Multi-purpose web framework design based on websockets over HTTP Gateway

A case study of GNU Artanis

Mu Lei
SZDIY Community
mulei@gnu.org

Abstract

Traditional Internet communication is largely based on HTTP protocol, which provides hyperlinked, stateless exchange of information. Although HTTP is convenient and easy to understand, it is problematic for real-time data exchange. The websockets protocol is one of the ways to solve the problem. It reduces overhead and provides efficient, stateful communication between client and server [1].

This paper introduces a web framework design in Scheme programming language, which takes advantage of the websockets protocol to provide both convenience and efficiency. In addition to expressiveness, Scheme provides powerful abstraction ability to implement co-routines with *continuations* for the server core.

One of the key goals is to design a high-performance generic server core with *delimited continuations*. We will also show how it is useful for Internet of Things (IoT).

GNU Artanis provides useful extension modules for web programming, RESTful, web caching, templating, MVC, and a novel abstraction for relational mapping to operate databases. We will provide a summary in the rest of the paper.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

General Terms Scheme, framework

Keywords Scheme, web framework, websockets, delimited continuation, co-routine, generic server

1. Introduction

A web framework is a library for rapid prototyping in web development, and usually provide CLI (Command Line Interface) or GUI (Graphical User Interface) tools to generate code and reduce workload according to certain patterns specified by the developer. In addition to web specific languages such as PHP, the older approach for web programming is CGI (Common Gateway Interface). Although many people think CGI is outdated, it is simple to understand and easy to use. It can still be regarded as a practical way for web programming.

Scheme and Functional Programming Workshop September 18th, 2016, Nara, Japan
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM [to be supplied]. . . $15.00

But the industry requires more: a fast way for rapid prototyping, an approach for both productivity and reliability, clean and DRY (Don't Repeat Yourself) for maintaining, high-level abstractions to hide details, code less, and provide security, etc. Most of the modern web frameworks such as Ruby on Rails provide the necessary tools to produce a web application model, which is an underlying program itself [2].

This paper will use the term *HTTP Gateway* to indicate the unified connection management interfaces and verification methods of websockets over HTTP. The *HTTP Gateway* is not a new concept, it is actually hidden in many web server designs. It is worth to discuss it explicitly for it is important to understand how GNU Artanis manages all the websocket connections. We will discuss it in 3.2.

In addition to the discussion about HTTP Gateway, this paper will show a different way to implement a concurrent server. It is different from the callback mode used in C or Node.js. *Continuations* are commonly used to implement concurrent processes [3][4][5]. It is proved that *continuations* could be a model for directly expressing concurrency in Scheme language [6]. In this paper, we will take GNU Artanis as a study case to explore this model. Moreover, GNU Artanis uses GNU Guile which is a practical Scheme implementation to provide *delimited continuations* to better implement co-routines. We will demonstrate the feature named *delimited continuations*, which is a better abstraction than traditional *continuations*. We will discuss it in the section 2.1.1.

The C10K (concurrent in 10,000) and C10M (concurrent in 10,000,000) problem tells us that the performance is not the only thing to be concerned about for high concurrent servers, but also for the purposes of scalability. The old method of using select() or poll() has $O(n)$ complexity, which is a problem to hold more connections [7]. The new method is kqueue()/epoll()/IOCP which has O(1) complexity to query available connection sockets. We'll take epoll() as an example to discuss the scalability issue in 3.3.

Websockets is a protocol for two-way communication between client and server over TCP connection. The client, in a broad sense, does not have to be a web browser. One of the benefits of websockets is that the developers could take advantage of TLS used in HTTPS for encryption. Another benefit is the unified listening port for arbitrary services. We will discuss websockets in the 3.1 section.

Finally, we will discuss important features of GNU Artanis in 4.

2. Some background knowledge

It is straight-forward to explain *continuations* and *delimited continuations* in continuation passing style (CPS), and we will show how to transfer their definitions in semantics to Scheme code for practical purposes.

