Eagle Framework: Network Server and Web Development with Swift

Design Project Summary

Prepared By
Joshua Beam
Auburn University COMP 7986

ABSTRACT

Network server applications, including web servers, are often written in the C and C++ programming languages, which provide high performance at the expense of builtin safety features and ease of use. Developers of web applications, however, typically use higher level scripting languages such as Ruby, Python, and JavaScript, which provide safety and ease of use at the expense of performance. The Swift programming language, introduced by Apple in 2014, has the potential to bridge the gap between C/C++ and today's popular scripting languages, and in doing so provide an attractive language for both network server and web application development. This report explores the strengths and weaknesses of the Swift programming language for network server and web application development, and also describes the design and implementation of Eagle Framework, an open source framework for network server and web development written in Swift.

TABLE OF CONTENTS

1. Introduction....4
2. Swift Overview....7
    2.1. Type System and Type Inference....7
    2.2. Memory Safety and Overflow Protection....8
    2.3. Error Handling....10
    2.4. Package Manager....11
3. Server Implementation....12
    3.1. Testing Framework....12
    3.2. Base Module....14
    3.3. Network Module Data Structures....15
    3.4. Server Class....19
    3.5. Http Module....21
4. Web Development Framework....25
    4.1. Template Module....25
    4.2. Database Module....26
5. Example Web Application....29
6. Performance Analysis....33
7. Conclusion....35
8. References....37

1. INTRODUCTION

For decades, two of the dominant programming languages for systems programming and the development of server software have been C and C++. The TIOBE Index for tracking programming language popularity has shown these two languages to be the most popular for much of the past twenty five years, with Java challenging them at the top of the list in the last decade [1]. C and C++ are open standards that have been widely implemented and refined over a long period of time, with strengths that include giving the programmer a tremendous amount of flexibility and high performance. With this flexibility and performance comes a major disadvantage, however: a lack of builtin safety features (such as automatic bounds checking when accessing arrays, integer overflow detection, and automatic memory management) that has led to widespread security and stability issues such as buffer overflows and memory leaks.

A particular category of software in which C and C++ are dominant is web server software. Netcraft's January 2016 Web Server Survey shows that the three most popular web servers for active sites on the internet (with a combined market share of 75.97%) are the Apache HTTP Server, nginx, and Microsoft's Internet Information Services (IIS) [2]. The Apache HTTP Server and nginx are written in C [3, 4], whereas Microsoft IIS is written in C++ [3].

While web servers themselves are often implemented in C and/or C++, this is not the case for web *applications*, which run on top of web server software. The most popular programming languages/frameworks for web application development include PHP, ASP.NET (which is typically used with Microsoft's C# programming language), and Java [5]. In recent years, rich frameworks based on the Model-View-Controller (MVC) paradigm have gained popularity, led by Ruby on Rails [6] and Django [7].

The disconnect between implementation language for web servers and web applications is due to the different goals of each. The ultimate goal of web server software is to handle as many HTTP requests as possible. While high performance is a desirable trait for web applications as well, it is not as important as ease of development (because web application developers often need to prototype and iterate quickly) and security (because web applications often deal with user data).

This separation between web servers and web applications can make debugging complex web applications difficult; it's generally not possible to run an entire web server and application stack from a single debugger, so multiple tools must be used, or the developer must rely on less than ideal methods such as printf debugging [8].

The job of web application developers is further complicated by the number of languages that they must work with. Knowledge of HTML and CSS is a must for web developers, and in addition to the backend programming language (such as PHP, Ruby, or Python), they may also need to be comfortable with JavaScript, the client-side scripting language supported by all major web browsers. The situation is even more complex due to the rise of mobile applications for smartphones and tablets, which often need to communicate with web applications through Representational State Transfer (REST) calls [9]. Developers working on mobile applications with web-based backends may need to work with all of the above plus the native programming languages for their mobile platforms.

Given the situation described above, there would clearly be benefits to a more unified approach to web development, with the number of programming languages involved being reduced without significantly compromising performance and security. One language that could potentially solve this problem, particularly for developers who work with Apple's iOS platform, is Swift.

Introduced in 2014, Apple's Swift language is intended to make it easier for developers to create software for Apple's platforms, while maintaining a high degree of compatibility with the existing C and Objective-C frameworks that have been in use on its platforms for many years [10]. Another goal, however, is to create a new programming standard. Craig Federighi, Apple's Senior Vice President of Software Engineering, told Ars Technica in an interview upon the open source release of Swift in December 2015, "We think [Swift] is how really everyone should be programming for the next 20 years. We think it's the next major programming language." [11]

Swift aims to be a safe language while also providing high performance and ease of use [10]. If it is successful in meeting these objectives, it could be an attractive choice for server, web application, and mobile development. Swift has been usable on Apple's iOS mobile operating system since its introduction in 2014, and as of its open source release in December 2015, it runs on Linux, one of the most widely used server operating systems [12]. The possibility of using Swift for network server and web development is now a reality.

This report describes the development of Eagle Framework, an open source network server and web development framework written in Swift [13]. Eagle Framework was developed to explore how suitable Swift is for network server and web development in terms of performance, ease of use, safety, and compatibility — particularly compatibility with C code, since there is a large

ecosystem of C libraries and Application Programming Interfaces (APIs) available on both OS X and Linux.

Section 2 of the report includes an overview of the Swift programming language, exploring its key characteristics and their implications for network server and web development. Section 3 describes the implementation of the network server components of Eagle Framework. Section 4 describes Eagle Framework's web development components, and Section 5 presents a sample web application created with the framework. Section 6 presents a performance analysis of the software, comparing it to other HTTP servers. Section 7 contains the conclusion of the report.

2. SWIFT OVERVIEW

We begin with an overview of the Swift language (particularly version 3.0 of the language, which is currently in development [14]), exploring its defining characteristics and their implications for network server and web application development.

A general theme that can be seen in the Swift language is that it encourages the developer to be explicit about behavior that occurs implicitly in typical programming languages, in order to avoid mistakes that can lead to bugs and security issues. Although this can make the language a little more cumbersome to work with than the high-level scripting languages that web developers are used to, it leads to improved safety and stability in Swift code.

Anyone interested in learning more about the Swift language is encouraged to consult Apple's "The Swift Programming Language," an official resource for the language available on the web and as an e-book [15].

2.1. Type System and Type Inference

Swift is a strongly typed language, more so than C or C++ because it generally does not allow the developer to use pointers to directly manipulate memory (except when interfacing with C code) and does not perform implicit type conversion. It is also statically typed, which is in contrast to scripting languages such as PHP, Ruby, and Python, where the type of a variable or parameter can change depending on the value assigned to it.

The programmer sacrifices some flexibility with a type system such as this when compared to scripting languages, but it promotes good software design and helps prevent a class of mistakes that is common to scripting languages. Consider the following Python code:

```
def format_user_details(user_id, first_name, last_name):
    return first_name + " " + last_name + " (user ID: " + str(user_id) + ")"

details = format_user_details(1, "Joe", "User")
```

The above function call is correct, but imagine if the programmer mistakenly put the user_id argument last; the error would not be detected until run-time, when the call to the function is actually made and the Python interpreter prints an error message:

```
TypeError: cannot concatenate 'str' and 'int' objects
```

This sort of mistake can easily occur in large code bases, particularly when code is refactored, because it may be difficult to find and update every occurrence of frequently used functions.

