nginx

Andrew Alexeev

nginx (pronounced "engine x") is a free open source web server written by Igor Sysoev, a Russian software engineer. Since its public launch in 2004, nginx has focused on high performance, high concurrency and low memory usage. Additional features on top of the web server functionality, like load balancing, caching, access and bandwidth control, and the ability to integrate efficiently with a variety of applications, have helped to make nginx a good choice for modern website architectures. Currently nginx is the second most popular open source web server on the Internet.

14.1. Why Is High Concurrency Important?

These days the Internet is so widespread and ubiquitous it's hard to imagine it wasn't exactly there, as we know it, a decade ago. It has greatly evolved, from simple HTML producing clickable text, based on NCSA and then on Apache web servers, to an always-on communication medium used by more than 2 billion users worldwide. With the proliferation of permanently connected PCs, mobile devices and recently tablets, the Internet landscape is rapidly changing and entire economies have become digitally wired. Online services have become much more elaborate with a clear bias towards instantly available live information and entertainment. Security aspects of running online business have also significantly changed. Accordingly, websites are now much more complex than before, and generally require a lot more engineering efforts to be robust and scalable.

One of the biggest challenges for a website architect has always been concurrency. Since the beginning of web services, the level of concurrency has been continuously growing. It's not uncommon for a popular website to serve hundreds of thousands and even millions of simultaneous users. A decade ago, the major cause of concurrency was slow clients—users with ADSL or dial-up connections. Nowadays, concurrency is caused by a combination of mobile clients and newer application architectures which are typically based on maintaining a persistent connection that allows the client to be updated with news, tweets, friend feeds, and so on. Another important factor contributing to increased concurrency is the changed behavior of modern browsers, which open four to six simultaneous connections to a website to improve page load speed.

To illustrate the problem with slow clients, imagine a simple Apache-based web server which produces a relatively short 100 KB response—a web page with text or an image. It can be merely a fraction of a second to generate or retrieve this page, but it takes 10 seconds to transmit it to a client with a bandwidth of 80 kbps (10 KB/s). Essentially, the web server would relatively quickly pull 100 KB of content, and then it would be busy for 10 seconds slowly sending this content to the client before freeing its connection. Now imagine that you have 1,000 simultaneously connected clients who have requested similar content. If only 1 MB of additional memory is allocated per client, it would result in 1000 MB (about 1

GB) of extra memory devoted to serving just 1000 clients 100 KB of content. In reality, a typical web server based on Apache commonly allocates more than 1 MB of additional memory per connection, and regrettably tens of kbps is still often the effective speed of mobile communications. Although the situation with sending content to a slow client might be, to some extent, improved by increasing the size of operating system kernel socket buffers, it's not a general solution to the problem and can have undesirable side effects.

With persistent connections the problem of handling concurrency is even more pronounced, because to avoid latency associated with establishing new HTTP connections, clients would stay connected, and for each connected client there's a certain amount of memory allocated by the web server.

Consequently, to handle the increased workloads associated with growing audiences and hence higher levels of concurrency—and to be able to continuously do so—a website should be based on a number of very efficient building blocks. While the other parts of the equation such as hardware (CPU, memory, disks), network capacity, application and data storage architectures are obviously important, it is in the web server software that client connections are accepted and processed. Thus, the web server should be able to scale nonlinearly with the growing number of simultaneous connections and requests per second.

Isn't Apache Suitable?

Apache, the web server software that still largely dominates the Internet today, has its roots in the beginning of the 1990s. Originally, its architecture matched the then-existing operating systems and hardware, but also the state of the Internet, where a website was typically a standalone physical server running a single instance of Apache. By the beginning of the 2000s it was obvious that the standalone web server model could not be easily replicated to satisfy the needs of growing web services. Although Apache provided a solid foundation for future development, it was architected to spawn a copy of itself for each new connection, which was not suitable for nonlinear scalability of a website. Eventually Apache became a general purpose web server focusing on having many different features, a variety of third-party extensions, and universal applicability to practically any kind of web application development. However, nothing comes without a price and the downside to having such a rich and universal combination of tools in a single piece of software is less scalability because of increased CPU and memory usage per connection.