2.1 First Class Continuations

A *continuation* is "the rest of the computation". In Scheme, *continuation* is implemented as *first class continuation*, which is often referred to as *call/cc* standing for *call-with-current-continuation*. It captures (reifies) the state of underlying machine state named the *current continuation*. The captured *continuation* is also referred as *escape procedure* passed into the scope by the argument (as a function) of *call/cc*. When the *escape procedure* is called within the context, the run-time discards the *continuation* at the time of the calling and throws the previous captured *continuation* back into the machine state.

The *continuations* captured by *call/cc* often implies *first class*, which means a continuation could be passed and returned from functions, and stored in a data structure with indefinite extent. For its "lightweight fork ability for parallel histories" avoiding to trap into the operating system kernel, it's widely considered to be the ideal method to implement lightweight threads (co-routines/green threads), although it's also useful in many areas, such as backtracking, exception handling, and flow control analysis, etc.

variable:

$$[\![x]\!]\rho \quad = \quad \lambda\kappa.(\kappa\ (\rho\ x))$$

λ–abstract:

$$[\![\lambda x.M]\!]\rho \quad = \quad \lambda\kappa.(\kappa\ \ \lambda v\kappa'.([\![M]\!]\rho[x \mapsto v]\ \ \kappa'))$$

application:

$$[\![(M\ \ N)]\!]\rho \quad = \quad \lambda\kappa.([\![M]\!]\rho\ \ \lambda m.([\![N]\!]\rho\ \ \lambda n.((m\ n)\ \kappa)))$$

Figure 1: Continuation Semantics in the spirit of CPS [8][9].

The $[\![...]\!]\rho$ as a compound form means a simplified one-pass CPS transformation with an environment ρ which maps variables to values. This specific CPS transformation should be constrained by two conditions: (1) Should not introduce any administrative redex (the reducible expression operated by a continuation capturing lambda); (2) Would not reduce the source term. And $E[... \mapsto ...]$ means capture-avoiding substitution in expression E. The term $(E_1\ \ E_2)$ represents the application of a function E_1 to an input E_2. The κ is *continuation* which has the type of a function from values to values. In spirit of CPS, the *λ–abstract* in Figure 1 denotes a function from an environment ρ to a function accepting an argument v and a *continuation* κ'.

And we give semantic of *call/cc* according to Figure 1:

$$[\![(\textbf{call/cc}\ F)]\!]\rho \quad = \quad \lambda\kappa.([\![F]\!]\rho\ \ \lambda e.(e\ \lambda v\kappa'.(\kappa\ v)\ \kappa))$$

Call/cc accepts a function evaluated from F. The e is the previously mentioned *escape procedure*. The κ' is the *continuation* of inner context which is trivial for the throwing, so it is never used, and actually replaced by the *continuation* κ captured by *call/cc*. As we mentioned previously, this is what happens when the *escape procedure* throws *current continuation*. With this semantic definition, we could easily rewrite it as Scheme code:

For the application of $[\![(\textbf{call/cc}\ F)]\!]\rho$, we could simplify all the forms to make it clearer:

```
((lambda (k)
   (lambda (e)
    (e (lambda (v _) (k v)) k)))
 F)
```

We have the definition of *call/cc* in Scheme code:

```
(define call/cc
 (lambda (e k)
   (e (lambda (v _) (k v)) k)))
```

The common placeholder "_" is κ' in semantic definition, which is trivial, could be ignored. The ρ is dismissed since the environment is managed by Scheme inexplicitly.

In most of Scheme implementations, *call/cc* is rarely in CPS which helped us to analyze the flow control for better understanding of *continuations*. But, practically we have to dismiss the explicit continuation passing (k or κ above) which makes the expression complex. It is hard to show how the *continuation* κ is processed in non-CPS form, so we just need to know that the *continuation* will be captured and thrown in Scheme inexplicitly.