Equivalent Swift code would look like the following:

```
func formatUserDetails(userId: Int, firstName: String, lastName: String) ->
String {
    return first_name + " " + last_name + " (user ID: " + String(userId) +
")"
}

formatUserDetails(userId: 1, firstName: "Joe", lastName: "User")
```

If the userId argument was passed to the *formatUserDetails* function in the wrong order, the error would be caught at compile-time by the Swift compiler. In this example, we also see a convention Swift follows to help alleviate issues such as this: by default, arguments must be labeled with the parameter name, making it easier to spot mistakes related to the wrong value being given for a parameter (it's possible for the programmer to override this behavior by putting an underscore and a space before the parameter name in the function declaration).

Strong type systems can be cumbersome to work with if the language requires the programmer to explicitly specify the type of every variable and parameter, but Swift alleviates this by inferring types whenever possible. For example, the following two lines of Swift code both define constants of type *String*:

```
let s1: String = "Hello, "
let s2 = "world!"
```

In the first line, the type of the constant is explicitly specified, whereas with the second line, the Swift compiler infers the type to be *String* based on the fact that it is assigned a string literal.

2.2. Memory Safety and Overflow Protection

Swift's type system goes a step further than typical strongly typed languages in that it requires the programmer to specify whether or not values can be nil (the Swift equivalent of "NULL" in C/C++, or "None" in Python). For example, the following code is invalid in Swift:

```
var name: String = nil
```

It's invalid simply because the explicit type given to the variable is a non-optional type, and therefore it cannot contain the value nil. The variable can be made optional by including a question mark at the end of the type:

```
var name: String? = nil
```

Requiring optional values to be explicitly declared in this way helps alleviate NULL-pointer exceptions, a class of errors that can occur in programming languages like C/C++ and Python when a variable or parameter that is expected to contain a non-NULL value actually contains NULL.

Additionally, Swift performs automatic bounds checking for its native array types, in order to prevent memory outside the bounds of an array being overwritten when a buffer overflow occurs. While Swift does provide data types to represent C pointers in order to interface with C code, programmers typically do not need to work with pointers at all in Swift code, which minimizes the occurrence of bugs related to incorrect memory allocation and utilization.

Swift arrays must have all of their elements initialized to some known value before they can be used. While this prevents errors related to using uninitialized data, it can potentially decrease performance when interfacing with C code. For example, when receiving data using a C sockets API function, the user may allocate a buffer to store the data read from a socket; each byte in the buffer will have to be initialized with some value (such as 0), only to have those bytes be overwritten by the C function.

Another safety feature of Swift is its builtin integer overflow protection. Consider the following C code:

```
unsigned int i = (~0) + 1;
printf("%u\n", i);
```

This is valid C code that adds 1 to the largest value that can be represented by an unsigned integer. This results in an integer overflow and the variable *i* will contain the value 0.

The equivalent Swift code is:

```
var i: UInt32 = (~0) + 1
```

```
print(i)
```

Upon attempting to compile this, however, the Swift compiler will exit with an error informing the user that the operation results in an integer overflow. In the event that the error cannot be detected at compile-time (for example, if an overflow results from two variables being added together), then the program will stop when the overflow occurs.

In the event that the programmer wishes for overflows to be allowed when performing mathematical operations, Swift includes overflow operators that can be used instead of the standard mathematical operators, denoted by placing an ampersand before the standard operator. For example, the following Swift code behaves the same as the above C code, resulting in the variable *i* containing a value of 0:

```
var i: UInt32 = (~0) &+ 1
print(i)
```

2.3. Error Handling

Swift supports a form of exception handling reminiscent of other languages, although exceptions are typically just referred to as *errors* in Swift. Error types should conform to the *ErrorProtocol* protocol; for example:

```
enum BlogError: ErrorProtocol {
    case NoTitleGiven, NoMessageBodyGiven
}
```

Functions that can throw errors are required to be declared as such with the *throws* keyword:

```
func response(to request: HttpRequest) throws -> HttpResponse? {
    ...
}
```

Another function that uses a throwing function must either *catch* any errors thrown by that function, or it must be declared as a throwing function itself. Later in the report, in the discussion of Eagle Framework's implementation, we'll see how this error handling mechanism is used to catch errors encountered when generating responses to HTTP requests and to show generic error messages informing the user that an error occurred.

2.4. Package Manager

One of the major new features of Swift 3.0 is the inclusion of the Swift package manager [16]. The package manager allows the developer of a Swift package to easily include other packages, distributed through Git repositories, by referencing them in a manifest file named Package.swift, which describes the targets and dependencies of a Swift package [16]. The package manager is expected to simplify the process of distributing and building Swift packages on OS X and Linux. Although it includes the ability to generate Xcode project files, it is currently unclear how (or if) future versions of Xcode will interoperate with the package manager.

3. SERVER IMPLEMENTATION

In the following sections, we will explore the design and implementation of Eagle Framework. The bulk of the project's code is broken up into multiple Swift modules (the different modules and their relationships are shown in Figure 1; each arrow points from one module to another module that depends upon it). With the release of the Swift package manager in December 2015, the convention is for a Swift package to have a directory tree that includes a *Sources* directory, and each sub-directory of the *Sources* directory should contain the Swift source code for a module with the same name as that sub-directory. For example, if *EagleFramework* is the root directory of our Swift package, then *EagleFramework/Sources/Network* contains the Swift source code for the *Network* module.

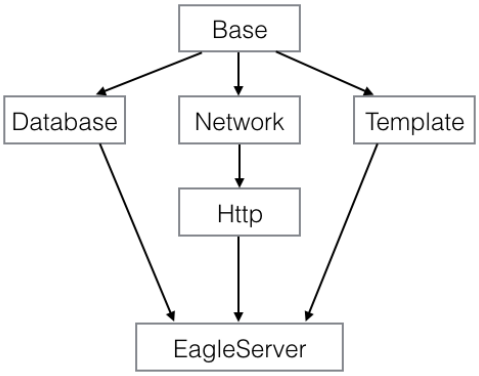

Figure 1. Module relationships

For the most part, each class and structure is stored in its own .swift file (for example, *Address.swift* for the *Address* structure in the *Network* module), although in some cases, a small class, structure, or other type that simply supports a larger class/structure may be defined in the latter's file.

We will now describe the implementation of the software in more detail, beginning with the custom testing framework that was used.

3.1. Testing Framework

Test-Driven Development (TDD) was practiced during the implementation of the project through the use of unit and integration tests. Initially, Apple's XCTest framework was used, as this is the official testing framework and has builtin support in Xcode. It later became apparent that XCTest was not yet well supported for projects using the Swift package manager's build system, and so a simple, custom testing framework was created for the project. The output of the

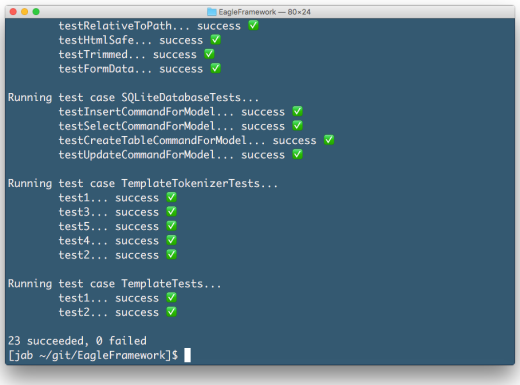

Figure 2. Test output

EagleTests target that includes the project's tests can be seen in Figure 2.