Thus, when server hardware, operating systems and network resources ceased to be major constraints for website growth, web developers worldwide started to look around for a more efficient means of running web servers. Around ten years ago, Daniel Kegel, a prominent software engineer, proclaimed that "it's time for web servers to handle ten thousand clients simultaneously" and predicted what we now call Internet cloud services. Kegel's C10K manifest spurred a number of attempts to solve the problem of web server optimization to handle a large number of clients at the same time, and nginx turned out to be one of the most successful ones.

Aimed at solving the C10K problem of 10,000 simultaneous connections, nginx was written with a different architecture in mind—one which is much more suitable for nonlinear scalability in both the number of simultaneous connections and requests per second. nginx is event-based, so it does not follow Apache's style of spawning new processes or threads for each web page request. The end result is that even as load increases, memory and CPU usage remain manageable. nginx can now deliver tens of thousands of concurrent connections on a server with typical hardware.

When the first version of nginx was released, it was meant to be deployed alongside Apache such that static content like HTML, CSS, JavaScript and images were handled by nginx to offload concurrency and latency processing from Apache-based application servers. Over the course of its development, nginx has

added integration with applications through the use of FastCGI, uswgi or SCGI protocols, and with distributed memory object caching systems like *memcached*. Other useful functionality like reverse proxy with load balancing and caching was added as well. These additional features have shaped nginx into an efficient combination of tools to build a scalable web infrastructure upon.

In February 2012, the Apache 2.4.x branch was released to the public. Although this latest release of Apache has added new multi-processing core modules and new proxy modules aimed at enhancing scalability and performance, it's too soon to tell if its performance, concurrency and resource utilization are now on par with, or better than, pure event-driven web servers. It would be very nice to see Apache application servers scale better with the new version, though, as it could potentially alleviate bottlenecks on the backend side which still often remain unsolved in typical nginx-plus-Apache web configurations.

Are There More Advantages to Using nginx?

Handling high concurrency with high performance and efficiency has always been the key benefit of deploying nginx. However, there are now even more interesting benefits.

In the last few years, web architects have embraced the idea of decoupling and separating their application infrastructure from the web server. However, what would previously exist in the form of a LAMP (Linux, Apache, MySQL, PHP, Python or Perl)-based website, might now become not merely a LEMP-based one (`E' standing for `Engine x'), but more and more often an exercise in pushing the web server to the edge of the infrastructure and integrating the same or a revamped set of applications and database tools around it in a different way.

nginx is very well suited for this, as it provides the key features necessary to conveniently offload concurrency, latency processing, SSL (secure sockets layer), static content, compression and caching, connections and requests throttling, and even HTTP media streaming from the application layer to a much more efficient edge web server layer. It also allows integrating directly with memcached/Redis or other "NoSQL" solutions, to boost performance when serving a large number of concurrent users.

With recent flavors of development kits and programming languages gaining wide use, more and more companies are changing their application development and deployment habits. nginx has become one of the most important components of these changing paradigms, and it has already helped many companies start and develop their web services quickly and within their budgets.

The first lines of nginx were written in 2002. In 2004 it was released to the public under the two-clause BSD license. The number of nginx users has been growing ever since, contributing ideas, and submitting bug reports, suggestions and observations that have been immensely helpful and beneficial for the entire community.

The nginx codebase is original and was written entirely from scratch in the C programming language. nginx has been ported to many architectures and operating systems, including Linux, FreeBSD, Solaris, Mac OS X, AIX and Microsoft Windows. nginx has its own libraries and with its standard modules does not use much beyond the system's C library, except for zlib, PCRE and OpenSSL which can be optionally excluded from a build if not needed or because of potential license conflicts.

A few words about the Windows version of nginx. While nginx works in a Windows environment, the Windows version of nginx is more like a proof-of-concept rather than a fully functional port. There are certain limitations of the nginx and Windows kernel architectures that do not interact well at this time. The known issues of the nginx version for Windows include a much lower number of concurrent connections, decreased performance, no caching and no bandwidth policing.

Future versions of nginx for Windows will match the mainstream functionality more closely.

