UNIVERSITY OF APPLIED SCIENCES

WEDEL

COMPUTER SCIENCE DEPARTMENT

Master's Thesis

A Dynamic Webserver with Servlet Functionality in Haskell representing all internal data by means of XML

Christian Uhlig

23.10.2006

Table of Contents

List of Figures

List of Examples

1. Preface

The objective of this thesis is the design, development and discussion of a dynamic web server in the pure functional programming language Haskell. This server system will be called **Janus**. The property "dynamic" will be based on the concept of Servlets known from the object oriented language Java. A central aspect of the resulting system will be the extensive usage of XML data structures to represent internal processing. For the XML representation and processing, the project will revert to the Haskell XML Toolbox of Uwe Schmidt at the University of Applied Sciences Wedel.

The design and its implementation will be inferred stepwise throughout later parts of the thesis, after a summary of technical basics, standards and existing work. It will focus on the programming of web applications with a complexity and capabilities comparable to Java Servlets while introducing an unprecedented level of flexibility, reusability and extensibility. As an example, a fixation on HTTP for transport (including techniques like CGI) and HTML for interface description (including forms and scripting) can be considered to be of limited value for today's applications. If one imagines a web shop system, an existing human interface to browse the assortment, dispatch orders and track shipping might need an additional formal access facility. E.g. software systems of partner companies could use this to query the stock or price of a certain product. Typically, such an extension would be implemented by an XML based web service nowadays. Another idea would be to provide an email interface for the web shop logic - not only to send mails accompanying the order procedures, but eventually allowing to dispatch orders simply by an email or query stock and prices of products. Obviously there should be a distinction between communication protocols, presentation and business logic even in this simple example. Therefore Janus will try to avoid any restriction on a specific implementation regarding these three aspects of an application.

Much work already exists concerning advanced and interoperational hosting of web applications in Haskell. Near the conclusion, we will spend some time on comparing Janus to these solutions and discussing several pros and cons.

Starting with chapter 2 this thesis will give an introduction to functional programming in general and programming in the pure functional programming language Haskell (based on the Haskell 98 standard) in particular. Afterwards, in chapter 3 we shall

have a look on what is required to build a dynamic web server and discuss these requirements in the context of Haskell. In the same chapter we will have a look at dynamic web server techniques, especially Servlets like they were introducted in the Java community. And finally chapter 3 will give an introduction to several advanced programming techniques and libraries available for Haskell which are not necessary to build a web server in Haskell but which contribute useful properties to our actual system design. Here we will introduce the Haskell XML Toolbox, the foundation for XML processing in Janus and for the utilized Arrows abstraction. Up to this point, the thesis will have build up the foundation for the Janus system design, which will be introduced next in chapter 4. It will be described on an abstract level first, stepwise deriving conceptual extensions and discussing design decisions. Chapter 5 will tie up to this by explaining the concrete implementation of the primary design components. Additionally there will be an insight into interfaces and helper modules to complete the chapter with a simple web application example.

Chapter 6 will provide a series of server configuration examples with increasing complexity. These examples shall illustrate the system design's core ideas and the practice of building a concrete Janus server. Next we will show two concise examples for Janus extensions in chapter 7 and 8: First a wrapper to provide a Java Servlet-like programming style and second a general purpose SOAP extension implemented directly by means of servlets.

Last but not least, chapters 9 and 10 provide our reflection. Chapter 9 discusses comparable solutions like WASH/CGI, chapter 10 concludes our own view on Janus and states some examples for future work.

2. Functional Programming and Haskell

2.1. Imperative vs. Functional Programming

Because Janus is to be implemented in a functional programming language, we have to examine what a functional programming language actually is, what is special about implementing a web server system in it and which language and library extensions are possibly required to do it.

During the history of computer science, a number of so called programming paradigms have emerged, which in essence represent different approaches to describe algorithms. While there are programming concepts considered to be "paradigms" although not based on a unique formal algorithm concept[1], the two most important paradigms to really denote different formal algorithm concepts are the imperative and the functional programming paradigms. However, please note that there are further and less common paradigms also representing unique formal algorithm concepts (e.g., logic programming).

Imperative programming can be considered to be the most common programming concept, which is teached to almost every computer scientist during his education. The fundamental idea is to incrementally transform an initial state into a final state by means of statements **[1]**, where the algorithmical problem's instance and its solution are contained in the state. The essential statement is the assignment statement which allows assigning a new value to a part (typically a variable) of the state. The whole idea is similar to the model of a Turing Machine. The Turing Machine is an imaginary model of computation established by Alan Turing **[2]**, based on the stepwise reading and writing of an infinite band and the transition of a machine state. Backed up by the Church-Turing-Thesis **[2]** (which is unproved, but commonly considered to be true), the set of algorithms represented by the Turing Machine is equal to or a superset of any other set of algorithms defined by a computational model or an intuitive concept of algorithms. The property of Turing Completeness is used to express the equivalence of a given algorithmical concept or general purpose programming language to the Turing Machine.

---

[1] E.g., object-oriented programming as an extension of imperative programming **[1]** and design-by-contract as an extension to the object-oriented paradigm **[3]**

In the case of imperative programming a possible definition to gain Turing Completeness for a language based on the idea of state transformation demands the following elements **[1]**:

- A program is comprised of a sequence of statements and a state (variables) to operate on.
- There exist assignments to change the state (value) of variables.
- There exist iterations (statement loops).
- There exist selections (conditional statements).
- There exist integer variables and support for integer arithmetic.

In practice no imperative language is restricted to these language components; although sufficient to formulate every algorithm imaginable, additional facilities are required to improve expressiveness of the language (how much code is required to describe a given algorithm?), software quality aspects (modularisation, reusability, readability, ...), etc.. For example, every current imperative language provides functions and procedures to structure program code.

As seen in imperative programming a program is a sequence of statements transforming a state. While there typically exists the notion of functions, in a mathematical sense these functions are procedures delivering a value because they can (and often do) depend on the program state and/or cause side effects on this state. This implies that an imperative function can deliver different results for the same input, and for two identical results it may cause different effects on the state (influencing subsequent statements). Functional programming, in contrast, defines a function in the mathematical sense: For the same input it shall deliver the same value and a function shall have no result (like a side effect on state) beside its return value. A program is considered to be simply an expression; the evaluation of this expression represents the program execution. Assignments do no exist in this model, just as little as iterations (because without an updateable state, a loop cannot produce any effect). One may already anticipate that this paradigma, if implemented strictly, may bring up questions regarding user interaction or interaction with the outside world in general. We will consider these issues later, for the moment we are talking about algorithms only. It may not be obvious that a program without state

(which is represented by updateable variables) and loops can express the same things as an imperative language. Functional programming, in contrast to the Turing Machine of the imperative paradigm, is based on the Lambda Calculus **[1]**. The Lambda Calculus, defined by Alonzo Church even before the Church-Turing-Thesis and the Turing Machine were established, is proven to be equivalent to the Turing Machine and its set of algorithms. It consists of constants, variables (not updateable ones, naturally), application of a function on a single argument and the so called Lambda Abstraction, which defines new (anonymous) functions. For a functional language to be Turing-complete, a possible minimal definition contains the following elements **[1]**:

- Like with imperative languages, integer values and arithmetic have to be supported.
- There has to be a facility to define new functions based on existing functions.
- There exist selections (conditional expressions).
- There exists recursion in functions (this is the central control structure in contrast to loops used in the imperative world).

We shall have a look at a simple example to illustrate the most striking differences of functional and imperative programming. Let's define a function to increment each element of a list of integer numbers by an argument value. First the imperative solution will be coded in Java utilizing an array to represent the list. Afterwards the functional solution will be coded in Haskell utilizing Haskell lists:

Example 1 - Java iteration vs. Haskell recursion

```
int[] incList (int[] numbers, int add) {
   for(int i=0 ; i<numbers.size ; i++)
      numbers[i] += add;
   return numbers;
}

incList []     _   = []
incList (x:xs) add = (x+add):(incList xs add)
```

We will not explain the code in detail, the two implementations only shall emphasive two important differences: The Java code replaces values in the original array and therefore entails a side effect on a state and any code working it (assuming this

array to be part of a shared state). It incorporates a loop to traverse the array. The Haskell code, on the other hand, constructs a new list (due to the lack of state and an assignment statement) and utilizes recursion to traverse the original list.

Another important property of functional programming is the treatment of functions as first-class values, enabling the programmer to dynamically construct functions and hand them to higher order functions. However, it has to be emphasized that this property can also apply to imperative languages while typical and necessary (in terms of Turing-completeness) for functional programming.

At this point, we finish with introducing the difference between functional and imperative programming with a short conclusion. As mentioned in the introduction, the Janus system shall be coded in the functional programming language Haskell. The discussion above is important to understand why this is not a self-evident target - a web server can be considered to be little algorithm-intense and highly I/O-oriented. Given the interpretation of I/O as a side effect on the world state and the absence of side effects in functional languages, a functional language seems to be very bad suited for our project. We will resume the discussion about the challenge of implementing a web server in Haskell and reasons to face these difficulties later. For now we continue with the introduction of the Haskell language and parts of its libraries.

2.2. Haskell Language Properties

2.2.1. Introduction

Haskell as defined in the Haskell98 standard **[9]** used throughout this project is a pure functional language with static, polymorphic typing and lazy evaluation. We will shortly discuss these primary properties before introducing some of the most common language elements.

First we have pureness. Actually pureness implies that Haskell is a functional language like defined in the preceding section: Given the same input, a function in Haskell is required to always deliver the same result (also called referential transparency). The language is free of implicit side effects. This is not true for all functional languages - to perform I/O, most functional languages include imperative constructs. This is the straight approach, interpreting I/O as side effects on a very large environmental state. However, these impure languages lose several useful

properties of functional programmming, for example lazy evaluation and unrestricted mathematical reasoning. These points will be explained throughout the next sections.

2.2.2. Lazy evaluation

Lazy evaluation describes the fact that Haskell evaluates expressions only in case (and at the time) their value is required to continue with program evaluation **[1, 4]**. By this expressions not required to compute the value of the program expression are not processed, saving resources. This is possible due to the pureness of the language. In an imperative environment an expression might cause a side effect with impact on the overall program result. So, even if the value of an expression is known to be not required the evaluation of the expression can be significant for program semantics and cannot be omitted easily. Furthermore, the order of evaluation can be of relevance for the result (again due to side effects and the order of their application on the state). In contrast, Haskell functions are free of side effects by definition. Beside the omittance of unnecessary evaluations lazy evaluation enables Haskell to provide the concept of endless lists. The definitions of such lists may describe endless structures; however, if only parts of these lists (computable without constructing the whole list) are required throughout the program, lazy evaluation allows the system to terminate because the unnecessary (endless) tail of these lists remains uncomputed automatically. This adds to the expressiveness of the language, allowing for more natural mathematical definitions of certain constructs (e.g., Fibunacci numbers may be defined recursively:

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs))
```

However, lazy evaluation comes at a price. Deferring evaluation of expressions requires the construction of function values in memory - for certain code this can heavily increase memory consumption. To handle such situations, the language provides facilities to manually and selectively enforce strict evaluation. Although this allows solving the problems in most cases, it adds to language complexity and weakens transparency - programmers are forced to cope with implementation properties in a descriptive style language, which is nothing less than contradictory.

2.2.3. Reasoning

The pureness of the language makes Haskell an ideal foundation for formal reasoning about function definitions. As functions work in the mathematical sense (without side effects) mathematical transformation rules can be applied. Therefore if the equality of two expressions is proven or stated by some law, occurences in program code can be automatically replaced by each other definition. This can be useful to do automatic program transformation, e.g. for optimization. Last but not least one can prove laws by means of induction. For example associativity of the string concatenation in Haskell (++) can be shown by utilizing the operator's definition in an induction proof **[4]**.

2.2.4. Data Types

Now that we have stated the main properties of the language, we give a short introduction to its syntax and semantics. First of all, data types can be declared by means of the data statement, in the simplest case by enumerating the possible values **[5]**:

```
data myType = Value1 | Value 2 | Value3
```

Some predefined data types can be interpreted to be represented this way or are actually defined this way:

Figure 1 - Integrated types as enumerations

```
Bool (data Bool = True | False)
Int, Integer (data Integer = ... | -1 | 0 | 1 | ...)
Char (data Char = 'a', 'b', ...)
```

`Int` and `Integer` both represent integer types, where `Int` is mapped onto a machine based representation (and therefore is limited to typical machine integers, e.g., 32bit) and `Integer` onto an arbitrary precision recursive representation (which theoretically allows for infinite numbers, although practically restricted by the machine's memory). Beside the types shown, there are further predefined types in Haskell. `String` actually is equal to `[Char]` (a list of characters) and therefore implemented in the library instead of the language itself. The `Float` and `Double` types accord to the common IEEE floating point specification (single and double precision) **[9]**.

While we used it for simple enumerations so far, the right side of a `data` definition actually consists of a series of constructors, where each one may be parameterized with a series of variables to represent structured data types. A simple example is the product or tuple type:

```
data Pair = Pair Char Char
```

This example shows a type `Pair` with a single constructor, which builds a structured value containing two `Char` values. It can be lifted easily to a polymorphic version based on type parameters:

```
data Pair a b = Pair a b
```

Here each value of type `Pair` contains a value of type `a` and a value of type `b`. Likewise we could have defined `Pair` with both contained values of type `a` to enforce that both values are of the same type. The concept of a product type has been syntactically integrated into Haskell by means of tuple notation: `(a,b)` would be equal to `Pair a b`, where the Haskell notation can be used for arbitrary arity tuples, e.g. `(a,b,c)` etc..

The notation for structured data types based on the `data` keyword as shown relies on order. E.g., one can access each element of a structured value by applying a pattern and bind the required values to variables. Another notation resembles record sets of languages like Pascal and C and allows to directly access elements by names. For the `Pair` example, we could define:

```
data Pair a b = Data {
   first  :: a,
   second :: b
}
```

To construct a `Pair` of the integers 4 and 5, we would write `(Data 4 5)` in the first notation and `Data { first = 4, second = 5 }` in the second. While this might look unattractive, the named fields are useful to access and change elements directly. For example, to denote the second element we would write `(second x)`, where x is a value of type `Data`. If we would like to change the first value into 3, we would write `x { first = 3 }`. **[9]**

2.2.5. Functions

As already mentioned, Haskell treats functions as values, where a function always takes exactly one argument **[5]**. In the type system a function is simply denoted by an arrow symbol `->`, for example yielding the following signature for a function from integer to integer:

```
f :: Int -> Int
```

To define functions depending on more than one argument, two approaches can be used. First, one can use the facilities for structured data types already shown, i.e. tuples. The integer addition function could be expressed this way:

```
add :: (Int, Int) -> Int
```

An elegant alternative to this concept is Currying. Here we take advantage of higher-order functions in Haskell, the ability to use functions as arguments and results of other functions. Based on this a function can take its argument and deliver a single result, which again is a function taking another argument. The addition function with Currying would look like:

```
add :: Int -> (Int -> Int)
```

And because the `->` symbol is right-associative anyway, we can omit the parentheses and simply write `add :: Int -> Int -> Int`, which builds up the illusion of a function taking two arguments. And beside the more convenient and explicit type signature, Currying allows for another elegant means of programming: partial application. While the tuple in the previous version of `add` has to be delivered completely when applying the function, the curried version may be applied to a single argument first and a second argument later. Utilizing partial application, we can define a function adding 5 to its argument in a very intuitive way based on the existing 2-arity `add` function:

```
add5 :: Int -> Int
add5 = add 5
```

Up to now we only saw functions in the familiar way of named definitions. In the Lambda Calculus as the foundation of functional programming, everything is constructed by means of so called Lambda expressions, which can be considered to be anonymous function definitions. Lambda expressions denoting function values also exist in Haskell, written `\x -> f(x)`, where `x` represents a list of arguments and

`f(x)` an expression using `x`. As a simple example, we can rewrite our `add5` function by using a Lambda expression instead of partial application:

```
add5 = (\x -> x + 5)
```

Naturally, Lambda expressions and partial application can be combined, for example in `add5 = (\x -> x + y) 5`, where `(\x -> x + y)` is simply an anonymous version of the `add` function.

To complete this section about functions in Haskell, we extend the concept of type variables to function definitions. We already saw data types with type variables (`data Pair a b`). Additionally, function definitions may contain type variables to make a function definition applicable to a set of types. For example, the identity function is of type

```
id :: a -> a
```

and simply returns its parameter. As the implementation of `id` is independent of the parameter's representation, the definition works for all types, which is expressed by the type parameter `a` in the signature. Moreover, the signature expresses the fact that the result is of the same type as the argument.

2.2.6. Type Classes

We already introduced type parameters to establish parametric polymorphism in Haskell. Additionally, Haskell supports ad hoc polymophism. While parametric polymorphism allows using a single function definition for a set of argument types, ad hoc polymorphism allows providing several distinct implementations for a single function to adequately treat different argument types. The language mechanism to support ad hoc polymorphism in Haskell is the type class **[5]**. Type classes allow defining a set of functions which have to be implemented for a given instance (an implementation of the type class for a data type). Default implementations independent of actual instances and based on other functions of the type class are possible. For example, in the `Eq` type class the two present functions (`==` and `/=`) are defined by means of each other:

Figure 2 - Default implementation for the Eq type class

```
x == y    = not (x/=y)
x /= y    = not (x==y)
```

Therefore, to define an instance of the `Eq` class, only one of the two operations actually has to be implemented (which is called a minimal complete definition). Haskell type classes are similar to abstract classes in object oriented languages like Java. Ad-hoc polymorphism (type classes) and genericity (type variables) in Haskell are connected by the context of type variables, a set of type class constraints. If one would define a multiplication function based on addition, its signature would be

```
mul :: Num a => a -> a,
```

expressing that such a function would work for all types installed in the `Num` class, which provides the `+` operator. Of course, this is only an example as the `*` operator is actually declared directly in the `Num` class.

2.2.7. Lists

To complete this short introduction to Haskell, we explain the usage of lists. In contrast to tuples, which allow to handle a fixed amount of elements, where each one may be of a different type, lists are meant to handle dynamic and arbitrary ordered sets of elements, where each element must be of the same type. It is worth noting that lists in Haskell are actually not part of the language but of the `Prelude` standard library. There, lists are represented by a recursive definition. However, the language provides a concise syntax to express list types and work with list data. A list is represented by the type `[a]`, where `a` can be any other valid Haskell type. For example, a function counting the number of elements in a list may have the following signature and definition:

Example 2 - A list length function

```
count :: [a] -> Int
count []    = 0
count x:xs  = 1 + (count xs)
```

Naturally, such a function already exists in the standard library and is named `length`. However, we can see the recursive definition based on pattern matching (first the empty list and second the selection of the current head), which is independent of the values contained and therefore may be applied to any list. So, a list can be processed by recursive definitions and they can be constructed using the cons operator (`:`) or by a language integrated literal syntax. For example, the list 1,2,3 may be constructed by 1:2:3:[] or by [1,2,3]. **[4]**

2.2.8. Monadic I/O

As already indicated in the general introduction of functional programming, I/O is an important issue here. Pure functional programming, as it allows mathematical reasoning and lazy evaluation, depends on functions being free of side effects. And without side effects, the dependence of a program on the outside world (for example, inputs of the user) can be hardly expressed. Many functional languages give up the pureness and incorporate imperative constructs (especially updateable variables, assignments and side-effecting functions) for this purpose. Another approach would be to make the side effect explicit to maintain functional properties. A side effect can be considered to be a change of state and a state can be made explicit input and output to a function. To give an example, a function `a -> b` takes a value of type `a` and delivers a value of type `b`, where a function `(a, s) -> (b, s)` does the same, but additionally transforms a state of type `s`. This way, a dependence on a state and a side effect on it become explicit. For I/O, the state of interest is the world state. After all, a function of type `(a, WorldState) -> (b, WorldState)` would make I/O pure functional. However, values of type `WorldState` are not very useful (because they can't be handled), beside other pitfalls of simply mapping I/O onto state transformers.

So, although the idea is the same (making a possible side effect explicit in the type system), Haskell maintains pureness by utilizing another concept: Monads **[4, 5]**. Monads are not limited to the problem of I/O, instead they abstract a more general concept which is also suitable to represent I/O. The interface any data type has to implement to become a Monad is the following:

Figure 3 - The Monad type class

```
class Monad m where
   (>>=)  :: m a -> (a -> m b) -> m b
   return :: a -> m a
```

There are two more operations, `(>>)` and `fail`, but they do not contribute to our understanding of Monads in this thesis. Their behaviour is defined by some laws regarding the aforementioned functions. Intuitively, Monads are an approach to encapsulate side effects. A value of a Monad, which can be any data type installed in the `Monad` class, encapsulates some value of type `a`. "Encapsulate" does not necessarily mean that the value is stored explicitly in the Monad value, for example the Monad might be represented by some function delivering the value. Typically,

the underlying data type adds something to the encapsulated value representation. You may think of such a Monad value to be a value inside some state representation, which also might be empty (denoting the identity Monad only representing the encapsulated value itself). Monad values are sequencially combined by means of the bind operator `(>>=)`. This operator uses the value stored in its left side Monad value, feeds it to a Monad construction function on its right side and finally builds a new Monad value to be the expression's result (which often equals the right side's return value). Altogether, Monads resemble some abstraction of imperative style programming, adding a stepwise state transformation parallel to a functional transformation.

For the purpose of I/O, Haskell provides the Monad `IO a`. A value of type `IO a` may be considered to be a computation (in constrast to a functional evaluation), delivering a value of type `a`. As we already know, a monadic data type typically contains more than the delivered value itself. When examining the `IO` Monad, one may consider the underlying data type to be a representation of the world state. For example, the Monad value delivered by

```
putStr :: String -> IO ()
```

can be interpreted to do a side effect on the world when evaluated and to deliver the nullary value. The Monad value

```
getLine :: IO String,
```

although not applying a side effect to the world, depends on the world state and delivers a string entered at the keyboard when evaluated. This string might be different every time the Monad is evaluated. This is no problem because `getLine` does not represent the string itself, but a computation to deliver this string. Indeed, while for many Monads there exist extraction functions (or the possibility to extract the value stored by means of pattern matching because the implementation is known), this is not the case for the `IO` Monad. This property is crucial for the whole concept of the `IO` Monad - if the implementation would be known or if there would be an extraction function, one could define a function

```
extract :: IO a -> a.
```

This function would break referential transparency as it could not be replaced by its result without changing the program's semantic. To be honest, there exists a function of this type, `unsafePerformIO`. The name hits the mark - the usage of this

function is unsafe. However, let's assume that no such function exists. The secret of the `IO` Monad implementation protects the pureness of the language in Haskell - the only functions aware of the implementation and able to extract the value of type `a` from an `IO` Monad value are the functions of the `Monad` class. To be precise, only the bind operator needs the ability to extract the contained value to invoke the right side function with it. And the right side function has a very useful property in this context - it always delivers a value of type `IO a`. After all, the only function able to obtain the value from an `IO` Monad results in a new `IO` Monad. The quintessence: When you entered the world of side effects, you cannot leave it anymore. Any function based on side effects explicitly shows this in its signature ending with some `IO` value. However, you may use pure functions inside a monadic transformation (on the right side of the bind operator before delivering a new `IO` Monad). A typical Haskell program is a value of type `IO ()`, which uses `IO` Monads to perform I/O and pure functions to work on the input values and to produce the output values.

A simple example for a self defined `IO` Monad would be

Example 3 - A simple I/O program

```
example :: IO ()
example = getLine >>=
               (\a -> getLine >>=
                  (\b -> putStr b >>=
                     (\_ ->   putStr a)))
```

Here, the sequence character of monadic programming is obvious. The example reads a line from the keyboard, binds it to the variable a (in the right side function of the first bind operator occurence), reads another line and binds it to b, prints out the second line and finally the first line. The last right side function, `\_ -> putStr a`, is actually independent of its argument, which is `()` from the previous `putStr`. One could use the `>>` operator of the `Monad` class here - this operator combines two Monad values in sequence, ignoring the result value of the first. But even with the `>>` operator, the code does not get elegant. The main problem is the binding of returned values to variables. There exists an additional notation, the so called do-notation, which addresses especially the binding of variables. The example above can be written in do-notation:

Example 4 - A simple I/O program in do-notation

```
example :: IO ()
example = do
   a <- getLine
   b <- getLine
   putStr b
   putStr a
```

Each line represents a Monad value, where each result can be bound to a name on the left side of an optional arrow symbol. Here you can see the imperative style programming inherent to Monads and especially to the **IO** Monad, looking more like a sequence of statements than a functional expression.

3. Web Servers and Servlets

3.1. HTTP

At this point, we introduce the concept of a web server as the foundation for two complementary topics which will lead us to the type of server we are about to build: Servlets and web services. So, what is a "web server"? In general, it is the server-side processing component of HTTP (Hypertext Transfer Protocol) as defined in RFC2616 **[10]** for version 1.1, which on its part is the foundation for the so called World Wide Web (WWW). One may extend this definition by the proposition that a web server typically delivers HTML documents and corresponding files (e.g. pictures). Hence the World Wide Web (WWW) can be considered to be a network of web servers according to the preceding definition, providing HTML **[11, 12]** pages by means of HTTP. However, for this thesis we would like to weaken this traditional definition insofar as we do not impose a restriction on the document types transferred. For example, when talking about web services (which typically rely on XML over HTTP) we would still like to call the host machines “web servers”.

Next we informally describe the properties of HTTP:

- HTTP is a request-response-protocol. That is, a connection in general consists of a query by the client and an according server response to the client. Logically an HTTP connection is closed after this pair of messages. Consequently a server is unable to send any data to the client if not explicitly requested as an immediate response to a request. To get notice of state changes, a client has to poll the server continuously on its own. Moreover, queries of a client are in no defined relationship to earlier ones, there is no concept of state or session for the communication between server and client.

- Subject of HTTP messages are so called resources, which are identified by means of URIs (Uniform Resource Identifier) **[13]**. Beside the access scheme (which is HTTP in our context) a URI denotes the server or location to serve the request and the logical path on the server (where the path has to be processed relative to the service accessed, not for example to the server's file system). Typically, a resource accessed on an HTTP server is an HTML file or a file referenced in an HTML file, but this is no necessity - HTTP is independent of the

resource type transferred. Especially, it is not even required that a resource is physically existing or that the underlying path exists in the file system. Actually the server may generate a resource dynamically and map a path to any location or generator software desired.

- As already indicated, the protocol defines two message types: Request and Response. A Request message is sent from the client to the server, describing the operation and resource in question, while the Response is sent from the server to the client to deliver the resource and/or information about the success of the Request.

- Both Request and Response messages are composed of a start-line, a list of headers (in the sense of name-value-pairs) and optionally a message body. The message body contains the resource in question if applicable.

- The operation to be executed by the server is being coded in the Request as the HTTP method. There exist the following methods: GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT. We will not discuss each single method here, but the most common method is GET to retrieve a resource from a server identified by the accompanying URI. Beside URI and method, the HTTP version is delivered in the start-line of the Request (also called Request-Line).

- In the Response, the respective operation's success or failure is represented by a status code. A code of the form 2xx denotes a successful operation, a code of the form 3xx denotes the delivery of a different resource than requested (Redirect), a code of the form 4xx denotes a client side error (for example, an invalid or unpresent URI) and finally, a code of the form 5xx describes a server side error. The status code is sent to the client along with a textual description of the status and the HTTP version in the start-line (also called Status-Line in the Response message).

- While it is typical that a message body is present only in the Response and represents the resource in question, this is not necessarily the case. E.g., in the PUT method the client transfers a resource to be stored on the server in the Request's message body. The corresponding Response will not contain a message body (but hopefully a status code 201 "Created").

- Although HTTP 1.1 does not define a mechanism to associate Request-Response pairs with a particular connection (to achieve session tracking functionality), it supports and encourages the usage of so called persistent TCP connections. With persistent TCP connections, a TCP connection is not closed after completion of a Request-Response pair but left open for further pairs. Some overhead for opening and closing TCP connections can be avoided this way. Furthermore, persistent TCP connections allow for the so called pipelining mechanism of HTTP 1.1, where a client does not wait for the Response to a particular Request but immediately dispatches more Requests. Pipelining hides communication latency on the underlying network. However, persistent TCP connections are a means of efficiency, they do not contribute (at least by standard definition) to any session mechanism.

Following this short summary of HTTP, we list the properties and requirements of an HTTP 1.1 web server implementation in terms of software engineering:

- A web server has to support network connections via TCP to receive Requests and to send Responses. Therefore the underlying software environment has to provide interfaces to access network facilities of the hardware and the operating system. An example for a widely used software interface is Sockets.

- A web server has to process the HTTP protocol, i.e., it has to parse the plaintext content of a Request, to interpret the abstract request (if a valid one is present in the message) and to generate a syntactically and semantically correct Response message based on the Request. These tasks are pure algorithmical (i.e., no I/O is required), wherefore any general purpose language provides the necessary means. However, an expressive and efficient parser library present in the language or its environment is advantageous.

- There is no control over the accessing systems, their state and their quality of software. Hence, a web server has to be fault-tolerant, i.e., it has to recognize faulty Requests and react in a defined manner (for example, with standard complying status codes). Moreover, timeouts have to be utilized in compliance with the standard to deal with faulty clients (to limit the impact of these on the server's performance).

- There should be logging facilities to record Requests, Responses, warnings and errors. This supports the debugging of both server and client software and establishes a certain level of security, as administrative users may track operation of the server and its history of Requests. Last but not least statistical data may be computed from the logging information.

- The server has to be enabled to control resources, i.e., according to the Request URIs it should be able to generate resources dynamically, to load them statically for example from a file system or to delegate their creation to external software. The underlying software system should provide useful I/O facilities to serve these functions.

- The server should be configurable regarding its network configuration, its data sources, it security parameters, its logging, possible extensions, etc. The software environment may provide functionality to easily manage configuration data.

- Queries may arrive at any time and several of them may occur at the same time. The server has to be able to accept accesses anytime and should be able to process them in parallel. This is on the one hand to exploit parallel hardware and on the other hand to hold down latencies.

3.2. Servlets and Web Services

We already saw that a resource denoted by a URI in HTTP does not have to be a physical file. Instead it may be computed in the moment of access. A common mechanism to dynamically create resources is CGI, the Common Gateway Interface **[14]**. CGI bases on forwarding certain information to an external program. This set covers variables describing the server context (environment variables) and the Request message, where especially the URI's query part (called QUERY_STRING in CGI) is of interest. The query part of a URI contains a set of attribute-value pairs. The interface to deliver information to the program is specific to the system and language used (which might be a simple shell script). The result of a CGI program is a list of headers (CGI control headers and arbitrary headers to be added to the HTTP Response) and the HTTP Response body. Thereby CGI is restricted to HTTP as the transport protocol. The more important restriction is the lack of a supportive container. Dynamic state information, database access as well as every piece of

data not included in the Request or the server environment is within the responsibility of the CGI application.

So, although every HTTP based web application can be constructed using CGI, more sophisticated mechanisms are desirable in terms of software quality. For example this concerns expressiveness, reusability, simplicity and flexibility. The technique to achieve this is the concept of a Servlet. Naively a Servlet can be considered to be a small program running on a server to serve requests and which is not able to run independently (for example, at command line). In the following, to infer a more precise definition we will refer to the Java Servlet specification as the most common Servlet definition **[15]**.

According to the Java Servlet Specification, "a servlet is a Java technology based web component, managed by a container, that generates dynamic content" and "the servlet container is a part of a web server or application server that provides the network services over which requests and responses are sent, decodes MIME based requests, and formats MIME based responses. A servlet container also contains and manages servlets through their lifecycle." These definitions become more precise by collecting requirements for a Servlet and the according Servlet container, where it should be obvious that Servlet properties to a certain extent directly result from the container's properties and vice versa. Therefore we collect both Servlet properties and Servlet container properties in one go:

- **Java:** A Servlet is a Java software component which implements the `Servlet` interface and in most cases derives from the `HttpServlet` class. The developer overrides methods concerning the different methods of HTTP (for example GET) to define the Servlet's behaviour. The expressiveness of a Servlet is equivalent to the Java programming language and its standard library. Due to the Java programming language, Servlets are largely platform independent and may use facilities to dynamically load further classes, serialize objects and concurrently execute certain tasks supported by the Java monitor concept.

- **Life Cycle:** A Servlet has a defined life cycle. A Servlet's class is loaded and objects of this class are instantiated by the container. The container may delay this initial phase until the first request for the Servlet is received. Next, a constructed Servlet object is initialized by means of a method of the `Servlet` interface which may be overriden to perform initial one-time tasks. Now, the life

cycle has reached the phase of request handling, during which the Servlet object processes requests arriving. The fourth life cycle phase is the end of service, where a destructor method in the `Servlet` interface is invoked before reclaiming the Servlet's memory. The end of service is arbitrarily controlled by the container, which may construct and destruct a Servlet object for each single request (although this would be very inefficient).

- **Context:** A Servlet has access to a so called `ServletContext` object, which is provided and managed by the container. Each web application and all contained Servlets are associated with such an object to gain access to a shared state. Servlets not included in a web application are associated with a default `ServletContext`. The `ServletContext` allows getting and setting configuration attributes originally defined in a static startup configuration for the web application. Furthermore it allows storing and retrieving arbitrary objects, doing logging and gaining access to static resources (HTML files, images, etc.).

- **Requests:** Requests are executed by delivering the `ServletContext`, a Request representation and a Response representation (to deliver results by means of side effects) to the Servlet's `service` method. In case of HTTP, there exist specializations, e.g., a mapping of the `service` function to methods for each HTTP method (not to be confused with the concept of an object oriented method). In this case, there are also specialized Request/Response classes (`HttpServletRequest`, `HttpServletResponse`) to be delivered with the invocation. The Request object delivers a set of parameters (by the client) and a set of attributes (by the container). In addition, an `HttpServletRequest` provides access to HTTP specific features like Cookies or HTTP headers.

- **Responses:** The Servlet's answer is delivered by means of a `ServletResponse` transmitted with the request service invocation. In general, the answer can be simply supplied by a stream constructed from the `ServletResponse`. Again there exists a specialized version `HttpServletResponse` to handle HTTP features. For example, Response headers may be controlled this way.

- **Filters:** We omit the concept of a Filter in the Java Servlet 2.3 specification because there is no fundamental contribution to our own understanding of Servlets.

- **Sessions:** A Servlet container provides a mechanism to track sessions. As discussed, sessions or states in general are not a native property of HTTP communication as each Request-Response pair is completely autonomous. There are ways to build session tracking on top of HTTP, for example encoding a state or a session reference into the URI query part or a Cookie to associate requests with a server generated session. With Java Servlets, the Servlet programmer shall be widely freed from managing these things on his own. Instead the Servlet container does this automatically and provides an `HttpSession` object to a Servlet invocation. This object not only describes the session identity, but also allows storing objects to constribute to the session state.

- **Forwarding:** There exists a mechanism allowing a Servlet to forward requests to other Servlets.

- **Web Applications:** There exists the concept of a web application, which is mainly a set of Servlets with some optional resources deployed to the server as an atomic unit. Based on web applications, there are many concrete facilities, for example for configuration, loading, installation, removement as well as some features already mentioned (like `ServletContext`).

- **Denotation by URIs:** Both web applications and Servlets are denoted by URIs. When a Request is retrieved, the web application in question is determined by means of longest match concerning Context URIs. Given the web application, the local URI (Request URI minus Context URI) is used to determine the Servlet inside the web application to serve the Request.

After all, this definition is strongly related to object oriented programming in general and Java as both implementing language and library in particular. For the purposes of this thesis, we provide a less specific and language independent definition to be the foundation for the Janus server:

Figure 4 - The Janus Servlet Definition