Although *call/cc* is a fine way to implement threads, it is well known that *call/cc* captures too many things which are an overkill for most control features. To avoid this problem, we introduce *delimited continuations*.

2.1.1 Delimited Continuations

Delimited continuations are more expressive than *call/cc*. Nevertheless, it captures less things to make the *continuations* more lightweight.

As an ordinary *continuation* stands for "the rest of the computation", a *delimited continuation* represents "the rest of the computation up to somewhere" [10].

Although many competing delimited control operators exist in the language research community, **shift/reset** are mentioned often. Following the semantics in Figure 1, **shift** and **reset** has the definition [8]:

$$[\![(\textbf{reset}\ E)]\!]\rho \quad = \quad \lambda\kappa_r.(\kappa_r\ \ ([\![E]\!]\rho\ \ \lambda x.x))$$

$$[\![(\textbf{shift}\ c\ M)]\!]\rho \quad = \quad \lambda\kappa_s.([\![M]\!]\rho[c \mapsto \lambda v\kappa'.(\kappa'\ (\kappa_s\ v))]\ \ \lambda x.x)$$

Note $\lambda x.x$ is a common procedure named *identity* which is used to indicate an empty *continuation* here. Apparently, if there's no **shift** within **reset**, the evaluated expression E will be returned without any change from **reset** because it is applied by *identity* function as the empty *continuation*. But if **shift** is evaluated within, the κ_r has no chance to be applied because the whole context returns by applying κ_s. What does it mean? It means that the *continuation* was truncated (delimited) to κ_s, and the trivial κ_r will not be captured. That's why it can reduce the cost compared to *full continuation*. It is very different from *full-continuation* capturing in **call/cc**. This feature would solve the problem of overkill we mentioned in the end of the last section.

So far we have seen the preliminary principle of *delimited continuation*. It is necessary to stop the discussion of semantics and get back to our topic. We are going to depict how GNU Guile handles *delimited continuations*, and how it helps our main purpose of implementing co-routines.

Although it is natural to implement *delimited continuations* by complex CPS transforming in semantics. It never has the best performance. GNU Guile implemented *delimited continuations* in the direct way that uses the language's native stack representation, not requiring global or local transformations [11]. Rather than CPS transformation, the direct implementation will copy the *continuation* chain which resides in the heap, and involves copying of *continuation* frames both during reification and reflection.

According to test results, it shows that the direct-implementing approach is fabulously faster than CPS transforming way [9]. The direct implementation captures *continuation* by copying the stack directly. Nevertheless, we could possibly further optimize of stack-/heap management and copying methods to make it better.

There are three equivalent terms of interfaces to handle *delimited continuations* in GNU Guile. In addition to the common term

shift/reset, **%/abort**, and **call-with-prompt/abort-to-prompt**. The only difference is their operational approach.

We chose **call-with-prompt/abort-to-prompt** for its third argument as a function will receive the thrown *continuation*, and will be called automatically each time the **abort-to-prompt** is called. This feature is very useful to implement the scheduler of co-routines. We will discuss it in 2.2. A common usage of it could be the following:

```
(call-with-prompt
 '(the-tag-to-locate-prompts)
 (lambda ()
  ... ; Do your job
 (abort-to-prompt
  '(the-tag-to-locate-prompts))
  ... ; continue the work
  ...)
 (lambda (k . args)
  ;; k is current continuation
  (save-the-continuation k)
  (scheduler ...))))
```

Delimited continuations have been implemented in few languages: GNU Guile, PLT Scheme, Scheme48, OCaml, Scala. Considering that many mainstream dynamic languages have *first class continuations* (call/cc), optimistically it is possible that more languages will implement *delimited continuations* as well. In view of the reasons mentioned above, the study case in this paper may shed some light on server design issues for the future.

2.2 Co-routines

Co-routines are essentially procedures which save states between calls. It has become a very important form of concurrency which avoids practical difficulties (race conditions, deadlock, etc.) to reduce complexity. Developers do not have to take care of synchronization by themselves, but leave it to this paradigm with this built-in feature. Co-routines is not generic enough to solve any concurrency problem, however, it is a possible solution for server-side development.