14.2. Overview of nginx Architecture

Traditional process- or thread-based models of handling concurrent connections involve handling each connection with a separate process or thread, and blocking on network or input/output operations. Depending on the application, it can be very inefficient in terms of memory and CPU consumption. Spawning a separate process or thread requires preparation of a new runtime environment, including allocation of heap and stack memory, and the creation of a new execution context. Additional CPU time is also spent creating these items, which can eventually lead to poor performance due to thread thrashing on excessive context switching. All of these complications manifest themselves in older web server architectures like Apache's. This is a tradeoff between offering a rich set of generally applicable features and optimized usage of server resources.

From the very beginning, nginx was meant to be a specialized tool to achieve more performance, density and economical use of server resources while enabling dynamic growth of a website, so it has followed a different model. It was actually inspired by the ongoing development of advanced event-based mechanisms in a variety of operating systems. What resulted is a modular, event-driven, asynchronous, single-threaded, non-blocking architecture which became the foundation of nginx code.

nginx uses multiplexing and event notifications heavily, and dedicates specific tasks to separate processes. Connections are processed in a highly efficient run-loop in a limited number of single-threaded processes called `worker`s. Within each `worker` nginx can handle many thousands of concurrent connections and requests per second.

Code Structure

The nginx `worker` code includes the core and the functional modules. The core of nginx is responsible for maintaining a tight run-loop and executing appropriate sections of modules' code on each stage of request processing. Modules constitute most of the presentation and application layer functionality. Modules read from and write to the network and storage, transform content, do outbound filtering, apply server-side include actions and pass the requests to the upstream servers when proxying is activated.

nginx's modular architecture generally allows developers to extend the set of web server features without modifying the nginx core. nginx modules come in slightly different incarnations, namely core modules, event modules, phase handlers, protocols, variable handlers, filters, upstreams and load balancers. At this time, nginx doesn't support dynamically loaded modules; i.e., modules are compiled along with the core at build stage. However, support for loadable modules and ABI is planned for the future major releases. More detailed information about the roles of different modules can be found in Section 14.4.

While handling a variety of actions associated with accepting, processing and managing network connections and content retrieval, nginx uses event notification mechanisms and a number of disk I/O performance enhancements in Linux, Solaris and BSD-based operating systems, like `kqueue`, `epoll`, and `event ports`. The goal is to provide as many hints to the operating system as possible, in regards to obtaining timely asynchronous feedback for inbound and outbound traffic, disk operations, reading from or writing to sockets, timeouts and so on. The usage of different methods for multiplexing and advanced I/O operations is heavily optimized for every Unix-based operating system nginx runs on.

A high-level overview of nginx architecture is presented in Figure 14.1.

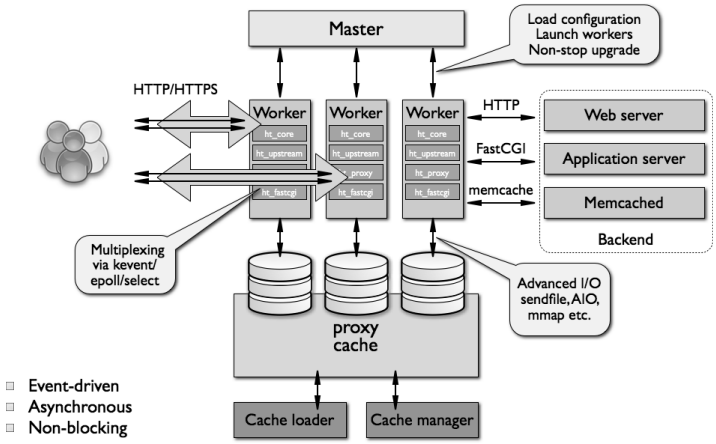

Figure 14.1: Diagram of nginx's architecture

Workers Model

As previously mentioned, nginx doesn't spawn a process or thread for every connection. Instead, `worker` processes accept new requests from a shared "listen" socket and execute a highly efficient run-loop inside each `worker` to process thousands of connections per `worker`. There's no specialized arbitration or distribution of connections to the `worker`s in nginx; this work is done by the OS kernel mechanisms. Upon startup, an initial set of listening sockets is created. `worker`s then continuously accept, read from and write to the sockets while processing HTTP requests and responses.