The testing framework includes a class named *TestCase*, which can be extended by a class to include one or more related tests. For example, the *Base* module discussed below includes extensions for Swift's builtin *String* type, and tests for these extensions are included in a class named *StringExtensionTests*. The beginning of this class is shown below:

```
class StringExtensionTests: TestCase {
    override var tests: TestDictionary {
        return [
            "testIsDirectory": {
                try assertFalse("/bin/ls".isDirectory)
                try assertFalse("/this/path/does/not/exist".isDirectory)
                try assertTrue("/tmp".isDirectory)
            },

            "testIsFile": {
                try assertFalse("/tmp".isFile)
                try assertFalse("/this/path/does/not/exist".isFile)
                try assertTrue("/bin/ls".isFile)
            },

            ...
        ]
    }
}
```

The *TestCase* class defines a property named *tests*, which is overridden in an extending class to return a dictionary of test names to functions (or closures) implementing tests. As shown above, the framework includes functions such as *assertTrue* and *assertFalse* to check for certain conditions and throw errors if those checks fail. The *TestCase* class includes a method called *run* that can be used to run all tests in the *TestCase* instance. A static *runTestCases* method is also included to run each *TestCase* given in an array, and print some statistics afterwards. This function is used as follows in the *EagleTests* target's *main.swift* file:

```
TestCase.runTestCases([
    JSONTests(),
    ModelTests(),
    StringExtensionTests(),
    SQLiteDatabaseTests(),
    TemplateTokenizerTests(),
    TemplateTests()
])
```

3.2. Base Module

The *Base* module includes a *Settings* class that is used to access settings passed to the server on the command line (one of which is a path to a directory containing static assets to be served as-is).

Also included in the *Base* module are extensions to Swift's builtin *String* type. Swift allows the programmer to add additional properties and functions to types that have already been defined, even builtin types and types defined by system libraries. This functionality is used to provide a number of useful utility properties and functions that are used throughout the rest of the modules.

Two of the computed properties added to the *String* type are *isDirectory* and *isFile*, which return boolean values indicating whether or not the path contained in the string is a directory or file path, respectively; for example, *"/tmp".isDirectory* would return true on both OS X and Linux, since /tmp is a standard directory available on both platforms.

Some functions added to the *String* type are intended to simplify methods that are builtin, but cumbersome to use. For example, the builtin mechanism for getting a substring of a string is to use the *substring(with: Range)* function:

```
let start = s1.index(s1.startIndex, offsetBy: 3)
let end = s1.index(start, offsetBy: 5)
let s2 = s1.substring(with: Range(uncheckedBounds: (start, end)))
```

The *Base* module provides a *substring* function that can be used as follows to perform the same task as the code snippet above:

```
let s2 = s1.substring(from: 3, length: 5)
```

Some additional string properties provided by the *Base* module are geared specifically towards web development. A common task for developers is to encode text content in an HTML-friendly manner for inclusion in an HTML page; this is especially important when displaying user-generated content, in order to prevent attacks that involve injecting malicious code into a web page's output. HTML encoding can be performed using the *htmlEncoded* property; for example, *"Hello".htmlEncoded* would return *"<i>Hello</i>"*.

The last two properties are useful for working with query strings and form data. The text after the question mark in a URL such as "http://localhost:5000/blog/?search=swift%20linux&page=2" is a query string, and in this example, it contains two keys, *search* and *page*, which have the original values "swift linux" and "2", respectively (form data is encoded in the same manner as query strings). *urlDecoded* converts a URL-encoded string back to its original form. For example, *"Hello%2c+world%21".urlDecoded* returns *"Hello, world!"* The *formData* property returns a Swift dictionary containing the URL-decoded keys and values found in query strings and data submitted via forms.

3.3. Network Module Data Structures

The Hypertext Transfer Protocol (HTTP) used on the web is an application layer protocol that utilizes the transport layer Transmission Control Protocol (TCP); this means that our server must be able to listen for connections to one or more TCP endpoints (a combination of an IPv4/IPv6 address and port), accept those connections, and be able to send and receive data over them.

The API traditionally used by UNIX applications written in C or C++ is the POSIX sockets API, which is implemented by the operating system's C library. Although there are higher level APIs available for use on OS X, Eagle Framework was designed to run on Linux as well; the POSIX sockets API is available on both OS X and Linux, and Swift includes builtin support for using C APIs, making this standard API the best choice for implementing the lower level networking code of the server.

Perhaps the most basic data structure utilized by the *Network* module is the IPv4 or IPv6 address, represented by the *Address* structure. At the beginning of the *Address.swift* file containing the implementation of this structure, we see the first example of operating system-specific code, as well as an enumeration used to specify the type of an address:

```
#if os(Linux)
import Glibc
#else
import Darwin
#endif

public enum AddressType {
    case IPv4, IPv6
}
```

The *import* keyword is used to import a module for use in the current Swift file. On Linux, C library functions are wrapped in the *Glibc* module, while on OS X, they are available in the *Darwin* module; the Swift compiler supports conditional statements similar to those seen in a C preprocessor, and this functionality is used in the code above to specify different modules to import on Linux and OS X. As mentioned above, we wish to support both IPv4 and IPv6 addresses, so the *AddressType* enumeration is used to specify one of these two types of addresses.

The beginning of the *Address* structure appears as follows:

```
/// Represents an IPv4 or IPv6 address.
public struct Address: CustomStringConvertible {
    public var type: AddressType
    public var address: [UInt8]
    public var hostname: String?
    ...
```

In Swift, you can define both structs and classes. The primary difference between the two is that struct instances are immutable by default and follow pass-by-value semantics, whereas classes are mutable by default and follow pass-by-reference semantics. Because an IPv4/IPv6 address is a relatively small data structure with no need for mutable data, it makes sense to make *Address* a struct.

The *Address* struct conforms to Swift's builtin *CustomStringConvertible* protocol; this means that the struct contains a computed property named *description* that returns a string representation of the address.

The first property of *Address* is the *type*. Next, the *address* property is an unsigned byte array containing the bytes that make up the address; if the type of the address is IPv4, then this array will contain 4 bytes, whereas it will contain 16 bytes for an IPv6 address.

The last property is the *hostname*, which is marked as an optional property utilizing the "?" symbol after the variable type. This is an example of how Swift attempts to enforce safety: if a variable or parameter is not marked as optional, then it can never contain nil; only optional variables/parameters can be nil. When used properly, this can help eliminate issues such as NULL-pointer exceptions, which can plague code written in languages like C++ and Python. The reason the hostname property is marked as optional is that, although we may wish to associate a hostname with an address, it is not required, because not all addresses have hostnames.