It is demonstrated that co-routines may easily be defined by users given first class continuations[12]. The term "first class" here means a continuation that could be passed and returned from functions, and stored in a data structure with indefinite extent. The co-routine could be implemented as a procedure with local state. In server-side development, in addition to traditional HTTP requesting, most servers need to maintain long live session for a connection. It is required that these procedures could be broken for a while, and sleep until the next packet arrives. In old the fashion way, people used OS-level threads (say, pthread) to avoid blocking. But, this approach creates some critical overhead (trap into kernel, locks, synchronization), not to mention the difficulties in debugging programs with threads. Ironically, people even complain that the threads model is a bad idea in practice [13].

Since Scheme provides full support for continuations, implementing co-routines is almost trivial, requiring only that a queue of continuations be maintained. This paper introduces one way to implement co-routines with *delimited continuations*. We will see how this approach is convenient and clean later in this section.

As described in 2.1.1, GNU Guile provides several similar abstract interfaces to handle *delimited continuations*. Here we choose the pair functions *call-with-prompt* and *abort-to-prompt* for it, since it is easier to invoke scheduler procedure which is used to resume the stopped co-routines while throwing the *delimited continuations*.

The basic principle of co-routine implementation is that saving context to a first class object then adding it into a queue, and scheduling around till the queue is empty. So, the first step is to initialize a queue:

```
(define *work-queue* (new-queue))
```

And the *spawn* interface: it is common to spawn a new co-routine. The *call-with-prompt* function was introduced in 2.1.1.

```
(define-syntax-rule (spawn body ...)
 (call-with-prompt
  (default-prompt-tag)
  (lambda ()
   body ...)
  save-context))
```

As described in 2.1.1, the last argument of *call-with-prompt* is a function which receives current continuation as the first argument. Obviously, we should save it to the queue when it needs to sleep. The second argument is optional, and we customize it as the index of the request in our example.

```
(define (save-context k idx)
 (format #t
         "Request~a␣EWOULDBLOCK!~%"
         idx)
 (queue-in! *work-queue* (cons idx k)))
```

The sleeping feature is implemented with *abort-to-prompt*. When it is called, the run-time will throw the *current continuation* bound to *k* as the first argument of *save-context* function.

```
(define-syntax-rule (coroutine-sleep idx)
 (abort-to-prompt
  (default-prompt-tag)
  idx))
```

Each time when a certain condition is met, the related *delimited continuation* would be resumed. The *resume* function will resume the task properly. Note that the resumed continuation should be re-delimited again to avoid stack issues. When calling *k* for resuming continuations, it is necessary to pass *idx* as the argument.

```
(define-syntax-rule (resume k idx)
 (call-with-prompt
  (default-prompt-tag)
  (lambda ()
    (k idx))
  save-context))
```

Finally, we need a scheduler to arrange all the tasks to be completed automatically.

```
(define (schedule)
 (cond
  ((queue-empty? *work-queue*)
   (display "Schedule␣end!\n"))
  (else
   (let ((task (queue-out! *work-queue*)))
    (resume (cdr task) (car task))
    (schedule)))))
```

Now we have a simple co-routine framework. The code to implement co-routines in GNU Artanis is far more complex than the listed code. Nevertheless, they are similar in principle, and the listed code is much easier to understand. For now, it's time to use them for requests handling.