```
A Servlet shall be a software entity transforming a request into a
response. To support modularization, a response can also be
considered to be the request for another Servlet, where the last
Servlet of a chain of Servlets delivers the ultimate response for
```

the client. There shall be a set of states, where each state is concurrently shared among a set of servlets at runtime. There are web applications, either explict or implicit, comprised of one or more Servlets. There is a Servlet container managing a set of web applications, which is responsible for receiving requests, selecting the handling Servlet based on a given configuration and invoking the Servlet with a unified representation of the request. The container shall take a Servlet's response in a unified representation and transfer it back to the requester. Moreover the container shall be responsible for loading its Servlets, for providing a structured configuration for each Servlet, for creating and providing the shared states where applicable and for providing a defined means for tracking sessions. The container might be embedded into a server software entity to gain access to network facilities. There should be no strict determination on any specific transport mechanism or content representation.

As already mentioned in the context of Java Servlets and made explicit in our own definition, Servlets do not necessarily rely on the HTTP protocol or HTML as content representation language. Extending the common understanding of Servlets as the basis for a human related web application, we come to the concept of a web service. A web service can be considered to be a set of Servlets utilizing XML based structured data to describe requests and responses. Furthermore there exist mechanisms to discover web services and formally describe their respective interfaces (i.e., their XML format). By this, a web service is in first place a machine accessible web application or a machine accessible interface to an existing web application.

This thesis and the Janus server concentrates on the aforementioned definition of Servlets with a concentration on HTTP/HTML (as with Java Servlets), but it should and will be a design goal to inherently support the implementation of web services. To achieve this goal, the system has to be made independent of the transport facility (here: HTTP) and the content representation (here: HTML), as demanded from our definition. Moreover, we anticipate at this point that Janus will be strongly based on XML structures to describe its internal processing. Starting from this, support for web services should be only a little step ahead.

3.3. XML

We mentioned that Janus will be based on XML for internal representation. Additionally there is the topic of web services, which is also strongly related to XML as a communication format. Therefore, to be complete, we now introduce XML basics (without going into details) **[6, 16]**. XML (Extensible Markup Language) is a data markup language based on SGML to store structured information in so called XML documents. Logically an XML document contains a tree of XML elements, where each XML element has a name, a set of attributes (where each one is a simple name-value pair) and a list of children. These children might be further XML elements, character data, so called processing instructions, comments, so called CDATA-sections and so called references. Children (if they are elements) might share the same name.

Physically, an XML document is a textual representation of this element tree, with additional meta information. In the textual representation, each element is represented by a starting tag, its content and an ending tag, where the starting tag contains the element name and the set of attributes: `<tagname attr="value" anotherattr="anothervalue">somecontent</tagname>`. An empty element may be abbreviated as a combined start and end tag: `<tagname/>`. Based on this, an XML document consists of a single labeled element. Preceding this root element, there might be a prolog. This prolog might contain an XML-declaration (which declares the XML-version of the document and the encoding utilized) and if such declaration exists, it comes first in the prolog. Furthermore the prolog might contain a single document type declaration (defining a so called DTD to impose certain structural requirements on the document). A document type declaration follows the XML declaration (if one exists) and may be surrounded by comments, processing instructions and whitespace (in arbitrary combination). After the prolog (which might be empty), the single document root element follows. As every element, the root element might be empty. Following the root element, there might be further comments, processing instructions or white space (in arbitrary combination). Based on the XML specification, the simplest XML document is an empty root element.

Here we have an example for a simple XML document:

Example 5 - A simple XML document

```
<?xml version="1.0"?>
   <myrootnode>
      Hello, world!
      <achild/>
      <anotherchild>
         <achild>Hi!</achild>
      </anotherchild>
   </myrootnode>
```

There are two possible properties for a given XML document: It may be well-formed and it may be valid. Well-formedness concentrates on the XML syntax and describes conformance to the XML syntax productions (starting with the document production). Moreover, to be well-formed some respective constraints given in the XML standard have to be maintained and each so called parsed entity referenced directly or indirectly within the document has to be well-formed. Validity is a much stronger statement. It requires the existence of a document type declaration and the conformance of the document to the according DTD (Document Type Definition). Such a DTD contains structural information about a document of its type, for example, which children an element has to contain.

XML specifies a namespace mechanism, i.e., for each element and its children there may be namespace declarations to be used by element names. Therefore, element names can be distinguished even though they (respectively their local part) might be identical. The respective mechanisms also apply to attribute names.

An XML processor reads XML documents in their textual representation and provides abstract access to the document's properties and tree. It may be a validating or a non-validating processor, that is, a processor which checks the accordance of a document to the declared document type definition or which does not. Beside the core standard describing the XML data model, its textual representation and the DTD concept, there are many additional standards extending the basic XML functionality. For example, XPath **[6, 17]** describes expressions to select parts of a document by means of a string path. XSLT **[6, 18]** allows transforming XML documents to arbitrary (also non-XML) target documents and XML Schema **[6, 19]** allows defining a document type description exceeding the expressiveness of DTDs (for example, data types may be used here).

3.4. Challenges in Haskell

Up to now, we summarized the general properties and software requirements for static and dynamic web servers (where a dynamic web server utilizes CGI or some Servlet concept to process requests). At this point, before outlining the concept of Janus, we should discuss the suitability of Haskell.

First of all, we remember the remarks already done regarding I/O in functional languages. One can easily see that these issues are especially relevant for a web server. A web server heavily depends on I/O to achieve network functionality and to access resources on the system. The important concept of explicit concurrency may also be considered to be an I/O phenomen - we will discuss this concept in the Haskell context in a later section. So we may start evaluating Haskell with regard to our web server with I/O:

- Pure functional programming needs advanced concepts to incorporate I/O. The worst one can say about these is that they are simply unfamiliar to an imperative style programmer. However, they do the job and maintain useful properties of the language.

- Efficiency is an issue for functional programming in general and Haskell in particular. For example, lazy evaluation requires constructing function values and buffered data in memory. This is useful (in terms of efficiency) if many expressions represented by such buffered values do not get evaluated, altogether CPU time can be saved and the garbage collection effectively reclaims used memory. But if confronted with a strongly imperative problem for which strict evaluation might be more adequate, lazy evaluation can be a burden and render some implementation unusable for big instances of the problem. One can handle this problem by explicitly enforce eager evaluation, but nevertheless this is a disadvantage compared to eager evaluation languages.

- While many algorithms can be formulated more descriptively in a functional language, this is usually not the case for I/O intense problems, where the imperative concept of a stepwise state transformation gets important. If one looks at the Haskell do-notation, the resulting code might heavily resemble a native imperative program.

- Haskell is statically typed and does not provide any language elements to do dynamic loading. By design, a program is completely translated and linked. For a dynamic web server to compete with the Java concept of Servlets, a mechanism to load Servlets at runtime is desirable.

- So far, Haskell does not seem to look attractive to build the Janus server, although the problems shown can be overcome by existing language extensions and libraries. But making Janus in Haskell does not end in itself ("done because it can be done"). There are advantages. First of all, Haskell provides flexibility, especially due to its handling of function values. Furthermore, some features of the language may enhance expressiveness of imperative style programming too, for example the integrated syntax for lists and tuples.

- XML is a stronghold of Haskell, as language processing in general. This is due to the parsing capabilities inherent to functional languages.

- As seen, Haskell provides a sophisticated type system, which significantly contributes to type safety (while maintaining flexiblity by means of parameterized and ad-hoc polymorphy) and software quality.

- Haskell now features a competitive concurrency model, Concurrent Haskell. We will discuss properties of this model soon; for the moment we summarize its advantages to be the ease of concept and the light weightiness.

So, these are the reasons and challenges faced when approaching our server in Haskell. So, let's put it together what we already have and what is still missing to start developing a design:

- We need I/O in general, for example to access file resources. This is achieved by the `IO` Monad in Haskell.

- We need access to the operating system's networking layer. This is achieved by according standard library functions based on the `IO` Monad.

- We still need explicit concurrency to process multiple requests in parallel.

- We still need a shared state. This is because a set of Servlets shall be able to work together on a given state. It shall be mutable, of course, because otherwise

Servlets would not have any memory to track sessions etc.. Pure functional programming does not support update assignments as this would break referential transparency. Therefore we need something new, which will be again based on the `IO` Monad to avoid breaking referential transparency.

- We still need a library to process XML.
- We still need a way to dynamically load Servlets. As there is no language integrated concept to do this and type safety cannot be maintained on the basis of object files, we will rely again on the `IO` Monad.

In the next section we start with the shared state first because the concurrency model chosen will be based on it.

3.5. Shared State

First, before talking about a shared state, one should think about an unshared state. A state is typically represented by a set of mutable variables. And as we already saw when discussing I/O in functional languages, this can be expressed by using state transformers: `(a, state) -> (b, state)`. So, mutable variables can be mapped easily onto a pure functional approach and likewise could be abstracted by means of Monads (as these are especially useful to work on a state hidden in some data structure beside the main value in question). Let's look at some interface for mutable variables defined by **[20]**:

Figure 5 - A state transformer interface for mutable variables

```
newVar    :: a -> ST s (MutVar s a)
readVar   :: MutVar s a -> ST s a
writeVar  :: MutVar s a -> a -> ST s ()
```

Here we have three functions to work on mutable variables stored in a state, where `MutVar s a` can be considered to be a reference to a variable in a state of type `s`, holding a value of type `a`. `ST s a` is a state transformer, working on a state of type `s` and delivering some value (such a state transformer is independent of any input beside the input state). `newVar` creates a new variable in the state holding an initial value, where the according state transformer delivers a new state containing the new variable and a reference to this variable. `readVar` reads a value from the state based on a given reference, `writeVar` puts a new value into it. The state transformer

of `writeVar` delivers the output value (), the nullary value. This implies that the transformer actually performs nothing except the side effect.

However, if we think about a shared state in the context of multithreading, changes made on variables by one thread should become visible to other threads. This behaviour would break referential transparency in the current solution, of course, as a thread might read different values from a variable in different expressions without having done assignments to these variables. Even worse, in this situation non-determinism might occur, as the order of changes caused by different threads is unpredictable.

Given these problems, the concept of mutable variables must be placed inside the `IO` Monad to encapsulate non-determinism and side effects not included in the explicit state description. The new interface is:

Figure 6 - An I/O based interface for shared mutable variables

```
newMutVar   :: a -> IO (MutVar a)
readMutVar  :: MutVar a -> IO a
writeMutVar :: MutVar a -> a -> IO ()
```

Here, the state transformer is replaced by the `IO` Monad, otherwise the concept stays unchanged. Examining this implementation one might consider the `IO` Monad to be a state transformer working on the `WorldState`. This could be expressed by a type synonym in Haskell **[20]**:

```
type IO a = ST WorldState a
```

Hence the state can be omitted from the `MutVar` type because it is known to be the `WorldState`. All this conforms to out discussion about I/O, where we considered the explicit transformation of the `WorldState` to be an approach to I/O in pure functional programming. We conclude the discussion of shared mutable variables with a simple example:

Example 6 - Demonstration of shared mutable variables

```
exampleMutVar :: IO ()
exampleMutVar = do
   a <- newMutVar "Example"
   b <- readMutVar a
   putStr b
   writeMutVar a "Text"
   c <- readMutVar a
   putStr c
```

This example creates a new variable, reads and prints out the value stored, rewrites the value and again reads it and prints it out. As we can see, using the reference bound to `a` delivers different values for the two invocations of `readMutVar`.

3.6. Concurrency

Now that we have a mutable shared state, we can introduce concurrency. Concurrency per se is nothing revolutionary to functional programming if one looks at the so called implicit parallelism. Implicit parallelism in general denotes parallel semantics inherent to program code without explicitly marking it. One useful example is the evaluation of expressions, for example `a(5) + b(2) + c(8)`, where `a`, `b` and `c` denote functions. Although mathematics dictate the associativity and therefore application order of the `+` operator on the values denoted by `a(5)`, `b(2)` and `c(8)`, it does not define the order in which a machine has to compute these values. However, in imperative languages there might be side effects caused by each of these subexpressions. This makes the order and the evaluation itself significant to the result of the overall expression and program. Arbitrarily evaluating the subexpressions affects the order of side effects. Omitting a subexpression which does not produce a value actually required completely omits its side effect (which might be required in contrast to the expression's value). Evaluating the subexpressions concurrently makes things even worse: Not only that the order of evaluation might miss the programmer's intentions, but the order also gets non-deterministic and easily differs for every program run. While the order of evaluation can be defined by the programming language to address the first part of our reasoning, this is not possible for a multithreaded environment. Or more precise, a mandatory order would make parallelism impossible.

In pure functional programming, these problems simply do not exist due to the lack of side effects. The language already exploits this fact to achieve lazy evaluation. Transparently multi-threading the evaluation of subexpressions could be easily added. However, there is another class of concurrency, which is explicit parallelism, where the programmer is able to mark programm code to get executed in parallel to other. This still does not conflict with functional programming, but it does if one considers the topic of interprocess communication. Many concurrent systems in the imperative world (like servers, databases, user interfaces etc.) heavily rely on communication between threads or processes. A very simple example for such

communication is the processing of files, which might be read and written concurrently by several threads. This is especially true for network applications, as it is typical for a network to get used and manipulated by many processes in parallel - most of them not even running on our machine and therefore immediately part of the `WorldState`.

All this leads to three requirements for our concurrency system in Haskell:

- We need facilities to mark up code for parallel execution.
- We need facilities to do thread communication while maintaining referential transparency.
- We need facilities of synchronization.

The last requirement is an inferred one because any concurrent system with interprocess communication requires synchronization to control order where required and to avoid racing conditions **[7]**. The requirements postulated are fulfilled by the Concurrent Haskell language extension, which Simon Marlow selected for his web server.

3.7. Concurrent Haskell

Concurrent Haskell **[21]** addresses our requirements with the introduction of a new function:

```
forkIO :: IO () -> IO ThreadId
```

`forkIO` obtains an action argument and creates a new thread executing this action. The thread created is a side effect of `forkIO`, which immediately terminates and delivers the new thread's id. Being independent of the thread created (in terms of control flow) is not self-evident, one could also imagine an action starting two threads, terminating with the concatenated results of both. Such an operation would be called symmetric in contrast to `forkIO`, but this approach is likely to get inelegant when infinite threads come into mind.

`forkIO` already accomplishes our first requirement (mark-up of code for parallelism) and our demand for type safety. Embedding parallism into the `IO` Monad makes dependence on side effects (for example, due to process communication) explicit and protects the pure paradigma. Furthermore, we can easily multi-thread any other

I/O related Haskell code. But we still have two points to be clarified: interprocess communication and synchronization. Concurrent Haskell builds the solution upon the mutable variable concept we already introduced. Let's remember the interface:

```
newMutVar   :: a -> IO (MutVar a)
readMutVar  :: MutVar a -> IO a
writeMutVar :: MutVar a -> a -> IO ()
```

Concurrent Haskell introduces a new interface, which is:

Figure 7 - Concurrent Haskell MVar interface

```
newEmptyMVar   :: IO (MVar a)
takeMVar       :: MVar a -> IO a
putMVar        :: MVar a -> a -> IO ()
```

Beside the insignificant distinction between `newEmptyMVar` (creating an empty variable) and `newMutVar` (creating a variable with an initial value), we got a new type: `MVar a`. This type is a simple extension to `MutVar a`, where synchronization based on a monitor has been integrated. In the original definition, `takeMVar` actually blocks until a value is present in the respective variable. In contrast to `readMutVar`, `takeMVar` removes the value and leaves the variable empty. But this is no significant difference again as there also exists a `readMVar` function. `putMVar` considers it to be an error if the variable already holds a value (where `writeMutVar` does overwrite the present value, but there also exists a `writeMVar`). Up to now, additional functions have been added to the respective library. The quintessence shall be that a monitor has been integrated into the `MutVar` concept, delivering the `MVar` type. Based on this type, processes can communicate (because the shared mutable variable semantics are still present, of course) and do synchronization at the same time. For example, a mutex can be easily implemented by using an `MVar ()`. The library already contains further constructs, for example semaphores and buffered channels based on MVars.

At this point, we already have completed the overview about Concurrent Haskell. We can easily recognize that a primary design goal was maintaining the simplicity of the Haskell language core. This has been achieved, as there is no new language element (although there has to be compiler support for the Concurrent Haskell extensions) but only very small changes to the standard library. These changes include `forkIO`, the `MVar` type and some primitives above this type. Many useful and complex facilities have been built based on these few core elements, proving the expressiveness of Concurrent Haskell.

3.8. Relevant Haskell Libraries

3.8.1. Overview

Now we start introducing some of the libraries and existing software systems which shall be used to implement Janus. These are:

- HWS-WP (Haskell Web Server with Plugins) by Simon Marlow and Martin Sjögren **[22, 23]**
- HXT (Haskell XML Toolbox) by Uwe Schmidt **[24, 25]**
- The Arrows extension to Haskell **[26, 27]**
- hs-plugins by Don Stewart **[28]**
- Bytestrings by Don Stewart **[29]**

Let's start with the web server.

3.8.2. HWS-WP

One may consider Janus to be the extension of an existing static web server (HWS-WP) by dynamic functionality. This is not the case, as the design of this server has proven to be inadequate to be taken as the framework for a completely new highly dynamic design like Janus with an entirely different approach to internal data exchange. So, instead of modifying HWS-WP, Janus will use components of HWS-WP to build its own HTTP interface.

HWS-WP started as HWS developed by Simon Marlow **[22]**. Later Martin Sjögren extended it by an Apache style Plugin interface to HWS-WP **[23]**. At the time of release, HWS was the first implementation of an HTTP server based on static files in the Haskell community. It has to be considered to be a case study to show that a highly I/O intense application like a web server can be implemented in Haskell and that such an implementation is competitive in practice. To be precise, Simon Marlow stated the following main reasons to build such a server in a pure functional language like Haskell:

- because it is possible

- because with Concurrent Haskell, there is a light weight concurrency model available

- because Haskell's sophisticated type system advances fault tolerance and respectively software quality

- because there exist useful extensions to Haskell providing mechanisms to handle errors (exceptions)

- because there exist asynchronous exceptions (as an extension) in Haskell to handle timeouts and react to external events

- because Haskell features a large library of tools to elegantly solve parts of the web server problem (for example, there are very powerful and expressive parsing solutions)

The decision to incorporate Concurrent Haskell is not justified with its light weight concept only, but with its felicitous compromise of programming style and efficiency. Marlow states the following alternatives when using concurrency in a programming language:

- **Operating system processes:** Every concurrent task is being executed as its own process.

- **Operating system threads:** There exists an interface in the operating system to directly provide the concept of threads. Threads of a process share memory among each other and therefore may be able to perform communication much more efficient than a non-threaded solution. Moreover, overhead is reduced. However, operating systems might map multi-threading on multi-tasking and implement each thread by a process, losing advantages in terms of efficiency.

- **Multiplexing:** This represents multi-threading simulated in program code by multiplexing inputs and outputs. While this approach consumes the least resources, it is more error-prone and less modular than the other. Additionally it causes the program to be less readable, it obfuscates algorithms and finally it can hardly take advantage of hardware parallelism.

- **User-Threads:** Technically user-threads are similar to the multiplexing approach, but in contrast, they are implemented in the runtime environment. The runtime environment provides an abstract API to allow applications utilize parallelism. Debugging, improvements and replacements can be done centralized in the environment without affecting application code. Because the runtime environment is closer to the operating system as well as the hardware and may legitimately use foreign language code (like C), exploitation of parallel hardware may be transparently implemented here.

Concurrent Haskell belongs to the last category. Marlow consideres this to be advantageous, because the approach is light-weighted (as no process switching on the operating system layer has to be performed) while not relying on error-prone and bad readable extensions by the programmer. By the way, Apache belongs to the first category as for each request a process is forked. According to Marlow, this resource-intense design is the reason for HWS performing quite well compared to Apache.

Now we discuss the design of HWS by means of several implementation code insights. These insights base on an according paper by Simon Marlow **[22]** and are simplified versions of actual program code to improve readability. We start with the server's main loop:

Figure 8 - HWS-WP's main loop

```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections conf sock = do
   (handle, remote) <- accept sock
   forkIO (
         catchAllIO
            (talk conf handle remote
               'finally' hClose handle)
            (\e -> logError e)
      )
   acceptConnections conf sock
```

The main loop (`acceptConnections`) is an action, which is parameterized with a network socket (a feature of the standard library) and a configuration structure (of type `Config`). The `Config` structure is pipelined through the whole system and delivers the configuration file's content to all subsequent functions. This forwarding starts in the main loop with the invocation of `talk`, the action processing a request. First, the main loop blocks upon the network socket. Please note that the socket is

created and delivered by the caller of `acceptConnections`, therefore interpretation of the network configuration has to be done earlier. When a request is received the main thread continues, now in possession of a tuple `(handle, remote)` containing an I/O handle for the request data stream and a data structure describing the requesting client. Next, a thread processing the request by means of `talk` is forked. Inside the thread action, talk is a parameter to a `catchAllIo` action. This action can be compared to the try-catch statement in Java, catching an exception thrown in `talk` and forwarding its abstract representation to the second argument action for handling. In the example the handler simply logs the exception by means of `logError`. We can also recognize the concept of a Java finally-block here, as `talk` is also delivered to a `finally` action. In contrast to `catchAllIo`, the second argument of `finally` is also performed if no exception has occured and a catched exception is thrown again after the second action has been executed. However, after the request handler thread has been started, `acceptConnections` recursively invokes itself and blocks again upon the socket to wait for the next request.

Before continuing with the definition of `talk`, we examine the data structures representing HTTP Requests and Responses:

Figure 9 - Type to represent an HTTP Request message

```
data Request = Request {
   reqCmd      :: RequestCmd,
   reqURI      :: ReqURI,
   reqHTTPVer  :: HTTPVersion,
   reqHeaders  :: [RequestHeader]
}
```

Here we can see the fields defined for an HTTP Request in RFC2616, the method (reqCmd), the URI (reqURI), the HTTP version (reqHTTPVer) and the HTTP headers as a list of `RequestHeader` values. Please note that the headers are defined as simple string-based name-value pairs. A value of type `RequestHeader` represents such a pair. However, we will not discuss the subsequent types in detail. Remarkably a field for the Request body is missing. Therefore one can assume that the current HWS implementation lacks support for the PUT and POST methods.

Figure 10 - Type to represent an HTTP Response message

```
data Response = Response {
   respCode       :: Int,
   respHeaders    :: [String],
   respCoding     :: [TransferCoding],
```

```
    respBody       :: ResponseBody,
    respSendBody   :: Bool
}

data ResponseBody
    = NoBody
    | FileBody Integer{-size-} FilePath
    | HereItIs String
```

The structure of Responses follows the `Request` structure and simply reproduces the fields defined in RFC2616. Now a field for the status code is present. The list for headers is simply string based here - presumably because there is no further processing of headers, they are simply streamed into the textual Response representation. Furthermore a list of encoding information is present and finally the Response body (which missed in the Request). The data type for the Response body is shown as well. The body can be empty (`NoBody` constructor) or it may contain data. In the latter case, the data might be immediately present in the structure (`HereItIs` constructor) or there may be a file path to retrieve the data from a file when sending the response (`FileBody` constructor). This is to implement an efficient way to transmit file data without loading it into memory. The meaning of `respSendBody` is undocumented.

Figure 11 - HWS functions to parse Request messages

```
getRequest     :: Handle -> IO [String]
parseRequest   :: Config -> [String] -> Either Response Request
```

The processing of a Request by `talk` starts with the retrievement of its textual representation. This is done by the `getRequest` action, which reads lines from the given handle until an empty line occurs. Please note that in contrast to the action's name actually not a value of type `Request` is delivered but simply an uninterpreted string list, where each string represents a line of the Request. A second operation called `parseRequest` (a pure functional one) is utilized to infer the corresponding `Request` value. Additionally, `parseRequest` is parameterized with the server configuration and delivers either a `Response` or a `Request` value. Other than one might expect, this function is not meant to also parse a Response message but to deliver a `Response` value if an error occurs during parsing. This `Response` value represents an according error message for the client.

Figure 12 - HWS functions to create Response messages

```
genResponse    :: Config -> Request -> IO Response
sendResponse   :: Config -> Handle -> Response -> IO ()
```

If no error occurs and a `Request` value is delivered, it is forwarded to a function called `genResponse` (along with the server configuration again). This action delivers an according `Response`, typically with the file content requested. To retrieve the file in question or at least check its existence and compute its length, this has to be an action. Finally the resulting `Response` value along with the network handle and the server configuration is forwarded to `sendResponse` to get transmitted to the client. This is also an IO action of course because it accesses the network layer and may access the source file. Please note that there is no intermediate step to generate a string representation of the Response message, this is done directly inside `sendResponse`.

Now let's come back to the `talk` action, here we have its definition based on the four functions and actions just introduced:

Figure 13 - Action to handle a single HTTP Request

```
talk :: Config -> Handle -> HostAddress -> IO ()
talk conf handle haddr = do
   req <- getRequest handle
   case parseRequest r of
      Left resp -> do
         sendResponse conf handle resp
         return ()
      Right req -> do
         resp <- genResponse conf req
         sendResponse conf handle resp
         logAccess req resp haddr
         if(isKeepAlive req)
            then talk conf handle haddr
            else return ()
```

The structure of `talk` is quite intuitive. `getRequest` delivers the string representation of the client's Request, which is transformed into a `Request` value by `parseRequest`. A case structure is used to recognize the error case in which a `Response` value is delivered by `parseRequest`. In this case, the Response message is directly transmitted to the client by `sendResponse`. Otherwise a `Response` is generated according to the `Request` value and sent to the client. Furthermore, the Request gets logged by `logAccess`. At the end (although not in case of an error), it is tested if the keep-alive header is set in the Request message. If so, the `talk` action does not

terminate but recursively invokes itself to wait for the client's next Request by means of the existing TCP connection and socket.

3.8.3. Arrows

Up to now we only talked about Monads to represent imperative style side effects in general and I/O in particular. However, there is a generalisation of Monads present in the Haskell world called Arrows **[26]**. Please note that Monads (respectively values of a Monad type) do actually not resemble functions. They may denote some value based on some state of its data structure, but in the moment of evaluation a monadic value does not depend on an input. The reason one easily gets the impression a Monad value depends on input is the bind operator, for example in:

