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

A case study on GNU Artanis

Mu Lei
SZDIY Community
mulei@gnu.org

Abstract

The 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's problematic for real-time data exchange. The websocket protocol is one of the ways to solve the problem. It reduces overhead and provides efficient, stateful communication between client and server [?].

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

One of the significant aim is to design a high performance generic server core with *delimited continuations*, and we'll show how it's useful for Internet of Things (IoT).

GNU Artanis also provides useful modules for web programming, RESTful, web caching, templating, MVC, and a novel abstraction for relational mapping to operate databases. We'll give it 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, websocket, delimited continuation, co-routine, generic server

1. Introduction

Web framework is a library for rapid prototyping web development, and usually provide CLI (Command Line Interface) or GUI (Graphical User Interface) tools to generate code for reducing workload according to certain patterns specified by the developers. In spite of using web specific language 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 Although many people think CGI is outdated, it is simple to understand and easy to use.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 yet be regarded as a practical way for web programming.

But the industry always 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 abstract for hiding details, less coding, securities, etc. Most of the modern web frameworks such as Ruby on Rails provides the tools necessary to produce a web application model, which is the underlying program itself [?].

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

Besides, this paper will show a different way to implement concurrent server. It is different from the callback way used in C or Node.js. We use *continuations* which has been already practiced as the abstract of concurrent processes in several cases [?][?][?]. It is proved that *continuations* could be the model for directly expressing concurrency in Scheme language [?]. In this paper, we'll take GNU Artanis as a study case to indicate the same purpose. Moreover, GNU Artanis uses GNU Guile which is a practical Scheme implementation which provides *delimited continuations* for better implementing co-routines. In this paper, we'll show a new way with the feature named *delimited continuations*, which is better abstract than traditional *continuations*. We'll discuss it in **??**.

The C10K and C10M problem has told us that the performance is not the only thing to concern for high concurrent server, but also the scalability. The old method to use select() or poll() has $O(n)$ complexity, which is the handicap to hold more connections [?]. 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 **??**.

The websocket is a protocol for two-way communication between client and server over TCP connection. The client is, in a broad sense, unnecessary to be a web browser. One of the benefit of websocket is that the developers could take advantage of TLS used in HTTPS for encryption. And another is that unified listening port for arbitrary services. We will discuss websocket in **??**.

Finally, we'll discuss some of the notable features of GNU Artanis in **??**.

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

2. Some background knowledge

It is simplified to explain *continuations* and *delimited continuations* in continuation passing style (CPS), and we'll show how to transfer their definition in semantics to Scheme code naturally for practical programming.

2.1 First Class Continuations

A *continuation* is "the rest of the computation". It is often referred as *call/cc* standing for *call-with-current-continuation*, which 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 trivial *continuation* at the time point 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 control flow 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 spirit of CPS [?][?].

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) Shouldn't introduce any administrative redex (the reducible expression operated by a continuation capturing lambda); (2) Wouldn't 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 that a function from values to values. In spirit of CPS, the *λ–abstract* in Figure **??** 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 **??**:

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

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)
```

Apparently, 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 that could be ignored. The ρ is dismissed since the environment is managed by Scheme inexplicitly.

In most of the Scheme implementations, $call/cc$ is rarely in CPS which helped us to analysis the control flow 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 the *continuation* will be captured and threw by 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 overkill for most other 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" [?].

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

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

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's 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 won't be captured that reduced the cost. It is very different from *full-continuation* capturing in **call/cc**. This feature would solve the "overkilled" problem we've mentioned in the end of last section.

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

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

According to the test results, it shows that the direct implement approach is fabulously faster than CPS transforming way [?]. Nevertheless, there could possibly be further optimizing of stack/heap management and copying methods to make it better.

There're three equivalent terms of interfaces to handle *delimited continuations* in GNU Guile. In spite of the common term **shift/reset**, **%/abort**, and **call-with-prompt/abort-to-prompt**. The only difference is their operational approach. It depends on you and your need to decide how to choose from them.

We choose **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'll show it in **??**. A common usage of it could be this:

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

2.2 Co-routines

Co-routines are essentially procedures which save states between calls. It has become very important form of concurrency, and avoid practical difficulties (race conditions, deadlock, etc) to reduce complexities. Developers are unnecessary to take care of synchronization by themselves, but leave it to this paradigm as its built-in feature. Co-routines is not generic enough to solve any concurrency problem, however, it's idea solution for server-side development.

It is demonstrated that co-routines may easily be defined by users given first class continuations[**?**]. The term "first class" here means a continuation 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 spite of 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 to the time when next packet arrives. In old fashion, people uses OS level threads (say, pthread) to avoid blocking. But it brings some critical overheads (trap into kernel, locks, synchronization), no mention the difficulties in debugging programs with threads. Ironically, it is even complained by people that threads model is bad idea for practices [**?**].

Since Scheme provides full support for continuations, implementing co-routines is nearly trivial, requiring only that a queue of continuations be maintained. Moreover, this paper introduce the way to implement co-routines with *delimited continuations*. We'll see how this approach is convenient and clean later in this section.

As described in **??**, 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's 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's common to spawn a new co-routine. The *call-with-prompt* function was introduced in **??**.

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

As described in **??**, 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 for 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's 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 certain condition were met, the related *delimited continuation* would be resumed. The *resume* function will resume the task properly. Note that the resumed continuation should be re-delimit again to avoid stack issues. And 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're in similar principle and easier to understand. For now, it's time to use them for requests handling.

