Communicating Process Architectures 2011
P.H. Welch et al. (Eds.)
IOS Press, 2011
© 2011 The authors and IOS Press. All rights reserved.

Serving Web Content with Dynamic Process Networks in Go

James WHITEHEAD II

Oxford University Department of Computer Science
Wolfson Building, Parks Road Oxford, OX1 3QD, United Kingdom;

jim.whitehead@comlab.ox.ac.uk

Abstract.
This paper introduces webpipes, a compositional web server toolkit written using the Go programming language as part of an investigation of concurrent software architectures. This toolkit utilizes an architecture where multiple functional components respond to requests, rather than the traditional monolithic web server model. We provide a classification of web server components and a set of type definitions based on these insights that make it easier for programmers to create new purpose-built components for their systems. The abstractions provided by our toolkit allow servers to be deployed using several concurrency strategies. We examine the overhead of such a framework, and discuss possible enhancements that may help to reduce this overhead.

Keywords. concurrency, web-server, software architecture, golang, Go programming language,

Introduction

The construction of a web server is an interesting case study for concurrent software design. Clients connect to the server and request resources using the text-based HTTP [1] protocol. These resources may include various types of content, such as static documents, images, or dynamic content provided by some web application. In ideal conditions, the web server would handle these requests sequentially, ensuring that each is served as quickly as possible. Unfortunately, the server is often capable of producing content much faster than the client is capable of receiving it. When confronted with actual workloads in real-world conditions, web servers must be capable of responding to many clients concurrently.

There are several approaches to providing this concurrent behaviour. The ubiquitous Apache 'httpd' web server uses a combination of process and/or thread pools in order to respond to requests [2]. Lighttpd [3] and Nginx [4] both utilize an event-driven asynchronous architecture [5] to ensure scalability. Yaws [6], written in Erlang, and occwserv [7], written in occam-π, both make use of lightweight threads. In addition to the different approaches for concurrency, each web server approaches the problem of serving web requests differently.

This paper introduces webpipes [8], a compositional web server toolkit, written as part of an investigation of concurrent software architecture. This toolkit enables the programmer to construct multi-purpose web servers where the configuration and program code directly reflect the actual behaviour of the server. Built upon the premise that any web request can be fulfilled by a series of single-purpose components, webpipes allows even complicated web configurations can be expressed in a way that is clear and understandable.

The webpipes toolkit is an example of a more general component-based architecture, where networks of components communicate with each other via message passing over ex-

plicit channels in order to process and fulfill requests. Although this particular implementation makes use of specific Go language features, the architecture and techniques used should translate well to any language that supports process-oriented programming and message passing.

The main contributions of this paper are a compositional architecture for constructing processing pipelines, and an implementation of this architecture for serving web requests. Combined, these are an example of using concurrency as a software design principle, rather than a feature addition to an otherwise sequential program. Additionally, we present our impressions of the Go programming language for concurrent programming.

The rest of the paper is organised as follows: Section 1 provides a short introduction to the Go programming language and the features that are relevant to the implementation of the `webpipes` toolkit. The design and implementation of the toolkit is presented in Section 2, including both an architectural overview and details of individual server components. Section 4 addresses the performance of the toolkit, while we examine the initial conclusions and discuss future work in Section 5.

1. Go Programming Language

Go is a statically-typed, systems programming language with a syntax reminiscent of C. Programs are compiled to native code and are linked with a small runtime environment that performs automatic memory management and scheduling for lightweight processes called 'goroutines'. It also features a concurrency model heavily inspired by CSP [9] where goroutines communicate with each other via message passing over explicit channels. Pointers are a feature of the language, however the type system does not allow for pointer arithmetic.

In this paper we focus on the features of the Go programming language that are used in the `webpipes` toolkit and may be unfamiliar to the reader. In particular we will not cover the syntax or basic semantics of the language, which can be found in the official language specification [10]. More comprehensive information on the language can be found on the language website [11].

1.1. Concurrency

The feature of Go that is most relevant to this work is the built-in support for concurrency. This includes a control structure called a *goroutine*, a cross between a lightweight thread and a coroutine. Spawning a new goroutine is a matter of prefixing a function call with the `go` keyword. The evaluation of the call will execute in a separate goroutine, while the calling goroutine continues execution with the next statement. The cost of creating a goroutine is mainly the allocation of the initial stack, plus the cost of the function call itself. The stack of a goroutine is segmented, starts small, and grows on demand. This allows larger number of goroutines to be spawned without the massive resource consumption associated with using operating system threads.

Once a goroutine has been created, its subsequent activity is completely independent of its creator, except that they can share memory and may communicate with each other through channels. Shared memory among goroutines allows for efficient implementation of certain algorithms, but its use is generally discouraged. The documentation for the Go language states: "Don't communicate by sharing memory; share memory by communicating." and this pattern can be seen throughout the existing code base.

A channel in Go is an explicitly typed, first-class value that provides synchronous many-to-many communication between goroutines. Channels may carry any first-class value, including functions or even other channels.

Channels are dynamically allocated using the `make` function, which takes the type of the channel to be created and optionally the size of the channel's buffer. By setting the buffer of a channel to a value greater than 0, sends are asynchronous as long as the buffer is not full, and similarly with receives when the buffer is non-empty. Here are two example channel declarations:

```
primes := make(chan uint, 10)     // buffered uint channel with 10 slots
writers := make(chan string)      // unbuffered channel of string values
```

Sending a value `v` on a channel `ch` is `ch <- v`. Receiving from a channel `ch` is the expression `<-ch`, which can be used anywhere an expression can be used, including control structures and assignments. Channels are bidirectional when created but can be constrained to allow only sending or receiving using type conversion, as shown in the following:

```
var primes_recv <-chan uint = primes
primes_send := chan<- uint(primes)
```

The first assignment is an example of an implicit type conversion to the reading end of a channel of unsigned integers (`<-chan uint`). This conversion may also happen implicitly during assignment and the resolution of function arguments and results. The second assignment shows an explicit cast from a bidirectional channel to the sending end (`chan<- uint`). These examples also illustrate the difference between explicit variable declaration and a short form of declaration using the `:=` operator. The type of a value declared this way is automatically inferred at compilation.