```
getLine >>= putStr
```

The value delivered by `getLine` is forwarded to the right argument of `>>=`. But the right side of `>>=` is not a monadic value, it is a function constructing a monadic value. The construction of this Monad value may depend on an input, of course. `putStr` is a function of type `a -> IO b` and can be used directly with the bind operator. But `putStr` is not a monadic value by itself, taking a string and printing it. It is a function constructing a monadic value printing exactly the value delivered to the `putStr` function.

So, where is the problem? By using constructor functions, forwarding of result values is possible. But there is a simple example where the asymmetry of the monadic bind operator becomes a burden. Parsers can be installed in monadic classes usefully. Here the bind and other monadic combinators are used to combine parsers (starting with primitive ones) to build e.g. sequences and alternatives. Now, optimizing parsers usually contain a static and a dynamic part, where the static part contains some preconditions checked before invoking the dynamic part. These preconditions for example include the initial tokens the parser accepts and if the empty sequence is accepted. To construct a new parser by the bind operator, the static properties of both source parsers have to be combined. The resulting parser accepts the empty sequence if both argument parsers do. If the first parser accepts the empty sequence, the combined parser accepts the initial tokens of both parsers united, otherwise only the initial tokens of the first parser. And here the asymmetry of bind becomes an insuperable obstacle - the right side is no instance of the parser data structure but a function constructing such a value. And the only thing one can

do with a function is applying it. Therefore, to build the static properties of the combined parser all subparser constructors have to be applied to the input first.

A solution to these deficiencies are Arrows, basically computations delivering an output based on an input (in contrast to Monads, which are computations delivering an output). Consequentially Arrows are polymorphic concerning both input and output types. The basic `Arrow` class defines the following functions:

Figure 14 - The Arrow type class

```
class Arrow a where
   arr      :: (b -> c) -> a b c
   pure     :: (b -> c) -> a b c
   (>>>)    :: a b c -> a c d -> a b d
   first    :: a b c -> a (b, d) (c, d)
   second   :: a b c -> a (d, b) (d, c)
   (***)    :: a b c -> a b' c' -> a (b, b') (c, c')
   (&&&)    :: a b c -> a b c' -> a b (c, c')
```

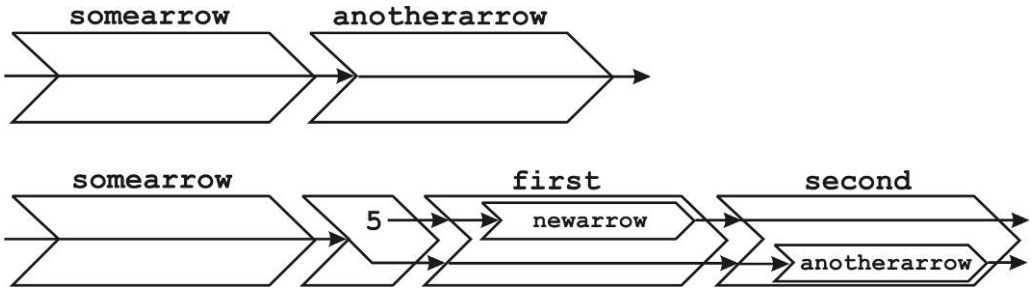
`arr` and `pure` (which are equivalent) correspond to the monadic `return` operation. While `return` constructs a monadic value simply returning a constant value without causing a side effect, `arr` returns an Arrow value without side effects based on a function. Functions are, of course, Arrows by themselve. A function to define an Arrow delivering a constant value (like `return` for Monads) is a special case of `arr` and can easily be defined for example by `arr (const x)`. The `>>>` operator corresponds to the monadic bind operator - in contrast to that, `>>>` is symmetric and expects Arrow values on both sides. Semantics are straight, the bind invokes the right side Arrow with the result of the left side Arrow. However, this symmetry implies that forwarding values through a sequence of Arrows has to be based on Arrows (while this can be done by functions on the right side of bind in the monadic world) and therefore, operations to lift Arrows into a multi-value computation chain have to be defined. This is achieved by the `first` operator, which lifts an Arrow to an Arrow operating on pairs and ignoring the second value. The `second` operator does the same, but applies the argument Arrow on the second value ignoring the first. `second` is defined based on the `first` operator. A simple example to illustrate this is:

```
somearrow >>> anotherarrow
```

We might want to put another arrow `newarrow` in between, which operates on some constant value, and let `anotherarrow` still work on the value of `somearrow`. So we have to use `first` to bypass `newarrow` with `somearrow`'s value:

```
somearrow >>> (arr (\x -> (5, x)) >>> (first newarrow) >>>
(second another)
```

Figure 15 - An Arrow sequence forwarding multiple values



`***` allows combining two Arrows to a single one, where the first Arrow operates on the first value and the second arrow on the second value. Again, this is by default implemented based on `first` and `second`. `&&&` applies two Arrows to the same input value and delivers a tuple with the two result values. A default implementation is present here, again.

Now we introduce further Arrow classes. First, the notion of error is added to the Arrow world by means of the `ArrowZero` class.

Figure 16 - The ArrowZero and ArrowPlus type classes

```
class Arrow a => ArrowZero a where
   zeroArrow   :: a b c
class ArrowZero a => ArrowPlus a where
   (<+>)       :: a b c -> a b c -> a b c
```

Here, the `zeroArrow` represents a null element in the according Arrow's domain and is usually interpreted as an error. Based on this class the `ArrowPlus` class provides the <+> operator typically applying the second Arrow to the input if the first one fails (i.e., it equals the `zeroArrow`). This behaviour corresponds to the `MonadPlus` class. However, these semantics are not mandatory, the formal definition only requires the <+> to be associative and to have `zeroArrow` as its unit **[26]**.

Figure 17 - The ArrowChoice type class

```
class Arrow a => ArrowChoice a where
   left  :: a b c -> a (Either b d) (Either c d)
   right :: a b c -> a (Either d b) (Either d c)
   (+++) :: a b c -> a b' c' -> a (Either b b') (Either c c')
   (|||) :: a b d -> a c d -> a (Either b c) d
```

`ArrowChoice` as another class is an outcome of the bind symmetry, like the tuple operators of the base `Arrow` class. Using Monads we can easily define functions on

the right side of bind to select some monadic value based on the input. With Arrows, the right side is an Arrow and while we can lift functions to Arrows, these have no side effects. We certainly could define Arrows selecting Arrows, but we had to use the respective internal data representation. So, as Arrows shall be abstract (which is even a necessity for I/O Arrows) and the combinators shall be generic, we need standard operators. The `ArrowChoice` type class provides such operators for selections.

Based on the `Either` type, uniting the domains of two existing types, we can select one of two Arrows based on the input. This is achieved by the `left` and `right` lift operations. `+++` allows to provide two arbitrary Arrows, each one with independent input and output types. The resulting Arrow therefore again delivers an `Either` value. `|||` bases on two input types but the argument Arrows deliver the same output type (as the combined Arrow does).

There are two further Arrow type classes in the standard library, `ArrowApply` and `ArrowLoop`. We will not discuss these classes here, please refer to **[26]**. However, we will see further user-defined Arrow classes in the next section when introducing the Haskell XML Toolbox. We now have introduced the Arrows concept and can summarize Arrows as the extension of Monads to input values or computations delivering some output based on some input. Obviously, combining Arrow values is more intuitive than combining Monad values and resembles function composition. The important bind operator now is symmetric, with some advantages as explained. However, as we cannot use arbitrary functions when combining Arrow values like with the monadic bind, the type classes turn out to be more complex than the respective monadic interface.

To be complete, we need to mention the do-notation for Arrows **[27]**. Like it delivers obfuscated code to distribute values across a Monad sequence by means of Lambda abstractions, it also proves to be confusing to distribute values across Arrows by means of the tuple primitives (`first`, `second`, `***`, `&&&`). And as the imperative style do-notation build upon the `Monad` class primitives solved this problem for monadic programming, the same has been defined for Arrows.

Example 7 - Monads vs. Arrows in do-notation

```
exampleM :: IO ()
exampleM = do
   str <- getLine
   putStr str
```

```
exampleA :: a b b
exampleA = proc val -> do
   str <- arrIO $ getLine  -< ()
   arrIO $ putStr          -< str
   returnA                 -< val
```

Please note that the Arrow example is based on the `ArrowIO` class later introduced in the HXT section - `ArrowIO` is not part of the standard library. However, one can see that the Arrow do-notation is mainly an extension of the monadic do-notation, providing a right side arrow symbol for the input value of each Arrow. The `arrIO` operation is used to lift `IO` Monad values to values of the respective (IO capable) Arrow type.

3.8.4. HXT

HXT is short hand for Haskell XML Toolbox and denotes a comprehensive library to process XML documents. Utilizing a general and flexible representation, HXT has been extended through the years both in terms of interface (starting with a pure functional filter interface **[24]**, continuing with a monadic interface **[24]** and now concentrating on an Arrow interface **[25]**), core capabilities (for example, XML validation) and additional XML facilities (for example, XPath). In this section, we introduce the core representation and the three programming interfaces. When discussing the interfaces, we will concentrate on the Arrow interface and its extensions to the Haskell Arrow type classes. We will conclude with some statements regarding states in Arrows (as this will be utilized in Janus) and the extensions beyond the XML core.

First, we discuss the primary data structure representing XML in HXT. The XML tree representation bases on a generic rose tree:

```
data NTree a  = NTree a (NTrees a)
type NTrees a = [NTree a]
```

This structure may store an arbitrary tree with nodes of a given type `a`, where tree operations are independent of the elements actually stored. These operations are condensed in the type class `Tree`, in which `NTree` has been installed. Moreover, a set of filter types and functions has been defined for generic NTrees. The type

```
type TFilter node = NTree node -> NTrees node
```

models filter functions which transform a tree of given content type to a list of trees with the same content type. This is actually a generalisation of a filter `NTree a -> NTree a` allowing substitution of a tree by more than one tree or by no tree at all (virtually deleting the filtered tree).

```
type TSFilter node = NTrees node -> NTrees node
```

`TSFilter` extends the original concept to work on a list of trees for the input. A rich set of functions has been defined to combine and transform `TFilter` values. For example, an operation working on the current node can be used throughout a specific traversion of the whole tree. Furthermore, some general functions to extend the standard monadic programming interface has been defined, which are independent of a concrete Monad and therefore have been implemented for all `Monad m`. E.g., `ifM` creates a Monad value which selects between two given Monads based on a given predicate. We will see these functions again in the Arrow interface. And we will see again the drawbacks of the Arrow bind symmetry, as for the Arrows world, `ifM` respectively `ifA` and similar operations have to be defined individually for any Arrow instance. For Monads, one can use functions for the right side of `>>=` to define such general combinators one and for all.

So far, we have a generic rose tree with according tree and filter functions. For XML, this tree is used together with the `XNode` data type as its payload, yielding the types:

```
type XmlTree  = NTree XNode
type XmlTrees = NTrees XNode
```

An `XNode` can be a tag (`XTag QName XmlTrees`), a text node (`XText String`), an attribute node (`XAttr QName`), a comment node (`XCmt String`), a DTD node (`XDTD DTDElem Attributes`), etc. Please note that for the `XTag`, the `XmlTrees` field represents the attribute list and not the element's children. The XML element structure is based on `NTree`, which allows using generic operations throughout the tree. This still continues with the processing of an element's attributes, although one has to select the attribute list of a tag afore. The `QName` type used for tags and attributes abstracts the identification of XML elements and attributes and may utilize namespaces.

Starting with `XmlTree` the HXT library provides a huge assortement of functions to create and process XML trees. For example, there are constructors for predicate filters. `hasAttr :: String -> XmlFilter` creates a filter delivering the input tree itself

if its top node is an `XTag` or `XPi` node with an attribute of the given name. All other trees are deleted. Other filters transform nodes, for example, `addAttr` adds an attribute of given name and value to an `XTag` or `XPi` node. All these filters can be easily extended to process a whole tree by means of the mentioned filter combinator functions, like processTopDown etc. Last but not least, XmlFilter values are used to construct a tree. For example, rootTag constructs a filter which independently of its input delivers a tree with a given list of children, an XTag node and according attributes. These children and attributes are also delivered by means of filter lists (where each filter independently of its input creates tree with XTag respectively XAttr nodes).

Now we are in possession of a pure functional processing system for XML trees represented by `NTree XNode` values and filters over them. But XML processing is usually embedded in a more complex system, which contains I/O (e.g., to read XML files or to print out XML documents to the console) and especially parsing of textually represented XML. So, what is further required is an extension into the world of I/O and side effects ("side effects" in the sense of utilizing a hidden state in addition to I/O). And we would like to get a unified solution, combining instances of a single expressive concept to fulfil all these requirements. We could easily build code based on the `IO` Monad and our pure functional XML interface. But it is a typical application for Monads to implement a hidden state and it became awkward if we would try to use a single newly defined state Monad together with the existing `IO` Monad. Therefore, we introduce a new Monad instance, the type `StateIO`. `StateIO` encapsulates functions of type `state -> IO (res, state)`, taking a state value and delivering a new state and a value by means of an `IO` Monad value. Basically, this combines the `IO` Monad with an arbitrary state Monad to a single new Monad. There exist several functions which allow working on the contained user specified state as a whole and to lift `IO` Monad values to the `StateIO` Monad (without affecting the contained state).

Figure 18 - HXT Monad state

```
data SysState = SysState {
   sysStateAttrs          :: !SysStateAttrs
   sysStateErrorHandler   :: !(XmlStateFilter ())
}

type SysStateAttrs = AssocList String XmlTrees
```

```
data XmlState state = XmlState {
   sysState    :: !SysState
   userState   :: !state
}
type XState state res      = StateIO (XmlState state) res
type XmlStateFilter state  = XmlTree -> XState state XmlTrees
```

The `Monad` instance actually used is `XState state res`, which uses `StateIO` with a state comprised of a system state and a user state. The user state is being represented by the type parameter `state`. The functional filter interface is lifted to the type

```
type XmlStateFilter = XmlTree -> XState state XmlTrees,
```

which provides state and I/O as required. The function `liftF :: XmlFilter -> XmlStateFilter state` allows to lift any `XmlFilter` to the new interface (without side effects and I/O, of course). Furthermore, there are functions to directly utilize the monadic interface, for example by reading a file and directly delivering the parsed content as `XmlTree`. Now we can write sequences of `XState` Monad values using lifted native functions, lifted `XmlFilter` functions and lifted `IO` Monad values.

At this point, we completed the introduction of the first two interfaces, the pure functional and the monadic ones. Now we proceed with discussing the Arrow interface, which is utilized in Janus. Fortunately, we can build upon much of the previous work as the Arrow interface is mostly an adaption of the monadic interface. First, we shall examine the Arrow type classes added by HXT to the standard library. These type classes have been announced earlier because they base on several monadic combinators defined. The monadic definitions have been made independent of concrete `Monad` instances due to the >>= operator issue. For Arrows, as with `ArrowChoice` and the tuple based Arrow combinators (`first`, `second`, `***`, `&&&`), we need overloading for each Arrow type installed. Therefore we provide additional type classes to abstract the new functionality:

Figure 19 - ArrowIO type class

```
class Arrow a => ArrowIO a where
   arrIO    :: (b -> IO c) -> a b c
   arrIO0   :: IO c -> a b c
   [...]
```

`ArrowIO` contains functions to construct Arrow values based on an `IO` Monad value or constructor function. `arrIO0` corresponds to the function

```
io :: IO a -> StateIO state a,
```

which again has been implemented for all `Monad` instances at once in contrast to `ArrowIO`. Of course, `ArrowIo` does only make sense for `Arrow` instances inherently supporting I/O.

Figure 20 - ArrowList type class

```
class (Arrow a, ArrowPlus a, ArrowZero a, ArrowApply a) => ArrowList a where
   arrL        :: (b -> [c]) -> a b c
   arr2L       :: (b -> c -> [d]) -> a (b, c) d
   constA      :: c -> a b c
   listA       :: a b c -> a b [c]
   this        :: a b b
   none        :: a b c
   withDefault :: a b c -> c -> a b c
   single      :: a b c -> a b c
   applyA      :: a b (a b c) -> a b c
   catA        :: [a b c] -> a b c
   seqA        :: [a b b] -> a b b
   [...]
```

The type class `ArrowList` addresses Arrows able to perform non-deterministic transformations, i.e., produce multiple results for a single input. A set of results is usually represented by a list type, hence the name. `ArrowList` provides operations to construct non-deterministic Arrows by means of functions (`arrL`, `arr2L`) and a deterministic arrow independent of its input by means of `constA`. By default, the output type of a list Arrow is not a list type, but the element type of such a list. This is because it is not intended to invoke a subsequent Arrow with the list as input, but one instance of the subsequent Arrow for each list element. The necessary mapping is done in the >>> implementation and is not made explicit in the type signature. This is actually the source of non-determinism, as each subsequent Arrow only sees a single input value and is unable to access possible parallel values. Each sequence of evaluation stays single-valued, the inherent concurrency is hidden.

However, one may want to operate on the whole result set of a list Arrow. To achieve this, non-determinism can be transformed into determinism by the `listA` function, which makes the set of results explicit. `this` represents the identity arrow, which simply delivers its input. This might be useful on a position where an Arrow is required, but no effect is wanted - for example, in the else part of an if-expression where the else part is mandatory, but unnecessary for the situation. `none` denotes the error Arrow, which can be considered to be the list Arrow always delivering the empty list. Actually, `none` equals the `zeroArrow` of the `ArrowZero` class, which is

consequentially a prerequisite for the `ArrowList` class to be instantiated for a given data type. `single` like `listA` transforms a given non-deterministic arrow into a deterministic one, this time by delivering at most a single value (discarding other elements of the result set). `withDefault` prevents a given Arrow to fail and returns a default value in this situation. `applyA` takes an Arrow delivering another Arrow based on the input and immediately applies the generated Arrow onto the input. `catA` and `seqA` combine a list of Arrows to a single one, where `catA` applies all Arrows to the input and concatenates the result (generalising the `<+>` operator of `ArrowPlus` to lists) and `seqA` serialises the Arrows based on their order in the list (generalising the `>>>` operator to lists).

Figure 21 - ArrowTree type class

```
class (ArrowPlus a, ArrowIf a) => ArrowTree a where
   getChildren       :: Tree t => a (t b) (t b)
   getNode           :: Tree t => a (t b) b
   setChildren       :: Tree t => [t b] -> a (t b) (t b)
   setNode           :: Tree t => b -> a (t b) (t b)
   changeChildren    :: Tree t => ([t b] -> [t b]) -> a (t b) (t b)
   changeNode        :: Tree t => (b -> b) -> a (t b) (t b)
   processChildren   :: Tree t => a (t b) (t b) -> a (t b) (t b)
   insertChildrenAt  :: Tree t => Int -> a (t b) (t b) -> a (t b) (t b)
   deep              :: Tree t => a (t b) (t b) -> a (t b) (t b)
   processBottomUp   :: Tree t => a (t b) (t b) -> a (t b) (t b)
   [...]
```

Next, we introduce the `ArrowTree` class. Although one could expect this to be a generalisation of `ArrowList`, there is no correlation. `ArrowList` bases on a property of the Arrow's inherent transformation (resembling a relation instead of a function) and is independent of the input and output types. In contrast, `ArrowTree` relies on a property of the input and output values, which the class expects to be tree structured. This property is enforced by requiring and utilizing the `Tree` type class for the input and output values. `ArrowTree` is comprised of the filter combinators' Arrow versions - for example, process and insert combinators can be found here. And again, to be complete, Arrows requires a type class to be installed while the Monad interface is realised by a single implementation for each function. However, if we consider an Arrow installed in this class to represent a filter of the functional and monadic interfaces, these functions can be used for any type installed in the `Tree` class. `XmlTree` as a specialisation of `NTree a` is such a type, making an Arrow `ArrowTree a => a XmlTree XmlTree` match the behaviour of the `TFilter` type.

Figure 22 - ArrowIf type class

```
class ArrowList a => ArrowIf a where
   ifA      :: a b c -> a b d -> a b d -> a b d
   neg      :: a b c -> a b b
   orElse   :: a b c -> a b c -> a b c
   [...]
```

The `ArrowIf` class extends the control capabilities of the Arrow interface and is actually the Arrow version of a set of the monadic interface functions. For the last time, we see some monadic functions being transformed into an Arrow type class. We will not introduce every single function of this interface, but only examples. `orElse` takes two Arrows and applies the second if and only if the first one fails. Failure is denoted by the empty result list, wherefore `ArrowList` is a prerequisite class for `ArrowIf`. `ifA` selects one of two Arrows by means of a third Arrow, which resembles a predicate. If this predicate Arrow delivers a value, the “true” Arrow is taken. If it fails (empty result list), the “false” Arrow applies. The “always false” predicate is equivalent to `zeroArrow`. A predicate Arrow can be negated by `neg`, which fails if the argument Arrow delivers a value and delivers a value (the input value) if the argument Arrow fails.

Figure 23 - ArrowState type class

```
class Arrow a => ArrowState s a | a -> s where
   changeState    :: (s -> b -> s) -> a b b
   accessState    :: (s -> b -> c) -> a b c
   getState       :: a b s
   setState       :: a s s
   nextState      :: (s -> s) -> a b s
```

`ArrowState` abstracts some functions of the `StateIO` Monad to access the Arrow's state. This time, the monadic interface would also require a type class to generalise these functions. The current implementation is actually restricted to the `StateIO` type while `ArrowState` is reusable.

We now introduced the new Arrow type classes of HXT. Yet, we do not know how to process XML with the Arrows interface. The functional interface uses filters of the type `XmlTree -> XmlTree`, the monadic interface uses filters of the type `XmlTree -> XState state XmlTrees`. As already mentioned, this can be represented by an Arrow `a XmlTree XmlTree`, where the Arrow utilized has to provide a suitable state for the HXT system. This Arrow is `IOSLA`. Like `StateIO`, this Arrow integrates state transformations as well as I/O and furthermore transforms a value `a` into a value `b`

(in contrast to `stateIO`, which simply delivers a value due to its Monad properties). `IOSLA` can be parameterized with a state, which is `XIOState us` in HXT. `XIOState` is comprised of a system state, which is the HXT processing state, and a user definable state (type parameter `us`). The type `IOStateArrow us b c` is defined as `IOSLA` with the (`XIOState us`) state (adding the user state `us` to the HXT state). Finally the `IOSArrow b c` is defined as `IOStateArrow` with the nullary user state `()`. The user state can be accessed by functions defined for the `IOStateArrow` and allows integrating an application specific state seamlessly into the HXT Arrow interface.

Figure 24 - The HXT IOStateArrow

```
newtype IOSLA s a b = IOSLA {
   runIOSLA :: (s -> a -> IO (s, [b]))
}

data XIOState us = XIOState {
   xio_sysState   :: !XIOSysState
   xio_userState  :: !us
}
data XIOSysState = XIOSys {
   xio_trace         :: !Int
   xio_errorStatus   :: !Int
   xio_attrList      :: !(AssocList String XmlTrees)
   [...]
}
type IOStateArrow s b c    = IOSLA (XIOState s) b c
type IOSArrow b c          = IOStateArrow () b c
```

The option to define a user state will be excessively used in Janus, but this will be discussed later. After all, the Arrows interface of HXT represents the XML filters by Arrows of type `IOstateArrow`. To complete the interface, there is a huge set of Arrow filters corresponding to filters of the functional interface. Given the example of `hasAttr :: String -> XmlFilter`, there exists an Arrow `String -> (IOSLA s) XmlTree XmlTree`. Actually, there exists a function `hasAttr :: String -> a XmlTree XmlTree` in a type class `ArrowXml`, in which `IOSLA s` has been installed.

To conclude this section, we would like to mention some extensions of HXT. For example, HXT meanwhile contains a facility to validate a given document against a DTD. Furthermore, there exists an XPath implementation. This allows doing selections and modifications based on an XPath expression. For example, the according interface provides a function `processXPathTree`, applying a given filter

Arrow to all nodes of a tree selected by an XPath expression and delivering the new tree. `getXPathTrees` simply returns the nodes denoted by a given XPath expression.

3.8.5. Haskell Plugins

An important property of Java Servlets in particular and configurable systems in general is the concept of Dynamic Loading. This concept allows loading object code into memory and linking it at runtime. An obvious advantage is that parts of the system can be coded, compiled and added although the application is already deployed. A simple example is the procedure of deploying a new web application to the Java Tomcat Server **[30]**. While the Tomcat server is running, one can build a new web application and force the server to load the Java class files belonging to this application. These files maybe did not even exist at the time of compiling and loading of the server.

Janus does not incorporate such facilities to load object code after booting, although such an extension is imaginable for the future. Nevertheless Janus already utilizes Dynamic Loading at startup for another important feature - configurability. The server configuration and its web applications will get described in an XML document - this is only possible if there is a way to interpret this XML description and afterwards build function values based on the contained information. The hs-plugins library by Don Stewart delivers this functionality **[28]**. It provides an interface to dynamically load object code while solving module dependencies transparently.

Figure 25 - hs-plugins interface with example

```
load_ :: FilePath -> [FilePath] -> Symbol -> IO (LoadStatus a)
data LoadStatus a
   = LoadSuccess Module a
   | LoadFailure Errors