```
(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))
```

Listing 1: Co-routine handling requests

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

```
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!
```

Listing 2: Coroutines running result

3. Server Core Design

Having finally learned about this powerful weapon named coroutines, we may consider how to bleed it out for the potentiality of a server program. We'll reasonably show the server core design in both functionality and performance concern.

The websocket protocol plays an important role in the functionality.

As is the case of the performance concern, non-blocking and edge-triggered I/O multiplexing, as we'll describe it in GNU/Linux environment, epoll() will be discussed in this case.

3.1 Websocket

Although the term websocket 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 pack and forth while keeping the connection open. This approach keeps a bi-directional ongoing conversation taking place between client (most of the time it's a browser) and the server, beyond, websocket provides full-duplex communication. And all the communications are done over TCP port 80, which is of benefit for those environments which block non-web Internet connections behind a firewall. This makes all the connections long living session rather than short session 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.

Another benefit to transport data over HTTP is that many existing HTTP security mechanisms also apply to websocket. With this unified security model, a list of standard HTTP security methods could be applied to a websocket connection. For example, the same encryption as HTTPS using TLS/SSL. It's 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. Websocket Secure (WSS) use the exactly the same way with handshake in HTTP, then upgrade to websocket protocol.

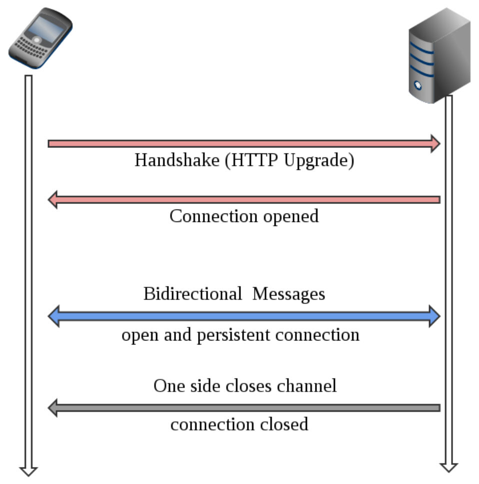

Figure 2: Websocket handshake and communication

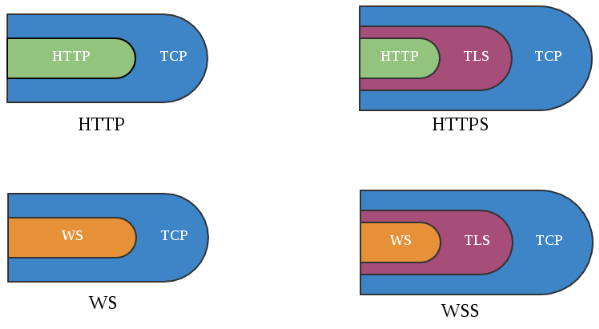

Figure 3: HTTPS and Websocket Secure

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's transparent in many server program.

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

The Figure **??** shows the HTTP Gateway architecture in GNU Artanis. It provides arbitrary protocol services over the websocket. Beyond the web server, it becomes a *generic server*, which handles connections of various protocols in the same time. A *generic server* seems useless and meaningless in the past. Because most of the application of server programs are for dedicated protocol, and to provide single service as its main duty.

For the *dedicated server*, FTP, HTTP as in web service, game server, IRC, etc. Usually, the remote server starts a service program as a daemon listening on a TCP/UDP port according to a convention, and provides just one service in a dedicated protocol. Such a decoupled pattern is obviously meets the KISS (Keep It Stupid and Simple) principle. Moreover, when one service is down, it won't effect the others.

For the *generic server*, people don't have to listen on many TCP ports, there's just one port, 80 or 443 (for HTTPS). And the

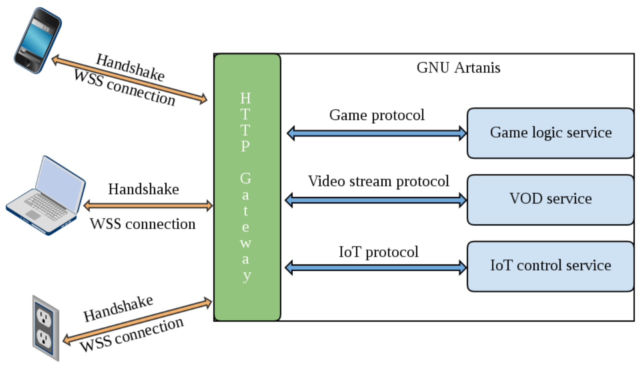

Figure 4: HTTP Gateway architecture in GNU Artanis

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 way, all the services are running in GNU Artanis rather than standalone, and it's good for both quick developing and deployment. But it's tight coupled, if one of the services is broken, the whole GNU Artanis may down. Fortunately, it is allowed to make each service standalone, similar to CGI, but in bytevector way according to websocket configuration, rather than HTTP requests redirecting.