Following the execution of this code there are three versions of the channel: `primes` which is still bidirectional, and two references to the send and receive ends of the same channel. Although the directionality of a channel can be reduced in this way, you cannot obtain a bidirectional channel reference from a unidirectional one.

1.1.1. Idiomatic goroutines

Since the execution of a goroutine is independent of the calling code, it is a common Go idiom to pass a channel into the goroutine so it can communicate its status to the caller. This might look something like this:

```
func doSomething(done chan bool) {
    // do some work here
    done <- true
}

done := make(chan bool)
go doSomething(done)      // spawn a new goroutine
isDone := <-done          // wait for it to finish
```

The type of the channel could be anything, in this case we chose the `bool` type, but it could just as well be a channel of numbers or strings. It is not necessary to assign the value that is received over the channel, it could be discarded by not assigning it.

In the event that a process needs to wait for more than one channel event, but cannot be sure which event will occur next, the `select` statement can be used. Similar to the `ALT` provided by `occam`, if multiple cases can proceed, the runtime system makes a pseudo-random fair choice to decide which communication event proceeds. If no case can proceed, the default case (if supplied) is taken. This allows for the implementation of non-blocking tests of send and receive events, where the communication only proceeds if it would be non-blocking.

1.2. Objects and Methods

Go does not have classes and does not support the kind of class-based object oriented style with inheritance popularised by C++ and Java. However, Go does allow the developer to declare new named types, and then allows for the definition of methods that act on these types. This style of programming is very general in that methods can be defined for any sort of data.

Take the following type definitions for binary and unary functions on integers:

```
type UnaryIntFunc func(int) int
type BinaryIntFunc func(int, int) int
```

Although it may seem a bit odd to readers who are more familiar with traditional object-oriented programming, we can define methods on these new types. For example, consider the `Curry` method for binary integer functions:

```
func (fn BinaryIntFunc) Curry(x int) UnaryIntFunc {
  return func(y int) int {
    return fn(x, y)
  }
}
```

This method takes a `BinaryIntFunc` as its *receiver*, the terminology used in Go to indicate the value on which the method is invoked, and takes a single `int` argument. It then returns a unary function in the form of an anonymous closure. This function represents the partially evaluated form of the binary function with the first argument already fixed. These definitions can be used to manually curry any binary function on integers, as in the following example:

```
var Add BinaryIntFunc = func(x, y int) int {
    return x + y
}

AddTwo := Add.Curry(2)
five := AddTwo(3)
seven := AddTwo(5)
```

We should point out a few subtleties in this example. Firstly, the declaration of the `Add` function uses the full form of variable declaration where the type is explicitly specified. This is because any implicit form of declaration will assume the simplest type, in this case the underlying `func(int, int) int` type. Since there is no `Curry` method defined for that type, the program will fail to compile. By specifying the type explicitly, we give the compiler the information it needs in order to locate the `Curry` method and compile the program. Secondly, in our definition of the `Add` function, we have omitted the type specifier for the first argument. When a function has multiple consecutive arguments with the same type, only the last type specifier is required.

The most important thing to take away from this example is that the notion of an "object" in Go is quite different to other object-oriented programming languages. Although the same sort of construct in possible in Java through the use of class wrappers and anonymous classes, the Go program is succinct and clear. This technique ends up being a very powerful feature in the design and implementation of the `webpipes` toolkit.

1.3. Interfaces Types

Rather than providing an explicit type hierarchy for objects, Go provides a way to specify the behaviour of types using interfaces. An interface type specifies a set of methods that a type must provide in order to implement the interface. For example, the `Writer` interface is defined by the `io` package in the standard libraries:

```
type Writer interface {
    Write(p []byte) (n int, err os.Error)
}
```

A type satisfies this interface if it implements a method called `Write` that takes a slice[1] of bytes and returns both an integer and an `os.Error` object (Go allows a function to return multiple results). The documentation for this interface notes that the return values should indicate the number of bytes that were actually written as a result of the call, and any error that occurred. Once the interface has been defined, it can be used as a type throughout a program: in function argument and result types, and even in further type definitions. Here are two further examples from the Go standard libraries.

```
func WriteString(w Writer, s string) (n int, err os.Error)

type WriteCloser interface {
    Writer
    Closer
}
```

The `WriteString` function is a utility function that takes in a string and converts it to a slice of bytes before writing it to the writer. By using the `Writer` interface, the `WriteString` function can be applied to any object that provides the `Write` method. This means that writing a string to a file, a network socket or to a gzipped network socket are all the same as far as the `WriteString` function is concerned.

In the specification of an interface type we can list another interface, as shown in the above definition of `WriterCloser`. This declares `WriterCloser` to be a sub-type of both the `Writer` and `Closer` interfaces; any type that implements the first can be used wherever the second or third is required.

2. Design and Implementation

The `webpipes` toolkit is a compositional web server framework that is centered around the observation that different classes of HTTP requests can be served by different single-purpose components. These components can be chained together for more complicated behaviour. This is in contrast to more conventional web server design, where requests of all classes are served by a single handler. Although there is clearly a potential advantage to the single handler design from a performance standpoint, web servers that are written in this way can be incredibly difficult to understand and configure.