demo = do
   mv <- load "Plugin.o" ["api"] [] "resource"
   case mv of
      LoadFailure msg -> print msg
      LoadSuccess _ v -> return v
```

The `load_` function (which is identical to the `load` function despite lacking support for package.conf files) allows to load a symbol (typically, a function value) from a module denoted by a file path. Furthermore, a list of file paths can be provided to search for modules needed to solve dependencies. The result reflects the

operation's success by means of an algebraic data type. Of course, it is a Monad value, as it performs I/O (at least file operations).

Besides there is another function, `pdynload`, adding type checking to `load`. Here the type of the object requested is checked against a given string represented type before actually loading it into memory. However, there is a runtime penalty of approximately 7% for this safety. Last but not least hs-plugins provides a runtime compilation interface for Haskell code, which is not utilized for Janus though.

3.8.6. Bytestring

This section is to emphasize a drawback of Haskell's list implementation and how it may be overcome in a useful way for certain applications. Lists in Haskell are generic recursive structures supported by language integrated syntax to describe list types and list construction. As they are defined polymorphic, lists and most functions over lists can be used with any list element type definable. Altogether the list facility in Haskell proves to be expressive, safe and flexible. As the compiler is able to map tail recursive functions onto iterations, lists are not restricted in size by their recursive implementation, fitting any real world requirement. But there is a major problem: efficiency. When compiled, Haskell lists are represented by dynamic data structures, where each element contains a pointer to the next element and a pointer to the payload value. Considering for example large string files, this representation is very problematic both in terms of memory efficiency and runtime efficiency for certain operations. A primary reason is genericity - as the size of list elements depends on the actual type and might be dynamic for each single element, an efficient implementation is hard to infer automatically.

If we drop genericity and concentrate for example on strings, there are solutions. This is because strings are lists of very simple structure, where each element represents a fixed length character. Additionally, strings are very common in applications and especially large strings and their processing is typical. Therefore an optimization of the special case of character lists is both simple to achieve and effective to accelerate real world programs. One such optimized implementation is the ByteString library by Donald Steward a.o. **[29]**. This implementation is actually not based on Haskell lists at all; instead it provides an independent data type named `ByteString`. This type is represented in memory by arrays and can be processed much faster than Haskell list structures. Furthermore some memory overhead for

pointers can be omitted. Acceptance of the library is assisted by an implementation of all major standard library list functions. This allows substituting a ByteString implementation and a [Char] implementation quite easily. Additionally, handling of string files by means of `IO` Monads has been provided. However, ByteString and Haskell string types are incompatible in general, which can make it hard to build applications combining both worlds. This may be necessary when using other libraries based on Haskell strings. Janus will actually use ByteString for some small but expensive parts of the system.

We do not give any example for ByteString program code as it is almost identical to an implementation based on the original Haskell lists. However, we would like to mention the usage of ByteStrings in Janus. As one will see Janus is based on representing requests and their processing by XML trees. Therefore situations occur where large files have to be inserted into such XML trees to be sent back to the client. Although consequent in terms of software engineering, it is an inacceptable solution to directly map arbitrary files into XML. In many cases it is even inacceptable to load a file into main memory at all. Janus implements an optimization based on ByteString to cope with the problem. Here a Monad value is constructed to read a file and deliver it to the client based on ByteString. This Monad is stored in a global state together with an identifier, which then is the only data actually inserted into the XML tree. By reading this identifier, retrieving the according Monad value and evaluating it, the Handler can finally perform the file operation.

4. Concept and Design

4.1. Overview

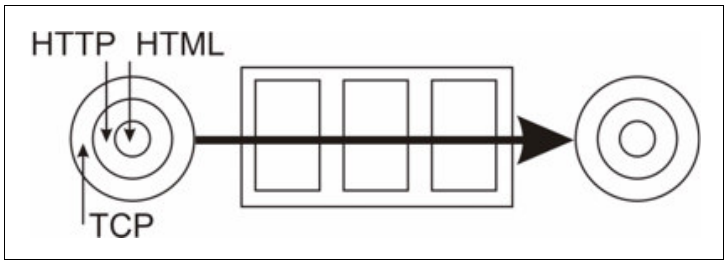
This section is to introduce the fundamental concept of Janus as well as the primary system entities and afterwards discuss each of these components in detail. The subsequent chapters will derive the current implementation from the concept and illustrate it with some examples. These examples will show how a Janus server is configured from the scratch for different purposes with increasing complexity. Both the server configuration and insights of actual processing in the server for the given configuration are shown. Moreover, there will be small code samples to give an impression of actual servlet programming in Janus.

The basic intention for Janus was to conceive and develop a server to host Haskell-based applications for the dynamic creation of HTML pages. Obviously, it would not be satisfactory to build a server simply invoking a certain function or executable and delivering the result to the client (which is, of course, the principle of CGI). Actually, the server should support the notion of a Servlet, like previously defined. The server should support the installed Servlets with according functionality, for example sessions, query parameters, creation of content like HTML, configuration, a mutable state and so on. Furthermore, one might question a possible concentration on TCP and HTTP as transport mechanisms. Like Java Servlets, Janus should not be limited to a predefined set of common protocols, as the logic of a Servlet might and should be reusable in several contexts (a simple example is to deliver files over FTP in addition to HTTP, where a mutual Servlet performs the identification and loading of files).

Taking this into account, Janus tries to define an abstraction of all major services in mind, which allows generalizing Servlets on one hand, while still delivering useful and especially reusable primitives on the other hand. This abstraction regards a server (or a service, since we would like to consider a server to be a set of services, for example on multiple TCP ports) as a pipeline transforming a request into a response, which is delivered back to the client. For the rest of this thesis, a transformation from request to response in whole is called a `Transaction`. Our Transaction term should not be confused with the transaction concept known from databases, where a transaction basically denotes an atomic state change which can

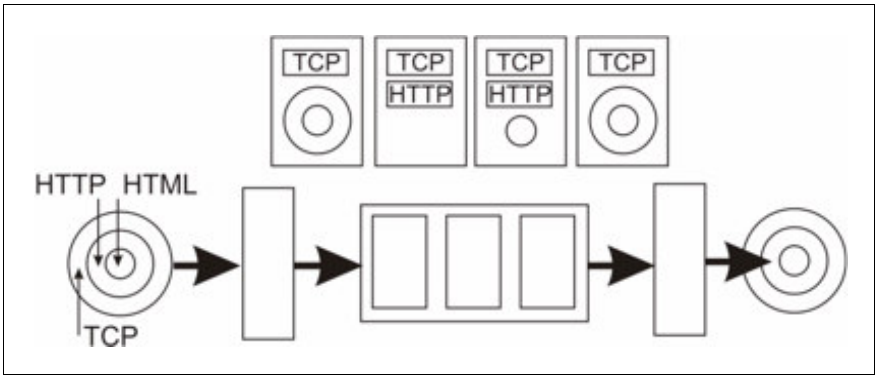
only succeed in whole or not at all. Although to enforce such behaviour for Janus Transactions could be an interesting task for the future, they currently only mark the logical combination of a request, its processing and the resulting response. The pipeline for a given service might be composed of several transforming components, which are plugged into each other to form a sequence. To let this well known general idea become useful for a server framework, a standard has to be defined which describes the way components of a concrete pipeline communicate with each other and make their results available to subsequent transformers in a unified manner.

Figure 26 - A pipeline transforming HTTP Requests to Responses



This leads to the idea to represent a Transaction by means of an XML tree or actually a sequence of XML trees, where each one represents a single snapshot of the Transaction. So there are at least two XML trees for any Transaction - the first representing the request and the second representing the request together with its computed response. In this case, the Transaction would have been transformed into its final form by a pipeline constituted by only a single component. However, in general there will be intermediate transformation states, i.e., a Janus service pipeline will be a sequence of an arbitrary number of XML tree transformers, each of which takes the current XML representation of a Transaction and delivers a new XML tree for the transaction.

Figure 27 - HTTP pipeline with Transaction "snapshots"



Each component of the pipeline is called a `Shader`. We also could have called this facility a “transformer”, but as the transformer term is quite overloaded in computer science, the term Shader has been chosen instead. The term references to lighting in 3D computer graphics, where a so called shader program gets applied to a set of pixels to compute a new color for each one of them based on some arbitrary algorithm. So we metaphorically consider the transformation of a Transaction to be some kind of lighting process, converging to the final result with every step.

Last but not least, we need a third component to embed a pipeline into. This component is to accept a request, to generate its initial Transaction form, to feed it to the Shader pipeline and to finally transmit a response build from the resulting Transaction state. This facility is called a `Handler`. Based on this concept of representing a network service as an XML transformation pipeline, the next three sections will discuss the three core elements just introduced in detail.

4.2. Transactions

So let's start with the simplest part of all, the Transaction. As already stated, a Transaction is considered to denote the process of transforming a particular single request into a response. Transactions are represented by XML trees according to some structural rules. It starts with an initial typically quite raw XML tree generated and issued by an according Handler, and is stepwise transformed by the Shader pipeline into a final XML tree. In the pipeline, each Shader takes the current XML representation of a Transaction and delivers a new XML tree (which might be identical to the input XML tree) representing the new processing state of the Transaction. By this a Transaction is represented by multiple (at least two, as mentioned above) XML trees, where each one defines the processing state of the Transaction at that moment in time (or more precise, at that stage in the Shader pipeline). Please note that a Transaction covers the whole request processing. Especially it is not valid to call the initial XML tree a request and the final XML tree a response – actually the final tree contains abstract information about both the request and the according response.

Now, how might a Transaction XML tree look like? At this point one might consider declaring DTDs and Schema definitions for such a tree to obtain a mandatory specification how to structure a valid Transaction (Which elements have to exist? Which elements are allowed to exist? Which values elements may contain? etc.).

Such definitions could be evaluated next to a Shader function to guarantee certain properties of the XML tree. By this, many checks inside the Shader itself could be omitted. Currently, Janus is not going this way. The main reason is that this idea looks quite expensive compared to the benefit. Before any Shader invocation, the respective tree has to be validated in whole and it is questionable if lazy evaluation can be used here to avoid unnecessary computations. The Shader itself only checks tree elements actually required for its work. Additionally, a strict and mandatory XML tree checking enforced by the surrounding server system would have a negative impact on flexibility, as any changes to structures (even small changes only required for debugging or with future applications in mind) have to be reflected in DTD and Schema definitions immediately. A conclusion might be that such a paradigm can be useful for testing new Shader implementations in a strict context. However, it should be removable for production usage and there should be no Shader functionality (including certain tests on the XML tree) relying on DTD/Schema evaluation. A less fundamental DTD/Schema facility according to these restrictions could actually get part of future extensions to Janus.

Now we shall have a look at the actual structure of Transaction trees. First, what is included in an initial Transaction as issued by a Handler?

- A Transaction shall be represented by an XML tree with a root element named `transaction`.

- There shall be attributes at the root element describing general properties of the Transaction: A Transaction state, a unique (regarding the runtime of the server) Transaction identifier, a creation timestamp and the Handler's identifier. The Transaction state shall indicate for example an error condition or the completion of processing.

- There shall be an element describing request meta information collected by the respective handler, for example `tcp` for a TCP Handler.

- There shall be an empty element named `messages` to contain Transaction level messages (in contrast to Handler or server level), including control messages, warnings and errors.

- There shall be an empty element `response_fragment`, which is to contain the plaintext response finally transmitted back to the client by the Handler.
- There shall be an element `request_fragment`, which contains the plaintext request right from the beginning.

Here we have an example for a typical initial Transaction generated by a TCP Handler receiving an HTTP request:

Example 8 - An initial Transaction

```
<transaction transaction_id="1" transaction_state="Init" handler="TCPHandler"
      start="1157647830796">
   <tcp remote_ip="127.0.0.1" remote_port="1174"/>
   <messages/>
   <request_fragment>
      GET /index.html HTTP/1.1
      Host: localhost
      [...]
   </request_fragment>
   <response_fragment/>
</transaction>
```

During processing, each Shader reads, adds, changes or deletes values or whole subtrees of the Transaction. For example, an HTTP related Shader may operate on a subtree `/transaction/http`. Please note that here and for the rest of the thesis we utilize XPath expressions to denote XML nodes. As subsequent Shaders rely on information added to the tree, it is mandatory for a Shader's specification to state the modifications the Shader does to a Transaction (and especially which information is added or changed). To conclude this section, we show a possible final Transaction tree for the small example above:

Example 9 - A final Transaction

```
<transaction transaction_id="1" transaction_state="Processing" handler="TCPHandler"
      start="1157647830796" end="1157647831044">
   <tcp remote_ip="127.0.0.1" remote_port="1174"/>
   <http>
      <response status="200">
         <body>[...]</body>
      </response>
      <request url="/index.html" uri_scheme=""
            uri_path="/index.html" uri_query="" uri_frag="" method="GET">
         <body></body>
      </request>
```

```
    </http>
    <messages/>
    <request_fragment>
       GET /index.html HTTP/1.1
       Host: localhost
       [...]
    </request_fragment>
    <response_fragment>
       HTTP/1.1 200 OK
       Server: Janus/0/240406
       Date: Thu, 07 Sep 2006 16:50:30 GMT
       Content-Type: text/html
       Content-Length: 12647
       [...]
    </response_fragment>
</transaction>
```

4.3. Shaders

In Janus, a Shader is a transformer which takes an XML tree and delivers a new XML tree. The transition between these two trees represents the work a particular Shader performs in the pipeline. For the moment, let's consider a Shader to be a function with signature

```
shader :: XmlTree -> XmlTree
```

As we will see, this approach is too naiv. However, it expresses the basic idea in the most explicit way. So, what do we expect an actual Shader to do? We already saw most of the concept throughout the previous section. First of all, we start with the Transaction describing the request. In general, a request will be plaintext, for example the textual representation of an HTTP Request. This is not necessarily the case, but it is the most common case. To do some work on it, we need to feed this "flat" request to a Shader. We do this by the aforementioned XML element `/transaction/request_fragment`, which contains the original plaintext representation. The first Shader might recognize the content of the request fragment to be HTTP, parse it and generate XML entities (elements and attributes) in its resulting XML tree to make the HTTP information available in a unified way. For example, the Shader might add the above mentioned `/transaction/http` subtree to the Transaction and insert attributes for the requested URI, the HTTP method, and so on. The resulting tree is forwarded to the next shader in the chain, which might be a Shader to insert a requested static file into the Transaction tree. Another Shader could be defined to not insert the according file but to analyse the filename

(extension) or even the file content to add a MIME type definition to the response. Because multiple Shaders actually work together (although in chain), it is not useful to build up the response directly in plaintext, but in a defined structured representation – for example a `/transaction/http/response` subtree. A final Shader in the pipeline could easily take the resulting response subtree, infer a plaintext representation by means of a generic implementation and store it in the `/transaction/response_fragment` element. Of course, this only happens if a plaintext representation exists, which might not be the case for an inconsistent response subtree or if an error occured.

After all, Shaders are our central element of Servlet programming. Indeed, we can consider a Shader to be some kind of generalized Servlet. The main deficiency currently is the lack of state. Functional XML transformation may store a state on the client, but to achieve Servlet functionality according to our definition, there must be a notion of server side state. However, we postpone discussion about these extensions to sections about the Shader implementation in Haskell. For the moment, our understanding of Servlets in Janus is an XML tree transformer, where the tree encapsulates request, response as well as intermediate data in a structured and unified way and allows access to previous Shader's results.

4.4. Handlers

As described in the previous sections, a service in Janus is build up by a pipeline of Shaders. This pipeline usually starts with a Transaction tree containing basically a `request_fragment` (the request plaintext) and ends with a Transaction tree containing the request, lots of additional structures inferred by intermediate transformations and a `response_fragment` with the resulting plaintext for the client. But how do we integrate such a pipeline into a real interactive server implementation? The corresponding system entity is called a `Handler` and it is responsible for:

- Accepting requests according to the Handler type (e.g. TCP/IP) and a given configuration (e.g. TCP listening port).

- Creating an initial Transaction tree. This might contain the `request_fragment` only (beside standard Transaction tree elements), but in general the initial Transaction can also contain arbitrary XML structures and besides, it is not required to even

contain data in the `request_fragment` (but is required to contain an empty `request_fragment` element in this case).

- Invoking a given Shader with the initial transaction.

- Delivering a response. Typically, this implies to retrieve the content of `response_fragment` and deliver the plaintext to the original client by means of the Handler's transport mechanism.

Additionally, a Handler may do some preprocessing and add meta information to the Transaction (the request data is not required to be the only information provided by the Handler). This preprocessing may contain information about the Handler itself (which Handler generated the original Transaction), its configuration and some data about the request (in the case of a TCP Handler, this might include the address of the requesting client etc.). Beside the possibility to add transport mechanism specific data to a Transaction, the Handler concept simply separates the technique requests and responses are transmitted by from the logical processing of requests. For example, one can use the same Shader pipeline to process HTTP requests for a TCP Handler and for some kind of debug handler, where the requests and responses are represented by text files or are exchanged by stdin/stdout streams.

4.5. Server

Given the concepts of Shaders and Handlers, the server itself becomes an easy part to describe. Basically, the server in Janus terms is a set of Handler definitions. Each Handler definition contains a Shader to process requests received by this Handler. While these descriptive and conceptual properties are quite concise, in reality there are lots of operational tasks to be settled by the actual server implementation. We will come to a detailed insight within the following chapter. However, a concept has to take operational system properties into account. Therefore we would like to discuss several responsibilities the server as a system entity has to live up to. Primarily, there are loading and configuration. If a system is made up of Shaders, which are systematically composed to pipelines (where a pipeline is still a Shader, though), which on their parts constitute the server, there must be a description how these parts are connected to a runtime system. One may code these connections directly in Haskell, compiling the whole server into a single, static binary. This does not make much sense and proves to be inferior to common practice in other web

server systems. So, system components should be written and translated separately, there should be a formal language to describe the system structure (i.e., the composition of Shaders, Handlers and their respective configuration) and the server should be the entity to understand this language and to build the actual runtime structure at load time.

To conclude this chapter, we can summarize that the idea fundamental to Janus is to see a network server as a set of services, where each one uses a pipeline of Shaders to transform an XML represented request into an XML represented response. The system structure (existing Handlers and the composition of their pipeline Shaders) is coded in a configuration language to be translated into a runtime system by Janus at load time.

5. Implementation

5.1. Introduction

The idea of a service pipeline constituted by Shaders implies a primary requirement, the ability to compose Shaders to form new Shaders and at last a pipeline. Given the signature of a Shader stated when introducing its concept (`XmlTree -> XmlTree`) composition of Shaders would be nothing else than simple function composition. However, we already mentioned that a pure functional point of view is not enough to fulfil our own definition of Servlets. We need a server-side state, which is missing in the Shader signature. Of course an existing server-side state could be copied into the Transaction by the Handler and back after the Transaction processing completed. Managing a shared state in Haskell could be completely hidden from the Shaders this way. Indeed, this approach is by far too inefficient when thinking of raisonable large server states. There should be state-aware (and I/O-aware) programming directly in the Shader code and if we look for a concept resembling functions and providing sophisticated state management capabilities connected with a combinator library, we will find Arrows. Moreover, given the possibility to define an arbitrary I/O-capable state, we can easily imagine to add further functionality important to Servlet programmers – for example messaging and error handling.

5.2. The Shader Arrow

We saw that serving a request can be modelled as a transformation of XML trees. A system to represent and process XML documents is required then. HXT, as shown, is such a system. Additionally HXT provides a straight approach to coding based on its capabilities by offering a fully featured Arrow interface. The Arrow filter used resemble the idea of Shaders given here, as they take XML and deliver XML. But these Arrows provide more than just filters to handle XML. They are based on a flexible implementation seamlessly integrating I/O and a user state when required. By using HXT we get a Shader representation and an environment to embed our own functionality.

So, the approach to model the system's most important component, the Shader, shall be:

Figure 28 - The Shader Arrow

```
type Shader             = XmlTransform Context
type XmlTransform s     = JanusArrow s XmlTree XmlTree
type JanusArrow s a b   = IOStateArrow s a b
```

`XmlTransform` is a type synonym for HXT's `IOStateArrow` configured for XML input and output. The `Shader` type adds a user-defined state of type `Context` to the `XmlTransform` Arrow.

Beside some denotational sugar concerning the Arrow type involved, we have to think about the user-defined state `Context`. If we consider Servlet functionality to be restricted to transform XML trees in a pure functional manner, the `JanusArrow () XmlTree XmlTree` or simply `IOSArrow XmlTree XmlTree` would have been enough. However, we already mentioned the requirement of additional facilities (state, messages, errors), which are not self-evident for the pure functional world of Haskell. The `Context` type extends the Arrow by exactly these functionalities. Let's have a look at the structure of Context:

Figure 29 - The Context state type

```
data Context = Context {
   c_cfg       :: ConfigArrow,
   c_sRep      :: Repository ShaderCreator,
   c_shadow    :: Shader,
   c_scopes    :: Map String SharedScope,
   c_channels  :: Map String SharedChannel,
   c_msgbuf    :: Messages
}
```

5.3. Shared Mutable State

First of all, beside the `Context` being the state for our Arrow we have to define the mutable state. From a programmer's point of view, the mutable state might be simply called "the state" regarding his Shaders. Nevertheless, formally the mutable state is only a subset of the Arrow state. So, let's define the properties of this mutable state:

- The mutable state shall be tree-structured with a hierarchical namespace to address elements. Every node, including inner nodes, shall be able to contain information (with the empty node expressible) and shall be able to maintain an arbitrary number of children (so this is a rose tree).

- To address elements, XPath expressions shall be utilized. This is because to denote elements in the Transaction-XML XPath will also be used, so we can introduce a unified scheme here to address elements in both the mutable state and the Transactions. However, despite the use of XPath, path expressions will not in general be interchangeable between the two. This is due to the fact that the mutable state will not support ambiguous names (i.e. childrens with the same name) and some XML facilities (the attribute axis, namespaces, etc.).

- The payload of each tree node shall be an arbitrary string value.

- The mutable state shall be used concurrently, i.e., multiple threads will work on it and should see each other's updates.

- There shall be mechanisms of safety and synchronisation to handle concurrency. These cover locking of tree nodes to prevent racing conditions and resulting inconsistencies as well as facilities to regard up-to-dateness of given nodes.

- There should be support for the concept of listening **[8]**, i.e., a thread shall be able to wait for a state changing operation to occur on a given node.

The mutable state in all is stored in the `c_scopes` field. Actually, this field manages a set of so called scopes, where each scope is represented by a name string and contains a complete mutable state of type `SharedTree` according to the requirements above. One could regard this solution to be a forest of mutable states or as the root level of the mutable state tree. But for the second interpretation to be valid an important property is lacking: The scope map is not based on a mutable variable and therefore, updates on this map wouldn't be visible to other threads. This first unshared map is used to allow differences between scopes of Handlers, because now it is possible to insert an exclusive scope with the same name for each Handler.

For the scopes, a simple interface can be defined:

Figure 30 - Mutable state scope interface

```
listScopes  :: JanusArrow Context a String
getScope    :: String -> JanusArrow Context a SharedScope
addScope    :: String -> JanusArrow Context a a
swapScope   :: String -> JanusArrow Context a a
delScope    :: String -> JanusArrow Context a a
```

`listScopes` delivers an Arrow which independently of its input delivers the list of names of the scopes present in the state. This Arrow is non-deterministic, for a real list to be delivered the `listA` function has to be used (converting `listScopes` into `JanusArrow Context a [String]`). `getScope` returns the scope selected by the string argument (ignoring the input value). `addScope` creates a new empty scope with the given name (where the Arrow simply returns its input). `swapScope` replaces an existing scope by a new one (again simply returning the input value). `delScope` finally deletes the scope denoted.

Figure 31 - The StateTree type

```
data StateTree = StateTree {
   t_id           :: String,
   t_cell         :: StateCell,
   t_ts           :: JanusTimestamp,
   t_children     :: Map String SharedTree,
   t_listeners    :: [StateListener]
}
```

A single scope (of type `SharedScope`) is equivalent to the type `SharedTree`, which represents a tree stored in a mutable variable. The `StateTree` type stored in this variable contains the local identifier of the node, the value of the node, the node's last change timestamp, a list of children and finally a list of listeners. The value is not simply implemented by a string field, but by an abstraction called `StateCell`. The reason is that there should be a way to separate structural updates and value updates because otherwise, for example a series of update operations on a node's children would have to be considered to imply value updates for the node concerned. To implement the separation, the timestamp information integrated into the node covers structural changes (changes of the children list), and additionally the `StateCell` structure contains a timestamp to cover updates on its value. A cell's value is again not implemented directly as a string, but by an algebraic data type: `StateValue`. This data type contains a constructor for string values (`SimpleVal`), but furthermore allows storing maps, trees and so on by means of specialized constructors. The main raison to provide such a structure is efficiency - clients of the mutable state might want to store large data structures without the need for the state's features (synchronisation, immediate XPath access, etc.). In this case, the data structure may be stored as a single value. Moreover, some constructors allow storing function values, which cannot be represented as strings. As a disadvantage

this solution requires a constructor for each special type requested and therefore proves to be hardly extensible by a user.

The children of a node are represented by the same type already used for the scope map - an immutable map of names to `SharedTree` values, where the keys are the local names of each child. In contrast to the scope map, we do not have the property of immutability regarding threads here, because the map is part of a type (`StateTree`) stored inside a shared mutable variable. Finally, the listeners are a list of so called `StateListener` values of type `MVar StateOperation`. To listen, a thread simply has to insert an empty `MVar` into the list and block on it by means of the `takeMVar` operation. All listeners are released by putting a value of type `StateOperation` into each of these `MVars`. The listeners immediately return with this `StateOperation` being the result of the `takeMVar` operation. A `StateOperation` is an algebraic date type, covering deletion, insertion, updating and unspecified operations together with the fully qualified XPath concerned. This way a listener not only recognizes the fact of a state change and which node is involved, but also the type of the state change without any additional querying. In the update case, the new value is transmitted along with the constructor.

Figure 32 - The shared mutable state interface

```
(!=>)             :: String -> String -> JanusArrow Context a StateCell
getStateCell      :: String -> String -> JanusArrow Context a StateCell
getStateTreeTS    :: String -> String -> JanusArrow Context a JanusTimestamp
listStateTrees    :: String -> String -> JanusArrow Context a String
listStatePairs    :: String -> String -> JanusArrow Context a (String, StateCell)
(<=!)             :: String -> (String, StateCell) -> JanusArrow Context a a
setStateCell      :: String -> String -> StateCell -> JanusArrow Context a a
setStateCellSafe  :: String -> String -> StateCell -> StateCell
                     -> JanusArrow Context a a
delStateTree      :: String -> String -> JanusArrow Context a a
(<=!>)            :: String -> (String, JanusArrow Context StateCell StateCell)
                     -> JanusArrow Context a StateCell
```

Next we discuss the interface visible to the programmer when accessing the shared state. A central idea is to use the scope functions to get in possession of a scope value and subsequently use a state operation Arrow on this value. Please note that there is no public function to retrieve a scope from the `Context` - this operation is hidden and implicitely invoked by the state Arrows themselve (where a string parameter for the scope involved is required). Although a sequence of operations on the same scope may get less efficient this way, code gets significantly more

readable without a scope access for every distinct state operation. Regarding state access, there are operations to list the children of a node (`listStateTrees`), to query a node's structural timestamp (`getStateTreeTS`), to get its value cell (`getStateCell`, which is equivalent to the `!=>` operator), to change a node's value cell (`setStateCell`, which is equivalent to the `<=!` operator), to delete a node (`delStateTree`), to swap the value cell of a node (`<=!>` operator) and to listen to a node. All operations base on XPath names to specify the nodes involved. For get, set and swap, there are specialized operations (denoted by new operators), which work on strings or `StateValues` instead of cells (e.g. `!->` retrieves a string and `!$>` retrieves the `StateValue`). This is especially useful if some features are not required, but would enforce much redundant code for boxing and unboxing. Fourth versions of the aforementioned functions (e.g. `!*>`) allow polymorphic storing and retrieving of string values - if the type in question has been installed in both the `Show` and the `Read` classes. Indeed, there is a pitfall involved here, because the `Show` representation of a string value (which is required for the respective `read` function) is not equivalent to the string itself (but equals the string with leading and trailing quotes). Therefore, one cannot store a string value with the string operator and retrieve it by means of the polymorphic operator (and vice versa - storing polymorphic and retrieving as string).

An additional version of the list operation exists (`listStatePairs`) which not only retrieves the list of child names, but at the same time delivers the value cell of each child (a list of `(String, StateCell)` tuples). Again, a specialized version for string values exists, delivering a list of `(String, String)` tuples. It has to be mentioned that there does not exist a node adding or creation operation as the set operations actually cover both construction and update functionality. If the node to be changed does not exist, it is created and every nonexisting intermediate node on the path as well. The intermediate nodes are created empty, of course. Furthermore there exists a "safe" version of the set operation for cells (`setStateCellSafe`). This operation is parameterized with a `StateCell` to be compared with the current cell of the node concerned. No update is performed if the cell stored does not equal this parameter. By this a thread may provide the previous cell known to him (and on which further computation was founded) to ensure no intermediate update of the node in question gets overwritten (and ignored by the current thread). To be complete, the timestamp of a value can be acquired by getting the cell in question and using a cell arrow to

extract the timestamp. Next, we show some simple examples how the state access may be used:

Example 10 - State access

```
1. ("global" <$! ("/somenode", SimpleVal "myval"))
2. ("global" !-> "/somenode")
3. (delStateTree "global" "/somenode")
```

The first example denotes an Arrow storing a `StateValue` representing the string “myval” in the global scope at the node `/somenode`. The second example reads the value stored in the same node in string representation. The third example removes this node from the state, including its subtree. Except the second Arrow, which returns the stored string independ of the Arrow’s input, these example Arrows simply return their input values, i.e., they completely rely on side effects regarding the hidden state.

At this point we conclude the introduction of our shared mutable state with scope functionality and continue the discussion of the `Context` type, representing the whole user state throughout the Janus system. `c_cfg` contains a `ConfigArrow`, which simply is an Arrow delivering an XML tree denoting some configuration. In the `Context`, this Arrow represents the system configuration, precisely the server configuration file. This part of the `Context` is not updateable shared among threads, as no changes to the server configuration are conceptually allowed for components at runtime. However, if they do, no other thread is affected. The `c_shadow` component contains an extendable Shader value which is meant as a replacement for the Shader currently processing. This way Shaders shall be enabled to provide an optimized version of themselve each time they are invoked. A simple interface allows extending and replacing the current shadow. Of course, the shadow value is not shared among threads as by definition it applies to a distinct Shader evaluation by a single thread only. Currently, the shadow functionality is unused. To get it working, optimizing Shaders and Handler support to replace the pipeline by the shadow value after a request would be required. `c_sRep` is a repository for `ShaderCreator` values. We will discuss the facilities to load Shaders when looking at the server level implementation. For the moment, it shall be enough to state that `c_sRep` allows Shaders to create and execute new Shaders based on types known to the system. This way, `c_sRep` supports a concept of higher order Shaders.

5.4. Messaging and Errors

For the last components of `Context`, `c_msgbuf` and `c_channels`, we enter the area of messaging, logging and error handling. We start with the simple idea to define a single concept for managing messaging, logging and error handling, where messaging for example covers outbound communication to the user and internal communication between system entities. It should be a single concept because these three functions seem to share application fields. For example, an error may be connected to a message to the user (or to another system component). A message to the user may also be logged, as well as an error message. And components may be interested in messages in the sense of events without them being errors. So, the idea is to combine the properties of messaging, logging and errors to define a set of unified operations and gain interchangeability.

We aggregate all this under the concept of a `Message`. For a message, we define a type, a timestamp, a source, a level, a code, a value and a state. We shortly introduce each of these parts:

- The **type** describes the context of the message and may be `ControlMsg`, `ErrorMsg`, `WarningMsg`, `EventMsg`, `LogMsg` and `PlainMsg`. While each system entity may define its own understanding for each of these types, there are no generally enforced semantics for one of them. Especially, there is no order in the sense of priorities - this functionality is delegated to the message level. Therefore, one may issue error messages at a low message level so that they are filtered out everywhere despite representing errors. However, there is some informal intention connected to each type. Event, warning and error types should be intuitively clear. Control messages shall represent messages between system components not meant for human reading. Log messages are especially meant to be written to a logging facility despite being displayed or further processed. Plain messages should be used for communication with the user (to display, for example, the messages of the boot up sequence).

- The **timestamp** represents the message's time of creation.

- The **source** may be used to represent the message's issuer. However, this field is of informal manner.

- The **level** represents the priority of a message, where lower values are more important. Some constants have been defined for unification, although arbitrary integer values may be used. The predefined constants are `l_control`, `l_debug`, `l_info`, `l_warn`, `l_error` and `l_mandatory`. The mandatory level denotes messages which are typically not more critical than warnings or errors, but shall be displayed to the user in any case. The boot up sequence may be an example for this. Please note again that the control, warning and error levels are not necessarily connected to the respective message types.

- The message **code** provides a formal way to denote message information. Automatic processing of messages can be implemented much easier and more stable if based on specific number represented values instead of human language strings (which are otherwise used for the message semantics).

- The message **value** contains the message itself as a string. With regard to the aforementioned message code, please note that the value is almost always meant to be read by a human. To deliver message semantics to a machine, the message code should be used.

- A message's **state** is a simple map of names to values (both implemented by strings). Athough it may be used arbitrarily, it is meant to support the message code with additional information. For example, the message code may select an operation in the context of a control message, and the state may deliver arguments to this operation.

Now that we know what a message contains, we may think of how to represent it in software. The straight idea would be to define a structured data type strictly containing the above mentioned information as fields. However, a different approach has been chosen: A message is represented by an XML tree. We see XML again here and here are the reasons:

- XML is already utilized throughout the system, so we can revert to existing libraries, code and experience.

- We gain the ability to extend Shaders to the processing of messages.

- We can easily store messages in the shared state without a specific extension beyond XML trees.

- We can easily insert messages into Transaction trees and use the same operations to process them as with messages stored in the Context.

- We could extend the concept of Shaders to processing messages, i.e. taking messages as input and delivering messages as output. Although this idea is not followed currently, utilizing XML to represent messages and to insert them into Transaction trees already allows for seamless integration of message handling in Shader code.

The mapping of the necessary message fields to XML is done by an according module. This module hides the structure of the XML messages by providing algebraic types and type synonyms for each field along with construction and access methods. Furthermore a filtering interface is defined. This interface bases on the type `MessageFilter s`, which is a type synonym for `XmlTransform s` or `IOStateArrow s XmlTree XmlTree`, where `s` is an arbitrary state. One can easily see that this type can be used inside Shader Arrows (where the state is instantiated to the `Context` type) and resembles the data type used for Shaders. With the `Context` state, the `MessageFilter` type is actually equivalent to the Shader type. `MessageFilter` values are meant to forward incoming messages fulfilling a given predicate and can be combined like any other Arrow to construct a more complex filter. Typical filters are, for example, filters to only forward messages of a given message type or with a message level lower or equal to a given one. There are filter constructors for these typical cases in the `Messaging` module, but no functions to combine or evaluate filters. For the latter purposes the system relies on the Arrow operations and additional functionality in the client modules for `Messaging` (`Shader` and `Transaction`). It has to be mentioned, after all, that although no direct access to the resulting XML trees is necessary regarding message functionality, programmers may use general HXT functionality to perform operations on the message XML trees. However, the hiding of structure by the `Messaging` module may break this way, making changes to the representation harder.

But back to the `Context` type. We just introduced a representation for messages suitable for general messages, errors and logging. Now we shall have a look at their usage in the `Context`. First, we have to introduce another concept based on messages, the concept of channels. A `Channel` in Janus is defined as:

Figure 33 - Channel definition