The primary way in which an instance of *Address* can be created manually is through its initializer that takes a hostname as a parameter, reproduced below in its entirety because it demonstrates the use of a simple C function from the POSIX sockets API:

```
/// Initialize an address with the given hostname.
///
/// - parameter forHostname: Hostname to resolve.
/// - returns: Address, or nil if the hostname could not be resolved.
public init?(forHostname hostname: String) {
    let hostnameCStr = hostname.utf8CString

    // Resolve the hostname. We try resolving an IPv6 address
    // first, and fall back to IPv4 if that fails.
    var type = AddressType.IPv6
    var host = gethostbyname2(hostnameCStr, AF_INET6)
    if host == nil {
        type = AddressType.IPv4
        host = gethostbyname(hostnameCStr)
    }

    guard host != nil else {
        return nil
    }

    // Create an array containing the address.
    var address: [UInt8]!
    if type == .IPv4 {
        address = [UInt8](repeating: 0, count: 4)
        for i in 0..<4 {
            address![i] = UInt8(bitPattern: host!.pointee.h_addr_list[0]![i])
        }
    } else {
        address = [UInt8](repeating: 0, count: 16)
        for i in 0..<16 {
            address![i] = UInt8(bitPattern: host!.pointee.h_addr_list[0]![i])
        }
    }

    self.type = type
    self.address = address
    self.hostname = hostname
}
```

Given a string containing a hostname, the above function first attempts to resolve the hostname as an IPv6 address using the *gethostbyname2* function provided by the system's C library; if that

fails, then the function instead attempts to resolve it as an IPv4 address using the *gethostbyname* function. The Swift guard statement is used to ensure that the hostname was resolved; if not, the function returns nil. The rest of the function sets up an unsigned byte array containing the IPv4 or IPv6 address corresponding to the given hostname, and finally sets the properties of the *Address* struct instance.

The building of the *address* array in the function above demonstrates a potential deficiency (discussed briefly in Section 2) in performance when interacting with C code due to the safety measures that Swift enforces. When Swift arrays are initialized, each element of the array must have some known value, so for an IPv6 address, each of the 16 elements of the address array is initialized to 0 (specified by the *repeating* parameter given to the array initializer), only to later be set to the actual value of the address from the C structure returned by the *gethostbyname* and *gethostbyname2* functions in the for loop that follows the array initialization.

An additional convenience structure named *Endpoint* simply contains an address and a 16-bit port, so that address/port combinations can easily be passed around as one value in the rest of the networking code:

```
public typealias Port = UInt16

/// Represents an address/port combination.
public struct Endpoint: CustomStringConvertible {
    public var address: Address
    public var port: Port

    public init(address: Address, port: Port) {
        self.address = address
        self.port = port
    }

    public var description: String {
        return "[\(address)]:\(port)"
    }
}
```

The above structure is used in the *Server* class to represent either a local address/port combination that the server should listen for connections to, or the address/port of a client that has connected to the server.

3.4. Server Class

The *Server* class is where we find the most conditional, operating system-specific code. On Apple's OS X, an efficient method to monitor socket descriptors for activity (such as data being received from a client) is to use the kqueue (or "Kernel Queues") API, which originated in the FreeBSD operating system [17]. This API is not supported on Linux, but there is an alternative that can be used on that platform: Linux's epoll API, based on an efficient O(1) algorithm, which acts as a successor to the POSIX polling mechanism that runs in O(n) time [18].

Before discussing the *Server* class further, let's take a look at a convenience class named *ServerConnection*, which represents a connection from a client to the server. This class stores the socket descriptor associated with a connection, as well as the local and remote endpoints; for example, if a client with address 192.168.1.2 connects from port 12345 to the server at address 192.168.1.1 listening on port 5000, then the local endpoint is 192.168.1.1 port 5000 and the remote endpoint is 192.168.1.1 port 12345.

```
/// Represents a connection to the server.
public class ServerConnection: CustomStringConvertible {
    public var shouldClose = false

    private(set) var descriptor: Descriptor
    private(set) var localEndpoint: Endpoint
    private(set) var remoteEndpoint: Endpoint
    ...
```

This class contains various functions for sending and receiving data; these functions essentially wrap lower level sockets API functions, so we will not discuss them in detail. If a call to the sockets API's *recv* function returns 0, this means that the connection has been ended and the socket should be closed, so when this happens, the *shouldClose* variable is set to true to indicate to the *Server* class that the connection can be closed and does not have to be monitored any longer.

Note that *ServerConnection* is a class, while previous data structures described have been structs. As mentioned earlier, structs adhere to pass-by-value semantics, while classes adhere to pass-by-reference semantics. With pass-by-value semantics, a copy of an object is created when it is passed from one function to another. If a connection has to be passed from one function to another, it makes no sense to create a copy of the connection, because, logically, there is just one connection; just as the operating system presents a connection as one socket descriptor, a

connection in the server code should be presented as only one instance of the *ServerConnection* class. Additionally, this will allow the *ServerConnection* class to be extended by another class to provide application layer network functionality, which, as we will see later, becomes useful in the *Http* module.

The following conditional compilation statement appears directly before the *Server* class:

```
#if os(Linux)
import CEpoll
import Glibc
#else
import Darwin
#endif
```

Here we see a new module used on Linux. The *Glibc* module does not expose Linux's epoll API, so a custom module named *CEpoll* was created for this project [19]. This module simply contains a *module.modulemap* file, which maps and exports a C header and/or library so that the functions/structures defined in the header can be used from Swift code:

```
module CEpoll [system] {
    header "/usr/include/x86_64-linux-gnu/sys/epoll.h"
    export *
}
```

Swift module maps can be built and used in a Swift project using the Swift package manager. At the time of writing, this module map mechanism is limited and awkward to use; module maps must contain absolute file paths, making each module map very platform-specific, and they must be placed into their own Git repositories separate from the packages that utilize them [20]. The above module map includes the epoll header for the 64-bit version of Ubuntu 15.10, which may not be a valid file path in other Linux distributions.

The *Server* class begins as follows:

```
public class Server<ServerConnectionType: ServerConnection> {
#if os(Linux)
    private var epollDescriptor: Descriptor
    private var events: [epoll_event] = []
#else
    private var kqueueDescriptor: Descriptor
    private var events: [kevent] = []
#endif
```

```
    private var localEndpoints: [Descriptor: Endpoint] = [:]
    private var connections: [Descriptor: ServerConnectionType] = [:]
    ...
```

Here we see an example of Swift's *generics*, which are functionally similar to templates in C++. The ultimate purpose of the *Server* class is to keep track of a number of connections, which it stores in its *connections* dictionary (the connection's socket descriptor value is the key into the dictionary). The generic parameter clause <*ServerConnectionType: ServerConnection*> specifies that the server will keep track of connections of type *ServerConnectionType*, a generic type which represents any class that extends the *ServerConnection* class.

As discussed earlier, the kqueue API is used on OS X while the epoll API is used on Linux. Both APIs require a special descriptor to be created and used in subsequent kqueue/epoll operations, and both have their own event type which describes an event to monitor (such as data becoming available for reading on a particular socket). The *events* array uses the appropriate type on each platform.

The rest of the *Server* class (as well as the various static utility functions it uses that are found in the *ServerUtil* class) is fairly standard code that uses the sockets, kqueue, and epoll APIs to listen for, accept, and monitor connections.

3.5. Http Module

The *Http* module builds upon the *Network* module to implement the core of a functional HTTP server. As mentioned previously, the *ServerConnection* class can be extended to support a certain type of application layer connection. This is what is done with the *HttpConnection* class, which extends the *ServerConnection* class to provide HTTP-specific functionality such as buffering the lines of text that form a complete HTTP request (represented by the *HttpRequest* class) and sending HTTP responses (represented by the *HttpResponse* class).