Listing 1: Co-routine handling requests

```
(define (coroutine-1)
  (display "Accepted␣request␣1\n")
  (display "Processing␣request␣1\n")
  ;; If EWOULDBLOCK
  (coroutine-sleep 1)
  ;; Resumed when condition is met
  (display "Continue␣request␣1\n")
  (display "End␣coroutine-1\n"))
```

```
(define (coroutine-2)
  (display "Accepted␣request␣2\n")
  (display "Processing␣request␣2\n")
  ;; If EWOULDBLOCK
  (coroutine-sleep 2)
  ;; Resume
  (display "Continue␣request␣2\n")
  ;; EWOULDBLOCK again
  (coroutine-sleep 2)
  ;; Resume
  (display "End␣coroutine-2\n"))

(define (run)
  (spawn (coroutine-1))
  (spawn (coroutine-2))
  (schedule))
```

Let's see the result. Certainly, in real cases, we use meaningful functions to replace these string printing functions. These printing lines indicate what could be happening.

Listing 2: Coroutines running result

```
Accepted request 1
Processing request 1
Request1 EWOULDBLOCK!
Accepted request 2
Processing request 2
Request2 EWOULDBLOCK!
Continue request 1
End coroutine-1
Continue request 2
Request2 EWOULDBLOCK!
End coroutine-2
Schedule end!
```

3. Server Core Design

Having finally learned about this powerful weapon named coroutines, we may consider how to use it in a server program. We will now show the server core design with a concern for functionality and performance.

As it is the case for performance concerns, i.e. non-blocking and edge-triggered I/O multiplexing, as we will describe in GNU/Linux environments, epoll() will be discussed in this case. GNU Artanis uses epoll() for I/O multiplexing.

The websockets protocol plays an important role in this functionality. The idea is to have a generic server which implements the websocket library and uses it to parse requests wrapped in the websockets protocol, and, then, redirects a requests to a proper service handler (local or remote, decided by the user). This is called simply a HTTP Gateway. With such a design we could handle connections in various protocols (rather than one dedicated protocol) wrapped in websockets. It is called *generic*, for it supports arbitrary protocols to meet the generic needs.

When we talk about the generic server core for a web framework, it may sound confusing. Because a web framework should have a dedicated HTTP server. But one could implement a generic server over HTTP because websockets makes it possible. So, let's see how websockets make it possible.

3.1 Websockets

Although the term websockets looks like kind of web related stuff, it is an independent TCP-based protocol. The only relationship to HTTP is that the handshake process is based on HTTP protocol as an upgrade request.

This makes it possible to allow messages to be passed back and forth while keeping the connection open. This approach keeps a bidirectional ongoing conversation taking place between client (most of the time, it is a browser) and the server. Websockets also provides full-duplex communication. All the communications are done over TCP port (usually 80), which is of benefit for those environments which block non-web Internet connections behind a firewall. This makes all the connections long-living sessions rather than short sessions as traditional HTTP does and provides the possibility for implementing Comet technology, the old way for full-dumplex communication on web, in a more convenient way.

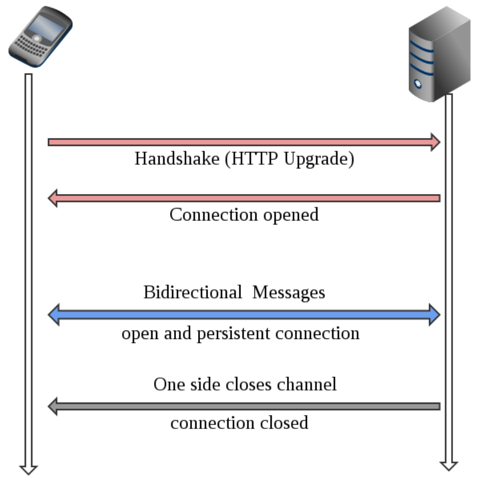

Figure 2: Websockets handshake and communication

Another benefit to transporting data over HTTP is that many existing HTTP security mechanisms also apply to websockets. With this unified security model, a list of standard HTTP security methods could be applied to a websockets connection. For example, the same encryption as HTTPS using TLS/SSL. It is the same way to configure TLS encryption for websocket as you do for HTTPS with certificates. In HTTPS, the client and server first establish a secure envelop which begin with HTTP protocol. Websockets Secure (WSS) uses the exact same way with handshake in HTTP, then upgrades to the websockets protocol.

