Yars: Multi-Threaded Application Server for Ruby

April 27th, 2015

Overview

Yars is a multi-threaded, non-blocking cache/web-server in Ruby. It is compliant with the Ruby rack framework so it can act as a server for robust frameworks such as Ruby on Rails and Sinatra. As such, it is very important to ensure performance, concurrency, and integrity of data. Fortunately, the client-server model of computing is an inherently parallel form of computation. Generally, the amount of state shared between different clients and requests is small and can be easily isolated.

High-Level Implementation

The basic (high-level) mechanism for the server is this:

1. It takes HTTP requests in a nonblocking manner.
2. Checks to see if the request has been made before.
3. Responds with the cached resource if it has been sent before 4. If the request is new, then it will forward the request to the application 5. The application decides what to do with the request, and dynamically creates a response 6. The response is sent from the application to the client 7. Our server will then close the HTTP transaction with an HTTP response.

Parallelization challenges

The parallelization challenges involve thread-safe use of the request-response cache, and the queue used to store pending requests. Both of these data structures represent hotspots or critical sections in the application, and will potentially serve thousands of clients at once. Therefore it is crucial to minimize any sequential bottlenecks found in the implementation, while making sure that all clients are served in a first-in-first-out ordering.

Yars::RequestQueue

The concurrency pattern for the Yars queue is known as the Producer-Consumer model. In our application, we have two concepts which equate to Producers and Consumers. They are called Frontend workers and Backend Workers, respectively.

The Frontend workers open a TCPSocket listening on a port of choice, upon receiving a request, a thread is started which pushes data to to the queue and subsequently notifies the Backend workers of the fact that there are now clients to serve. The Backend Workers safely pop data from the queue and begin reading the request buffer.

The queue built for Yars is a unique design which is somewhat of a combination of the concepts from a Bounded Partial Queue and an Unbounded Total Queue. Here are some of the requirements for the queue and how they affected the design choices for Yars:

1. Firstly, it is neccesary that this queue be unbounded since it is unknown how many clients may need to be serviced at any given time.
2. We want to minimize the amount of locking required to push and pop from the queue.
3. We, however, want to benefit of of being able to sleep threads which have no work to do. This way we wont waste valuable thread resources spinning and waiting for work when they could be used for other services (database, etc.). Therefore, we want to be able to use condition variables to signal that there is work to be done.

As a result our queue is lock-free in the case that there is work on the queue; however, when there is no work on the queue, we have the threads acquire a lock solely for the purpose of waiting on a condition variable. This gives us the benefits of being lock free for the majority of the time while keeping the useful signalling properties of locks and condition variables.

The implementation can be seen in `lib/yars/request_queue.rb`.

Yars::ConcurrentCache

If the request has been made before, we assume the response will always be the same (this is an immutable server, if you will). It is important to note that in many cases this property will not be desired; but in certain cases it is wise to cache certain requests for static material. This option is configurable by setting a Header in the HTTP response object known as an E-Tag. When Yars sees this E-Tag, it can safely cache the request response.

The concurrency pattern for the cache is known as the Readers-Writers model. All Backend Workers access the shared cache at the same time, some reading and some writing, with the constraint that no thread may access the cache for reading or writing while another process is in the act of writing to it.

To implement the cache, we used the concept of a Refined Striped Hash Set. The way this hash works is by creating a creating two lists: `@table` is a table of lists which key-value objects are stored, `@locks` is an array of re-entrant locks used to lock access to each bucket. When Yars caches a response, all it does is attempt to write the response:request key-value pair to the concurrent hash table; and upon reading a request it attempts to lookup they request key in the hash set.

In this hash set, we use an atomic markable reference which indicates an owner thread and a boolean representing whether or not a resize is in progress. If there is not resize in progress, we can acquire the lock, hash the value to determine which bucket our value is in, then linearly search the list for the value. Afterwards we release the locks.

The interesting about this structure is that it is able to adjust the number of locks as the number of entries grow. Therefore, as the structure gets larger we are actually reduce the amount of locking that is needed as the probability of any given item being in any single bucket reduces.

The implementation can be seen in `lib/yars/concurent_hash_set.rb`.

Non-Blocking IO

Since Ruby is an interpreted language, we will need to squeeze out every ounce of performance in the implementation. Since servers are typically sending data back and forth with clients, they are typically IO bound, meaning they are limited by the sending data over a wire rather than expensive CPU calculations.

In YARS, all IO operations with the client are done in a non-blocking and asynchronous manner. This ensures that threads do not get stuck waiting on the kernel to send the request and response bytes to the client.