Our toolkit is a layer that runs on top of the basic web server functionality provided by the `http` standard library. It provides components that enable various response behaviours, such as interacting with external applications via the CGI [12] protocol or compressing server output. These components can be composed together into processing pipelines that will han-

[1] Although slices are a higher-level contiguous view of an array, they are distinct types. In this paper, as in Go, we refer to these simply as slices.

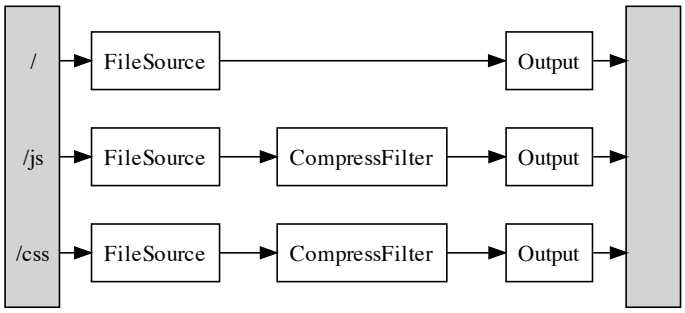

Figure 1. Process network for static file server with path-based compression

dle different classes of HTTP requests. As an example, consider a web server that serves static files but compresses any of those files served from the `js/` or `css/` subdirectories. Figure 1 shows the process network for this server, which directly reflects the desired behaviour of this server.

In this figure the leftmost and rightmost entities represent the portions of the web server that are responsible for handling the details of the HTTP protocol. The front-end of the server accepts an incoming network connection, reads a request from the client, and then dispatches it to the appropriate handler. The back-end of the web server either closes the connection or attempts to re-use it to handle another request from the same client. Handlers are represented by the paths leading from the server, and represents distinct processing pipelines that are distinguished by URL. Each of these paths ends in an `Output` component, a toolkit-provided core component that performs the actual writes to the client.

The basic function of web servers is to implement the HTTP protocol, and we have used the `http` package from the standard Go libraries to do this. The focus of the `webpipes` toolkit is on providing the content. This involves both the header and the body of the HTTP response.

2.1. Allowing for Collaborative Response Generation

In the simple web server model, where each request is fulfilled by a single handler, the logic is very simple: the handler first sets and outputs the headers, and then writes the body of the response directly to the client. This matches the output required by the protocol. Unfortunately, this does not work as soon as we allow multiple components to be involved in the generation of a response— if the first component started writing the content, then it would already be too late for any of the later components to set headers or examine the content being generated.

The `webpipes` package addresses this problem by introducing a set of new types that are used to de-couple the setup of a response from content generation, by removing direct access to the underlying network connection and providing an alternate means of transmitting a response. This is accomplished through the `Conn` type, which is a `struct` that represents an HTTP client connection. This type encapsulates the request itself and provides a way of generating a response by defining the following methods:

```
// Response headers and status code methods
func (c *Conn) SetHeader(string, string)
func (c *Conn) GetHeader(string) string
func (c *Conn) SetStatus(int)
```

```
// General utility
func (c *Conn) HTTPStatusResponse(int)

// Content methods
func (c *Conn) NewContentWriter() io.WriteCloser
func (c *Conn) NewContentReader() io.ReadCloser
```

The first three methods allow a component to set and get the headers and the numeric status code for the response. The `HTTPStatusResponse` method is a utility that provides a canned HTTP status response, such as 404 (Not Found) and 500 (Internal Server Error). These responses normally include content that indicates what sort of error has occurred to the user, and is used frequently in writing web components, so it is included as a utility method.

The final two methods are the ones that enable the package to break response generation into two steps, by allowing for the allocation of pipelines of content readers and writers. A content writer is an object that can write to the output steam of the response, while a content reader provides a way to read content has been written by components earlier in the pipeline. A component that produces content is called a *source* and uses a content writer, while *filter* components use both a content reader and writer to examine or transform the content.

These readers and writers are created in pairs, starting with a content source, and thus a call to `NewContentWriter`. The `Conn` object creates both a writer and a corresponding reader, storing the reader so it can be returned from the next call to `NewContentReader`. Subsequent invocations create new pairs of writers/readers as needed. This means that an individual component will only have one half of a pair, and all pipeline ends will be connected into a content pipeline. Operations on these readers and writers are synchronous.

Figure 2 shows a version of the same server from Figure 1 augmented to display the content pipeline that is created for each of the different paths. Each "W" represents a content writer, and each "R" indicates a content reader.

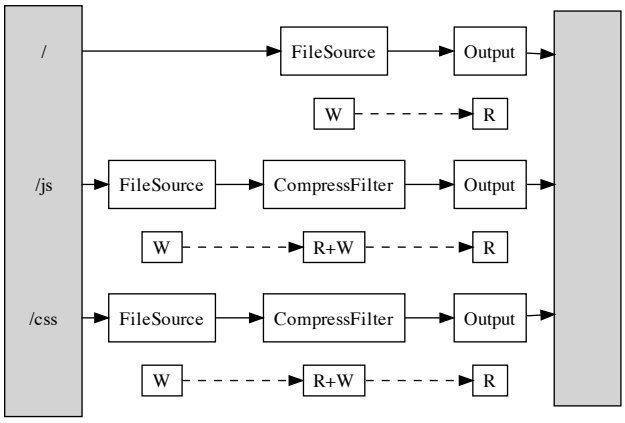

Figure 2. Static file server process network with dynamic content pipeline

2.2. Understanding Components

Up to this point, we have used the term *component* to mean a portion of the web server that is partially responsible for responding to a request. The `webpipes` package explicitly defines

the `Component` interface type with a single method that takes a `Conn` object along with the request and returns a boolean value:

```
type Component interface {
    HandleHTTPRequest(*webpipes.Conn, *http.Request) bool
}
```