The run-loop is the most complicated part of the nginx `worker` code. It includes comprehensive inner calls and relies heavily on the idea of asynchronous task handling. Asynchronous operations are implemented through modularity, event notifications, extensive use of callback functions and fine-tuned timers. Overall, the key principle is to be as non-blocking as possible. The only situation where nginx can still block is when there's not enough disk storage performance for a `worker` process.

Because nginx does not fork a process or thread per connection, memory usage is very conservative and extremely efficient in the vast majority of cases. nginx conserves CPU cycles as well because there's no ongoing create-destroy pattern for processes or threads. What nginx does is check the state of the network and storage, initialize new connections, add them to the run-loop, and process asynchronously until completion, at which point the connection is deallocated and removed from the run-loop. Combined with the careful use of `syscall`s and an accurate implementation of supporting interfaces like pool and slab memory allocators, nginx typically achieves moderate-to-low CPU usage even under extreme workloads.

Because nginx spawns several `worker`s to handle connections, it scales well across multiple cores. Generally, a separate `worker` per core allows full utilization of multicore architectures, and prevents thread thrashing and lock-ups. There's no resource starvation and the resource controlling mechanisms are isolated within single-threaded `worker` processes. This model also allows more scalability across physical storage devices, facilitates more disk utilization and avoids blocking on disk I/O. As a result, server resources are utilized more efficiently with the workload shared across several workers.

With some disk use and CPU load patterns, the number of nginx `worker`s should be adjusted. The rules are somewhat basic here, and system administrators should try a couple of configurations for their workloads. General recommendations might be the following: if the load pattern is CPU intensive—for instance, handling a lot of TCP/IP, doing SSL, or compression—the number of

nginx `worker`s should match the number of CPU cores; if the load is mostly disk I/O bound—for instance, serving different sets of content from storage, or heavy proxying—the number of `worker`s might be one and a half to two times the number of cores. Some engineers choose the number of `worker`s based on the number of individual storage units instead, though efficiency of this approach depends on the type and configuration of disk storage.

One major problem that the developers of nginx will be solving in upcoming versions is how to avoid most of the blocking on disk I/O. At the moment, if there's not enough storage performance to serve disk operations generated by a particular `worker`, that `worker` may still block on reading/writing from disk. A number of mechanisms and configuration file directives exist to mitigate such disk I/O blocking scenarios. Most notably, combinations of options like sendfile and AIO typically produce a lot of headroom for disk performance. An nginx installation should be planned based on the data set, the amount of memory available for nginx, and the underlying storage architecture.

Another problem with the existing `worker` model is related to limited support for embedded scripting. For one, with the standard nginx distribution, only embedding Perl scripts is supported. There is a simple explanation for that: the key problem is the possibility of an embedded script to block on any operation or exit unexpectedly. Both types of behavior would immediately lead to a situation where the worker is hung, affecting many thousands of connections at once. More work is planned to make embedded scripting with nginx simpler, more reliable and suitable for a broader range of applications.

nginx Process Roles

nginx runs several processes in memory; there is a single master process and several `worker` processes. There are also a couple of special purpose processes, specifically a cache loader and cache manager. All processes are single-threaded in version 1.x of nginx. All processes primarily use shared-memory mechanisms for inter-process communication. The master process is run as the `root` user. The cache loader, cache manager and `worker`s run as an unprivileged user.

The master process is responsible for the following tasks:

- reading and validating configuration
- creating, binding and closing sockets
- starting, terminating and maintaining the configured number of `worker` processes
- reconfiguring without service interruption
- controlling non-stop binary upgrades (starting new binary and rolling back if necessary)
- re-opening log files
- compiling embedded Perl scripts

The `worker` processes accept, handle and process connections from clients, provide reverse proxying and filtering functionality and do almost everything else that nginx is capable of. In regards to monitoring the behavior of an nginx instance, a system administrator should keep an eye on `worker`s as they are the processes reflecting the actual day-to-day operations of a web server.

The cache loader process is responsible for checking the on-disk cache items and populating nginx's in-memory database with cache metadata. Essentially, the cache loader prepares nginx instances to work with files already stored on disk in a specially allocated directory structure. It traverses the directories, checks cache content metadata, updates the relevant entries in shared memory and then exits when everything is clean and ready for use.