Results

IO Bound Tasks

Table 1: Small (14byte) load

Concurrency	Requests/sec	Transfers/sec
2	5238.91	608.82KB
4	4094.25	475.80KB
6	1055.15	122.62KB
8	2409.62	280.02KB
12	5502.08	639.40KB
16	5220.45	606.67KB
24	2662.31	309.39KB
32	6209.57	721.62KB
64	4691.21	545.17KB

Table 2: Medium (100,000byte) load

Concurrency	Requests/sec	Transfers/sec
2	1865.19	1.76GB
4	1725.89	1.63GB
6	1661.79	1.56GB
8	1718.24	1.62GB
12	1619.92	1.52GB
16	1603.94	1.51GB
24	1468.67	1.42GB
32	1085.72	1.03GB
64	1021.02	1.00GB

Table 3: Large (1,000,000 byte) load

Concurrency	Requests/sec	Transfers/sec
2	1865.19	1.76GB
4	1725.89	1.63GB
6	1661.79	1.56GB
8	1718.24	1.62GB
12	1619.92	1.52GB
16	1603.94	1.51GB
24	1468.67	1.42GB
32	1085.72	1.03GB
64	1021.02	1.00GB

These sets of experiments represent the IO bound tasks in the YARS server. It may seem odd, but the data shows that increasing the number of threads does not actually increase performance or throughput within the system. However, this behavior is expected with how YARS is structured. Since it is a non-blocking server, threads do not have to wait to perform IO. Therefore, there is no benefit in having multiple threads for IO tasks since it does not allow for significantly more parallelization. In fact, having more threads linearly decreases performance with IO. The fundamental reason for this is that threads must be scheduled and maintained by the interpreter, and every 10ms a new thread is scheduled for its quanta on the CPU. This makes maintaining threads for IO tasks not particularly useful. In fact, we have seen a right of single threaded web-servers which follow the non-blocking paradigm (Node.js, etc.) in recent years.

CPU Bound Tasks

Table 4: fibonacci(25); caching: true

Concurrency	Requests/sec	Transfers/sec
2 504	8.07 433.	82KB
4 456	9.89 392.	72KB
6 295	4.32 253.	89KB
8 399	8.10 345.	21KB
12 122	9.38 105.	65KB
16 305	7.86 262.	78KB
24 345	7.86 308.	78KB
32 427	6.24 367.	49KB
64 404	2.12 347.	37KB

Table 5: fibonacci(27); caching: true

Concurrency	Requests/sec	Transfers/sec
2	4030.29	346.35KB
4	3962.40	340.52KB
6	272.32	23.40KB
8	2259.60	194.18KB
12	2552.31	198.94KB
16	885.61	76.11KB
24	277.50	23.85KB
32	2361.84	202.97KB
64	1255.45	107.89KB

Table 6: fibonacci(24); caching: false

Concurrency	Requests/sec	Transfers/sec
2	66.90	5.68KB
4	57.15	4.86KB
6	53.08	4.51KB
8	45.15	3.84KB
12	50.05	4.25KB
16	48.72	4.14KB
24	49.71	4.22KB
32	49.98	4.25KB
64	51.48	4.37KB

These results were truly confounding to me when I first saw them. One would assume that having having more threads in a computational task equate to faster running times in accordance with Ahmdahl's law. However, we have shown in Ruby that this is *not* the case. Why is that?

When pondering some of my results, I researched and found out that Ruby uses a Global Interpreter lock to control execution of the code. This means that when the interpreter is interpreting a line of Ruby code, it cannot execute Ruby code from another thread. This is a *huge* source of sequential bottlenecking. There are other implementations of Ruby now which were created for the sole purpose of addressing this issue (Jruby and Rubinius).

With the inability to evaluate code simultaneously, the number of threads you use is not longer relevant, and the number of *threads of execution*, otherwise known as N in Ahmdahl's law, becomes 1.

However, it must be noted that although threads cannot execute code simultaneously, the code is not guaranteed to be thread safe. In fact, you are still subject to almost all of the issues of concurrent programming, with nearly none of the benefits.

Discussion

Challenges

When writing this application, there were a number issues I faced that were both new and challenging, forcing me to learn to concepts outside of my comfort zone. I will address a few of these challenges in the next section.

State of Concurrency in Ruby

Concurrency in Ruby is a touchy subject. The design principles of the language were more focused on elegance of the language and object oriented design than robust concurrency support. The language designer himself, Matz, does not consider himself a concurrency expert. Needless to say, there is some work that needs to be done with respect for concurrency in the language. The most challenging aspect for me was the lack of concurrency primitives in the language. For example, there is no native AtomicReference, and I had to find a library to support the features I needed.

My Open Source Contribution to Concurrent-Ruby

Even when I found the library that I needed it *still* did not have everything that I wanted. As I result, I decided to take matters into my own hands and support open-source software that I benefit from daily.

In order to implement the ConcurrentHashSet I needed the AtomicMarkableReference primitive. This primitive allowed me to keep track of the owner of a hash table, so I could check to see if there was currently any resize operations in progress.

I actually took the time to properly implement the idea and send it upstream to concurrent-ruby. As of now it is a pure Ruby implementation of the primitive, but I plan to create bindings for it in C and Java.

You can see my pull-request here

Conclusion

In conclusion, I have discovered the importance of understanding the underlying implementations of your operating system and language when designing a concurrent application. You must understand how your language with do threading in order to have upfront guarantees about the nature of your application.

Overall, I think designing a server it is of utmost importance to maintain both non-blocking I/O, and parallel computation. However, the requirements for every differ, so in some cases you can have one and not the other. For example, I think Yars would be well-suited for tasks which serve static data to a client, or any task that is heavily IO bound. This is because the server responds quickly under those conditions. However, it would not be suited for an application where intensive parallel processing is needed. It would be optimal to implement those behaviors in a language with access to lower level concurrency primitives.