So, what's 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'll be many IoT nodes as the massive clients, with various sensors or monitors, which may require various application level protocols, although their messaging protocol maybe the same (MQTT over websocket). For *generic server* reduces the workload of customizing and management of various protocols, and IoT based applications are well-known to be the next big thing, the *generic server* may act important role in next decades.

3.3 Concurrency

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

No matter how the number is increasing, the main purpose is to hold more connections concurrently as possible. Unfortunately, it is useless if one just purchase stronger machines with more rams. In this way, the performance of single connection processing could be higher, but increase few connections concurrently.

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

Beyond, 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 the optimizing of the language implementation, and better exception handling for robustness of the server system.

Of course, it's necessary to provide advanced scaling methodology for higher concurrency need to meet the future cloud computing. But it's out of our purpose in this paper.

4. Some features in GNU Artanis

In spite of the server core, there're 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.

To the extent that systems conform to the constraints of REST they can be called RESTful. It is often communicate over HTTP with the same method name (GET, POST, PUT, DELETE, etc.) that web browsers use to retrieve web pages and to send data to remote servers. 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. And there'll be more than one interfaces with the *Model*, they're called *Views*. The *Views* could be GUI, CLI or API. Although *Views* are very often graphical, they dont have to be. A *Controller* is an object that lets you manipulate a *View*. In a brief, *Controller* handles the input whilst the view handles the output [**?**].

GNU Artanis provides CLI tools for generating MVC template code, this will be introduced in **??**.

4.3 Relational Mapping

Usually, Relational Mapping (RM) stands for ORM (Object Relational Mapping). 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 choose not to use OO for programming. It is enough to use the features of Functional Programming (FP) to replace the essentials in OO, and it's proved in GNU Artanis development. Because of this reason, GNU Artanis doesn't implement ORM, but closures to replace classes, and message passing for dispatching the methods to mimic half-baked OO, which is more lightweight and less complexity than OO. It is called Functional Programming Relational Mapping (FPRM) in GNU Artanis.

4.4 Sessions

HTTP sessions allows 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're 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 a fashion for most of the 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 developers work:

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

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

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

```
art migrate operator name
```

The 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, and very high scalability with serializable continuations to scale the whole server system by adding infinite nodes. These issues are open to be discussed and practiced in the future.

Acknowledgments

First, thanks free software movement and its community for all the contributions.

Thanks SZDIY community for providing all the conditions, and all the kindly encouraging. And I would like to thank my second daughter who is only 18 days. It came suddenly in advanced that disorganized my writing plan. Fortunately, I have strong server core to handle these two concurrent tasks, exhausted.