The cache manager is mostly responsible for cache expiration and invalidation. It

stays in memory during normal nginx operation and it is restarted by the master process in the case of failure.

Brief Overview of nginx Caching

Caching in nginx is implemented in the form of hierarchical data storage on a filesystem. Cache keys are configurable, and different request-specific parameters can be used to control what gets into the cache. Cache keys and cache metadata are stored in the shared memory segments, which the cache loader, cache manager and `worker`s can access. Currently there is not any in-memory caching of files, other than optimizations implied by the operating system's virtual filesystem mechanisms. Each cached response is placed in a different file on the filesystem. The hierarchy (levels and naming details) are controlled through nginx configuration directives. When a response is written to the cache directory structure, the path and the name of the file are derived from an MD5 hash of the proxy URL.

The process for placing content in the cache is as follows: When nginx reads the response from an upstream server, the content is first written to a temporary file outside of the cache directory structure. When nginx finishes processing the request it renames the temporary file and moves it to the cache directory. If the temporary files directory for proxying is on another file system, the file will be copied, thus it's recommended to keep both temporary and cache directories on the same file system. It is also quite safe to delete files from the cache directory structure when they need to be explicitly purged. There are third-party extensions for nginx which make it possible to control cached content remotely, and more work is planned to integrate this functionality in the main distribution.

14.3. nginx Configuration

nginx's configuration system was inspired by Igor Sysoev's experiences with Apache. His main insight was that a scalable configuration system is essential for a web server. The main scaling problem was encountered when maintaining large complicated configurations with lots of virtual servers, directories, locations and datasets. In a relatively big web setup it can be a nightmare if not done properly both at the application level and by the system engineer himself.

As a result, nginx configuration was designed to simplify day-to-day operations and to provide an easy means for further expansion of web server configuration.

nginx configuration is kept in a number of plain text files which typically reside in `/usr/local/etc/nginx` or `/etc/nginx`. The main configuration file is usually called `nginx.conf`. To keep it uncluttered, parts of the configuration can be put in separate files which can be automatically included in the main one. However, it should be noted here that nginx does not currently support Apache-style distributed configurations (i.e., `.htaccess` files). All of the configuration relevant to nginx web server behavior should reside in a centralized set of configuration files.

The configuration files are initially read and verified by the master process. A compiled read-only form of the nginx configuration is available to the `worker` processes as they are forked from the master process. Configuration structures are automatically shared by the usual virtual memory management mechanisms.

nginx configuration has several different contexts for `main`, `http`, `server`, `upstream`, `location` (and also `mail` for mail proxy) blocks of directives. Contexts never overlap. For instance, there is no such thing as putting a `location` block in the `main` block of directives. Also, to avoid unnecessary ambiguity there isn't anything like a "global web server" configuration. nginx configuration is meant to be clean and logical, allowing users to maintain complicated configuration files that comprise thousands of directives. In a private conversation, Sysoev said, "Locations, directories, and other blocks in the global

server configuration are the features I never liked in Apache, so this is the reason why they were never implemented in nginx."

Configuration syntax, formatting and definitions follow a so-called C-style convention. This particular approach to making configuration files is already being used by a variety of open source and commercial software applications. By design, C-style configuration is well-suited for nested descriptions, being logical and easy to create, read and maintain, and liked by many engineers. C-style configuration of nginx can also be easily automated.

While some of the nginx directives resemble certain parts of Apache configuration, setting up an nginx instance is quite a different experience. For instance, rewrite rules are supported by nginx, though it would require an administrator to manually adapt a legacy Apache rewrite configuration to match nginx style. The implementation of the rewrite engine differs too.

In general, nginx settings also provide support for several original mechanisms that can be very useful as part of a lean web server configuration. It makes sense to briefly mention variables and the `try_files` directive, which are somewhat unique to nginx. Variables in nginx were developed to provide an additional even-more-powerful mechanism to control run-time configuration of a web server. Variables are optimized for quick evaluation and are internally pre-compiled to indices. Evaluation is done on demand; i.e., the value of a variable is typically calculated only once and cached for the lifetime of a particular request. Variables can be used with different configuration directives, providing additional flexibility for describing conditional request processing behavior.