This boolean value indicates whether or not the connection should be passed to the next component. If a component ever returns `false` it should be prepared to manually perform output and close the connection, however this operation is rate. Consider a component that responds to every request with the contents of a static string, in a plain text response[2]:

```
type StrComp struct {
    content string
}

func (s StrComp) HandleHTTPRequest(c *Conn, r *Request) bool {
    writer := c.NewContentWriter()
    if writer == nil {
        c.HTTPStatusResponse(http.StatusInternalServerError)
        return true
    }
    c.SetHeader("Content-Type", "text/plain; charset=utf-8")
    c.SetStatus(http.StatusOK)

    generate := func() {
        io.WriteString(writer, s.content)
        writer.Close()
    }

    go generate()
    return true
}
```

Since this component is a source, the first thing we do is request a content writer from the connection. If the connection is unable to return one for some reason, this means there has been an internal server error, so we indicate this to the client using the `HTTPStatusResponse` method. This should never happen, as it indicates a semantic error in the component pipeline, such as a source component appearing somewhere other than the start of the pipeline.

Once we've obtained the content writer, we set the header and the status on the response. Next we create a function that will actually perform the output of the content and then invoke it in a new goroutine. We then return `true` to indicate that the connection should be passed onto the next component.

The operations that are taken in the main goroutine, such as allocating the content writer, setting the status code and header, are all part of the *setup phase* of responding to a request. The work performed in the `generate` function, and thus in the "content goroutine", is part of the *content generation phase* of request fulfillment. When the `HandleHTTPConnection` method call returns, it indicates that the setup phase is complete and that the connection can be sent to the next component in the network.

[2]Package names in this example have been omitted for formatting reasons. As indicated in the interface declaration, the `Conn` type is defined by the `webpipes` package and the `Request` type is defined by the `http` package. In a typical program, all type references are normally fully qualified.

In order to use this component, we would need to create a new instance of the `StrComp` struct type, set to contain the string of our content. For example:

```
hello := StrComp{content: "Hello World!"}
```

2.3. Classifying Components

Although the `Component` interface type allows very general components, we have already identified two kinds of components that might exist in a processing pipeline: sources and filters. Each of these is a specialisation of a more general type, called a *pipe*. These components do not normally concern themselves with the content stream, although they can always request content readers and writers from the connection object. Pipes are often used to examine the request or response headers and take some action, such as re-routing the request to another portion of the network or logging a message to the console.

The `webpipes` packages takes these classifications and provides a set of corresponding type declarations that assist in the creation of new components by automatically handling the boilerplate allocation of content readers and writers:

```
type Pipe func(*Conn, *Request) bool
type Source func(*Conn, *Request, io.WriteCloser) bool
type Filter func(*Conn, *Request, io.ReadCloser, io.WriteCloser) bool
```

These type declarations allows the programmer to write components that are simply functions. This avoids the need to create a type wrapper and define an `HandleHTTPConnection` method. Here's a `TextStringSource` that works quite similar to the `StrComp` component we've just defined in a more compact and readable form:

```
func TextStringSource(content string) Source {
    return func(conn *Conn, req *Request, writer io.WriteCloser) bool {
        conn.status = http.StatusOK
        conn.SetHeader("Content-type", "text/plain; charset=utf-8")

        generate := func() {
            io.WriteString(writer, content)
            writer.Close()
        }

        go generate()
        return true
    }
}
```

This code defines a function that produces a text string component based on the `content` argument. Because this component is declared as a `Source`, it will automatically inherit the `HandleHTTPConnection` defined by the `webpipes` package.

2.4. Writing a Filter Component

Filter components allow the programmer to examine and transform the contents of a response before it is output to the client. This may be as simple as performing compression on the output stream or something as complicated as performing semantic analysis of the output and altering it in some way. The body of a filter component looks quite similar to that of a source component, with the exception of the content goroutine. Rather than being able

to unconditionally write the content to the response, the filter first use the content reader to obtain the data. Here is an example of a filter that performs rot13 encryption, a simple substitution cipher, on any alphabetic characters in a content stream:

```
var Rot13Filter Filter =
    func(conn *Conn, req *http.Request, reader io.ReadCloser,
        writer io.WriteCloser) bool {

        filter := func() {
            // Wrap the reader so all reads come out rot13'd
            rot13 := NewRot13Reader(reader)
            io.Copy(writer, rot13)
            writer.Close()
            reader.Close()
        }
        go filter()
        return true
    }
```

This filter makes use of two functions that we haven't seen before. The `NewRot13Reader` is a function provided by the toolkit. This function takes in an `io.Reader` and returns another that performs on-demand reads from the inner `io.Reader`, encrypts the data using the rot13 cipher, and returns the result. The `io.Copy()` function is a utility provided by the standard libraries to copy all of the data from a writer to a reader. If the programmer chooses, they could manually read in the blocks of data from the content reader and output them to the content writer; the `io.Copy` is just a utility that helps to automate this mechanical process when the transformation can be expressed using readers and writers.

Other than spawning the content goroutine to perform the filtering, this component does not perform any other work: it does not need to change the status of the response or set any additional headers. In a more realistic example, such as changing the encoding or compressing the stream, the client would need to be notified via headers so they would know how to decode the data.

Although the filter components that are presented in this work focus on textual transformations, it is also possible to alter binary content such as images. This could be used to add watermarks to images on-the-fly, replace images with lower-quality versions on older clients or any number of other complicated behaviours.