```
newtype Channel      = Ch (Messages, MessageHandler, [ChannelListener])
type MessageHandler  =
   JanusArrow Context (Channel, Messages) (Channel, Messages)
type ChannelListener = MVar Messages
```

So, a `Channel` is a tuple of a list of messages, a `MessageHandler` and a list of `ChannelListener` values. The message list shall be a buffer for messages, which can be utilized by message handlers to store messages for later use. Without according functionality of a message handler, the message list stays empty by concept. A message handler is an Arrow to infer a new channel and a new list of messages from an input list of messages and a Channel (which usually should be the Channel the handler is contained in). The idea behind this Arrow, which is based on the `Context` state itself, is to filter the incoming messages (with the result of a new message list) and possibly do a side effect based on the `Context`. The parallel Channel transformation is required to easily put messages into the message buffer. A side effect could be the writing of a message into a log file, for example. By concept, a filter (which could be a composed list of filters) is processed for each single message. However, to maintain flexibility a message list is used. The `ChannelListener` type bases on the same ideas as the `StateListener`: A listener may provide an empty `MVar` for a Channel's listener list, block on this `MVar` and wait for a message being put into the `MVar`. The invocation of listeners is not implemented by default (like the usage of the message buffer) but can be achieved by an according invocation handler. By this one can decide at which point of filtering the invocation of listeners shall occur - the alternatives would be at the beginning of filtering for every message or at the end for messages which pass the whole filter, but this seems to be too restrictive. An invocation handler requires the transformation of the Channel, like a buffering handler does.

For the `Context` state and its `c_channels` field, Channels are used in a shared way, like the scopes of the shared state:

```
type SharedChannel    = MVar Channel
```

Therefore changes on a Channel are visible to any thread operating on the same `Context`. However, the addition or deletion of Channels is local to the respective thread. This behaviour equals the behaviour of scopes. As for the scopes, there exists a simple interface to enumerate, add, retrieve and delete Channels from the `Context`. To work with messages, the Shader module provides functions to configure

a Channel's handler (by removing, replacing or adding a `MessageHandler` value), to retrieve and remove messages from a Channel's buffer, to send a message to a Channel (where an equivalent shortcut operator exists, expecting the Channel name on the left side and a message Arrow on the right side), to define a message send Arrow catching a failed Arrow (`zeroArrow`), to apply a `MessageFilter` value to the message buffer and to listen on a Channel or invoke it. Like with the shared state, the message operations are all connected to a Channel name. The necessary querying of the Channel value from the `Context` is a hidden operation implicitely invoked by each message operation.

For the configuration of message handlers, a set of predefined `MessageHandler` constructors exists. These include for example a handler applying a given `MessageFilter` to its input, a handler storing each message passing by in a log file, a handler invoking the Channel listeners for each message and a handler copying each message into the Channel's message buffer.

Last but not least, there is a field `c_msgbuf` in the `Context` state. This field is required to support error handling. We have seen the processing of messages in the Arrow state based on Channels. There will be another usage based on Transactions, managing messages local to the current Transaction. In this case Shaders add (or generally process) messages to a specific subtree of the current Transaction (`/transaction/messages`). A problem arises if an error is inferred from the failure of an Arrow. To generate an error message there has to be an alternative coded for the failing Arrow (for example, by the `orElse` operation in `ArrowIf`). Typically, after such an error message is generated, one would want the whole Shader to fail - especially to avoid further error handling after the critical point. But if the whole Shader fails (which is catched by a wrapper added to the Shader at loading time), it has no result (because this is the way errors are coded in HXT Arrows: empty results lists). So the Transaction tree containing the new error message would not be returned from the Shader. The solution is to push all new messages into a `Context` message buffer and let the Shader fail. The aforementioned wrapper detects new messages inside the buffer and adds them to the Transaction state before the failed Shader's invocation. This way messages can be added to the Transaction inside a failed Shader and furthermore, all effects of the failed Shader on the Transaction are implicitely rolled back (because the previous Transaction state is utilized for adding the messages). The `Shader` module provides functions to

forward the buffer's content to a Channel, to clear the buffer and to retrieve its content. More important, there are operators to catch failing Arrows, store an according message into the buffer and fail again. These Arrows resemble similar Arrows mentioned above to catch `zeroArrow` and send a message to a Channel - the main difference is the usage of the buffer in one case (to finally get the message into the Transaction structure) and the transmission of messages to a `Context` Channel in the other case.

5.5. Handler

Given the implementation of Shaders, Handlers are easy to introduce. As already mentioned, conceptually Handlers are configured with a Shader value (representing its processing pipeline for requests). When invoked, they start retrieving requests to be handled by its Shader. As most Handlers can be considered to be non-terminating (as they implement an infinite loop to retrieve requests) each one should be launched inside its own thread. A straight implementation could be `Handler :: IO ()`, which would be immediately compatible to the `forkIO` signature (`IO () -> IO ThreadId`). However, this would lead to a mixup of both monadic and Arrow code in the server. It would be much more elegant to hide all monadic code (which is inherently necessary to perform I/O) and to encapsulate the program into Arrow representation only. As another advantage of such a design, the `Context` state could be easily distributed among system components; it would be present everywhere automatically. The Handler type is defined this way then:

```
type Handler = JanusArrow Context () ()
```

This type is almost equivalent to the `IO ()` type in that it takes nothing and delivers nothing (except a side effect). However, it is equipped with the `Context` state type. For most I/O operations, the transition to such a paradigma would be easy, utilizing the `arrIO` functions in the `ArrowIO` class. But for example, `forkIO` is a problem. While there are generic operations to transform `IO` Monads into Arrows, the same is not true for the opposite direction. However, there exists a specialized operation, `runX :: IOSArrow XmlTree c -> IO [c]`, which transforms an `IOSArrow` taking an `XmlTree` value as input and delivering an arbitrary type `c` into an `IO` Monad value, delivering a list of type `c`. By using this operation, we can define:

```
processA :: JanusArrow s a b -> JanusArrow s a ThreadId
```

`processA` takes a generic `JanusArrow` and transforms it into a new Arrow. This new Arrow delivers a thread id instead of the original value and evaluates the argument Arrow by its own thread. Obviously, the original return type `b` cannot be returned, as the new Arrow cannot wait for the forked thread to complete. Furthermore, we have to regard the user state. `runX` evaluates the given Arrow starting with the empty state. Furthermore, `runX` does not allow to directly forward the input value of type `a` to the threaded arrow (because an `XmlTree` value is expected). So, inside `processA` a new Arrow is constructed, which ignores its `XmlTree` input value and adds a constant Arrow in front of the argument Arrow to provide the input value of type `a`. Another operation (`withOtherUserState :: s1 -> IOStateArrow s1 b c -> IOStateArrow s0 b c`) is used to issue the current `Context` state to the new Arrow.

Beside the `forkIO` operation, some more `IO` Monad values have to be encapsulated into new primitives. For example, monadic exception handling is problematic, too. While it is easy to transform the `catch` structure into an IO aware Arrow, the `catch` still requires an `IO` Monad value as argument (the value which may throw an exception). Using `runX` to transform Arrows to Monads for usage in the `catch` structure creates a mess of code. It seems to be more useful to create another generic operation of the following shape:

```
exceptZeroA :: (a -> IO b) -> JanusArrow s a b
```

This operation encapsulates a single `IO` Monad value and catches all exceptions thrown. To lift such an error condition upwards the invocation chain, the `zeroArrow` is utilized. There are specialized versions, for example taking an exception handler as an argument (which may even utilize the `Context` state for error propagation).

5.6. Server

5.6.1. Introduction

At this point, we have Shaders and Handlers implemented, but the system has no skeleton. As described in the abstract discussion, the server requires a central component to glue things together and do the management. This component is implemented as the `Server` module. Which tasks do this module and its operations have to fulfil?

- Maintaining a configuration file which defines server properties as well as Handlers and their pipelines.
- Loading the object code for the Handlers and Shaders utilized in the configuration.
- Configuring and assembling Handlers and Shaders. Assembling may include adding of functionality by means of Arrow composition (e.g. >>>).
- Creation of the initial `Context`.
- Starting of Handlers.

5.6.2. Configuration

It is an obvious necessity to provide a server configuration, i.e., by means of a file. Regarding Janus, this file shall not only contain configuration parameters, but describe the server structure in general. A major functionality of Janus is the dynamic, configuration based definition of Handlers and composition of Shaders.

So, Janus gets a configuration file. It is quite self-evident to utilize XML for this, as many system components (Shaders, Transactions, parts of the shared state, messages) already rely on XML. Integration of XML configuration data therefore imposes little efforts and interface additions to the system. As we will see more complex configuration throughout the examples in the following sections, we only introduce the basic elements of the Janus configuration file here without showing really useful files.

The Janus configuration starts with a root element named `janus`. Its children describe the general server configuration in a single element `config`, some kind of boot sequence in a single element `init` and each Handler installed by means of a `handler` element:

Example 11 - Configuration file skeleton

```
<?xml version="1.0" encoding="UTF-8"?>
<janus>
   <config/>
   <init/>
   <handler/>
   ...
</janus>
```

The content of the general server configuration may be arbitrary subtrees and attribute-value pairs. However, there is an important mandatory child of the `config` element, called `loader`. This element defines all Handlers and Shaders which shall be known to the system at runtime. The definition is done by a mapping of reference names (used throughout the configuration file when referring to the respective Shader/Handler) to function values (which are defined by a module name and a value name). For each Handler to be known, a single `handler` child is inserted into `loader`, analogous `shader` elements for Shader values. The mapping of references to module and value names is done by attributes:

Example 12 - The loader subtree

```
...
<loader>
   <handler reference="tcpHandler" name="tcpHandler"
      module="Network.Server.Janus.Handler.TCPHandler"/>
   <shader reference="local_echo" name="localEchoShader"
      module="Network.Server.Janus.Shader.ShaderLib"/>
   ...
</loader>
...
```

Please note two things:

First, the definition of a `handler` or `child` element in `loader` not only implies the creation of an according reference, but actually the loading of the described value into memory at server startup. This is even one of the first things the server does when started. Due to the lazy nature of Haskell one may expect a different behaviour, e.g. loading of object code in the moment of its first usage. As a disadvantage, such a lazy behaviour would mask erroneous Shaders if they are not used (which means that no incoming request forces their invocation, which could change at any time). The implemented solution enforces the loading of object code before any Handler is started and propagates errors arising. However, to prevent simple errors blocking the server startup, loader failures are reported but ignored (i.e., starting continues).

Second, the values denoted in `loader` are actually not of type Shader, but of type `ShaderCreator`. While a Shader is an Arrow transforming an XML tree into an XML tree, a `ShaderCreator` builds a Shader according to a given configuration. For example, a Handler might be installed at TCP port 80 or TCP port 3128. This difference is a configuration property forwarded to a `HandlerCreator`, which

constructs according Handlers. The configuration of a Handler or Shader value itself is immutable. However, the separation into values and their creators for both Handlers and Shaders will be introduced in more detail throughout the next section.

For now, we continue describing the configuration file elements. The next one is `init`, which may contain a list of `shader` definitions. A Shader definition is a subtree with a root element `shader`, some mandatory and optional attributes for the root element, an optional `config` child subtree and optional further `shader` subtrees. The `shader` element in this context actually is the central structure to define Shaders for the runtime system. It may look like:

Example 13 - Shader configuration element

```
<shader id="mime" type="init_mime">
  <config typefile="./conf/mime.types"/>
</shader>
```

First, we see some attributes at the root element. The `type`-attribute is mandatory; it contains a reference string as defined in the `loader` section and defines the `ShaderCreator` used to construct the according Shader. The `id` attribute is optional. If unspecified, a system-wide unique identifier is assigned at startup time. There exist two further optional attributes typically unused: `state` and `accepts`. Per default, both the `id` attribute and the `state` attribute (where the `state` attribute equals the `id` attribute) are added to the configuration provided to the `ShaderCreator`. Furthermore, an empty subtree with the `state` attribute as its name is added to the `local` scope of the Handler in question. A Shader may access the `state` attribute to get in possession of a state subtree exclusive to this instance created by the `ShaderCreator`. As said, per default the `state` and `id` attributes are equal. However, if a `state` attribute is provided, the subtree created is of this name. Therefore one can assign the same state to several instances of the same Shader or assign different states to several Shader instances with the same identifier. Last but not least the `accepts` attribute defines a list of Transaction states for which the Shader is invoked. For example, the `accepts` attribute allows a Shader to work on a Transaction of state "Failure". Per default, all Shaders are implicitely configured to only accept the state "Processing" to ensure that a failed Transaction simply falls through the pipeline until the end of it or an error handling Shader is reached (which, on the other hand, may be configured to ignore Transactions in state "Processing").

The `config` subtree of a Shader is optional and may contain arbitrary data understood by the `ShaderCreator` in question. Now let's come back to the `init` element. As said, it contains a sequence of `shader` elements. Semantics are that this sequence of Shaders is composed to a single Shader and afterwards invoked with an empty Transaction at startup time. By this, one can configure the startup environment (e.g., adding scopes or Channels) by means of Shaders as the `init` element gets processed before any Handler is started.

Finally we reach the sequence of `handler` elements. A `handler` element may look like:

Example 14 - A handler element

```
...
<handler id="tcp_80" type="tcpHandler">
   <config>
      <port>80</port>
   </config>
   <init>
      <shader id="shop" type="load">
         <config scope="local" to_state="/catalogue"
            file="./conf/shop2.db" _1="product" _2="price"/>
      </shader>
   </init>
   <shader type="trace_shader"><config>This is text</config></shader>
</handler>
...
```

It defines a Handler to get started on the server. The structure is quite similar to the structure of `shader` elements. As for Shaders, a type is required to select a `HandlerCreator` in the `loader` section by means of a reference string. The `id` attribute shares its properties with the Shader `id` attribute. Again, there is an optional `config` subtree with arbitrary data known to the utilized `HandlerCreator`. The optional `init` subtree shares its meaning with the Server's `init` subtree: A sequence of `shader` elements is used to build a single Shader invoked with the empty Transaction during the Handler's startup. In contrast to the Server's `init` section, the Handler's `init` section of course only affects the `Context` state as seen by this Handler. Finally the `handler` element contains a single `shader` element to describe its pipeline. This is constructed in the same way as each single `init` Shader.

After all, the Janus configuration incorporates a description of the server structure as well as Shader, Handler and Server level configuration data. The configuration file is

provided as an XML tree in the `Context` state, but its main purpose is to support the startup of the server.

5.6.3. Booting and Dynamic Loading

We already listed the missions of the `Server` module. Now we shall have a look how the configuration file is used to construct a running server. First of all we still lack the precise concept of `ShaderCreators` and `HandlerCreators`. As we said, a Shader value is not configurable by itself, everytime it is invoked it does the same. But we want to use the same Shader or Handler at different locations of the server with different behaviour. The simple example already mentioned is a Handler for TCP. There is only one TCP Handler source code, but we want to use it multiple times in our server for different ports. So what we actually define is not the Handler or Shader, but a constructor for a class of Handlers or Shaders based on a given configuration. The type for a `HandlerCreator` shall be:

```
type HandlerCreator = ConfigArrow -> Shader -> Handler
```

This type should be easy to understand: A `HandlerCreator` takes a configuration Arrow, a Shader and creates a Handler value. The configuration Arrow represents the `config` subtree of the according `handler` element. The Shader represents the Handler's pipeline, as generated from the `handler` element's `shader` subtree. To construct a Handler based on the respective `handler` element, the system has to acquire the `HandlerCreator` denoted by the `type` attribute, to create the Shader as well as the configuration Arrow and to invoke the `HandlerCreator` with them. The `init` element and its Shader representation is simply put in front of the Handler value (as both are Arrows of compatible type and can be composed) and is therefore executed before the actual Handler. What we miss is the construction of the Shader values as required for the `shader` and `init` elements. The `ShaderCreator` looks like:

Figure 34 - The ShaderCreator type

```
type ShaderCreator = ConfigArrow -> Associations -> Shader
data Associations  = Assoc ([Shader], Map String Shader)
```

The `ConfigArrow` value again corresponds to the `config` subtree of the `shader` element, where the `id` and `state` attributes got added. The `Associations` value

delivers a way to model nested Shaders. If there are `shader` subtrees inside a `shader` definition like in

Example 15 - Subshader definitions

```
<shader id="mime" type="init_mime">
   <config typefile="./conf/mime.types"/>
   <shader type="shader1"/>
   <shader type="shader2"/>
</shader>,
```

the subshaders are constructed and forwarded to the `shaderCreator` of the parent Shader. The `Associations` value contains these immediate subshaders. The according data type provides two representations for these: A list expressing the order of subshaders and a map allowing to efficiently access subshaders based on their identifiers. This implies that system behaviour is undefined if the same identifiers are used for Shaders with the same father node. Such a situation may only occur by programmer intervention (by defining `id` attributes) as the system automatically assigns unique identifiers. Access to the Associations is restricted to a set of interface functions hiding the internal representation, but providing exactly the functionality stated: Efficient access based on identifiers and a way to express the order of Shaders (the order of definition in the configuration tree).

Now we have to glue things together:

- First of all, the server loads the configuration by means of the HXT parser.
- The empty `Context` server configuration is replaced by the loaded XML representation.
- Besides, some general changes are done to the `Context` state, for example default states and Channels ("global", "local", ...) are created.
- The `loader` tree is processed. For each `shader` and `handler` subelement, a dynamic loader is invoked to get the according Creator value. These values are stored in Repositories (which will be explained soon).
- The Shader repository is added to the `Context` state (to be precise, the existing empty repository is replaced).

- The `init` tree of the server is processed, i.e., the according Shader value is constructed and invoked with an empty Transaction.
- For each `handler` subtree, the according Handler value is created. This is done by constructing the according Shader value and using the `HandlerCreator`, the Shader value and the `config` subtree to build the Handler value.
- To get a `HandlerCreator`, the according repository is forwarded to the creation Arrow. For Shaders, the `ShaderCreator` can be taken from the `Context` state where the respective repository has been loaded to.
- Each Handler value gets enhanced by combining it with its `init` Shader.
- Each Shader value gets enhanced by combining it with code based on the `accepts` attribute. Afterwards, the constructed Shader value only processes Transactions in states listed in the `accepts` attribute.
- Each Handler value gets started by means of the `processA` primitive.

To complete this section, we shall have a look at the `Repository` type. This type not only provides an encapsulation for a set of values, it actually incorporates the dynamic loading functionality. Beside functions to get, add or remove values of the stored type (in our case, `HandlerCreator` and `ShaderCreator`), there is a function to add a value based on a module name and an object name. Furthermore, a key is provided for later lookup of the stored value. This key corresponds to the references used in the `loader` configuration tree:

```
loadComponent :: Repository a -> String -> ModuleName -> ObjectName
                 -> IO (Maybe (Repository a))
```

The function takes a repository, the key for the new value, the module name and the object name of the new value and eventually returns a new repository with the value added. The loading is based on the hs-plugins library.

5.7. Janus Shader Servlets

After all, we completed the introduction of the requirements for Janus, its concept, major parts of its implementation and basic facilities to build servers. Now, what can we do with it? First of all one may recognize that there is no strict distinction

between Servlets and some system components. A web application programmer can add `ShaderCreators` with their respective classes of Shaders. And a system programmer can add `ShaderCreators` to do general request transformation, for example parsing HTTP Requests. A concrete server will be comprised of system Shaders doing reusable and low-level request transformations (as a request starts with being a plaintext node) and embedded user-defined Shaders which, for example, represent a business web application unaware of a request's technical details (or at least some of its technical details). These are two completely different purposes, which got mapped onto the same fundamental concept. For Janus, one may even consider the parsing of HTTP, the computation of a static HTML file's path in the file system, etc. to be system level web applications in a technical sense (where web has to be disconnected from its TCP/IP-HTTP meaning).

Inside a Shader one can use HXT to process the Transaction tree and one can access the `Context` state by means of the aforementioned interfaces as well as the messaging facility. Of course, Haskell library functions (including I/O) can be used. And the Arrows extensions provided by HXT, especially `ArrowIf`, and its implementations of `ArrowZero` and `ArrowPlus` provide both convenient and general structures to combine Arrows. These combinations might be control structures, Arrows throwing errors and so on. Some helper functions already shown allow catching the `zeroArrow` and mapping it onto messages. Furthermore, they allow to do exception handling and multithreading directly with Arrow primitives. Well, now let's discuss a simple example Shader, where we first present the source and afterwards explain its meaning and introduce new interfaces used. It has to be emphasized again that a programmer actually defines `ShaderCreators` instead of Shaders, where a `ShaderCreator` can be compared to a Java Servlet class and an according Shader (build regarding a configuration) resembles an instance of such a Servlet class. However, we typically say that a programmer is defining a Shader.

Example 16 - Demo-Shader to display a text

```
demoShader :: ShaderCreator
demoShader =
   mkCreator $ \config _ ->
      (proc in_ta -> do
         text <- config >>> getVal "/config/@text"
               <+!> ("demoShader", ValueNotFound, "No text found.", [])  -< ()
         "global" <-@ mkSimpleLog "demoShader" ("demoShader found text: " ++ text)
               l_info -<< ()
         let html_text = "<html><header><title>Demo</title></header><body><h1>"
```

```
            ++ text ++ "</h1></body></html>"
        setVal "/transaction/http/response/body" html_text -< in_ta
    )
```

This small Shader displays an HTML page with a configuration-supplied H1-Tag. We could configure it with

```
<shader type="demoShader"> <config text="MyText"/> </shader>
```

in the configuration file to let it display "MyText" in the internet browser (if embedded into an HTTP server). How is it done? First, there is the `mkCreator` function. This function takes a `ShaderCreator` (with the arguments for its configuration and its Associations) and enhances it, i.e. it adds error handling. This error handling has been explained when discussing the `Context`: It catches a `zeroArrow`, empties the `Context` message buffer, inserts these messages into the Transaction and continues execution (with the Transaction possibly in Failure state). The Shader itself is represented by the `proc in_ta -> ...` do-notation Arrow. It reads the XML attribute `/config/@text` from the Shader configuration and binds it to `text`. Please note that the configuration is represented by an Arrow delivering `XmlTree`, and `getVal` builds an Arrow taking `XmlTree` and delivering a string. Therefore `config` and `getVal` can be composed immediately by the `>>>` operator. In the do-notation, we here have an input value `()` for the according Arrow. As `config` is independent of its input, the input value is actually insignificant. Last but not least the `<+!>` operator is used to generate an error message if `config >>> getval` fails. The error is stored in the `Context` message buffer to be inserted into the Transaction by the `mkCreator` environment. There exists a similar operator `<!>` with the same semantics, but operating on the `Context` and sending the message to a Channel.

The next line shows a simple debug message, sent to the global Channel on `l_info` level by means of the `<-@` operator. Afterwards, the HTML text is constructed by a let-clause and finally stored in the HTTP Response body. Please note that in contrast to the first line, this last Arrow receives `in_ta`, the input Transaction tree. `setVal` therefore operates on the Transaction, the resulting Transaction is the whole Shader's result.

We postpone examples for the actual structure of Transactions to our examples in the next chapter. For now we introduce some additional helper functionality already partially used in the Shader above. As we already mentioned, a subset of XPath expressions is utilized to access the state trees. So it seems to make sense to

extend this paradigma to Transaction handling, i.e., use XPath expressions to parameterize operations over the Transaction tree (or other XML trees, of course). If we think of XPath expressions as configuration data for Shaders, the unified usage of XPath may improve interoperability between state and Transaction. HXT provides operations based on XPath, but these are of general manner and would make code redundant and hard to read if used frequently. Therefore we introduce a set of XML helper functions. These include some functions already known, for example the primitives to create multithreaded Arrows (`processA`) and to do exception handling. More important, there are operations to do typical XML operations in an easy fashion:

Figure 35 - Transaction XML Arrows

```
getVal          :: String -> XmlAccess s String
getValDef       :: String -> String -> XmlAccess s String
getValP         :: (Read a, Show a) => String -> XmlAccess s a
listVals        :: String -> XmlAccess s String
listValPairs    :: String -> XmlAccess s (String, String)
setVal          :: String -> String -> XmlTransform s
setValP         :: (Read a, Show a) => String -> a -> XmlTransform s
delVal          :: String -> XmlTransform s
swapVal         :: String -> String -> XmlAccess s (XmlTree, String)
getTree         :: String -> XmlTransform s
insTree         :: String -> XmlTransform s -> XmlTransform s
insEmptyTree    :: String -> XmlTransform s
addTree         :: String -> XmlTransform s -> XmlTransform s
delTree         :: String -> XmlTransform s
```

The `Tree` operations work on whole subtrees, whereas the `Val` operations work on string values only. The interface tries to resemble the according functions to handle the shared mutable state. For example, there are some polymorphic versions to use `Show` and `Read` class instances to store and retrieve arbitrary typed values. There are functions to list the names of a set of nodes, with a special version to directly include the respective string values. Of these we just saw `setVal` and `getVal`.

As already mentioned XPath expressions for the state and for the functions above are not equivalent in expressiveness. The state XPath expressions do not support attributes or predicates, they are primitive location paths. The XPath expressions for the general XML access functions above are restricted as well, but less strong. Attributes are allowed, of course, as predicates are too. Furthermore, the `*` character may be used to express a selection of all children. For the `listVals`

functions, the `*` is even mandatory: `*` and `@*` are used to separate the selection of all children from the selection of all attributes.

To illustrate the similarities between tree and state access, we extend our Shader above. Now it reads an XPath value from its configuration to denote a state value, which holds the text to display:

Example 17 - Enhanced Demo-Shader to display a text

```
demoShader :: ShaderCreator
demoShader =
   mkCreator $ \config _ ->
      (proc in_ta -> do
         node <- config >>> getVal "/config/@node"
               <+!> ("demoShader", ValueNotFound, "No node found.", [])  -< ()
         text <- "global" !-> ("/text/" ++ node)
         "global" <-@ mkSimpleLog "demoShader" ("demoShader found text: " ++ text)
               l_info -<< ()
         let html_text = "<html><header><title>Demo</title></header><body><h1>"
               ++ text ++ "</h1></body></html>"
         setVal "/transaction/http/response/body" html_text -< in_ta
      )
```

We easily recognize the additional line binding a `Context` state value `/text/<node>` to `text`, where node represents the node loaded from the configuration. We utilize the `!->` operator, which expects the scope in question on the left side and the node XPath on the right side.

To conclude this section about Shader programming, we do some remarks about a small extension library of Janus, `HTMLBuilder`. The aim of this library is to provide a set of Arrows to abstractly construct HTML strings. Please note a representative selection of interface functions:

Figure 36 - Excerpt from HTMLBuilder

```
html2Str    :: XmlAccess s String
(+>>)       :: XmlTransform s -> [XmlTransform s] -> XmlTransform s
html        :: XmlSource s a
headers     :: XmlTransform s
title       :: String -> XmlTransform s
htmlbody    :: XmlTransform s
link        :: String -> String -> XmlTransform s
heading     :: Int -> String -> XmlTransform s
form        :: String -> String -> XmlTransform s
formHidden  :: String -> String -> XmlTransform s
```

We demonstrate the usage of these functions by coding the HTML in our recent Shader example by means of `HTMLBuilder`:

Example 18 - Demo-Shader with HTMLBuilder primitives

```
demoShader :: ShaderCreator
demoShader =
   mkCreator $ \config _ ->
      (proc in_ta -> do
         text <- config >>> getVal "/config/@text"
               <+!> ("demoShader", ValueNotFound, "No text found.", [])  -< ()
         "global" <-@ mkSimpleLog "demoShader" ("demoShader found text: " ++ text)
               l_info -<< ()
         html_text <- html2Str -< (body text)
         setVal "/transaction/http/response/body" html_text -< in_ta
      )
   where
      body text =
         html
            +>> [ headers
                     += title "Demo"
                  , htmlbody
                     += heading 1 text
                ]
```

The coding is done in the where-clause. As we see in the interface and in the code sample, HTML is represented here by XML (we are actually talking about XHTML **[12]**). Each Arrow of the `HTMLBuilder` interface delivers an `XmlTree` denoting a subtree of the later XHTML document. For example, `heading 1 "test"` delivers on XML tree with a root node called `h1` and a child text node `test`. By means of the HXT `+=` operator, one can insert a subtree into the current node. In `headers += title "Demo"`, a node named `title` with a child text node `Demo` is inserted into a tree with root node `headers`. This way, a XHTML document can be composed recursively. `HTMLBuilder` adds another operator: `+>>` extends `+=` to lists, i.e. all trees denoted by the list in the where-clause are in orderly inserted into the XHTML root node created by `html`. Finally, `html2Str` is utilized to render the XHTML tree constructed to a string.

In this example, the advantages of `HTMLBuilder` are not obvious. Although one might format the where-clause in a more concise (and less readable) way like the HTML text concatenation above was formatted, there is no advantage here in terms of code length. However, the usage of an abstract interface provides better readability and eases later changes (with text concatenation, changes to the element structure

are cumbersome and error-prone). Furthermore, the `HTMLBuilder` interface simplifies the recursive generation of complex XHTML documents and can be used to define additional and more complex XHTML generation primitives. To conclude, changes to the internal XHTML translation can be implemented in the `HTMLBuilder` library transparently for existing code.

6. Examples

6.1. Introduction

At this point, we summarize the preceding chapters about the Janus design and the implementation of its core components by traversing the facilities seen:

- An implementation for Shaders, which is based on HXT Arrows to transform XML into XML, where the XML trees represent our notion of Transactions.
- The HXT Arrows throughout the system are parameterized with a Janus specific state type, the `Context`.
- The `Context` type incorporates a shared mutable user state, represented by rose trees, which supports timestamp tests for both structure and data. There exists synchronisation over these trees to ensure thread safety. There is support for listener threads. To improve flexibility and modularity, there is a set of scopes, where each one represents its own shared state. Each scope can be replaced for a single thread to enforce locality of state changes.
- The `Context` type incorporates a messaging mechanism, which unifies errors, logging and general messages. The mechanism is based on the concept of Channels, which are comprised of a Handler for new messages, a buffer for messages and a listening facility. As with the shared mutable states, a set of Channels is supported to allow for thread-exclusive Channels.
- The `Context` type provides access to the server configuration and dynamically loaded Shader objects.
- The `Context` type provides a facility called “shadow" to allow the construction of optimized Shaders during evaluation.
- There exist Handlers to wrap services around Shaders. Typically these are network services.
- There exist helper functions to ease access of state and Transaction nodes by means of XPath.

In this chapter, we will show some examples how to configure a Janus server. The examples start with a simple one and stepwise add functionality. By the way, we will add general Shaders to the system to improve expressiveness of the configuration file and modularity of user-defined Shaders. To remember, please note the empty configuration file:

Example 19 - Empty configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<janus>
   <config>
      <loader/>
   </config>
   <init/>
</janus>
```

This configuration is valid, but defines no `HandlerCreator` or `ShaderCreator`. Consequently, they are no concrete Handlers defined and the boot sequence (`init` element) is empty as well.