The `try_files` directive was initially meant to gradually replace conditional `if` configuration statements in a more proper way, and it was designed to quickly and efficiently try/match against different URI-to-content mappings. Overall, the `try_files` directive works well and can be extremely efficient and useful. It is recommended that the reader thoroughly check the `try_files` directive and adopt its use whenever applicable.

14.4. nginx Internals

As was mentioned before, the nginx codebase consists of a core and a number of modules. The core of nginx is responsible for providing the foundation of the web server, web and mail reverse proxy functionalities; it enables the use of underlying network protocols, builds the necessary run-time environment, and ensures seamless interaction between different modules. However, most of the protocol- and application-specific features are done by nginx modules, not the core.

Internally, nginx processes connections through a pipeline, or chain, of modules. In other words, for every operation there's a module which is doing the relevant work; e.g., compression, modifying content, executing server-side includes, communicating to the upstream application servers through FastCGI or uwsgi protocols, or talking to memcached.

There are a couple of nginx modules that sit somewhere between the core and the real "functional" modules. These modules are `http` and `mail`. These two modules provide an additional level of abstraction between the core and lower-level components. In these modules, the handling of the sequence of events associated with a respective application layer protocol like HTTP, SMTP or IMAP is implemented. In combination with the nginx core, these upper-level modules are responsible for maintaining the right order of calls to the respective functional modules. While the HTTP protocol is currently implemented as part of the `http` module, there are plans to separate it into a functional module in the future, due to the need to support other protocols like SPDY (see "SPDY: An experimental protocol for a faster web").

The functional modules can be divided into event modules, phase handlers, output

filters, variable handlers, protocols, upstreams and load balancers. Most of these modules complement the HTTP functionality of nginx, though event modules and protocols are also used for `mail`. Event modules provide a particular OS-dependent event notification mechanism like `kqueue` or `epoll`. The event module that nginx uses depends on the operating system capabilities and build configuration. Protocol modules allow nginx to communicate through HTTPS, TLS/SSL, SMTP, POP3 and IMAP.

A typical HTTP request processing cycle looks like the following.

1. Client sends HTTP request.
2. nginx core chooses the appropriate phase handler based on the configured location matching the request.
3. If configured to do so, a load balancer picks an upstream server for proxying.
4. Phase handler does its job and passes each output buffer to the first filter.
5. First filter passes the output to the second filter.
6. Second filter passes the output to third (and so on).
7. Final response is sent to the client.

nginx module invocation is extremely customizable. It is performed through a series of callbacks using pointers to the executable functions. However, the downside of this is that it may place a big burden on programmers who would like to write their own modules, because they must define exactly how and when the module should run. Both the nginx API and developers' documentation are being improved and made more available to alleviate this.

Some examples of where a module can attach are:

- Before the configuration file is read and processed
- For each configuration directive for the location and the server where it appears
- When the main configuration is initialized
- When the server (i.e., host/port) is initialized
- When the server configuration is merged with the main configuration
- When the location configuration is initialized or merged with its parent server configuration
- When the master process starts or exits
- When a new worker process starts or exits
- When handling a request
- When filtering the response header and the body
- When picking, initiating and re-initiating a request to an upstream server
- When processing the response from an upstream server
- When finishing an interaction with an upstream server

Inside a `worker`, the sequence of actions leading to the run-loop where the response is generated looks like the following:

1. Begin `ngx_worker_process_cycle()`.
2. Process events with OS specific mechanisms (such as `epoll` or `kqueue`).
3. Accept events and dispatch the relevant actions.
4. Process/proxy request header and body.
5. Generate response content (header, body) and stream it to the client.
6. Finalize request.
7. Re-initialize timers and events.

The run-loop itself (steps 5 and 6) ensures incremental generation of a response and streaming it to the client.

A more detailed view of processing an HTTP request might look like this:

1. Initialize request processing.
2. Process header.
3. Process body.

4. Call the associated handler.
5. Run through the processing phases.

Which brings us to the phases. When nginx handles an HTTP request, it passes it through a number of processing phases. At each phase there are handlers to call. In general, phase handlers process a request and produce the relevant output. Phase handlers are attached to the locations defined in the configuration file.