2.5. Pipe Components and Branching Networks

The most general component type, `Pipe` is only a thin wrapper around the base `Component` type, providing a `HandleHTTPConnection` method that just invokes the pipe function itself. By default, a pipe component does not need any content readers or writers, so there's no additional work for the method to perform.

A good example of a simple pipe component is one that logs incoming requests to a file for later review. Such an access logging component does not need to know anything about the content being written. It just takes information about the client, request, and response and logs an entry.

A more complicated pipe example is one that makes routing decisions based on details provided in the request, such as an authentication component. The HTTP simple authentication protocol requires clients to specify credentials using headers and allows the server to examine and validate these credentials before proceeding with the request. If the credentials are not valid, the client may be prompted by their web browser or may simply be denied ac-

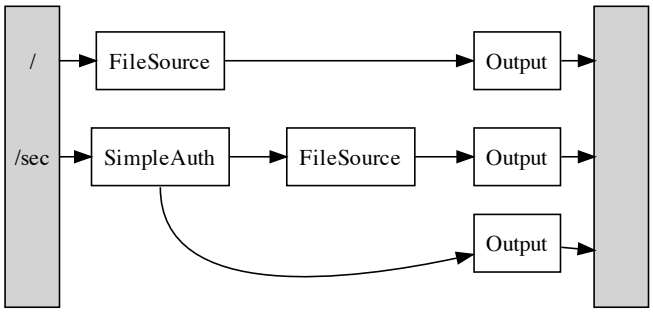

Figure 3. File server with private area requiring authentication

cess to the resource, depending on configuration. This sort of construct requires a branching network, an example of which is shown in Figure 3.

Rather than a non-branching path from start to finish, there are two edges leading from the authentication component that correspond to these different behaviours. Although branching networks are slightly more difficult to construct, the components have the same type and semantics as any other component.

2.5.1. Conditional Components

It is possible that a source or filter component may not apply to every single request it encounters. This is a problem since the `HandleHTTPConnection` method provided by the toolkit automatically allocates the content readers and writers and the toolkit doesn't provide a way to de-allocate them. In this case, the programmer can either construct their own type that implements the `Component` interface or they can utilize a pipe component that then invokes another component when the conditions are correct.

For example, not all clients are able to support compression of content. This information is only available from the headers that are sent by the client along with the request. General compression components should ensure that the content is only compressed by a supported method, if any. This could be accomplished by taking the following steps:

1. Check the HTTP protocol version to see which headers need to be examined.
2. Check the headers to see if any of the compression methods that are supported by the server are also supported by the client.
3. If no supported compression methods are supported, return `true` and make no alterations.
4. Otherwise, invoke the `HandleHTTPConnection` method of the appropriate compression filter, passing in the connection and request in order to insert the compression filter into the content stream.

Accordingly, the `webpipes` toolkit provides unconditional components that perform gzip and flate compression, called `GzipFilter` and `FlateFilter`, and a conditional component that performs the appropriate header checks, called `CompressionFilter`. The programmer is free to include whatever component is most appropriate for the desired configuration.

3. Creating Component-based Web Servers

We have introduced the compositional web server architecture provided by the `webpipes` toolkit and introduced the different classifications of server components. This section explains how we can use the very basic server provided by the `http` package along with our toolkit to construct function web servers using these components.

3.1. Default Multiplexing Server

The `http` package provides a very simple multiplexing server. That is, the server utilises a single network connection, but allows multiple handlers to respond to incoming requests. In order to accomplish this, handlers must be explicitly registered with the server along with a URL prefix string that is used to match requests to the appropriate handler. The `webpipes` toolkit allows you to take advantage of this server by ensuring that the created component chains conform to the `http.Handler` interface that is required by this server.

The following is an example server with two handlers:

```
func main() {
    http.Handle("/", webpipes.Chain(
        webpipes.FileServer("http-data", "/"),
        webpipes.OutputPipe,
    )
    http.Handle("/hello", webpipes.Chain(
        webpipes.TextStringSource("Hello, world!\n"),
        webpipes.OutputPipe
    )
    server := http.Server{
        Addr: ":12345",
    }
    log.Printf("Starting test server on %s", server.Addr)
    err := server.ListenAndServe()
    if err != nil {
        log.Fatalf("Error: %s", err.String())
    }
}
```

The first handler is a chain of components that serves files from the `http-data` subdirectory using a component created using the `FileServer` function. This function takes a base directory and a prefix and serves files from that directory, stripping the prefix from the start of the URL. If this were the only handler registered with the server, any well-formed request would be routed to it, since any such request would begin with `/`.

The second handler is for any request that begins with the URL `/hello`. The server automatically finds the longest prefix match between the request and the registered handlers, so this handler is able to provide a more specific pattern than the file server without any issues. Each client that is routed through this handler is sent the text string `"Hello, world!"` as text/plain content using the previously introduced `TextStringSource`. Both component chains include the `OutputPipe` component, which handles the transmission of the response to the client's network socket.

The last few lines of this program are boilerplate that create a new `http.Server` object with the `Addr` member set to the host and port on which the server should bind. We then invoke the `ListenAndServe` method on this server to begin listening for client connections. The server will continue until an error occurs in the accepting of new connections

and will report this error before exiting. There are a few additional attributes you can set on the `http.Server` object including socket read and write timeouts, but for many servers the above code is sufficient.

3.2. Constructing Component Chains

In the preceding example, the `webpipes.Chain` function was used without explaining its purpose. This function takes any number of components as arguments and returns an object that implements the `http.Handler` interface, allowing it to be used directly as an argument to `http.Handle`. These components are stored in an array and when a connection arrives, each component is applied to the connection in turn; a simple form of functional composition. For many web server configurations, this sequential execution of components is likely the most efficient. Each connection is served as quickly as it can be, subject to the number connections currently being processed by the server.

