needed and the listener thread needs to be shut down in the NapiAdapter destructor. The Windows API provides the function TerminateThread, which allows for immediate termination of the listener thread as required. A std::thread instance can only be ended by joining or detaching the thread. This means that some inter-thread communication would be necessary to notify the listener loop with a state variable to break and return so that its thread could be joined. That is not a particularly complicated feature to add, but since the notification variable would not be checked until the next iteration of the listener loop, it could block execution for the timeout period of the Nymi API update operation. So even though std::thread is more simple to use on its own, in this specific scenario it would have increased the complexity and harmed the performance of the NapiAdapter implementation overall.

A secondary reason for choosing the Windows library is that many other Windows specific libraries are used throughout the credential provider, so it is consistent with the rest of the code base. A general advantage of std::thread is that it is portable to any operating system, however since the credential provider is a Windows-only API, portability is not a design consideration for this code. Because of the functionality which the Windows thread library provides, I made the decision to use it over std::thread.

7 Reflection

My work of updating the Nymi credential provider to use the new generation of Nymi API was a great opportunity to practice applying design principles to a real world problem. Over the course of the term I was given a lot of independence and opportunity to experiment, which led me to make some mistakes and learn a great deal in the process of developing a robust and effective solution. I took time to design the interface and review it with my supervisor before starting the implementation, and I did not have to make any significant changes to the design during the implementation phase. I did make an effort to re-evaluate my design at various intervals during the implementation phase, and by doing this I was able to correct some assumptions and decisions I had previously made. I was also fortunate to work on a problem that tied together the knowledge of algorithms, data structures, object-oriented programming and communication protocols that I have gained over the first three terms of the Software Engineering degree program.

8 Works Cited

”Event Handling Patterns”. *Books on Pattern-Oriented Software Architecture*, `http://www.cs.wustl.edu/~schmidt/POSA/POSA2/event-patterns.html`. Accessed 19 April 2018.

“Service Access and Configuration Patterns”. *Books on Pattern-Oriented Software Architecture*, `http://www.cs.wustl.edu/~schmidt/POSA/POSA2/access-patterns.html`. Accessed 22 April 2018.