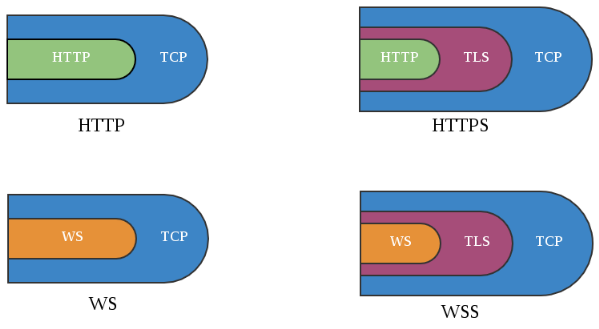

Figure 3: HTTPS and Websockets Secure

3.2 HTTP Gateway

One of the important concept of GNU Artanis is the HTTP Gateway. As mentioned earlier in this paper, HTTP Gateway is not a new concept, for it is transparent in many server programs.

The HTTP Gateway, as described intuitively, is a portal between the client and the customized protocol processing module of the server program, taking the HTTP negotiation to allow them to share information by communicating with another protocol over HTTP.

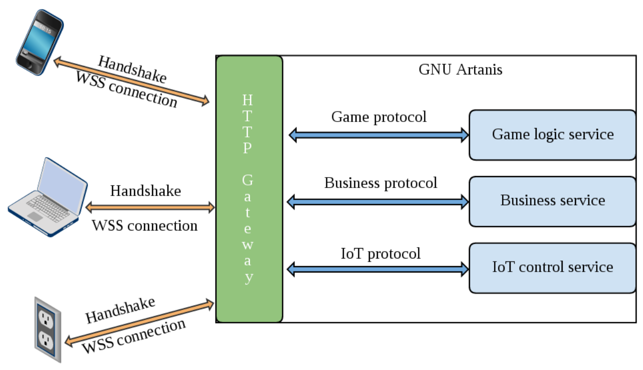

Figure 4: HTTP Gateway architecture in GNU Artanis

The Figure 4 shows the HTTP Gateway architecture in GNU Artanis. It provides arbitrary protocol services over websockets. Beyond the web server, it becomes a *generic server*, which handles connections of multi-protocols at the same time. A *generic server* was rarely mentioned before, because most of the applications for server programs are for dedicated protocols as well as to provide a single service as its main duty.

For the *dedicated server* which usually provides just one protocol service, for instance, a HTTP server. Usually, the remote server starts a service program as a daemon listening on a TCP/UDP port according to a convention, and provides one service for a dedicated protocol. Such a decoupled pattern meets the KISS (Keep It Smart and Simple) principle. Moreover, when one service is down, it will not effect the others.

For the *generic server*, people do not have to listen on many TCP ports, there is just one port, 80 or 443 (for HTTPS). And the HTTP Gateway will dispatch the requests to the related service, and maintain the long live session for each connection in a high concurrent way. In the simplest form, all the services are running in GNU Artanis rather than standalone, and this is good for both quick development and deployment. Some people may think it is too tightly coupled, if one of the services is broken, the whole GNU Artanis may go down. How we solve this? Fortunately, it is possible to make each service standalone, similar to CGI, but in bytevector way according to the websockets configuration, rather than HTTP requests redirecting. This would be good enough to solve the issue.

So, what is the value of a *generic server*?

It is obvious that the massive protocol-customized connections will appear in such a scenario like Internet of Things (IoT). There will be many IoT nodes as massive clients, with many sensors or monitors, which may require many application-level protocols, although their messaging protocol maybe the same (MQTT over websockets). Since the *generic server* reduces the workload of customizing and managing multiple protocols, and once IoT-based applications become the next big thing, the *generic server* may play an important role in next decades.

3.3 Concurrency

The eternal subject of server-side development is always concurrency. Many years ago, the industry was focusing on *C10K* problem with at least $10,000$ concurrent connections. Nowadays, it is becoming *C10M* which means at least 10 million connections concurrently.