The `Component` interface defined by the `webpipes` package, however, only defines the behaviour of components and how they interact with each other; it doesn't specify anything about how the components are connected together or how they are actually executed when handling a client connection. The `Chain` function is just one example of how component networks might be connected and executed.

In the case where a processing pipeline is not necessarily sequential, such as the previously mentioned `SimpleAuth` component that requires branching, the `webpipes` toolkit provides additional methods of constructing handlers from components. The `ProcNetwork` function and its variants can be used to turn a list of components into a static process network, while the `NetworkHandler` function can take the input and output channel for a network of processes and allow it to be used as a handler in the default web server.

When a list of components is passed to the `ProcNetwork` function, a small goroutine is started for each of them that sits and accepts incoming connections from the channels that connect each component to its neighbors. When a connection is received, the component function itself is invoked and when this function concludes, the connection is passed along the network.

The `ProcNetworkInOut` function is the same, but allows you to specify the input and output channels for the overall network. This function is really only used when the programmer is constructing their own process networks and channels, but helps connect portions of a process network together. Either of the channel arguments may be `nil` and the function will create a new channel for you. Both versions of the `ProcNetwork` function always return the input and output channels for the constructed network.

For example, the following code allocates four channels (two for overall network input and output and two to connect the components together) and spawns three goroutines that will act as server farms for each of the components.

```
in, out := webpipes.ProcNetwork(
    webpipes.FileServer("http-data", "/"),
    webpipes.CompressPipe,
    webpipes.OutputPipe,
)
```

Once created, these channels can then be passed to `NetworkHandler`, enabling the process network to be used as a handler in the web server:

```
http.Handle("/compressed", webpipes.NetworkHandler(in, out)
```

When a request is received, the web server invokes the `ServeHTTP` method that is defined by the `webpipes` package for network handlers. This injects the connection into the process

network via the input channel and then waits for the connection to emerge on the output channel. Once received, the function returns, allowing the http server to close or re-use the client connection as appropriate.

3.2.1. Manual Process Networks

It is conceivable that a programmer would like to create their own network of processes and then use it as a handler in the default web server. The `NetworkHandler` function can also be used for this purpose, as long as the network has an input and an output channel that are exclusively open-ended; that is no other process should attempt to write to the input channel or read from the output channel.

In reality, anyone who is creating their own process network and server farms should probably implement the `http.Handler` interface directly, since it will give them better control over how requests are handled.

4. Performance and Comparison

Web server performance is typically measured using some combination of throughput and latency. Throughput indicates the number of requests that can be served per second, while latency indicates the amount of time taken to serve a request. These micro-benchmarks are performed under varying loads, used to simulate a multiple clients attempting to access the web resources simultaneously.

We have measured the performance of both `webpipes` and the stock Go `http` package against the Apache 'http' web server, representing the ideal case in these benchmarks. It is not entirely fair to compare a fledgling programming language with an unoptimised `http` library against a well-tested production web server. However, this enables us to show that there are no artificial bottlenecks in the testing environment by measuring a server that with ideal performance within our testing ranges.

The benchmark we performed involved requesting a static 4KB file multiple times across a multitude of concurrent connections, to illustrate the throughput of the server in terms of requests per second. These tests were performed in the following test environment:

- Server: 3.00Ghz Intel Core 2 Duo with 4GB of RAM, running Ubuntu 10.4.1 and Linux kernel 2.6.32-24.
- Clients: 1.66Ghz Intel Centrino Duo with 1GB of RAM, running Ubuntu 10.10 and Linux kernel 2.6.35-22.
- Switch: HP Procurve 2724 Gigabit switch.

The 'httperf' [13] program was used to perform the benchmarks by generating a sustained load against the server. In addition, we created a tool called 'autohttperf' [14] that distributes the load generation across multiple client machines and can perform automated stress testing of web servers.

Figure 4 shows the number of successful requests per second against the number of attempted requests per second and Figure 5 shows the percentage of connection/request errors against the number of attempted requests per second. You can see that Apache maintains the ideal result, being able to serve all incoming requests within the client timeout window. On the other hand, the Go `http` web server with no framework overhead is able to sustain just under a third of this amount before it reaches its saturation point. This saturation point is artificial [3]– it is a hardcoded number of connections that are allowed to be active at any point in time. In my test environment, without this limit, the web server would reliably crash due

[3]There is currently a bug in Go that causes performance measurements of Go web servers to be unreliable above this point. At the time of publication, these bugs have been reported but are outstanding.