6.2. A monolithic static webserver

As a first step, we now add static web server functionality to this Janus server.

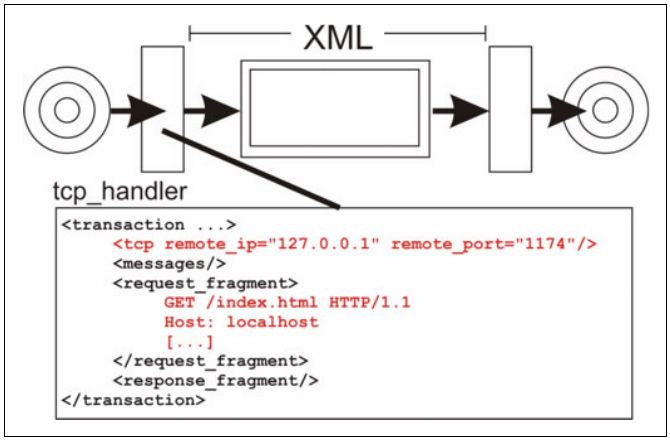
Example 20 - A monolithic webserver configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<janus>
   <config>
      <loader>
         <handler reference="tcpHandler" name="tcpHandler" module="..."/>
         <shader reference="static_webserver" name="static_webserver" module="..."/>
      </loader>
   </config>
   <init/>
   <handler id="tcp_80" type="tcpHandler">
      <config><port>80</port></config>
      <init/>
      <shader type="static_webserver">
         <config base_url="/" maps_to="/usr/srv/www"/>
      </shader>
   </handler>
</janus>
```

`tcpHandler` is a Handler basically following the hull of HWS-WP. It is configured with a port (in the example it is 80, the standard HTTP port), acquires a socket on this port and waits for a request. If one approaches a thread is forked to process it.

Afterwards the server continues waiting for further requests. The freshly arrived one is first transformed into an initial Transaction like this:

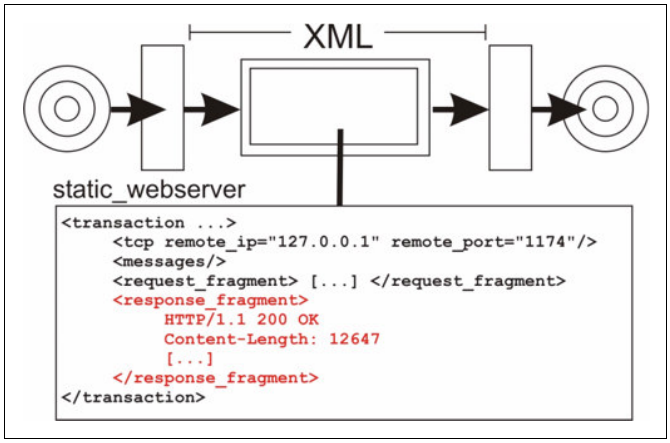
Example 21 - An initial HTTP Request Transaction



You can find the plaintext HTTP Request in the `request_fragment` node. Parts of the request have been left out because they are insignificant for this example. Next the Transaction is set to state "Processing" (defined by an attribute `/transaction/@transaction_state`, which has been omitted in the example) and forwarded to the Shader.

The `static_webserver` Shader concentrates all transformation work to serve a request. It is configured with a base URI, where the Shader has its root, and a file system location, where this root maps to. The Shader's work include parsing HTTP based on the delivered `request_fragment`, interpreting the HTTP Request to locate the requested file, loading the file, inferring some HTTP meta data (for example, the status code) and finally creating the HTTP Response message in the Transaction `response_fragment`:

Example 22 - A final HTTP Request Transaction



You can see the HTTP Response in the `response_fragment` node; the file content itself has been omitted. At last, the plaintext `response_fragment` is transmitted back to the client by the TCP Handler and the Transaction is completed.

6.3. A modular static webserver

The Shader incorporated in the first example is a monolithic one. If there is a job inside the Shader which could be utilized for another service, it would not be possible to install it there by means of configuration. So, our second example concentrates on splitting up this Shader into useful and closed subtasks. To identify these subtasks, we go back to the first Shader's explanation and introduce Shaders for: 1. HTTP parsing, 2. locating and loading the file requested, 3. inferring meta data, 4. creating the response. As an interface among these Shaders we utilize a new Transaction subtree: `/transaction/http`.

Example 23 - A modular webserver configuration

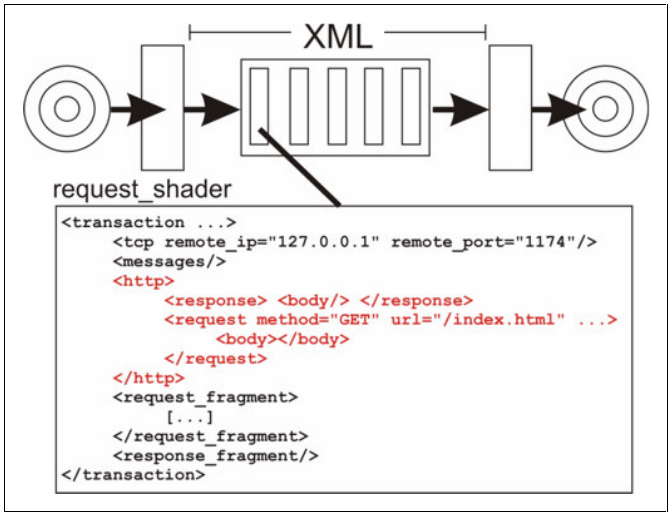
```
<?xml version="1.0" encoding="UTF-8"?>
<janus>
   <config>
      <loader>
         <handler reference="tcpHandler" name="tcpHandler" module="..."/>
         <shader reference="request_shader" name="requestShader" module="..."/>
         <shader reference="file_shader" name="fileShader" module="..."/>
         <shader reference="http_status_shader" name="httpStatusShader"
            module="..."/>
         <shader reference="html_status_shader" name="htmlStatusShader"
            module="..."/>
         <shader reference="response_shader" name="responseShader" module="..."/>
         <shader reference="sequence" name="sequence" module="..."/>
      </loader>
   </config>
   <init/>
   <handler id="tcp_80" type="tcpHandler">
      <config><port>80</port></config>
      <init/>
      <shader type="sequence">
         <shader type="request_shader"/>
         <shader type="file_shader">
            <config base_url="/" maps_to="/usr/srv/www"/>
         </shader>
         <shader type="http_status_shader" accepts="[Processing, Failure]"/>
         <shader type="html_status_shader">
            <config page_404="./conf/404.html"/>
         </shader>
```

```
        <shader type="response_shader"/>
      </shader>
    </handler>
</janus>
```

Regarding the loader, there is little to add. We easily recognize the new `ShaderCreators`, their effect on Transactions will be explained soon. However, there is a new `ShaderCreator` called `sequence`. When discussing the Handler implementation, we mentioned that a Handler is actually configured with a single `shader` subelement. As we now use five Shaders to define the service, we need to compose these five to a single Shader. In terms of programming languages, we need a block concept or a sequence control structure. This is achieved with the `sequence ShaderCreator`. This `ShaderCreator` composes the Shader values delivered as Associations to a new single Shader, which evaluates the original Shaders in the order stated in the configuration file. We would like to define `sequence` to be our first `ShaderCreator` participating in an informal special class, the control structure Shaders. Members of this class typically concentrate on the combining of Shaders to new Shaders while having no own effect on Transactions passing by. They can be considered to be Shaders of higher order as they operate on other Shaders.

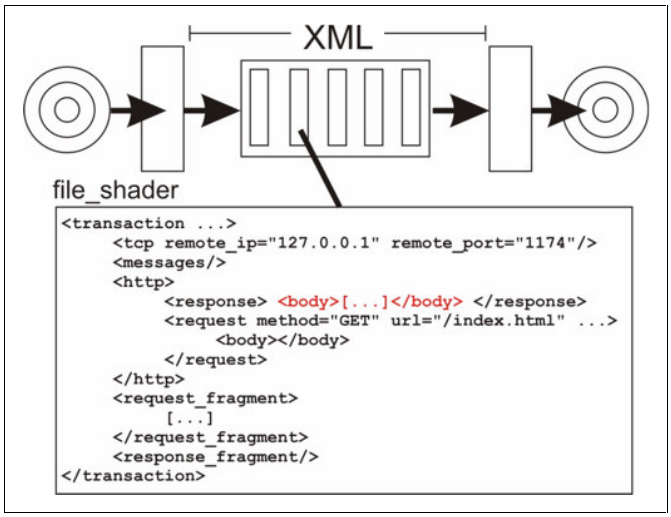
Now let's start with processing the request, which at first is identical to the initial one in the previous example. This Transaction arrives at the `request_shader`, which parses the HTTP included and generates a new Transaction subtree describing the request:

Example 24 - A Transaction after the HTTP Request Shader



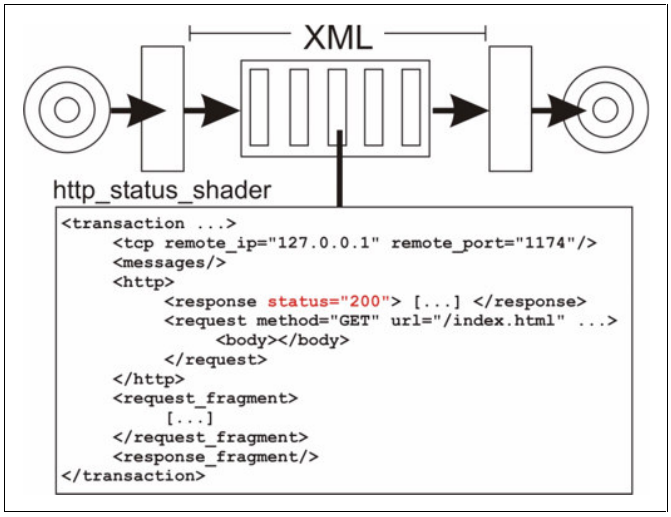
There exists an `http` subtree in the Transaction, describing the request. For now, the method and the URI components are defined and there is a `body` subtree, but it is empty for a GET request, of course. Furthermore, an empty `response` subtree has been created to be filled with response meta data. Next, the `file_shader` starts working. Its purpose is to infer the filename by means of its configuration and the URI requested. This information can be taken from the Shader configuration and the `/transaction/http/request` subtree. As a result, the Shader loads the file in question and put its plaintext into the response body.

Example 25 - The HTTP Response body has been inserted by the File Shader



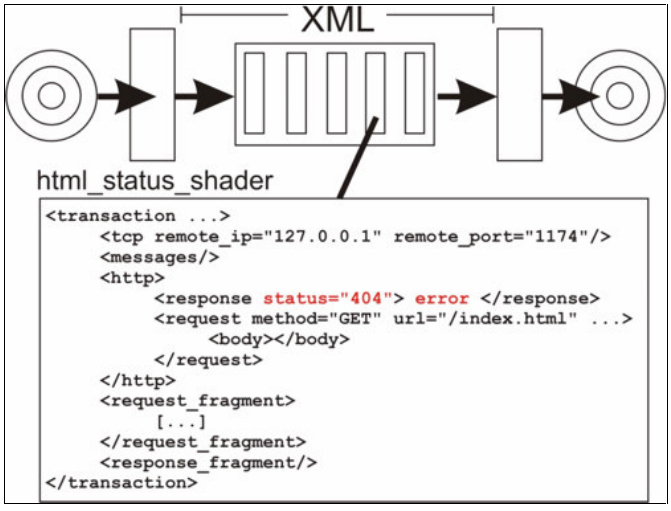
The Transaction continues in the `http_status_shader`, which has an `accepts` attribute adding the Transaction Failure-state to the Shader's workspace. A very simple implementation can completely rely on this Transaction state. If the `file_shader` recognizes an error condition (for example, a requested file does not exist), it uses the respective interface to send an error message and to terminate with the `zeroArrow`. The environment transforms the `zeroArrow` into the Failure state for the Transaction and continues operation. In the `http_status_shader`, the state is checked and a status code is set in the `/transaction/http/response` subtree: For state "Processing" 200 is set, for state "Failure" 404 is set. In the "Failure" case the Transaction state is set back to "Processing", as the error condition now is considered to have been handled. Simple extensions may provide more useful status codes, for example the messages in the `Context` state or in the Transaction can be considered. For our example we assume the file requested to exist, resulting in the following Transaction after the `http_status_shader`:

Example 26 - A Transaction after the HTTP Status Shader



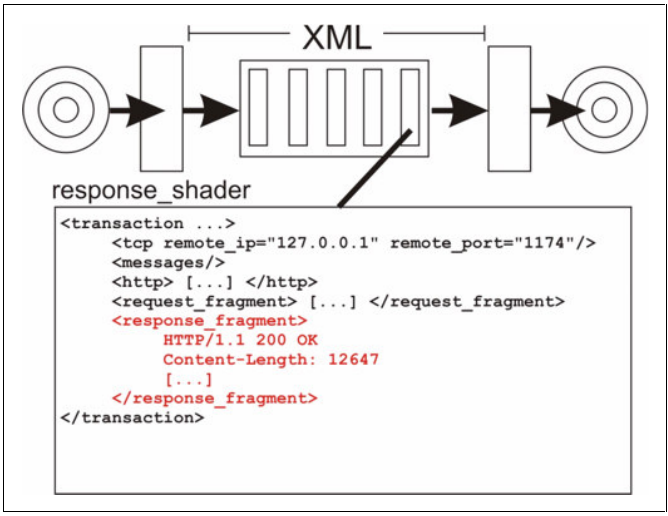
The `html_status_shader` as the one but last Shader involved is a simple one, it reads the status code inferred and, in case of an error code, puts an appropriate message document into the `/transaction/http/response/body` node. The document relation (code to file) is based on the Shader's configuration, as one can intuitively recognize in the configuration above for status code 404.

Example 27 - A Transaction after the HTML Status Shader



Last but not least, the Transaction arrives in the `response_shader`, which uses the `/transaction/http/response` subtree to generate a plaintext HTTP response:

Example 28 - A Transaction after the HTTP Response Shader



The result matches the previous example's one, except the meta data used to interface the Shaders. We have seen several important aspects of the Janus concept, especially the configuration of components, the composition of Shaders to a service, the division of a large task into a set of smaller subtasks and the idea of forwarding data to other Shaders by XML. Moreover, the benefits of such a modularization are obvious in the example, as the `request_shader` and `response_shader` components can be expected to occur in any HTTP based service. The same can be expected for status code and error page generation, although these might be replaced by specialized implementations in certain cases.

6.4. Adding of query parameter processing

To continue our examples, where we just divided an existing solution into a modular solution, we now extend the service functionality by adding an additional Shader. This one parses and interprets the provided URI's query part and forwards the results by means of an addition to the Transaction tree:

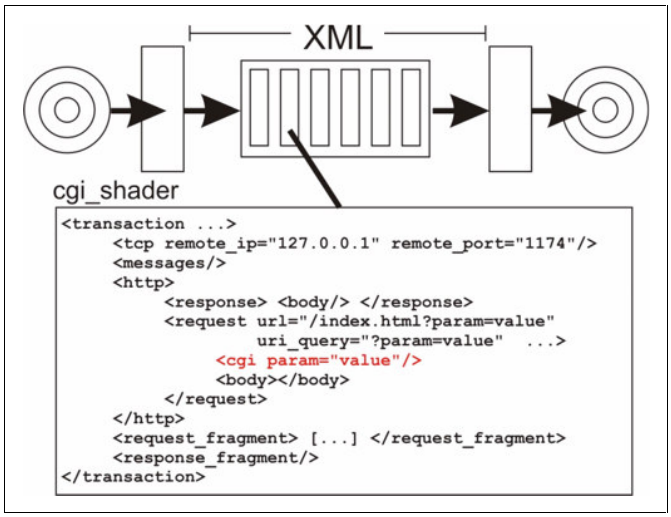
Example 29 - Configuring a query parameter Shader

```
<?xml version="1.0" encoding="UTF-8"?>
<janus>
   <config>
      <loader>
         [...]
         <shader reference="cgi_shader" name="cgiShader" module="..."/>
         [...]
      </loader>
   </config>
```

```
    <init/>
    <handler id="tcp_80" type="tcpHandler">
       <config><port>80</port></config>
       <init/>
       <shader type="sequence">
          <shader type="request_shader"/>
          <shader type="cgi_shader"/>
          <shader type="file_shader">
             <config base_url="/" maps_to="/usr/srv/www"/>
          </shader>
          <shader type="http_status_shader" accepts="[Processing, Failure]"/>
          <shader type="html_status_shader">
             <config page_404="./conf/404.html"/>
          </shader>
          <shader type="response_shader"/>
       </shader>
    </handler>
 </janus>
```

The extension is achieved by the `cgi_shader` `ServletCreator`, which was named this way because CGI is the historically most important field of application for the URI query part (or search path, as it was called for URLs). It is instantiated directly after the `request_shader`, however it may be placed anywhere in the service pipeline as long as it follows the `request_shader` and precedes the first Shader relying on query parameters. Considering the request URI /index.html?param=value, we achieve the following Transaction after the `cgi_shader` stage:

Example 30 - A Transaction after the query parameter Shader



Here one can see the extracted name-value pair param=value added as an attribute to a new subtree `/transaction/http/request/cgi`. So, any Shader interested in the request parameters may access the list denoted by

`/transaction/http/request/cgi/@*`. There is nothing else to add as the service continues operation in the same way as in the previous example, no further interpretation of the parameters occurs.

6.5. Adding of MIME type processing

After showing a simple extension to the service pipeline (again a very reusable one), we now do the same for MIME types. A task for a static web server is to compute the type of a document to deliver and express this type based on a standardized relation. This type is called a MIME type and is expressed by a simple string, for example text/html denotes an HTML file. However, this extension is more interesting as we want to utilize a table to compute the MIME type based on a file's extension. This table shall be loaded from a file and stored in the shared mutable state to be used by every respective Shader instance. Let's have a look at the new configuration file:

Example 31 - Configuring a MIME type Shader

```
<?xml version="1.0" encoding="UTF-8"?>
<janus>
   <config>
      <loader>
         [...]
         <shader reference="mime_shader" name="mimeShader" module="..."/>
         <shader reference="init_mime" name="initMimeDB" module="..."/>
         [...]
      </loader>
   </config>
   <init>
      <shader type="init_mime"><config typefile="./conf/mime.types"/></shader>
   </init>
   <handler id="tcp_80" type="tcpHandler">
      <config><port>80</port></config>
      <init/>
      <shader type="sequence">
         [...]
         <shader type="file_shader">
            <config base_url="/" maps_to="/usr/srv/www"/>
         </shader>
         <shader type="mime_shader"/>
         [...]
      </shader>
   </handler>
</janus>
```

We already have practice in locating the changes. There is a new `ShaderCreator mime_shader` and an according instance in our service pipeline. But moreover, there is a new `ShaderCreator init_mime`, instantiated in the server level `init` element. Although irrelevant for the current example, please note that the `init` element may contain a list of `shader` elements; a `sequence` Shader would not be required here. Now, `init_mime` denotes a Shader being configured with a file (in a well defined format, of course). When evaluated, the Shader loads this file and puts a name-value pair into the `Context` state for each connection included (for example, .html -> text/html). The respective state component is the `/mime-node` in the “global” scope. As the global scope may be accessed by any Handler and is never replaced, the `init_mime` instance could also have been placed into a Handler's `init` sequence. However, semantically it should be put into the server's `init` sequence as it actually affects all Handlers. To finish this example, we have a look at a Transaction after the new mime_shader:

Example 32 - A Transaction after the MIME type Shader

```
<transaction transaction_id="1" transaction_state="Processing" handler="TCPHandler"
      start="1157647830796">
   <http>
      <response mime="text/html">
         <body/>
      </response>
      <request url="/index.html" uri_scheme="" uri_path="/index.html" uri_query=""
            uri_frag="" method="GET">
         <cgi/>
         <body></body>
      </request>
   </http>
   <tcp remote_ip="127.0.0.1" remote_port="1174"/>
   <messages/>
   <request_fragment>
      [...]
   </request_fragment>
   <response_fragment>
      [...]
   </response_fragment>
</transaction>
```

6.6. A simple web application

With query parameters and MIME types added to our server, we now leave the pure static webserver and introduce our first web application. A very small application,

primarily intended to show a general structure for Janus web applications. It is a navigation tool for the `Context` state, where one can select from a list of available scopes and afterwards move in the state tree by clicking on nodes. The functionality is split up into two Shaders, one to display the front page with the list of available scopes (`browser_door`) and one to display a state tree (`browser_scope`). As usual, we first show the (quite different) configuration tree and then discuss the changes:

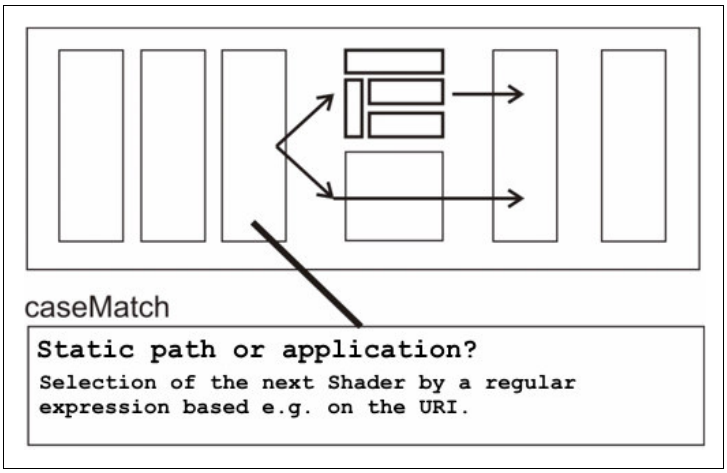
Example 33 - Configuring a web application

```
<?xml version="1.0" encoding="UTF-8"?>
<janus>
   <config>
      <loader>
         [...]
         <shader reference="browser_door" name="doorShader" module="..."/>
         <shader reference="browser_scope" name="scopeShader" module="..."/>
         <shader reference="sequence" name="sequence" module="..."/>
         <shader reference="caseMatch" name="caseMatchControl" module="..."/>
         <shader reference="neg" name="negPredicate" module="..."/>
         <shader reference="taExpr" name="taExpr" module="..."/>
         <shader reference="valExists" name="taPredicate" module="..."/>
      </loader>
   </config>
   <init>
      <shader type="init_mime"><config typefile="./conf/mime.types"/></shader>
   </init>
   <handler id="tcp_80" type="tcpHandler">
      <config><port>80</port></config>
      <init/>
      <shader type="sequence">
         <shader type="request_shader"/>
         <shader type="cgi_shader"/>

         <shader type="caseMatch">
            <shader id="value" type="taExpr">
               <config path="/transaction/http/request/@url"/>
            </shader>
            <shader id=".*/browser.*" type="sequence">
               <shader type="while">
                  <shader id="predicate" type="neg">
                     <shader id="predicate" type="valExists">
                        <config path="/transaction/http/response/body"/>
                     </shader>
                  </shader>
                  <shader id="body" type="caseMatch">
                     <shader id="value" type="taExpr">
                        <config path="/transaction/http/request/cgi/@operation"/>
                     </shader>
```

```
                    <shader id="default" type="browser_door"/>
                    <shader id="init" type="browser_door"/>
                    <shader id="browse" type="browser_scope"/>
                 </shader>
              </shader>
           </shader>
           <shader id="default" type="sequence">
              <shader type="file_shader">
                 <config base_url="/" maps_to="/usr/srv/www"/>
              </shader>
              <shader type="mime_shader"/>
           </shader>
        </shader>

        <shader type="http_status_shader" accepts="[Processing, Failure]"/>
        <shader type="html_status_shader">
           <config page_404="./conf/404.html"/>
        </shader>
        <shader type="response_shader"/>
     </shader>
  </handler>
</janus>
```

The two new `ShaderCreators` for the application can be found in the loader, beside a set of further new `ShaderCreators`: `caseMatch`, `neg`, `taExpr`, `valExists`. Their meaning will be explained soon. Before that, let's have a look at the service pipeline. Beginning and end stayed unchanged, whereas the `file_shader` and `mime_shader` have been replaced by a more complex structure. To start, the idea is to invoke the web application based on the URI, where each URI not selecting the web application shall still be handled by the static server path. This is achieved by a new control structure Shader, the `caseMatch`.

Example 34 - A caseMatch structure



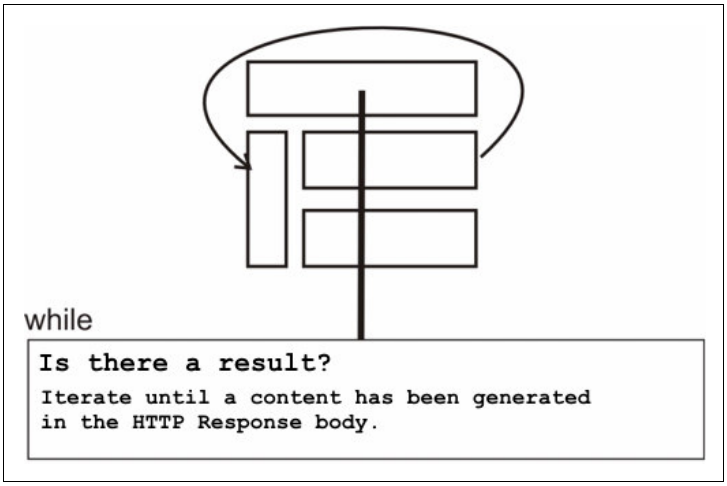
This Shader expects a subshader (via Associations) named `value`, which somehow delivers a value `x`. Given this value `x`, the `caseMatch` Shader tries to match `x` against its other subshaders' identifiers based on regular expressions (the subshader expressions are considered to be regular expressions). Order is significant, so if multiple identifiers match `x`, the first one is selected. If no identifier matches, the `default` subshader, if present, is selected. The `caseMatch` Shader can be configured to utilize other names than "value" and "default" to avoid ambiguities.

The `value` Shader in the example is of type `taExpr`, which is a new one too. As the name suggests, it can be considered to be a so called expression Shader (in contrast to regular Shaders and control structure Shaders). An expression Shader shall take a Transaction tree as usual, but delivers a tree with a root node containing a string value. A `taExpr` Shader deliveres an XPath selected part of the current Transaction. In the example `/transaction/http/request/@url` is delivered, containing the URI identified by the `request_shader`. So the structure based on `caseMatch` and `taExpr` selects a subshader based on the request URI. For any URI containing `/browser`, the subshader ".*/browser.*" with the web application is selected, otherwise the default subshader with the static path.

When analysing the web application, we immediately encounter another new Shader type, the `while` Shader. The reader might have an intuition what this Shader does, but we first provide an idea how the web application works. As already stated, the application consists of multiple Shaders or Servlets, so we obviously have to select one to serve the current request. To do so, we use a new state called "operation" encoded into the URI as a parameter. If this parameter misses (which will typically be the case for the first request by a client), a default is used (pointing to the door page, of course). To complete our idea, we consider another aspect: Does every request have to be fulfilled by a single Servlet? Or in other words: Does every Servlet deliver a user interface result, for example an HTML document in the HTTP Response body? Obviously, the answer "yes" would be very restrictive. Therefore, we consider Shaders causing some side effects (for example, on the shared mutable state or the Transaction), but delivering no final result and leaving further work to other Shaders. A possible approach to model this is an iteration, a loop. And this is what the `while` Shader does. It is our second member of the control structure Shader class and represents a while loop, repeating the `body` subshader

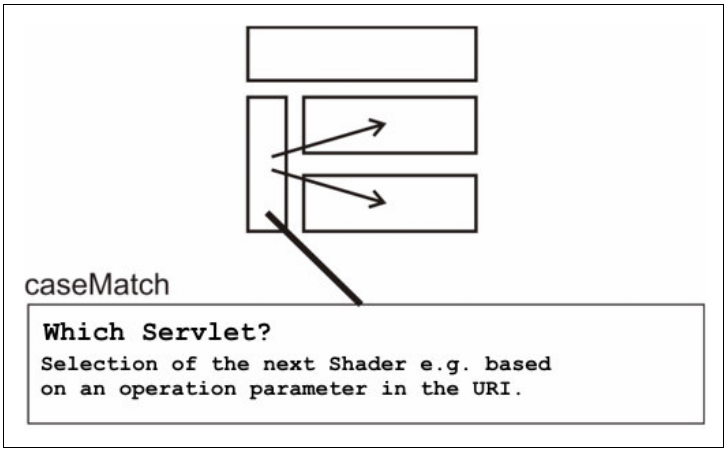
as long as the `predicate` subshader holds (where the predicate is evaluated before the body is, as common for while loops). The predicate Shaders are the third class of special Shaders, which deliver the input Transaction if successful and fail if not successful. As in other languages, predicates can be considered to be Boolean expressions, but there is a difference: A Boolean expression Shader might exist and deliver a Boolean value (in string representation), whereas our predicate Shaders actually fail to represent the Boolean false (which is a quite convenient way to express the Boolean false when using Arrows).

Example 35 - A while loop



The expression Shader used is `valExists`, which checks the existence of a subtree or attribute in the Transaction based on an XPath expression. To be precise, in our example `valExists` is wrapped by another predicate Shader, `neg`. `neg` inverts its argument predicate, delivering True for False and vice versa. Altogether, the predicate defined succeeds if the node `/transaction/http/response/body` does not contain a value in the Transaction. So, the `while` Shader loops as long as no HTTP body has been specified by any Shader in the iteration. The loop body is comprised by a `caseMatch`, performing the aforementioned selection based on the operation parameter.

Example 36 - Another caseMatch usage



The `default` as well as the `init` subshaders point to `browser_door`, the `browser` operation points to the `browser_scope` Shader. To illustrate the process: The first request with no parameter set enters the default Shader, which delivers links with the operation-parameter set to "browse". When used, these links invoke requests leading to the `browse` subshader.

One may argue that the loop does not constitute advantages here, as there seems to be a single iteration only in any case. This is true, but we tried to introduce a general structure to be also used by more complex web applications. However, beyond a simple web application, a redundant while loop and lots of other new Shader usages, the example effectively demonstrates the reusability of Janus entities. Servlets and the static path share five Shaders, so Janus proves to get an expressive service description language. And we continue with adding heavily reusable components: Sessions.

6.7. Adding sessions

Our approach to sessions is a Shader trying to detect the according session to a request by means of a `sessionid` query parameter. If one is present, the last state of this session is read from the shared mutable state (using the session id). A session state is stored atomically as an `XmlTree` value. Then it is inserted in an according `session` subtree of the Transaction. If no session can be found, a session id is generated and an empty session is created in the Transaction. Afterwards, Shaders can use the session state as stored in the Transaction and when finished, a second new Shader is used to push the session back into to shared mutable state. To do

so, the subtree `/transaction/session/state` is stored as a whole in a node defined by the session id. Here we have the according configuration file:

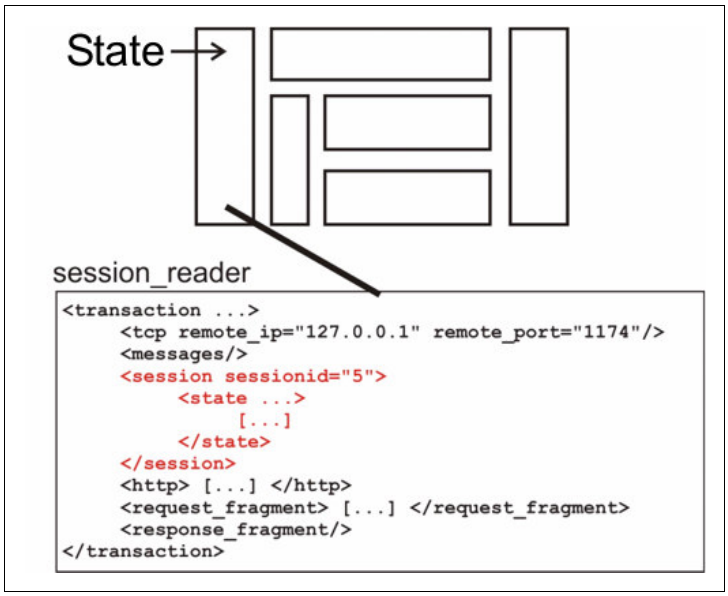
Example 37 - Configuring session Shaders