When the *HttpConnection* class has received a complete HTTP request and built an instance of the *HttpRequest* class, it calls the following member function:

```
private func process(request: HttpRequest) {
    print("\(self) request: \(request)")

    let response = ResponderRegistry.response(to: request)
```

```
    print("\(self) response: \(response)")
    send(response: response)

    shouldClose = true
}
```

First, the request itself is printed to the console for logging and debugging purposes. Next, a response is generated using the *ResponderRegistry*, which brings us to the concept of a responder. In Eagle Framework, a responder is simply a class that conforms to the following Swift protocol:

```
public protocol Responder {
    func response(to request: HttpRequest) throws -> HttpResponse?
}
```

A responder must implement one function: *response(to: HttpRequest)*. This function builds and returns a response for the given request, or nil if the responder cannot handle the request. Each responder should be registered with the simple *ResponderRegistry* class, reproduced in its entirety below:

```
public class ResponderRegistry {
    private static var responders: [Responder] = []

    public static func register(responder: Responder) {
        responders = [responder] + responders
    }

    public static func response(to request: HttpRequest) -> HttpResponse {
        for responder in responders {
            do {
                if let response = try responder.response(to: request) {
                    return response
                }
            } catch {
                return HttpResponse.error(message: "The following error
occurred when attempting to respond to your request: \(error)")
            }
        }

        return HttpResponse.fileNotFound(path: request.path)
    }
}
```

A responder can be registered by calling the static *register* function, which places the given responder at the beginning of the registry's list of responders. The registry can then be used to generate a response to an HTTP request by calling the static *response(to: HttpRequest)* function, which will loop through the list of responders until it finds one that handles the request. If a response could not be generated (which can happen if there is no responder that can handle the request), then a standard HTTP 404 File Not Found response is generated and returned. If a responder throws an error while generating a response, then a generic error message response is generated and returned.

An example of a simple responder is the included *FileResponder* class, which simply checks to see if the path in the given request corresponds to a file in the given *fileSystemPath* directory, and if so, returns an *HttpResponse* containing the contents of that file:

```
public class FileResponder: Responder {
    public private(set) var webPath: String
    public private(set) var fileSystemPath: String

    private static var cachedResponses: [String: HttpResponse] = [:]

    public init(webPath: String, fileSystemPath: String) {
        self.webPath = webPath
        self.fileSystemPath = fileSystemPath
    }

    public func response(to request: HttpRequest) -> HttpResponse? {
        if let path = request.safeFilePath {
            // If a response for the file path has been cached, return it.
            if let response = FileResponder.cachedResponses[path] {
                return response
            }

            if let relativePath = path.relativeToPath(webPath) {
                var fullPath = fileSystemPath + "/" + relativePath
                if fullPath.isDirectory {
                    if !path.isEmpty && !path.hasSuffix("/") {
                        return HttpResponse.redirect(to: path + "/")
                    }

                    fullPath += "/index.html"
                }

                if let response = HttpResponse.file(withPath: fullPath) {
                    FileResponder.cachedResponses[path] = response
                    return response
```

```
                }
            }
        }

        return nil
    }
}
```

The *FileResponder* class maps a file path from the given *webPath* (for example, if the user accesses *http://localhost:5000/files/test.txt*, then *files* would be the *webPath*) to a local file stored in the *fileSystemPath* directory. If such a file is found, then the responder generates a response, caches it in a dictionary so that it can quickly be reused later, and returns it.

The responder mechanism described above is the core of how a programmer can use Eagle Framework to build a web application. It provides plenty of flexibility, as the programmer is able to determine whether or not a responder should handle a request based on the request's properties, including its HTTP method (such as "GET" or "POST"), its file path, and its headers.

We next turn to the modules that together provide a framework to assist Swift programmers in building web applications.

4. WEB DEVELOPMENT FRAMEWORK

Web developers have a variety of frameworks to choose from for application development, including Ruby on Rails [6] and Django (based on Python) [7]. The primary appeal of these frameworks is that they provide the developer with a tremendous amount of functionality out of the box, including powerful template engines and Object-Relational Model (ORM) frameworks for interacting with databases. The scripting languages that these frameworks are based on also provide the programmer with powerful features and flexibility, but they often do so at the expense of performance, and the lack of a strong type system can open the door for bugs such as arguments being passed to functions in the wrong order or variables containing instances of unexpected data types. With Swift's goals of being a powerful and easy-to-use, yet fast, language, it could potentially avoid these issues.

Eagle Framework includes a simple template engine, as well as a basic Object-Relational Model and database solution based on SQLite [22]. These modules are described below.

4.1. Template Module

The *Template* module provides a simple template engine that's capable of embedding values from a Swift dictionary object into HTML content. Eventually, this will be expanded to include support for conditional and looping constructs. The *Template* module contains some of the more complex code in the project, including a tokenizer (or lexer), a parser, and classes to represent different types of nodes in a template (for example, sequences of plain text or a reference to a value in a dictionary). Figure 3 shows the classes in the module and their relationships to each other.

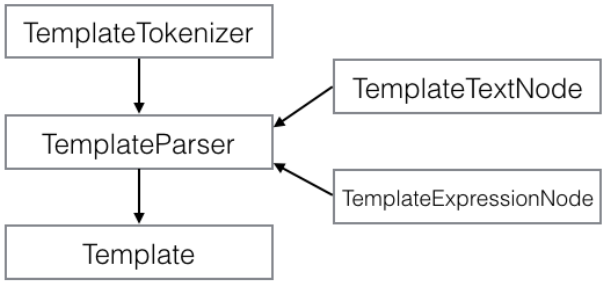

Figure 3. Template module classes

Consider the table below, showing the source code for a template on the left and the corresponding rendered content on the right. The syntax for the template source is based on ASP.NET's syntax [21].

Template Source	Rendered Content

`<html>`
`  <body>`
`  <h1><%=title%></h1>`
`  <p><%=content%></p>`
`  </body>`
`</html>`
	`<html>`
	`  <body>`
	`  <h1>Test</h1>`
	`  <p>Hello, world!</p>`
	`  </body>`
	`</html>`

If the above template were stored in a file named "index.html.template", a *Template* object could be created from it by calling *Template.fromFile("index.html.template")*. The template could then be rendered as a string by calling the *render* method of the template object, which accepts a dictionary as a parameter. If the dictionary contained the value "Test" for the *title* key and "Hello, world!" for the *content* key, the rendered content would appear as it does in the right side of the table above.

4.2. Database Module

The Object-Relational Model solutions provided by frameworks such as Ruby on Rails [6] and Django [7] allow the developer to define models simply by creating Ruby and Python classes that contain properties corresponding to the models' fields. These systems take advantage of the powerful reflection and introspection features of those languages to generate database schemas and queries based on class definitions.

Reflection and introspection is an area that Swift is currently lacking in, but it does include basic reflection functionality through the builtin *Mirror* type. Eagle Framework's *Database* module uses this type to achieve simple Object-Relational Model functionality.