No matter how the number increases, our main purpose is to hold more connections concurrently as possible. Unfortunately, our efforts are useless if one just purchases faster machines with more RAM. This way the performance of single-connection processing could be higher, but increasing few connections concurrently.

The bottleneck is not the performance of a machine, but the algorithm of event dispatching. The traditional select() and poll() are outdated, for their $O(n)$ complexity drags the performance down when it tries to query large number of sockets. The modern epoll() has constant time complexity for that, say $O(1)$, and obviously wins the title.

Furthermore, the edge-trigger mode of epoll() co-operated with non-blocking I/O is widely used in the industry for high-performance concurrent programming. It is believed that the concern of a server is not only about concurrency, but also better exception handling for robustness of the system.

Naturally, it is necessary to provide an advanced scaling methodology for higher concurrency to meet the future needs of cloud computing. But this discussion is out of scope in this paper.

GNU Artanis implements epoll() for its I/O multiplexing and takes advantage of non-blocking to implement asynchronous I/O in co-routines. This model is proved to be practical and well-known in the industry. We choose this model (epoll + non-blocking) to co-operate with co-routines and provide good performance and concurrency.

4. Some features in GNU Artanis

In addition to the server core, there are some notable features in GNU Artanis.

4.1 RESTful

REST stands for representational state transfer, which is an architectural style consisting of a coordinated set of components, connectors, and data elements within a distributed hypermedia system where the focus is on component roles and a specific set of interactions between data elements rather than implementation details.

Depending on the extent in which a system conforms to the constraints of REST, they can be called RESTful. To communicate using a REST service over HTTP, the same method name (GET, POST, PUT, DELETE, etc.) which web browsers use to retrieve pages and send data to remote servers is applied. REST systems interface with external systems as web resources identified by URIs.

The name “representational state” is intended to evoke an image of how a well-designed Web application behaves: a network of web pages, where the user progresses through the application by selecting links (state transitions), resulting in the next page (state) being transferred to the user and rendered for their use.

```
(get "/hello/:name"
  (lambda (rc)
    (format #f "hello␣~a␣~%"
            (params rc "name"))))

;; curl example.com/hello/mulei
;; ==> hello mulei
```

4.2 MVC

Model-View-Controller (MVC) is a software architectural pattern for implementing user interfaces on computers. It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user. Traditionally used for desktop graphical user interfaces (GUIs), this architecture has become extremely popular for designing web applications.

The *Model* is the unchanging essence of the application/domain. When there is more than one interface with the *Model*, they are called *Views*. The *Views* could be a GUI, a CLI or an API. Although *Views* are very often graphical, they dont have to be. A *Controller* is an object that lets you manipulate a *View*. In short, a *Controller* handles the input whilst the view handles the output [14].

GNU Artanis provides CLI tools for generating MVC template code. This feature will be introduced in the section 4.5.

4.3 Relational Mapping

Relational Mapping (RM) stands for ORM (Object Relational Mapping) in most cases. It is a programming technique for converting data between incompatible type systems in object-oriented (OO) programming languages.

However, although GNU Guile has an object system named GOOPS, GNU Artanis chose not to use OO for programming. It is enough to use the features of Functional Programming (FP) to replace the essentials in OO, and this is proved in the GNU Artanis development. Because of this, GNU Artanis does not implement ORM, but aims to replace classes, and message passing for dispatching the methods to mimic half-baked OO, which is more lightweight and less complex than OO. This is called Functional Programming Relational Mapping (FPRM) in GNU Artanis.

4.4 Sessions

HTTP sessions allow for associating information with individual visitors.

A session is a semi-permanent interactive information interchange, also known as a dialogue, a conversation or a meeting, between two or more communicating devices, or between a computer and user. A session is set up or established at a certain point in time, and then torn down at some later point. An established communication session may involve more than one message in each direction. A session is typically, but not always, stateful, meaning that at least one of the communicating parts needs to save information about the session history in order to be able to communicate, as opposed to stateless communication, where the communication consists of independent requests with responses.