```
<?xml version="1.0" encoding="UTF-8"?>
<janus>
   <config>
      <loader>
         [...]
         <shader reference="session_reader" name="sessionReadShader" module="..."/>
         <shader reference="session_writer" name="sessionWriteShader" module="..."/>
         [...]
      </loader>
   </config>
   <init>
      <shader type="init_mime"><config typefile="./conf/mime.types"/></shader>
   </init>
   <handler id="tcp_80" type="tcpHandler">
      <config><port>80</port></config>
      <init/>
      <shader type="sequence">
         [...]
            <shader id=".*/browser.*" type="sequence">
               <shader type="session_reader"/>
               <shader type="while">
                  <shader id="predicate" type="neg">
                     <shader id="predicate" type="valExists">
                        <config path="/transaction/http/response/body"/>
                     </shader>
                  </shader>
                  <shader id="body" type="caseMatch">
                     <shader id="value" type="taExpr">
                        <config path="/transaction/http/request/cgi/@operation"/>
                     </shader>
                     <shader id="default" type="browser_door"/>
                     <shader id="init" type="browser_door"/>
                     <shader id="browse" type="browser_scope"/>
                  </shader>
               </shader>
               <shader type="session_writer"/>
            </shader>
         [...]
      </shader>
   </handler>
</janus>
```

There is nothing to add, one can see the two new `ShaderCreators session_reader`, which copies existing session states into the Transaction or creates new sessions,

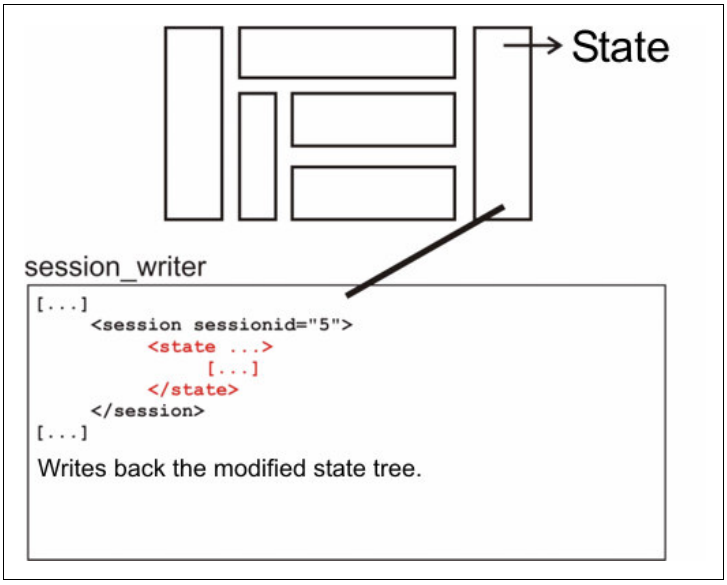
and `session_writer`, which copies session states back into the `Context`. To aid understanding, we show an example transaction immediately after the `session_reader` has been completed:

Example 38 - A Transaction after the session reader Shader



In this example, the session id is 5 and may have been selected e.g. by the URI query part. A `session` subtree has been inserted and its state has been filled with the state tree from the `Context`. If no session id had been supplied, the only structural difference would be that the `/transaction/session/state` subtree were empty and the id 5 would have been created by the `session_reader`.

Example 39 - The session_writer Shader



So, again we introduced an important and widespread concept by a simple and reusable extension coded and configured directly in the Shader language.

6.8. Adding authentication

Last but not least, we provide an example configuration to implement authentication. It shall only illustrate the idea, no concrete transaction is provided:

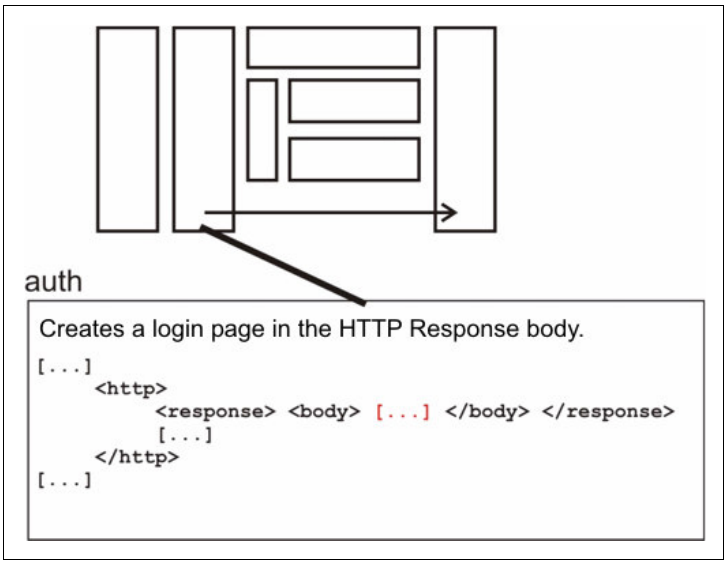
Example 40 - Configuring authentication

```
<?xml version="1.0" encoding="UTF-8"?>
<janus>
   <config>
      <loader>
         [...]
         <shader reference="auth" name="authShader" module="..."/>
         [...]
      </loader>
   </config>
   <init>
      <shader type="init_mime"><config typefile="./conf/mime.types"/></shader>
      <shader type="load"><config scope="global" to_state="/userdb"
            file="./conf/user.db" _1="password"/></shader>
   </init>
   <handler id="tcp_80" type="tcpHandler">
      <config><port>80</port></config>
      <init/>
      <shader type="sequence">
         [...]
            <shader id=".*/browser.*" type="sequence">
               <shader type="session_reader"/>
               <shader type="auth">
                  <shader type="while">
                     <shader id="predicate" type="neg">
                        <shader id="predicate" type="valExists">
                           <config path="/transaction/http/response/body"/>
                        </shader>
                     </shader>
                     <shader id="body" type="caseMatch">
                        <shader id="value" type="taExpr">
                           <config path="/transaction/http/request/cgi/@operation"/>
                        </shader>
                        <shader id="default" type="browser_door"/>
                        <shader id="init" type="browser_door"/>
                        <shader id="browse" type="browser_scope"/>
                     </shader>
                  </shader>
               </shader>
```

```
            <shader type="session_writer"/>
          </shader>
        [...]
      </shader>
    </handler>
  </janus>
```

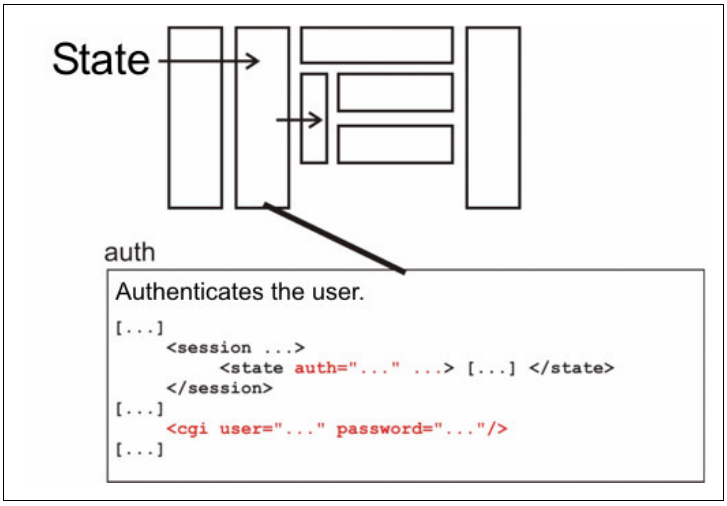
Authentication is performed by an additional `ShaderCreator`: `auth`. This Shader works like a conditional sequence Shader, executing its subshaders only in case authentication was successful. A successful authentication might be indicated simply by a session state attribute. If there is no such attribute present, the `auth` Shader skips its subshaders, but provides an HTML page to give the user the opportunity to enter his account data.

Example 41 - Creating a login page for unauthenticated users



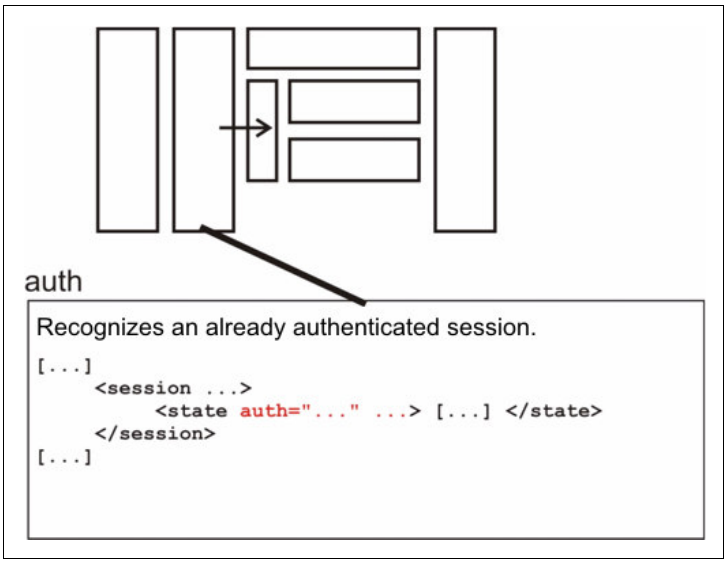
With the next request the `auth` Shader recognizes a new situation: Still there is no valid authentication in the session, but there are credentials present in the query parameters. Now, the Shader tries to authenticate these credentials. If it succeeds, the authentication is stored in the session state. Otherwise an according HTML message is transmitted to the client.

Example 42 - Authenticating a user based on e.g. URI parameters



Authentication in the example relies on a user database stored in the shared mutable state. This database is created by the `load` Shader in the server's `init` sequence, which reads a delimiter based text file and forwards the resulting tuples to the node `/userdb` in the global scope. For each user, a subnode with the username and a password node is constructed.

Example 43 - Further requests after a successful authentication



7. Java Servlet Style Extension

Janus provides a framework to define Servlets based on the transformation of XML, the concept of Transactions (in the Janus sense) and a context incorporating shared mutable states and messaging functionality. However, we started defining Servlets with regard to the Java Servlet specification. This chapter shall illustrate how the flexibility and extensibility of Janus can be used to imitate the Java Servlet style in Janus. Although not discussing the necessary source code, we will show a simple interface to map a programming style resembling Java Servlets onto Shaders.

Naturally, we are not talking about native Java code, but the conceptual facilities seen in Java Servlets. To start, we have a look at the `HttpServlet` class. The common procedure to implement a Servlet is to define a new class deriving `HttpServlet` and overriding the `doGet` method:

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
```

Actually, the main method of `HttpServlet` is the service method demanded by the `Servlet` interface. This method forwards requests to an overloaded definition of the service method, which expects specialized `ServletRequest` and `ServletResponse` instances (`HttpServletRequest` and `HttpServletResponse`, of course). Here, the HTTP method is detected and a handler for the method type is invoked (doGet, doPut, etc.). One could override the service method, but as the processing of GET requests is the usual case, typically the `doGet` method is overriden. However, all the methods provide data to the Servlet programmer by means of an `HttpServletRequest` object and expect the Servlet's effect by means of method calls to an `HttpServletResponse` object. The `HttpServletRequest` class provides an interface to select data from the respective HTTP Request (method, headers, cookies, URI, authentication, etc.) and to access the session inferred by the Servlet container. The `HttpServletResponse` primarily allows sending arbitrary data (which will typically be HTML code) and defining cookies, headers etc. Last but not least, the third primary factor is the `ServletContext`, an interface providing access to lots of meta data, for example initialization parameters provided by an according web.xml specification file. An object implementing the `ServletContext` interface can be obtained from the Servlet object itself.

So, to resemble this programming interface we define the following function and accompanying data types:

Figure 37 - Java Servlet counterparts in Janus

```
hostServlet :: (HttpServletContext -> HttpServletRequest -> HttpServletResponse ->
               JanusArrow Context a HttpServletResponse) -> ShaderCreator

data HttpServletContext = HSCON {
   servletid   :: String,
   attrs       :: [(String, String)],
   state       :: [(String, String)]
}

data HttpServletRequest = HSREQ {
   session        :: Int,
   session_state  :: XmlTree,
   url            :: URI,
   params         :: [(String, String)],
   user           :: String,
   client_ip      :: String,
   client_port    :: String
}

data HttpServletResponse = HSRES {
   header         :: [(String, String)],
   state_chg      :: [(String, String)],
   new_s_state    :: XmlTree,
   status         :: Int,
   body           :: String
}
```

`hostServlet` takes a function, which expects a context, a request and a response to build an Arrow delivering a response (without dependence on input) and constructs an according `ShaderCreator`. In the three data types (context, request, response) Janus entities get mapped onto the Java Servlet paradigma. The context value delivers the Shader identifier (`servletid`) defined in the server configuration and the attributes present at the root of the Shader's configuration (`attr`). Furthermore, a Servlet state is provided - these are string based name-value pairs, which are stored in the `Context` shared mutable state by utilizing the Shader's id. When invoking the Shader, the current Servlet state is copied into the `state` field. By means of the `state_chg` field in the response value, the Servlet state may be updated - the wrapper forwards the pairs contained in the `state_chg` field back into the state. The request value encapsulates session information (which are the session id and the current session state tree) and important information about the request (URI, query parameters, authentication, etc.). This data is taken from the respective Transaction subtree of previous facilities (TCP Handler, HTTP Shader, etc.). Finally, the

response representation (beside its state change functionality) allows defining a list of HTTP headers, a new session state, an HTTP status and the HTTP body string.

In contrast to the Java implementation the `HttpServletResponse` occurs twice in the signature. This is owed to Haskell pureness, wherefore we are forced to express changes to a value by delivering a new value. Interestingly the value is not simply delivered but wrapped into an Arrow value, which in contrast to the double occurence of `HttpServletResponse` is not a necessity. Arrows are used to do side effects, but we already can do side effects by means of some fields (especially `state_chg` and `new_s_state`) contained in the response value and the network I/O is done by the environment. However, expecting an Arrow value to deliver the Servlet's result significantly improves flexibility. Firstly, HXT Arrows can be seamlessly used to process XML (especially the session XML tree). Secondly, the Arrow based HTML building library can be integrated. And thirdly, arbitrary I/O beyond the provided mechanisms to affect the system's shared mutable state may be performed by the Servlet. As Java Servlets can easily perform I/O and often do, the third point seems to be mandatory to resemble Java Servlets.

Now, a `ShaderCreator` based on this paradigma can be for example created this way:

Figure 38 - The skeleton of a Java style Servlet in Janus

```
javaServlet :: ShaderCreator
javaServlet =
   hostServlet (\context request response ->
      proc _ -> do
         [...]
            returnA -< response { status = 200, body = ..., state_chg = [...] }
      )
```

This template is made up of a skeleton delivering the three function arguments in a Lambda expression and starting a do-notation Arrow definition. In this skeleton there may be arbitrary Arrow code, ending with a `returnA` arrow delivering the response value. In this case the input response value is taken, its status is set to 200 (OK), its body is set to an arbitrary string (denoted by "...") and the `state_chg` field is set to an arbitrary list of name-value pairs (denoted by [...]). When we justified the existence of two `HttpServletResponse` values in the `hostServlet` signature, one may have argued that without side effects and without meaningful response data at the beginning, the input response value could be dropped. The `javaServlet` example

shows a reason why this value is indeed useful: By delivering an input response value, we can define some kind of valid default response and furthermore ease the construction of the response (because only changes have to be stated - in the example we omitted the `header` and `new_s_state` fields). Besides, extensions of the response become transparent, at least as long as no construction of whole `HttpServletResponse` values is performed by a Servlet. However, as part of future work the three `HttpServletX` types could be lifted to abstract data types hiding their internal representation.

So, to conclude this chapter we discuss the achievements. Obviously, the shown extension resembles important characteristics of Java Servlet programming. Knowingly general access to the Transaction has been avoided, as this would mix up the universal concept provided by Shaders and the more application specific design of `hostServlet`. Nevertheless one has to state that Java Servlets actually provide a more generic interface than we adopted, as they are not HTTP-centric. But HTTP Servlets are the common case and our extension is a proof of concept only. A further extension to cover non-HTTP transport can be considered to be future work, as well as a more detailed adoption of the Java Servlet interface (obviously, we only took a core subset so far).

Last but not least we should not only mention the connection between a Java Servlet and a Janus `ShaderCreator` built by `hostServlet`, but also the relation between a Java Servlet container and a Janus server. A Java Servlet container loads web applications with according configuration data, instantiates Servlet objects on demand, operates a set of shared states, manages sessions as well as session states and provides the network functionality to its Servlets (by receiving requests, interpreting HTTP, starting Servlets, constructing HTTP Responses and transmitting them to the clients). As we see the functionality of Janus is quite similar, in that there is network functionality wrapped around Shaders and Shader instances of `ShaderCreator` are loaded and configured. But in contrast, huge parts of the system functionality which is inherently part of a Java Servlet container (sessions, HTTP parsing and interpretation) is implemented by Shaders and therefore easily configurable, rearrangable and extensible by the user.

8. SOAP Extension

The preceding chapter demonstrated an expressive Janus system extension to provide Java Servlet-style Shader programming. In this chapter, we are about to introduce another extension, this time not concerning programming style, but protocol support. As stated, the Janus design is especially characterised by the implementation of huge system parts in the Shader programming interface. For example, the HTTP protocol is integrated this way. Now, we would like to show the practical usefulness of this property by actually adding another protocol to the system - SOAP **[6, 31]**.

SOAP stands for "Simple Object Access Protocol" and represents an XML-based protocol to do RPCs (Remote Procedure Calls). RPC is a mechanism to invoke object code on remote systems and obtain the result, where both the invocation and the result transmission is network based. Although the abbreviation suggests an explicit support for object oriented method invocation, SOAP is not restricted to object oriented programming. However, SOAP is well suited for object oriented method calls as it provides more sophisticated selection options than RPC. Especially a URI is used to denote a request handler, where the URI of course may select a runtime object on the remote system. Next, we shortly introduce the structure of SOAP messages.

Example 44 - A SOAP message

```
<SOAP-ENV:Envelope
      xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m="Some-URI">
         <symbol>DIS</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A SOAP message contains a so called SOAP `Envelope` as the root element. The `Envelope` contains a mandatory `Body` element and an optional `Header` element. The `Header` widely follows the `Body` element structure, but is meant to allow e.g. application specific extensions. The `Body` may contain a `Fault` element in case of an error where the `Fault` element belongs to the soap namespace. Otherwise it contains the payload elements. These must not be in the soap namespace. In the

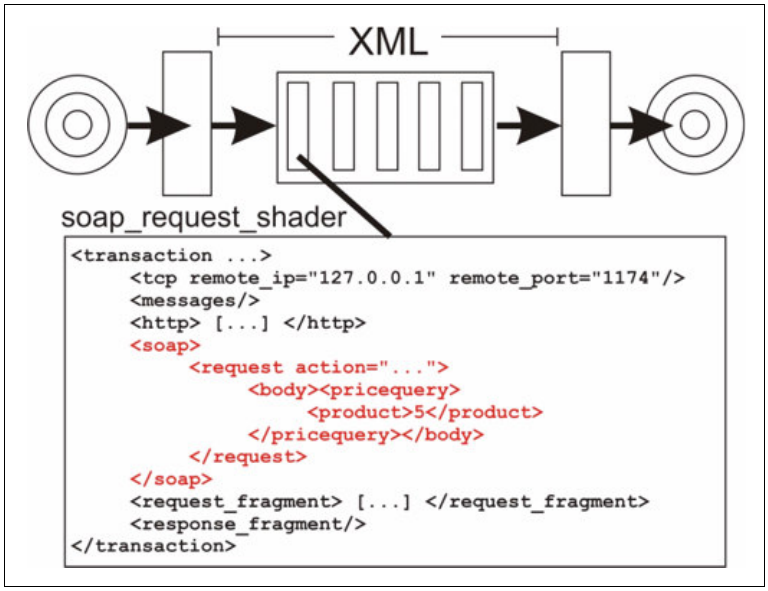
example above, the message payload is a `GetLastTradePrice` element (with its own namespace), containing a `symbol` element. One can consider this to model a function invocation of `GetLastTradePrice` with an argument named `symbol`. This argument has the value “DIS”. There are further specifications made regarding SOAP, e.g. encoding and typing of the payload elements. In Janus, SOAP currently is a proof of concept, therefore we ignore these here.

Actually, SOAP is only an example how useful the simple adding of system level Shaders is. We will not discuss the implementation in detail, but we will give a motivation for this particular extension. One may imagine a set of Shaders constituting a web shop, where some Shaders are responsible for doing model based work (independently of a user interface). There might exist a Shader to read a product price from a database, a Shader to add an article to a shopping cart, a Shader to remove an article from it, a Shader to calculate the value of a shopping cart, a Shader to do credit card billing etc. It would be possible to integrate these functions directly into the Shaders generating the user interface, but as this would be considered bad design in Java, we can assume it to be bad design in Janus too. So, given such Shaders we can think of integrating them into non-HTML (or even non-HTTP, of course) services. This is the point where SOAP enters the scene: We could think of a SOAP-based interface to access instances of the aforementioned Shaders, configured to work on the same state (i.e. article database, shopping cart representation, order storage, etc.). What do we need? Firstly, we can use HTTP POST as there is an appropriate binding for SOAP (which is indeed the typical binding). Secondly, we need to parse SOAP and interpret the message included. Then we can insert an abstract representation of the message into our Transaction tree. Thirdly, we need to map our SOAP abstraction onto our existing Shaders' interfaces.

Okay, we already have HTTP with POST method support. So we need a Shader to parse SOAP and to put it into the Transaction. This Shader is called `soapRequestShader` and it parses the HTTP body as SOAP. If there is a valid SOAP message, its content is afterwards reflected in a Transaction subtree `/transaction/soap/request`. In the root node there will be information about the binding type and the so called SOAPAction (which is retrieved by means of an HTTP header). In the subtrees `/transaction/soap/request/body` and `/transaction/soap/request/header`, the according name-value pairs are stored.

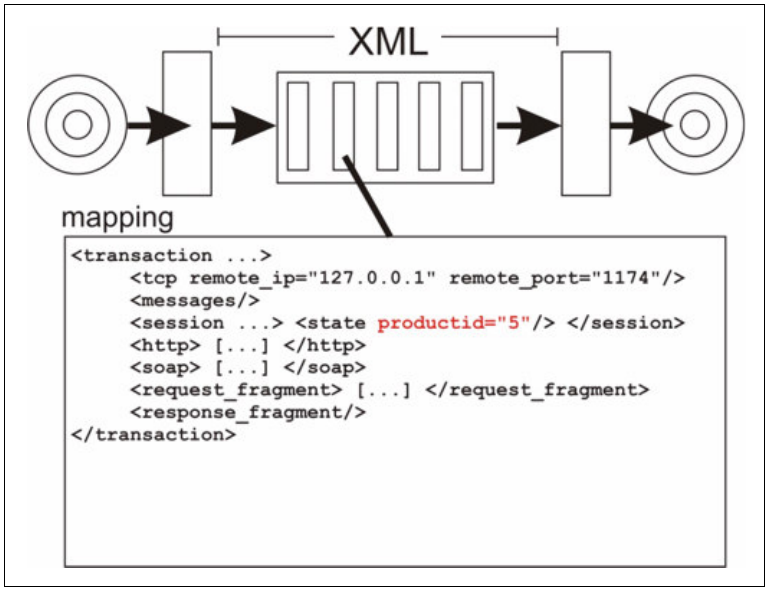
These two subtrees are essentially copies from the SOAP message tree, but in contrast to the received plaintext, they are accessible by tree operations.

Example 45 - Parsing a SOAP request

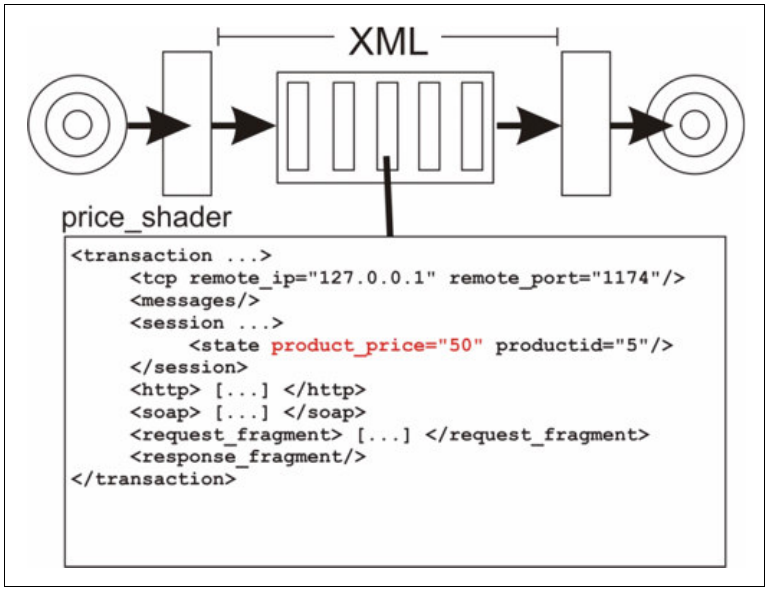


Last but not least, we need to map the SOAP structures onto the respective Shader's interface. For example when invoking the product price query, we need to map the product identifier included in the SOAP structure to the required Transaction node for the price Shader. We can utilize a generic mapping Shader for this task, e.g. to map `/transaction/soap/request/body/product` to `/transaction/session/state/@productid`:

Example 46 - Mapping of a price query's product id

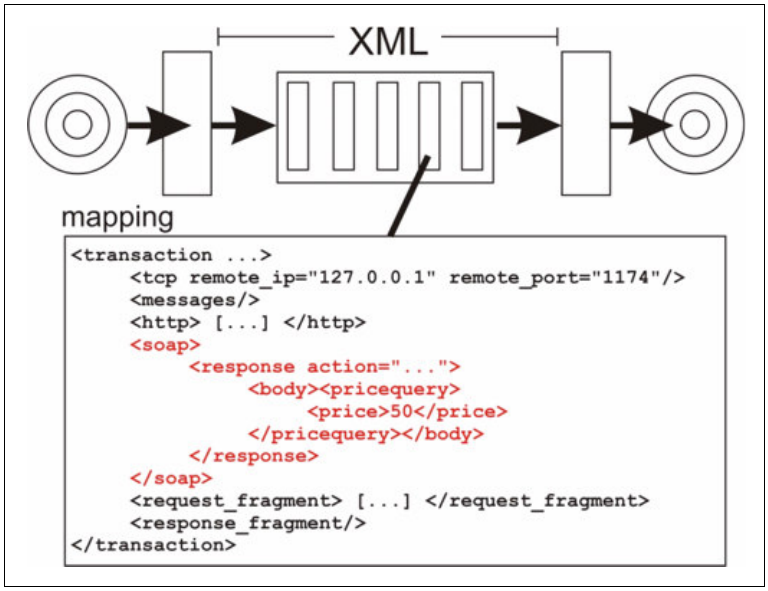


Example 47 - The actual price query Shader



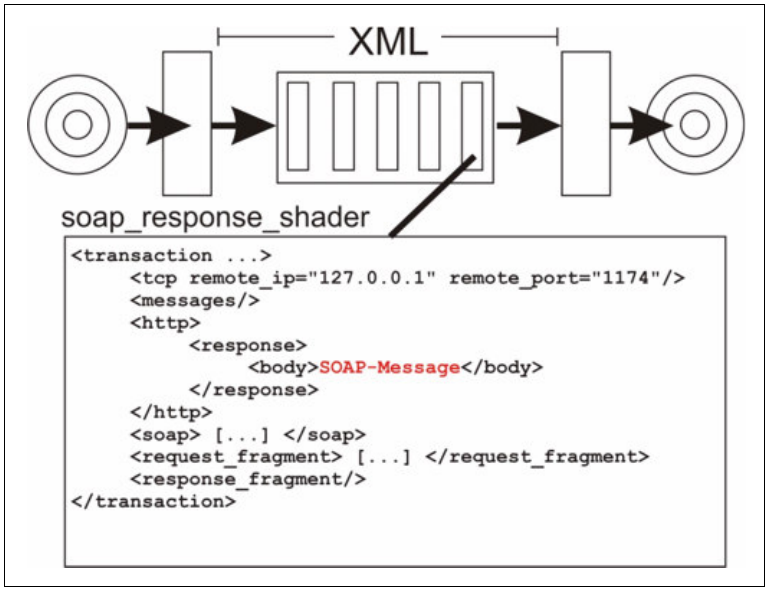
Continuing the process, we currently end up with the result price stored somewhere in the Transaction (`/transaction/session/state/@product_price`). Again, a mapping is required, this time back to a SOAP response structure. We simply define a subtree `/transaction/soap/response` structure analogous to the request subtree.

Example 48 - Mapping the price to the SOAP response



And finally, we finish with a new `ShaderCreator soapResponseShader`, creating a string representation of the according SOAP response message in `/transaction/http/response/body`. The rest is done by the usual HTTP service pipeline.

Example 49 - Building the SOAP envelope in the HTTP Response body



There is not much to add. Based on existing HTTP functionality, control structures and helper Shaders, we only had to add two `shaderCreators`: `soapRequestShader` and `soapResponseShader`. Given some Shaders independent of the user interface (although mapping might also work in this case) we completed a SOAP-based service pipeline simply by configuration. Using the same state components like the product database, which again is a matter of configuration, this service actually extends an existing web application to a web service. Now we show a configuration and a Transaction for our webshop price example:

Example 50 - A simple SOAP pipeline