Two generic classes, *RequiredModelProperty* and *OptionalModelProperty* (which both implement the *ModelProperty* protocol), are used to define properties in a model and store the values of those properties. The *Model* class includes type aliases based on the generic model property classes that can be used to define boolean, double-precision floating point, integer, and string properties. Each model also has a 64-bit integer *id* property used to identify it, and foreign keys can be defined to represent relationships between different models.

The *Model* class has a computed property named *properties*, which builds a list of the class instance's properties that conform to the *ModelProperty* protocol. This includes the following

code (where *NameAndProperty* is a type alias for a tuple containing the property's name as a string and its *ModelProperty* reference):

```
var result: [NameAndProperty] = []
let mirror = Mirror(reflecting: self)
for child in mirror.children {
    if let label = child.label {
        if let property = child.value as? ModelProperty {
            result.append((label, property))
        }
    }
}
```

The *Model* class can be extended to define a model. Here's an example from one of the project's unit tests:

```
class TestModel: Model {
    let boolProperty = Model.BoolProperty(defaultValue: false)
    let doubleProperty = Model.DoubleProperty(defaultValue: 1.23)
    let intProperty = Model.IntProperty(defaultValue: 42)
    let stringProperty = Model.StringProperty(defaultValue: "Hello")
    required init() {}
}
```

The properties defined in the model above are all non-optional, so default values are specified in their initializers.

As mentioned previously, the database module includes support for SQLite [22] databases. The *SQLiteDatabase* class implements the *Database* protocol:

```
public protocol Database {
    /// Create storage in the database for the given model.
    func createStorage(forModel model: Model) throws

    /// Saves a data model to the database.
    ///
    /// - parameter model: The data model to save.
    func save(model: Model) throws

    /// Loads a data model from the database.
    ///
    /// - parameter model: The data model instance to load the data into.
    /// - parameter withId: The unique identifier of the data model to load.
    /// - returns: The data model populated with the loaded data.
```

```
    func load(model: Model, withId id: Int64) throws -> Model

    /// Query for models of the given type.
    func query<T: Model>(model: T) throws -> [T]
}
```

SQLite stores a database in a single file, so the initializer for the *SQLiteDatabase* class accepts a single string representing the path to the database file as a parameter. The class works by generating strings containing SQL commands to perform the necessary operations to save and load models. For example, suppose that we have the following model that represents a user:

```
class User: Model {
    let name = Model.StringProperty(defaultValue: "")
    let password = Model.StringProperty(defaultValue: "")

    init(name: String, password: String) {
        self.name.value = name
        self.password.value = password
    }
}
```

Suppose also that the variable *database* is set to an instance of *SQLiteDatabase*, and the following function call was made:

```
database.save(model: User(name: "user", password: "test")
```

The *SQLiteDatabase* class uses SQLite's C API to execute commands and retrieve the results. Calling the *save* function as above would first cause the following SQL command to be run:

```
INSERT INTO User (name, password) VALUES ('user', 'test')
```

The above command will fail if the *User* table does not yet exist, in which case the following command to create the table is executed:

```
CREATE TABLE User (name TEXT NOT NULL, password NOT NULL)
```

The insert command is then executed again, and the model's ID is set to the ID of the newly inserted row in the database (as retrieved using the SQLite API's *sqlite3_last_insert_rowid* function). When saving or loading a model, the *SQLiteDatabase* retrieves the name of the model subclass and the names/values of its properties using Swift's *Mirror* functionality.

5. EXAMPLE WEB APPLICATION

As an example of how Eagle Framework can be used to build a simple web application, the software includes a basic blog system (shown in Figure 4), implemented in a class named *BlogResponder*. As its name implies, this class conforms to the *Responder* protocol discussed earlier.

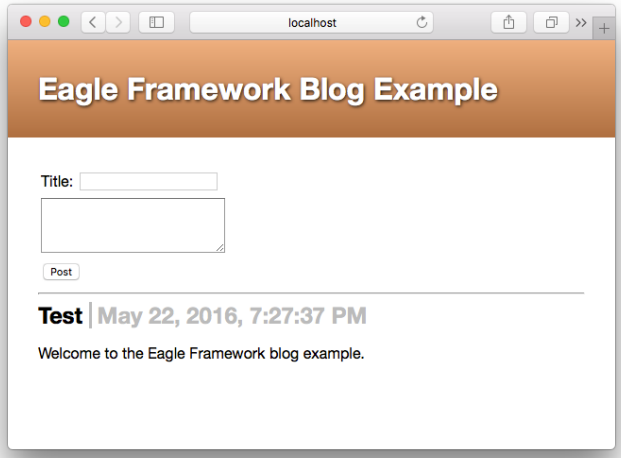

Figure 4. Blog example

First, a model named *BlogPost* is defined to allow blog posts to be stored in a database:

```
class BlogPost: Model {
    let title = Model.StringProperty(defaultValue: "Title")
    let body = Model.StringProperty(defaultValue: "Body")
    let timestamp = Model.DoubleProperty(defaultValue: 0.0)

    required init() {
        self.timestamp.value = NSDate().timeIntervalSince1970
    }

    init(title: String, body: String) {
        self.title.value = title
        self.body.value = body
        self.timestamp.value = NSDate().timeIntervalSince1970
    }
}
```

Each blog post contains three properties: the title, the body of the post, and a timestamp, which is stored as the number of seconds since the UNIX epoch.

Next, we'll look at the beginning of the *BlogResponder* class and its initializer:

```
class BlogResponder: Responder {
    private(set) var webPath: String
    private(set) var database: Database

    private let indexTemplate: Template
```

```
    init(webPath: String, databasePath: String) throws {
        self.webPath = webPath
        self.database = try SQLiteDatabase(filePath: databasePath)

        do {
            try self.database.createStorage(forModel: BlogPost())
        } catch {}

        self.indexTemplate =try Template.fromFile("blog_index.html.template")
    }
```

The *webPath* property simply contains the root path for the blog application on the HTTP server (for example, "/blog"). The next property is a database that is initialized as a SQLite database stored in the database file path given to the initializer. The *createStorage* method is called on the database in order to create the file if it does not yet exist. The final task performed by the initializer is to load a template for the blog index, which will include a list of posts and a form for adding a new post.

The *index* method generates a response containing the index:

```
    func index() throws -> HttpResponse? {
        var html = ""

        for post in try database.query(model: BlogPost()).reversed() {
            let formatter = NSDateFormatter()
#if os(Linux)
            formatter.dateStyle = .MediumStyle
            formatter.timeStyle = .MediumStyle
#else
            formatter.dateStyle = .mediumStyle
            formatter.timeStyle = .mediumStyle
#endif
            let datetime = formatter.string(from:
NSDate(timeIntervalSince1970: post.timestamp.value))

            html += "<h2>\(post.title) <span class=\"timestamp\">
\(datetime.htmlSafe)</span></h2>"
            html += "<p>\(post.body)</p>"
        }

        let content = indexTemplate.render(["blog_posts": html])
        return HttpResponse.html(content: content)
    }
```

The method includes a for loop that iterates through each *BlogPost* model stored in the database, sorted in reverse order so that new posts are shown first. The method builds up an HTML representation of the list of blog posts, and finally renders the index template and returns the response.