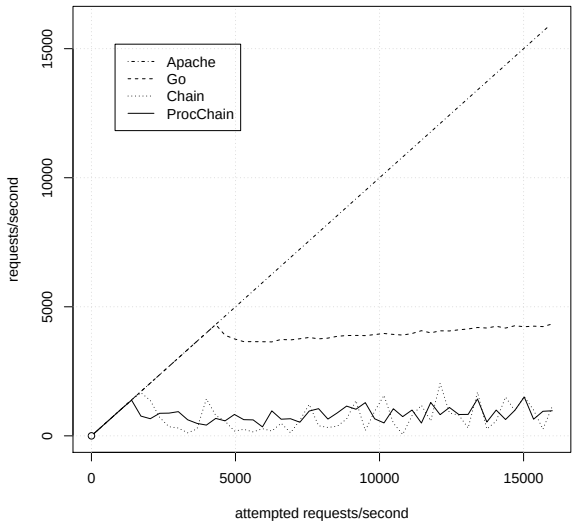

Figure 4. Web server throughput

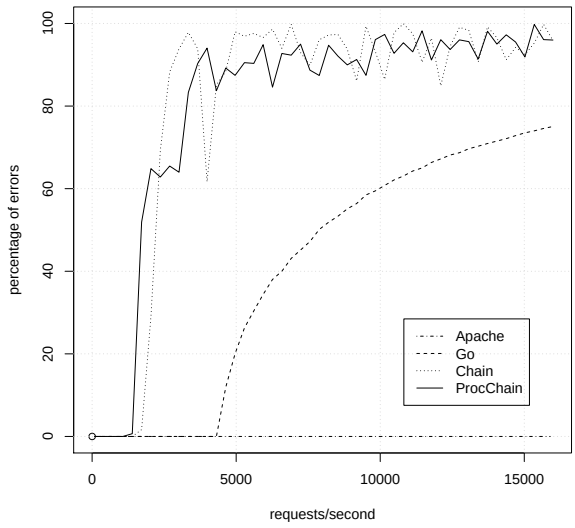

Figure 5. Percentage of connection errors

to memory allocation errors. These measurements do, however, represent a lower bound on what the server is capable of.

The performance of the `webpipes` framework is shown using two different methods of connecting components together, `Chain` and `ProcChain`. Recall that the first uses function calls and method invocation to invoke components, while the second has the additional synchronization overhead of a channel that is connecting each process. The load that is able to be sustained by a server using our toolkit is reasonable for low-traffic situations, it remains ideal until somewhere between 1500-2000 requests per second, but performance above that is difficult to measure due to the related bugs in Go.

4.1. Identifying overhead

For a compiled language with a relatively simple `http` package and web server, it is initially surprising that the Go-based web server is unable to achieve better performance under heavier loads. However, it turns out that this specific type of test, where a small static file is being delivered to the network client as part of the response, is an optimised case in the Apache server implementation. Specifically, it makes use of the `sendfile` [15] system call which can be used to copy data from a file to a socket. This system call is an optimisation over the standard read/write in that it attempts to avoid kernel/user context switches and in some cases is capable of performing a zero-copy [16] transmission of the file data. We believe that the performance of the Go `http` package could be improved through the use of this system call.

Currently, the test we are performing hits a non-optimal path through the I/O code in Go's libraries:

1. The headers are written by the file handler to the response. These writes are performed using a buffered network writer with the default limit of 4KB of buffer space. Since the header data is smaller than this limit, the write is buffered until later.
2. The file is read in 32Kb blocks using the `read` system call. Since the file in question is under this limit, it only requires a single call to `read`.
3. The data is written to the `http.ResponseWriter` object which then performs the write on the underlying buffered network connection without copying.

4. The data is then written to the buffered network connection. Because the buffer is not empty, this either requires the headers to be send in a different network packet from the payload or data to be copied. The `bufio` in this case will perform a copy of the file data to fill the non-full buffer and then flush that to the client. The remaining file data is then sent without requiring a copy.

The `webpipes` framework introduces even more overhead with its system of content readers and writers. Currently, these are implemented using the `io.PipeReader` and `io.PipeWriter` interfaces which explicitly copy data on every paired read/write operation. For the static file test performed above, this means that in addition to the buffering issue identified above, the file data is copied an additional time before ever being written.

Interestingly, in our tests we do not show a significant difference between the `Chain` and `ProcChain` versions of constructing processing pipelines despite the heavy use of short-lived goroutines in the latter. However, it is possible that any differences between the two may be masked by the outstanding issues with the Go web server. If these results hold, it will be an encouraging result for programs that make heavy use of goroutines.

5. Conclusions and Future Work

In this paper, we have presented a compositional architecture for constructing processing pipelines. This architecture was demonstrated through the `webpipes` toolkit, a framework for constructing component-based web servers using the Go programming language. This framework makes use of a series of type and semantic definitions that allow multiple components to participate in the generation of responses. This is accomplished by de-coupling the generation of response headers from the generation of the actual content of the response. This toolkit encourages the use of simple components that are strung together to create more complicated behaviours.

The behaviour of web servers constructed using the `webpipes` toolkit are easy to understand; often there is a direct correlation between the structure of the code used to create a server and the definition of that server's behaviour. The same might be accomplished in mainstream web servers through the clever use of domain specific configuration languages, however we believe the component model used by the `webpipes` toolkit is inherently easier to understand.

This work is the product of an initial case study in concurrent software design. Throughout the design process we have restricted ourselves to a model of concurrency where components communicate with each other solely via message passing over explicit channels. In order to accomplish this, it was necessary to determine an appropriate level of abstraction for processes and define the nature of the communication between these processes. Our design follows the premise that a web request can be fulfilled by a series of single-purpose components. We chose to model these notions of components as concurrent processes. The components communicate with each other, passing an abstraction connection object. This object contain the details of the request as well as a way for components to construct the response. By thinking about how web server behaviour might be modelled as a collection of concurrent processes, this abstraction came quite naturally.

5.1. Impressions of Go

Go is a relatively young entry in the field of programming languages, released in November 2009. It offers a syntax that is familiar to C, C++ and Java programmers while providing a novel interface-based type system and channel-based concurrency. We initially explored the Go programming language in order to understand how it relates to other concurrent program-

ming languages, specifically Scala, Erlang, and occam-π. After working with the language for some time, we chose it as the target language for our case study. This choice was primarily motivated by the following: /safe

1. The ability to define new concurrent processes without need for additional wrappers or classes. This enabled rapid prototyping and experimentation without verbose programs.
2. The simple concurrency model built around synchronous communication via first-class channels.
3. The "safety" provided by the static typing, lack of pointer arithmetic, and requirements for explicit casting.