```
<shader type="sequence">
   <shader type="request_shader"/>
   <shader type="soap_request_shader"/>
   <shader type="mapping" from="/transaction/soap/request/body/pricequery/product"
           to="/transaction/session/state/@productid"/>
   <shader type="price" scope="/global" db="/products"/>
   <shader type="mapping" from="/transaction/session/state/@product_price"
           to="/transaction/soap/response/body/pricequery/price"/>
   <shader type="soap_response_shader"/>
   <shader type="response_shader"/>
</shader>
```

One can recognize the Shaders to parse SOAP and to generate the response. More important, there is a `mapping` Shader copying the received product id to an attribute `productid` in the session state (we assume the `price` Shader to expect it there). The `price` Shader (which is configured to work on a product database in the global

scope) stores the respective price in an attribute `product_price`. This attribute is copied back to the SOAP response by another `mapping` Shader instance. Obviously this pipeline only processes price queries. However, we could easily add a control structure Shader to select a Shader based on the request URI or the SOAPAction. To illustrate the pipeline work, we show a Transaction after the first `mapping` Shader completed:

Example 51 - Transaction excerpt containing SOAP data

```
<transaction transaction_id="1" transaction_state="Processing" handler="TCPHandler">
   [...]
   <session sessionid="42"> <state productid="567"> [...] </state> </session>
   <soap>
      <request binding="HTTP" action="Some-URI">
         <body> <pricequery> <product>567</product> </pricequery> </body>
      </request>
   </soap>
   [...]
</transaction>
```

Some Transaction parts have been omitted (e.g. `http`). We can see the `soap` subtree with the request and the session state with the `productid` copied by the mapping Shader.

Finally, please note again that SOAP in Janus is a proof of concept only. E.g., SOAP namespaces, encoding and typing are ignored and left as future work.

9. Comparison

9.1. Introduction

Naturally there are many things left to discuss and to show regarding Janus, however its basic ideas and examples how these translate into real applications have been explained comprehensively now. So, to start reflection we will discuss several competing approaches in the area of dynamic web servers or application servers in general (as Janus itself is not restricted to the concept of a dynamic web server). Please note that we will concentrate on conceptual properties, this chapter is not intended to form a tutorial concerning the respective solutions. Moreover we will skip any further comparison to Java Servlets as these already have been discussed exhaustively when introducing Servlets in general, in our definition of Servlets and in the Java Servlet extension for Janus.

9.2. WASH/CGI

As a first alternative solution we have a look at WASH/CGI **[32]**. As already obvious by the name, this approach concentrates on the CGI interface. Therefore a useful comparison to Janus can only be done regarding HTTP/HTML functionality. However, as WASH/CGI chases some interesting ideas concerning its application domain, this should not be a deficiency.

The core purpose of WASH/CGI is to provide an expressive interface to create CGI executables to be invoked by a server (which is not inherently part of WASH/CGI itself). The programmer is supported by hiding the details of session handling, i.e. the stateless nature of HTTP is transparently lifted to a session state. Moreover, as HTML and especially HTML forms are essential to CGI, there is a library of HTML form primitives to construct pages for interaction. The resulting page descriptions may incorporate type information to let WASH/CGI do type checking of web form inputs.

Let's consider a little example:

Example 52 - WASH/CGI login reflection page

```
login = ask $ standardPage "LOGIN" $ makeForm $ table $
do
   nameF <- tr (td (text "Enter your name ") >>
```

```
            td (textInputField (attr "size" "10")))
   passF <- tr (td (text "Enter your password ") >>
            td (textInputField (attr "size" "10")))
   submitField (check nameF passF) (attr "value" "LOGIN")

check nameF passF =
   htell $ standardPage "LOGIN" $
         (text "You said " ## fromJust (value nameF) ##
          text " and " ## fromJust (value passF))
```

This CGI program asks the user to enter a username and a password by means of a web form (`login` function). Afterwards it reflects the entered data (`check` function). We will not discuss any type signatures, but it should be clear that the aforementioned functions deliver monadic values. To talk about the implementation, the `login` function uses the `ask` function on a composition of a basic web page constructor (`standardPage`), a form constructor, a table constructor and a series of field constructors to describe the login HTML page. For the input fields, the `textInputField` constructor is used, which delivers a string. Beside `textInputField`, there exists a polymorphic version (`inputField`) which is restricted to a type value, where the type is inferred by the result occurrences (as usual in Haskell). The return values (to be precise, the values these fields might return) are bound to variable names. `submitField` defines the submit button and points to the handling CGI Monad constructor: `check`, invoked with the two result variables. Finally, `check` simply creates and sends (by `tell`) a page containing the string values originally transmitted by the user.

Before discussing this notion of CGI programming, please note a naiv version of the same program coded in Janus. There is a Shader for the login page and a Shader for the reflection page. First, we provide the configuration file composing the two Shaders, utilizing a query parameter operation:

Example 53 - Janus reflection Shaders

```
<shader type="caseMatch">
   <shader id="value" type="taExpr">
      <config path="/transaction/http/request/cgi/@operation"/>
   </shader>
   <shader id="default" type="login"/>
   <shader id="check" type="check"/>
</shader>

login =
   mkStaticCreator (proc in_ta -> do
```

```
      html <- content -<< undefined
      htmlStr <- html2Str -<  html
      setVal "/transaction/http/response/body" htmlStr -<< in_ta
   )
   where
      content =
         html
            += (headers += title "LOGIN")
            += (htmlbody
               += (form "login" "/login"
                  += (block "" += formHidden "operation" "check")
                  += (block "" +>> [text "User" "", formText "username" "" 25])
                  += (block "" +>> [text "Password" "", formPass "password" 25])
                  += (block "" += formButton "action" "Login")
                     )
                  )

check =
   mkStaticCreator (proc in_ta -> do
      name <- getValDef "/transaction/http/request/cgi/@username" "" -< in_ta
      pass <- getValDef "/transaction/http/request/cgi/@password" "" -< in_ta
      html <- content name pass -<< undefined
      htmlStr <- html2Str -<  html
      setVal "/transaction/http/response/body" htmlStr -<< in_ta
  )
   where
      content name pass =
         html
            += (headers += title "LOGIN")
            += (htmlbody
               += (heading 1 ("You said " ++ name ++ " and " ++ pass))
                  )
```

The operation parameter misses in the first Transaction, therefore the default Shader is invoked (`login`). This Shader sets the parameter to `check`, so the next transmission will be directed to the respective Shader. One may note that the Janus implementation is significantly longer and less expressive than the WASH/CGI implementation. That is correct, but consider the reasons. Actually most of the additional code is owed to a more general but less powerful library to define HTML pages and forms. Primitives could be added easily to get rid of unwanted redundancies and would compress the HTML construction code. Further overhead is the composition in the configuration file. Although this is additional code compared to the WASH/CGI implementation, it is the foundation for a flexibility and configurability unknown to WASH/CGI. Sadly it does not give us any advantage regarding the concrete example.

Beside HTML representation, what are the main differences between Janus and WASH/CGI?

- WASH/CGI does composition in code and requires recompilation for modifications. Janus does not due to its configuration capabilities.

- Session handling is implicit in WASH/CGI, while it is explicit in Janus and requires an additional query parameter in the example. This is not a necessity though; with cookies for session handling the parameter could be omitted. Janus sessions are implemented as Servlets, whereas WASH/CGI sessions are hard coded into the framework.

- Parameters have to be taken explicitly from the Transaction in Janus, while they are delivered by means of function arguments in WASH/CGI. However, Transaction access provides more flexibility in Janus as for example meta data of preceding Shaders is present.

- Values are type safe in WASH/CGI, type checking is transparent to the application. This is not the case for Janus, existence and types of values have to be checked. Nevertheless the programmer might utilize the polymorphic operators to access the shared mutable state and the Transaction tree in a type safe manner. In the case of a type error the respective Shader simply fails, which is of course both less convenient and more configurable (by generating and handling errors) than the WASH/CGI solution.

- Janus operates on server level while WASH/CGI operates on CGI executable level.

- WASH/CGI provides a persistent mutable state, which is currently unmatched by Janus. However the Janus shared mutable state has been designed with later adding of persistency in mind. Therefore, many of the synchronisation mechanisms of WASH/CGI are actually present in Janus too.

After all, does WASH/CGI implement some kind of Servlet paradigma? When taking a server managing WASH/CGI executables in an appropriate way into account, we think yes, as there are arbitrary Haskell computations for the programmer, framework delivered sessions and state as well as synchronisation. Moreover HTML (and especially HTML forms) can be expressed in a very compact way and is

supported by type safety. Nevertheless many of these facilities are implications of WASH/CGI being CGI centric. Janus, on the other side, has been designed to be as independent of concrete protocols and presentation as possible.

WASH/CGI might be the better solution for exclusive HTTP/HTML web applications. Janus scores with a unified and configurable architecture, which allows extensive interoperation of Servlets, user defined system extensions (like SOAP in the previous chapter) and protocol independent programming (which again points at SOAP providing a straight web service extension to existing web applications).

9.3. HAppS

HAppS stands for the Haskell Application Server **[33]**. This name already suggests that the system is meant to provide more than just CGI or dynamic webserver functionality. The system is based on the following fundamental ideas and properties:

- There is a separation of protocols, application logic and presentation. Especially HAppS is not restricted by design to HTTP over TCP.

- Applications are considered to be of signature `Input -> m Output` in the most general case, where `m` is a Monad type. To be precise, it is `InputType -> MACID OutputType` or `Ev state event t`. The latter actually is the general type, parameterized with a state (which is application specific), an input (event) and a result type.

- Access to the current event (which is the Request when talking about HTTP) within an application function is possible by means of the function `getEvent`.

- `MACID` represents the HAppS ACID approach, which enforces transactional properties over events. Events can be considered to be requests, where a request for example might be a HTTP GET Request.

- The `MACID` Monad supports side effects. However, it is not an I/O capable Monad, actually it allows to schedule side effects to be performed by the environment. The according function is `addSideEffect`.

- Each application defines its own arbitrary Haskell state. A string serialization instance and a start state have to be implemented over this state. In the application code (which is a monadic do sequence) the state can be accessed by a Monad value `get`.

- Given a set of application functions, a Handler can be defined. This Handler is an argument to a `StdPart` constructor, which is the only argument for the system's main function to create a server.

Let's consider a small example taken from a HAppS tutorial, which provides a page to change the application state and a page to show the current state:

Example 54 - A HAppS application

```
data MyState = MyState String deriving (Read,Show)
instance StartState MyState where
   startStateM = return $ MyState "Hello World!"
instance Serialize MyState where
   typeString _ = "MyState"
   encodeStringM = defaultEncodeStringM
   decodeStringM = defaultDecodeStringM

app :: Method -> Host -> [String] -> Ev MyState Request Result
app GET _ ["change"] = do
   sresult 200 $ renderHtml page
   where
      page = header
         << thetitle << "Change State"
         +++ h1 << "Change State"
         +++ gui "/change"
         << ((textfield "state") +++ submit "change" "change")

app POST _ ["change"] = do
   rq <- getEvent
   put $ MyState (lookS 100 rq "state")
   redirect "/"

app _ _ _ = do
   MyState message <- get
   sresult 200 message

main = stdMain $ simpleHTTP "" [] app :*: End
```

The application state is `MyState`, a simple string with according instances in `StartState` and `Serialize`. The `app` function represents the web application in

question. Its signature is enforced by the `simpleHTTP` function. This function utilizes a function like `app` to create a Handler for the server creation. `app` gets the HTTP method and path segment list as first and third arguments. Based on pattern matching over these and access to the application state (by `get` and `put`), an HTML page is generated. The first `app` definition (GET method with a single path segment "change") delivers a page with a form to change the current state. `sresult` generates an HTTP response with code 200 and an HTML payload. The second definition matches against the according POST request, where the event (the HTTP Request) is obtained by `getEvent`. Taken from the event, the parameter is stored to the application state. The implementation concludes with a redirect to /. This will invoke the last definition, which matches against all methods (presumably, it will be GET) and all path segment lists other than "change". Here the current state is delivered. Like with WASH/CGI, respective Monad values are composed to describe HTML code.

We do not give an according Janus implementation, but it would be quite similar to our WASH/CGI example with an additional Shader definition for the state change. So, let's compare Janus and HAppS:

- Both systems achieve conceptual independence of transport protocols and presentation. However, HAppS requires specialized constructor functions to support additional protocols like SOAP. Furthermore a given application is fixed to request and result types as one can see in the Monad type signature. This is different in Janus where the usage of a generic XML representation does not fix a Shader to some environment. And Janus allows adding protocol support by means of Shaders, as discussed in our SOAP example.

- As the representations for events and results are different (in contrast to Transactions in Janus representing both in one entity), composition of application functions is not possible. Moreover no configuration and restructuring can be done without recompiling.

- For the moment, there is larger protocol support in HAppS (for example, SMTP) than in Janus, but as mentioned extension of Janus seems to be much easier than it is for HAppS.

- HAppS provides the ability to define an arbitrary application type in Haskell. On one hand this is more convenient than the Janus model of state. On the other hand, the Janus model provides a system supported state unified for all applications, tree structured and synchronized. And although no arbitrary types can be stored there, polymorphic access functions and some special payload types (for example, Shaders, `XmlTree` or `IO` Monad values) maintain a considerable level of convenience.

- HAppS provides ACID transactional functionality, Janus does not. Although the Janus system might be extended regarding an ACID-oriented Transaction term in the future, currently it relies completely on synchronization mechanisms to protect consistency.

- HAppS allows using XML for internal representation. An application can easily provide XML conversion functions for used Haskell data types and furthermore define an XSLT sheet to do transformation of XML to generate the final result. For Janus, internal XML representation is the native case; a mapping between XML and Haskell types is nonexistent here. A facility to provide XSLT transformations is missing and should be provided in the near future. This facility could be used to generate the response by means of XML instead of a plaintext in `/transaction/response_fragment`. Moreover XSLT could be a replacement for Shaders in certain simple situations as both are basically founded on the transformation of XML (representing Transactions).

Given these properties, the better suited system has to be selected each time for a given application type. It is especially important whether ACID properties are required, which is a domain of HAppS. One might expect Janus to be the better choice if interoperability among services and reusability of application logic in services is crucial. HAppS on the other side may provide a more concise syntax of application programming, especially as Haskell types are used in the state.

9.4. HAIFA

HAIFA (Haskell Application Interoperation Framework/Architecture) **[34]** does not compete with Janus in general, as it is specialized in web services. However, regarding our discussion of a SOAP extension to Janus in chapter 8, an at least partial comparison seems to be interesting.

HAIFA does not try to be an application server, it is a framework to model important parts of web service processing in Haskell to support almost arbitrary applications in this area without being strictly embedded into an out-of-the-box server system. Actually the framework provides a way to model SOAP web services in Haskell with XML for encoding and HTTP as the transport binding. Its architecture is not limited to these standards in general, despite the current implementation sticking to it due to common practice. There are two primary purposes fulfilled by the system:

- HAIFA provides an interface to access existing SOAP based web services via HTTP by creating a so called SOAP stub. The core function to achieve this is `soapCall`, which delivers a side effect capable function essentially transforming an input message into an output message. To support side effects, the output message is wrapped into an I/O capable Monad value. I/O is required here at least for network access, of course. The two messages involved have to be defined as XML serializable; respective type classes and functions are also provided by the framework. The request execution (which is considered to be an I/O aware string transformation) actually is parameterized, HTTP represents an instance only.

- Furthermore HAIFA provides facilities to create new web services, i.e. publishing Haskell functions as web services via SOAP. This functionality also utilizes the serialization capabilities already introduced for the client side mapping of Haskell functions to web services. Naturally, a web service is defined similar to web service access; it is a side effect enabled transformation of an input message into an output message. Pure functions of the signature `InputMessage -> OutputMessage` can be easily lifted to such functions. Given serializable message types, a service function of the stated type and some helper functions by HAIFA (a simple HTTP server to start a set of services and a service constructor function), one can build a web service system.

So, HAIFA represents client and server handling of web services, based on Haskell XML type serialization. In particular, there exists an implementation for SOAP and HTTP. Compared to Janus, HAIFA lacks the concept of system configuration and especially composition without recompilation. HAIFA does not provide explicit support for a shared mutable state or messaging facilities, like Janus does. Altogether as already stated HAIFA does not resemble an application server to

define a certain paradigma of Servlet programming, to support this paradigma by helper interfaces and to define a highly dynamic configuration framework. The published services in HAIFA may be considered to be Servlets, but actually they gain their functionality on their own, container functionality including dynamically managed Servlet composition and configuration is missing.

It is a framework to use and provide web services. To use remote web services, there is no counterpart in Janus at all. For the server component, HAIFA leaves the definition of services up to the programmer, arbitrary Haskell code might be used here. Composition can be done by means of the monadic bind operator. So it is especially useful to assist one with publishing existing or new Haskell functions. Interoperation between such published services is possible, but also left to the programmer.

Comparing SOAP server functionality of Janus and HAIFA, Janus may resemble the HAIFA interface quite easily by providing a wrapper Shader to contain functions of signature `InputMessage -> OutputMessage`. As with HAIFA, a serialization facility could be implemented to build Haskell values from Transactions. However, there currently is no serialization in Janus and furthermore, the rudimentary SOAP implementation lacks support for types at all. So while probably being suitable to match HAIFA server functionality in the future, HAIFA's sophisticated XML serialization system and type safe support for Haskell values are missing and will continue missing for a time as these extensions can be considered to be non-trivial.

10. Conclusion and future work

The thesis concludes now with a discussion of its achievements and some words about future work. One can state that Janus reaches a high level of flexibility, as it implements a concept (the Shader) suitable to implement both system level services and web application Servlets in almost the same way. This is because we abstracted both tasks to share the same general idea of describing their computations. But how useful does this idea perform when thinking of real applications? We can identify the following properties of Janus Shaders:

- **Flexibility:** As already said, the Shader abstraction can be used for both system level standard services as well as web application Servlets. The examples shown proved that a Janus server and its services can be described and changed liberately by a single configuration file. Throughout this file, one can arbitrarily combine predefined Shaders as well as user-supplied Shaders to define system behaviour. By implementing his own system level Shaders, a user may not only add web applications to a Server, but even for example transport protocols.

- **Services:** The Janus system provides network functionality, which is hidden from the applications as much as possible. One can rely on intermediate processing Shaders to provide data to a Shader or one may access a complete set of protocol meta information delivered by previous low level Shaders. Each new Shader adds to this functionality and can be utilized for future applications.

- **State support:** There are interfaces to easily access a shared mutable state and messaging. Typical activities, like error handling, persistent storing of data, logging, configuration etc. can be coded in an expressive and convenient way. Encapsulated in an Arrow state with according interface functions, functionality of the incorporated state may be improved transparently.

- **Unification:** Janus Shaders heavily rely on XML technologies. The transmission of data through a service pipeline is done by means of XML, messaging is mapped onto XML, configuration is mapped onto XML. Using the same concept and according libraries for a large set of important functions simplifies programming, eases learning and significantly advances interoperability. As a very simple example, one may store the XML document of a Transaction at some

point in a service pipeline by simply inserting an appropriate generic Shader. The resulting file of XML documents might be used directly by any XML aware application.

- **Modularity and Reusability:** Given the simple XML based combining of Shaders, their dynamic composition in a configuration file and the easy extension of the Transaction XML format, one may develop Shaders to be used in different contexts. For example, a web application Servlet may rely on parameter data for input and output, and completely ignore any Transaction tree parts regarding technical details (for example, HTTP transport). Other Servlets can be used to map transport specific facilities (e.g. Cookies or a URI's query part) onto abstract parameters. Such a parameter dependent Servlet can be reused in many situations - not only different services, but also different types of services. The same is true for the generic mapping Servlet mentioned; by means of configuration it may be used for a large set of transport mechanisms on one side and an even larger set of client Servlets on the other side.

- **Configurability:** Janus can be considered to be very configurable. This starts with one of its main aspects, the composition of the whole server behaviour by means of Shaders. Furthermore, each Shader may be instantiated with a freely definable configuration. Both description of server structure and Shader configuration base on XML and are unified in the server configuration file.

We already compared Janus to other approaches. It can be concluded that Janus (as any other approach) cannot be shown to perform better in every aspect in every application. Janus shows and provides a way to take advantage of functional programming in Haskell and many of its advanced libraries to effectively combine imperative concepts with pure functional programming (for example, Arrows). Many properties of Janus in competition stand and fall with its environment. However, the most remarkably aspect of Janus may be its "lifting" of Servlet principles to low level processing, which combined with functional programming and XML allows levels of flexibility and reusability unknown to Java Servlets.

That's for the moment. Of course there is still much to do and we would like to show some ideas that could contribute to the Janus system in the future:

- **XSLT:** XSLT (XSL Transformations, where XSL stands for Extensible Stylesheet Language) is an XML standard **[6, 18]** which allows describing complex XML transformations (where the result does not have to be XML) by means of an XML coded language with control structures and state access. The properties of XSLT show similarities to the Shader concept in general, as XSLT can perform the transformation of Transactions while utilizing a state. Although XSLT lacks expressiveness to compete with Shaders in general, there might be simple Shaders which can be coded both efficiently and effectively in XSLT. So, an extension to the Janus framework would be XSLT to replace certain Shaders. This could be done by a wrapper `ShaderCreator` parameterized with XSLT, which it compiles to a Shader value.

- **Persistence:** Currently, there is no persistent state support in Janus. I.e., Servlets willing to use persistent data have to do it by themselve. There could be an extension to the `Context` state, which allows mapping parts of a state tree onto a persistent representation. For example, each subtree could be exported to an XML file and imported back (as long as only string-representable data is used throughout the regarded subtree, i.e. no function values). Of course, such a mapping could also be done with regard to a relational DBMS. Altogether, Servlets could access persistent data transparently by existing interfaces, simply reading or writing state components. And this functionality could be added to existing Shaders by configuration - a Shader is not able to recognize if a read value originates from a previous runtime write or from a persistent representation.

- **DTD/Schema:** As discussed, Janus and all its XML formats are not restricted in any way. Especially, there is no DTD or Schema to achieve structural safety of trees or type safety of values. Although the discussion stated arguments denying the need of such facilities in Janus, an extension to integrate DTD/Schema **[6, 19]** format descriptions where useful should be future work at least for research. Formats could be defined for the configuration file, Transaction trees and messages.

- **Performance:** This thesis concentrates on concept, not on performance. However, Janus currently falls back behind other solutions (e.g. the Java Tomcat server) in terms of speed. Future work should try to enhance Janus performance. There are many approaches to achieve this, for example, speeding up frequently

used and expensive operations (e.g. XML tree operations). Additionally there are many standard components like HTTP keep-alive and pipelining to get implemented to improve efficiency.

- **Shader Primitives and Typing:** It should be clear that extending the existing libraries of Shaders and adding further primitives to construct Shaders may add significantly to the expressiveness of Janus. By the way, primitives naturally include HTML/XHTML construction. As shown when comparing to WASH/CGI, the Janus HTML support currently lacks expressiveness, which could be improved by adding more powerful primitives based on the current general ones. Furthermore, one may think of introducing automatic type checks for values in Transactions.

- **Graphical Configuration:** The server configuration file adds to the code required for any Servlet to get implemented and installed. For any `ShaderCreator` defined, the instances in the configuration and surrounding control structures have to be coded. But as these tasks are of simple and systematic nature (e.g. inserting a `loader` entry), one could imagine a graphical configuration tool to generate or modify the configuration file. This tool could be implemented as an administrative web application on a Janus server, changing the configuration of the server it is running on.

- **Automatic Installation:** It would be useful to define an installation mechanism like Java Servlet .war files. This facility would integrate the required `ShaderCreator` object code as well as an installation description file to modify the configuration file.

- **ACID Transactions:** As mentioned when comparing to HAppS, Janus Transactions should be enhanced with ACID transactional properties.

- **Namespaces:** As a small improvement, Janus XML formats should utilize XML namespaces to avoid ambiguities.

- **XML serialization:** By means of an XML serialization facility like in HAIFA, Janus could support arbitrary Haskell types in its shared mutable state.

- **Business Process Modelling:** Janus configuration resembles business process modelling when thinking of, for example, a web shop composed of Shaders. However, a step of a business process often requires a complete process (Transaction) in terms of Janus to implement user interaction. To really map business processes onto Janus (which would a really useful feature for practice), an intermediate (maybe Janus implemented) tool would be required to systematically infer a configuration based on a business process model and a set of Shader descriptions.

- **Abstract Response Body:** The body of a response is string represented currently and therefore hard to transform. It may be useful to define an optional abstract body format, which is independent of particular physical representations (e.g. HTML).

Bibliography

Print resources

[1] Louden, Kenneth C., 2003: Programming Languages Principles and Practice, 2nd edition, Brooks/Cole

[2] Hopcroft, John E. / Motwani, Rajeev / Ullman, Jeffrey D., 2000: Introduction to Automata Theory, Languages, and Computation, 2nd edition, Addison-Wesley

[3] Meyer, Bertrand, 1988: Object-Oriented Software Construction, Prentice Hall

[4] Bird, Richard, 1988: Introduction to Functional Programming using Haskell, 2nd edition, Prentice Hall

[5] Thompson, Simon, 1999: The Craft of Functional Programming, 2nd edition, Addison-Wesley

[6] Skonnard, Aaron / Gudgin, Martin, 2002: Essential XML Quick Reference, Addison-Wesley

[7] Tanenbaum, Andrew, 2001: Modern Operating Systems, 2nd edition, Prentice Hall

[8] Gamma, Erich / Helm, Richard / Johnson, Ralph / Vlissides, John, 1995: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley

Online Resources

[9] Peyton Jones, Simon, 2002: Haskell 98 Language and Libraries - The Revised Report, http://haskell.org/definition/haskell98-report.pdf

[10] Fielding R. / Gettys, J. / Mogul, J. / Frystyk, H. / Masinter, L. / Leach, P. / Berners-Lee, T., 1999: RFC2616 Hypertext Transfer Protocol - HTTP/1.1, http://www.ietf.org/rfc/rfc2616

[11] Raggett, Dave / Le Hors, Arnaud / Jacobs, Ian, 1999: HTML 4.01 Specification, http://www.w3.org/TR/html401

[12] Misc., 2002: XHTML 1.0 The Extensible HyperText Markup Language, 2nd edition, http://www.w3.org/TR/xhtml1

[13] Berners-Lee, T. / Fielding R. / Masinter, L., 2005: RFC3986 Uniform Resource Identifiers (URI): Generic Syntax, http://www.ietf.org/rfc/rfc3986

[14] Robinson, D. / Coar, K., 2004: RFC3875 The Common Gateway Interface (CGI) Version 1.1, http://www.ietf.org/rfc/rfc3875

[15] Coward, Danny, 2001: Java Servlet Specification Version 2.3

[16] Bray, Tim / Paoli, Jean / Sperberg-McQueen, C. M. / Maler, Eve / Yergeau, François, 2006: Extensible Markup Language (XML) 1.0, 4th edition, http://www.w3.org/TR/REC-xml

[17] Clark, James / DeRose, Steve, 1999: XML Path Language (XPath) Version 1.0, www.w3.org/TR/xpath

[18] Clark, James, 1999: XSL Transformations Version 1.0, http://www.w3.org/TR/xslt

[19] Fallside, David C. / Walmsley, Priscilla, 2004: XML Schema Part 0: Primer, 2nd edition, http://www.w3.org/TR/xmlschema-0/

[20] Launchbury, John / Peyton Jones, Simon, 1995: State in Haskell, in: Lisp and Symbolic Computation 8(4), Dec 1995, pp293-341

[21] Peyton Jones, Simon / Gordon, Andrew / Finne, Sigbjorn, 1996: Concurrent Haskell, 23rd ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida, Jan 1996, pp295-308

[22] Marlow, Simon, 2000: Writing High-Performance Server Applications in Haskell - Case Study: A Haskell Web Server, Haskell Workshop, Montreal, Canada, September 2000

[23] Sjögren, Martin, 2002: Dynamic Loading and Web Servers in Haskell, http://www.mdstud.chalmers.se/~md9ms/hws-wp/report.pdf

[24] Schmidt, Martin, 2002: Design and Implementation of a validating XML parser in Haskell, http://www.fh-wedel.de/~si/HXmlToolbox/thesis.pdf

[25] Ohlendorf, Manuel, 2005: A Cookbook for the Haskell XML Toolbox with Examples for Processing RDF Documents, http://www.fh-wedel.de/~si/HXmlToolbox/cookbook/doc/thesis.pdf

[26] Hughes, John, 1998: Generalising Monads to Arrows, Science of Computer Programming, 37:67-111, May 2000

[27] Paterson, Ross, 2003: Arrows and computation, The Fun of Programming, edited by Jeremy Gibbons and Oege de Moor, Palgrave, 2003, 201-222.

[28] Stewart, Don, 2005: hs-plugins - Dynamically Loaded Haskell Modules, http://www.cse.unsw.edu.au/~dons/hs-plugins

[29] Coutts, Duncan / Stewart, Don / Leshchinskiy, Roman: Rewriting Haskell Strings, http://www.cse.unsw.edu.au/~dons/papers/fusion.pdf

[30] Misc.: The Apache Jakarta Tomcat 5 Servlet/JSP Container, http://tomcat.apache.org/tomcat-5.0-doc/index.html

[31] Misc., 2000: Simple Object Access Protocol (SOAP) 1.1, http://www.w3.org/TR/soap

[32] Thiemann, Peter, 2002: WASH/CGI: Server-Side Web Scripting with Sessions and Typed, Compositional Forms, in: Practical Aspects of Declarative Languages, 4th International Symposium, pp192-208

[33] Jacobson, Alex: HAppS - Haskell Application Server v0.8.4, http://happs.org

[34] Foster, Simon, 2005: HAIFA: An XML Based Interoperability Solution for Haskell, in: 6th Symposium on Trends in Functional Programming, pp. 103-118

Affidavit

I hereby declare that this master thesis has been written only by the undersigned and without any assistance from third parties.

Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

Place, Date Signature