The *addPost* method handles saving a new post to the database:

```
func addPost(from request: HttpRequest) throws -> HttpResponse? {
    let formData = request.formData
    let title: String! = formData["title"]
    let body: String! = formData["body"]

    guard title != nil else {
        throw BlogError.NoTitleGiven
    }

    guard body != nil else {
        throw BlogError.NoMessageBodyGiven
    }

    let post = BlogPost(title: title, body: body)
    try database.save(model: post)

    return HttpResponse.redirect(to: "/" + webPath)
}
```

This method retrieves the title and body of the post from the given *HttpRequest* object's form data, throwing an error if either property is not found. Next, it creates a *BlogPost* model, saves it, and redirects the user back to the blog index, where the new post should be listed.

Finally, the implementation of the *response* method takes a request and calls either the *index* or *addPost* method depending on the path in the request:

```
func response(to request: HttpRequest) throws -> HttpResponse? {
    if let path = request.path(relativeTo: self.webPath) {
        let pathComponents = path.components(separatedBy: "/")
        if pathComponents.count == 0 || pathComponents[0] == "" {
            return try index()
        } else if pathComponents[0] == "addPost" {
            return try addPost(from: request)
        }
    }
```

```
        return nil
    }
```

The blog application is part of the *EagleServer* target included with Eagle Framework. The *main.swift* file of this target contains the main application code that registers the *FileResponder* and *BlogResponder* responders, and then creates and runs an HTTP server that can be accessed locally over IPv4 or IPv6:

```
import Base
import Http
import Network

print("WWW Path: \(Settings.wwwPath)")
print("Resources path: \(Settings.resourcesPath)")

ResponderRegistry.register(responder: FileResponder(webPath: "",
fileSystemPath: Settings.wwwPath))

let blogDatabasePath = Settings.getAbsoluteResourcePath("blog.db")
ResponderRegistry.register(responder: try BlogResponder(webPath: "blog",
databasePath: blogDatabasePath))

let port: Port = 5000
let server = try HttpServer(hostname: "127.0.0.1", port: port)
try server.addLocalEndpoint(hostname: "::1", port: port)

print("Server started")
try server.run()
```

The full *EagleServer* implementation (including *BlogResponder* and the above *main.swift* code) is about 115 lines of code, and provides an example of how simple the code for an Eagle Framework-based web application can be.

6. PERFORMANCE ANALYSIS

Performance was analyzed and improved with the use of the Instruments application included with Xcode, which can perform time-based profiling of an application. Figure 5 shows an Instruments window with a call tree showing how much time was spent in which functions of the *EagleServer* program after processing a large number of requests.

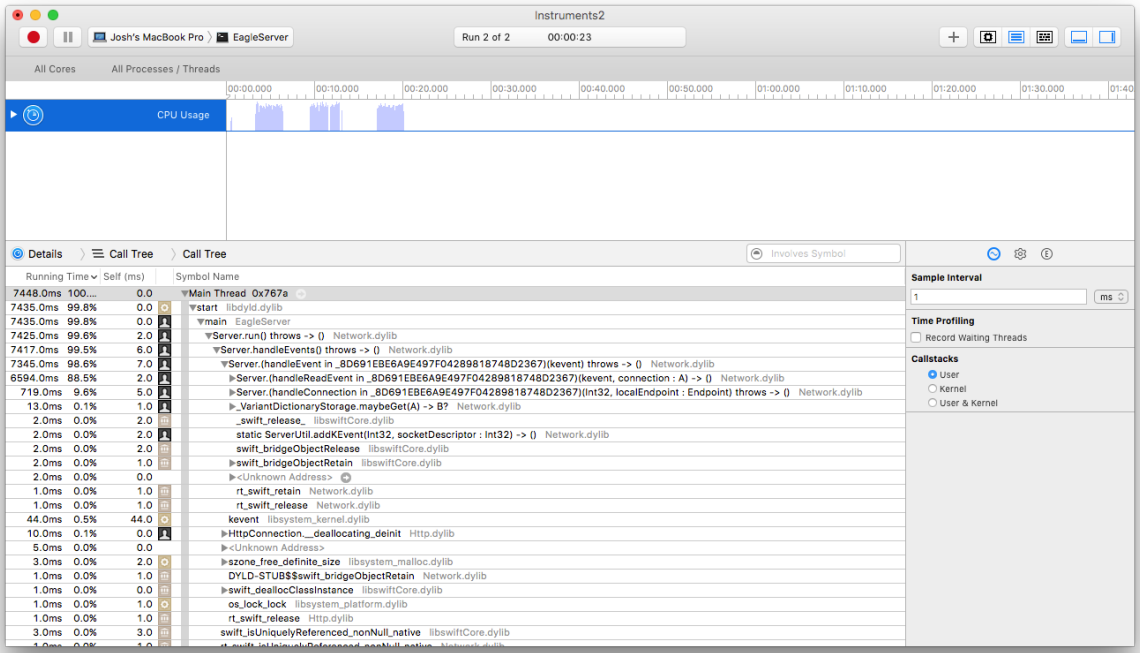

Figure 5. Instruments analysis of Eagle Framework

With the use of the Instruments profiler, it was discovered that some custom implementations of string manipulation functions (such as one for trimming whitespace from the beginning and end of strings) did not perform well, and they were replaced with implementations supported by the Foundation framework in Swift 3.0. Additionally, printing debug messages to the console for each connection and request consumed a measurable amount of time, so such messages were disabled in order to measure real-world performance outside of debugging scenarios.

The performance of Eagle Framework was also compared to that of other HTTP servers. The average number of requests per second was measured using *ab*, the benchmarking tool included with the Apache HTTP Server. The comparisons were performed on a MacBook Pro computer with a 2.3 GHz Intel Core i5 processor and 8 GB of memory. The *ab* tool was run on the same computer as the servers with a command such as the following:

```
ab -n 5000 -c 10 http://localhost:5000/
```

The *-n* option specifies the total number of requests, while the *-c* option specifies the number of concurrent requests, so the above command causes the *ab* tool to make a total of 5,000 requests, with 10 requests at a time running concurrently.

The test compared the performance of Apache HTTP Server 2.4.18, nginx 1.10.0, and Eagle Framework (with and without caching of file responses) when serving a small HTML file. The results are shown in Figure 6.

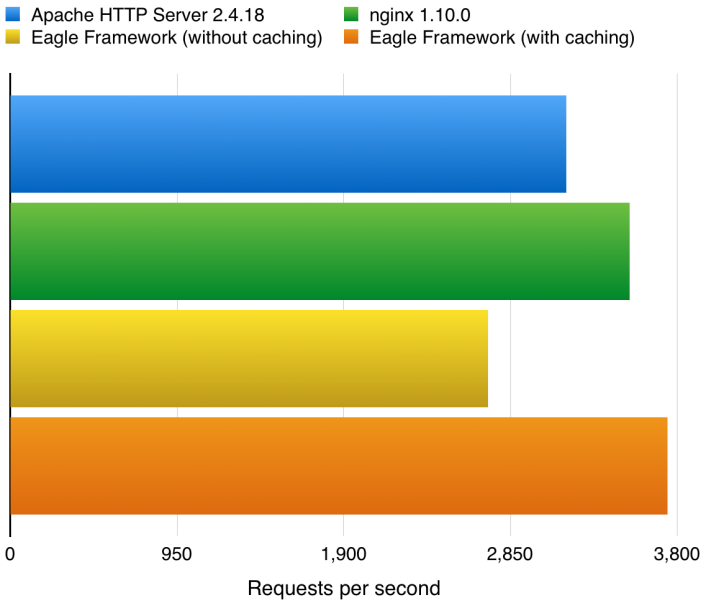

Figure 6. HTTP server performance comparison

This test was initially performed before caching of file responses was added to Eagle Framework; as the figure shows, Eagle Framework did not perform quite as well as the other servers before this optimization was implemented. After analyzing the Instruments results shown in Figure 5, it was apparent that generating the *HttpResponse* object for the HTML file took a large percentage of the time involved in responding to requests. A simple caching mechanism was then implemented in the *FileResponder* class, and the test was run again. With this simple optimization, Eagle Framework outperformed the other HTTP servers, processing almost 3,800 requests per second.