Through our experience implementing `webpipes`, we have found the first point, combined with the ability to define methods on any data type, to be incredibly important. This allowed us to create an abstraction for components that made them easy to use when constructing new web servers. In addition, the ability to provide methods for functions gave us the ability to provide boilerplate code for components, making it easier for developers to add new functional components to the system with minimal code.

Although the pervasive use of interface types can take some getting used to, their consistent use allows easy interoperability with the rich set of standard packages included with the language. It is a testament to these libraries that we can implement a fully functional compositional framework for constructing web servers in under 1500 lines of code!

On the other hand there are opportunities for improvement with the Go programming language. In particular the goroutine scheduler is immature and does not always scale well when multiplexing goroutines onto more than one operating system process. Additionally, there are currently two sets of compilers for the language. The `gc` compiler set is a very fast compiler that can produce reasonable code, while the `gccgo` front-end is much slower but can produce more efficient code. The latter is currently not able to multiplex goroutines and instead uses operating system threads, although this feature is expected to be implemented in the future.

Despite the relatively young age of the programming language, we believe that Go helps to fill an interesting niche in the field of programming languages. The unique feature-set and aims of the language make it worth investigation for systems-level concurrent programming.

5.2. Future Work

Through the development and evaluation of the `webpipes` toolkit, we have identified many opportunities that require further investigation. The overhead introduced by the content reader/writer system negatively affects the performance of the toolkit. In addition it is not a very idiomatic style of writing Go code. It would be worth investigating a design removes the explicit use of content readers and writers by providing a higher level of abstraction for the registration of filter components. This abstraction could remove an additional level of copying and synchronization from the toolkit and would be much closer to the way Go programs are normally written.

The use of pointers in a concurrent program introduces the possibility of race conditions and non-deterministic behaviour when components retain the reference and access it directly. This problem might be alleviated by introducing a concept of ownership to the type system, perhaps utilizing some form of linear types [17] or uniqueness typing [18]. This would allow us to specify at the language level that only one reference to an object should exist at any time.

The Go language provides for a form of distributed channels via the `netchan` package. It would be interesting to develop a web server that acts as a load balancer using the component-based architecture presented in this paper and these network channels in order to distribute

the incoming load to multiple back-end web servers. It is possible that may expose other advantages or drawbacks of our architecture.

References

[1] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC1945: Hypertext Transfer Protocol–HTTP/1.0. *RFC Editor United States*, 1996.

[2] Apache: HTTP Server - Multi-Processing Modules (MPMs). `http://httpd.apache.org/docs/2.0/mpm.html`, October 2010. URL `http://httpd.apache.org/docs/2.0/mpm.html`.

[3] Lighttpd Web Server. `http://www.lighttpd.net/`, August 2010. URL `http://www.lighttpd.net/`.

[4] Nginx Web Server. `http://nginx.org/en/`, April 2011. URL `http://nginx.org/en/`.

[5] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 230–243. ACM, New York, NY, USA, 2001. ISBN 1-58113-389-8. URL `http://doi.acm.org/10.1145/502034.502057`.

[6] Yaws - Yet Another Webserver. `http://yaws.hyber.org/`, September 2010. URL `http://yaws.hyber.org/`.

[7] Fred Barnes. occwserv: An occam web-server. In J.F. Broenink and G.H. Hilderink, editors, *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering Series*, pages 251–268. IOS Press, Amsterdam, The Netherlands, September 2003. ISBN 1-58603-381-6. ISSN 1383-7575. URL `http://www.cs.kent.ac.uk/pubs/2003/1720`.

[8] James Whitehead II. Webpipes: A Compositional Web Sever Toolkit. `http://github.com/jnwhiteh/webpipes`. URL `http://github.com/jnwhiteh/webpipes`.

[9] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985. ISBN: 0-131-53271-5.

[10] The Go Programming Language Specification. `http://golang.org/doc/go_spec.html`, March 2011. URL `http://golang.org/doc/go\_spec.html`.

[11] The Go Programming Language. `http://golang.org/`, March 2011. URL `http://golang.org/`.

[12] D. Robinson and K. Coar. The common gateway interface (CGI) version 1.1. Technical report, RFC 3875, October 2004.

[13] David Mosberger and Tai Jin. httperf: a tool for measuring web server performance. *SIGMETRICS Perform. Eval. Rev.*, 26:31–37, December 1998. ISSN 0163-5999. URL `http://doi.acm.org/10.1145/306225.306235`.

[14] James Whitehead II. autohttperf: Automated and distributed benchmarking using httperf. `https://github.com/jnwhiteh/autohttperf`. URL `http://github.com/jnwhiteh/autohttperf`.

[15] Erich Nahum, Tsipora Barzilai, and Dilip D. Kandlur. Performance issues in www servers. *IEEE/ACM Trans. Netw.*, 10:2–11, February 2002. ISSN 1063-6692. URL `http://dx.doi.org/10.1109/90.986497`.

[16] Dragan Stancevic. Zero copy I: user-mode perspective. *Linux J.*, 2003:3–, January 2003. ISSN 1075-3583. URL `http://portal.acm.org/citation.cfm?id=603771.603774`.

[17] P. Wadler. Linear types can change the world. In M. Broy and C.B. Jones, editors, *Programming concepts and methods: proceedings of the IFIP Working Group 2.2/2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, pages 561–581. North-Holland, 1990. ISBN 9780444885456. LCCN 90045970. URL `http://books.google.com/books?id=ycVQAAAAIAAJ`.

[18] Edsko Vries, Rinus Plasmeijer, and David M. Abrahamson. Implementation and application of functional languages. chapter Uniqueness Typing Simplified, pages 201–218. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-85372-5. URL `http://dx.doi.org/10.1007/978-3-540-85373-2_12`.