Traditionally, there are three kinds of session management in GNU Artanis:

- *Simple*, use hashtables for storing sessions in the memory;
- *Filesystem*, use files for storing sessions;
- *Database*, use Database for storing sessions.

4.5 CLI tools

Providing CLI tools is becoming very common for most web frameworks. Basically, there're four commands in GNU Artanis:

To initialize a new application folder:

```
art create project-name
```

And the *draw* command is useful to generate MVC template code to save development time:

```
art draw [controller/model] name
```

Note that *Views* are generated along with *Controllers*.

Sometimes it is necessary to move from one database vendor to another, or to upgrade the version of the database software being used. During a database migration, it could be useful to reconstruct the schema and tables, then import all the data to the new environment automatically.

```
art migrate operator name
```

Last but not least, the *work* command is used to start the server, and establish the service, listening on the specified port.

```
art work
```

5. Future work

It is possible to implement map/reduce for cluster applications, and very high-scalability with serializable continuations to scale the whole server system by adding an unlimited number of nodes. These issues are open for further discussion in the future.

Acknowledgments

First off, I would like to thank the Free Software community for all the contributions.

Special thanks to the SZDIY community for providing all the conditions, and all the kind encouragement for this work. Also, I would like to thank my second daughter who is only 18 days. She came into this world suddenly and in advance, which disorganized my writing plan. Fortunately, I have a strong server core to handle these two concurrent tasks, despite being exhausted.

References

[1] V. Pimentel and B. G. Nickerson. Communicating and displaying real-time data with websocket. *IEEE Internet Computing*, 16(4):45–53, July 2012. ISSN 1089-7801. doi: 10.1109/MIC.2012.64.

[2] D. Geer. Will software developers ride ruby on rails to success? *Computer*, 39(2):18–20, Feb 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.74.

[3] S. Krishnamurthi, P. W. Hopkins, J. McCarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the plt scheme web server. *Higher-Order and Symbolic Computation*, 20(4):431–460, 2007. ISSN 1573-0557. doi: 10.1007/s10990-007-9008-y. URL `http://dx.doi.org/10.1007/s10990-007-9008-y`.

[4] R. Hieb and R. K. Dybvig. Continuations and concurrency. *SIGPLAN Not.*, 25(3):128–136, Feb. 1990. ISSN 0362-1340. doi: 10.1145/99164.99178. URL `http://doi.acm.org/10.1145/99164.99178`.

[5] R. Hieb, R. K. Dybvig, and C. W. Anderson, III. Subcontinuations. *Lisp Symb. Comput.*, 7(1):83–110, Jan. 1994. ISSN 0892-4635. doi: 10.1007/BF01019946. URL `http://dx.doi.org/10.1007/BF01019946`.

[6] O. Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages.

[7] J. Lemon. Kqueue-a generic and scalable event notification facility. In *USENIX Annual Technical Conference, FREENIX Track*, pages 141–153, 2001.

[8] O. Danvy and A. Filinski. Representing control: a study of the cps transformation, 1992.

[9] M. Gasbichler and M. Sperber. Final shift for call/cc:: Direct implementation of shift and reset. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 271–282, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8. doi: 10.1145/581478.581504. URL `http://doi.acm.org/10.1145/581478.581504`.

[10] E. Sumii. An implementation of transparent migration on standard scheme. In *Proceedings of the Workshop on Scheme and Functional Programming, Technical Report 00-368, Rice University*, pages 61–64. Citeseer, 2000.

[11] A. Wingo. Guile and delimited continuations, 2010. URL `https://goo.gl/FKxtRf`.

[12] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 293–298, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: 10.1145/800055.802046. URL `http://doi.acm.org/10.1145/800055.802046`.

[13] J. Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, volume 5. San Diego, CA, USA, 1996.

[14] J. Deacon. Model-view-controller (mvc) architecture. *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*, 2009.