7. CONCLUSION

The implementation of Eagle Framework shows that the Swift programming language provides the performance, ease of use, safety, and compatibility necessary to make it a viable choice for both network server development and web development.

In Section 6, we saw that Eagle Framework performed almost as well as the Apache HTTP Server when serving static files before caching of responses was implemented. After that optimization was added, Eagle Framework outperformed the Apache HTTP Server and nginx. These results show that Swift can be used to create high-performance servers that perform similarly to servers written in C and/or C++.

The examples from the Eagle Framework source code shown in Sections 3 and 4, and the example web application covered in Section 5, show that Swift provides automatic memory management and has a simple syntax that's inspired by high-level scripting languages, despite the fact that it's a fast language intended to rival C and C++. Additionally, type inference frees the developer from having to explicitly specify types much of the time, even though Swift is strongly typed. This ease of use enables the sort of fast prototyping and iteration that web developers are used to.

Swift provides common safety features such as automatic bounds checking when accessing arrays. Swift's strong type system and its advanced safety features described in Section 2 (including optionals, integer overflow prevention, and measures to avoid unhandled exceptions) show that it's a language that emphasizes safety, which is especially important when developing web applications that work with user data.

Finally, Swift is able to interoperate with C code, providing access to a large ecosystem of libraries and APIs available on OS X and Linux. We saw this in Section 3 through Eagle Framework's use of the sockets API, as well as in Section 4 through the use of SQLite's C API. This compatibility ensures that developers can adopt Swift without abandoning existing C code, which can be advantageous for both server and web development.

There are issues to be aware of when using Swift, however. As mentioned earlier, Swift 3.0 is in development at the time of writing, and it introduces large source-incompatible changes from Swift 2.2, the current stable version; therefore, developers have to decide whether they want to support the stable version of Swift or the new, in-development version. A stable release of Swift

3.0 is planned for later in 2016, along with a new version of Xcode that includes it [14]. Recent releases of Xcode have included utilities to convert the syntax of old Swift code to the syntax of newer versions of the language, however, so it's possible the same sort of functionality will be available upon the stable release of Swift 3.0.

Another concern for anyone intending to do cross-platform development with Swift is the state of the Swift package manager, which is still early in its development at the time of writing. As mentioned earlier, it can be awkward to use (especially when attempting to interface with C libraries), and was found to have issues working with Apple's XCTest testing framework during the development of Eagle Framework. It is also unclear at this time how (or if) the package manager will integrate with Xcode in the future.

Despite these concerns, Swift is already a very capable language, and shows tremendous promise for the future. Eagle Framework shows that the language's performance, ease of use, safety, and compatibility make it a good choice for high-performance server development as well as web development.

Some of the future work that is planned for Eagle Framework includes:

- **Improved performance.** The software will continue to be optimized in order to achieve a high level of performance.
- **Richer API.** The web development framework will continue to be expanded to include features useful for the development of web applications. For example, the template engine will be improved to include features such as loops and conditional blocks.
- **SSL/TLS support.** This will allow the software to serve secure pages over HTTPS and the new version of the Hypertext Transfer Protocol, HTTP/2, since many browsers require HTTP/2 traffic to be encrypted [23].

Interested readers are encouraged to follow the Eagle Framework project on GitHub [13].

8. REFERENCES

[1]: TIOBE, “TIOBE Index.” [Online] Available: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2]: Netcraft, “January 2016 Web Server Survey,” Jan. 2016. [Online] Available: http://news.netcraft.com/archives/2016/01/26/january-2016-web-server-survey.html

[3]: V. Lextrait, “The Programming Languages Beacon.” [Online] Available: http://www.lextrait.com/Vincent/implementations.html

[4]: A. Alexeev, “nginx,” *The Architecture of Open Source Applications*. [Online] Available: http://www.aosabook.org/en/nginx.html

[5]: BuildWith, “Framework Usage Statistics.” [Online] Available: http://trends.builtwith.com/framework

[6]: Ruby on Rails, “Ruby on Rails.” [Online] Available: http://rubyonrails.org

[7]: Django, “Django overview.” [Online] Available: https://www.djangoproject.com/start/overview/

[8]: A. Binstock, “Embedded print Statements != Debugging,” *Dr. Dobb’s*, Feb. 2013. [Online] Available: http://www.drdobbs.com/architecture-and-design/embedded-print-statements-debugging/240148855

[9]: M. Vaqqas, “RESTful Web Services: A Tutorial,” *Dr. Dobb’s*, Sep. 2014. [Online] Available: http://www.drdobbs.com/web-development/restful-web-services-a-tutorial/240169069

[10]: Swift, “About Swift.” [Online] Available: https://swift.org/about/

[11]: A. Cunningham, “Craig Federighi talks open source Swift and what’s coming in version 3.0,” *Ars Technica*, Dec. 2015. [Online] Available: http://arstechnica.com/apple/2015/12/craig-federighi-talks-open-source-swift-and-whats-coming-in-version-3-0/

[12]: Apple, “The Swift Programming Language.” [Online] Available: https://github.com/apple/swift

[13]: J. Beam, “Eagle Framework.” [Online] Available: https://github.com/joshb/EagleFramework

[14]: T. Kremenek, “Swift 3.0 Release Process.” [Online] Available: https://swift.org/blog/swift-3-0-release-process/

[15]: Apple, “The Swift Programming Language.” [Online] Available: https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/

[16]: Swift, “Package Manager.” [Online] Available: https://swift.org/package-manager/

[17]: Apple, “Kernel Queues: An Alternative to File System Events.” *Mac Developer Library*. [Online] Available: https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/FSEvents_ProgGuide/KernelQueues/KernelQueues.html

[18]: O. Kovyrin, “Using epoll() For Asynchronous Network Programming.” [Online] Available: http://kovyrin.net/2006/04/13/epoll-asynchronous-network-programming/

[19]: J. Beam, “CEpoll.” [Online] Available: https://github.com/joshb/CEpoll

[20]: Swift, “System Modules.” [Online] Available: https://github.com/apple/swift-package-manager/blob/af7a6e88b1d3ffacce8d02acd328e86a5692ecfa/Documentation/SystemModules.md

[21]: Microsoft, “Code Render Blocks,” *Microsoft Developer Network*. [Online] Available: https://msdn.microsoft.com/en-us/library/k6xeyd4z(v=vs.71).aspx

[22]: SQLite, “SQLite.” [Online] Available: https://www.sqlite.org/

[23]: M. Nottingham, “HTTP/2 Implementation Status.” [Online] Available: https://www.mnot.net/blog/2015/06/15/http2_implementation_status