Phase handlers typically do four things: get the location configuration, generate an appropriate response, send the header, and send the body. A handler has one argument: a specific structure describing the request. A request structure has a lot of useful information about the client request, such as the request method, URI, and header.

When the HTTP request header is read, nginx does a lookup of the associated virtual server configuration. If the virtual server is found, the request goes through six phases:

1. server rewrite phase
2. location phase
3. location rewrite phase (which can bring the request back to the previous phase)
4. access control phase
5. try_files phase
6. log phase

In an attempt to generate the necessary content in response to the request, nginx passes the request to a suitable content handler. Depending on the exact location configuration, nginx may try so-called unconditional handlers first, like `perl`, `proxy_pass`, `flv`, `mp4`, etc. If the request does not match any of the above content handlers, it is picked by one of the following handlers, in this exact order: `random index`, `index`, `autoindex`, `gzip_static`, `static`.

Indexing module details can be found in the nginx documentation, but these are the modules which handle requests with a trailing slash. If a specialized module like `mp4` or `autoindex` isn't appropriate, the content is considered to be just a file or directory on disk (that is, static) and is served by the `static` content handler. For a directory it would automatically rewrite the URI so that the trailing slash is always there (and then issue an HTTP redirect).

The content handlers' content is then passed to the filters. Filters are also attached to locations, and there can be several filters configured for a location. Filters do the task of manipulating the output produced by a handler. The order of filter execution is determined at compile time. For the out-of-the-box filters it's predefined, and for a third-party filter it can be configured at the build stage. In the existing nginx implementation, filters can only do outbound changes and there is currently no mechanism to write and attach filters to do input content transformation. Input filtering will appear in future versions of nginx.

Filters follow a particular design pattern. A filter gets called, starts working, and calls the next filter until the final filter in the chain is called. After that, nginx finalizes the response. Filters don't have to wait for the previous filter to finish. The next filter in a chain can start its own work as soon as the input from the previous one is available (functionally much like the Unix pipeline). In turn, the output response being generated can be passed to the client before the entire response from the upstream server is received.

There are header filters and body filters; nginx feeds the header and the body of the response to the associated filters separately.

A header filter consists of three basic steps:

1. Decide whether to operate on this response.
2. Operate on the response.

3. Call the next filter.

Body filters transform the generated content. Examples of body filters include:

- server-side includes
- XSLT filtering
- image filtering (for instance, resizing images on the fly)
- charset modification
- `gzip` compression
- chunked encoding

After the filter chain, the response is passed to the writer. Along with the writer there are a couple of additional special purpose filters, namely the `copy` filter, and the `postpone` filter. The `copy` filter is responsible for filling memory buffers with the relevant response content which might be stored in a proxy temporary directory. The `postpone` filter is used for subrequests.

Subrequests are a very important mechanism for request/response processing. Subrequests are also one of the most powerful aspects of nginx. With subrequests nginx can return the results from a different URL than the one the client originally requested. Some web frameworks call this an internal redirect. However, nginx goes further—not only can filters perform multiple subrequests and combine the outputs into a single response, but subrequests can also be nested and hierarchical. A subrequest can perform its own sub-subrequest, and a sub-subrequest can initiate sub-sub-subrequests. Subrequests can map to files on the hard disk, other handlers, or upstream servers. Subrequests are most useful for inserting additional content based on data from the original response. For example, the SSI (server-side include) module uses a filter to parse the contents of the returned document, and then replaces `include` directives with the contents of specified URLs. Or, it can be an example of making a filter that treats the entire contents of a document as a URL to be retrieved, and then appends the new document to the URL itself.

Upstream and load balancers are also worth describing briefly. Upstreams are used to implement what can be identified as a content handler which is a reverse proxy (`proxy_pass` handler). Upstream modules mostly prepare the request to be sent to an upstream server (or "backend") and receive the response from the upstream server. There are no calls to output filters here. What an upstream module does exactly is set callbacks to be invoked when the upstream server is ready to be written to and read from. Callbacks implementing the following functionality exist:

- Crafting a request buffer (or a chain of them) to be sent to the upstream server
- Re-initializing/resetting the connection to the upstream server (which happens right before creating the request again)
- Processing the first bits of an upstream response and saving pointers to the payload received from the upstream server
- Aborting requests (which happens when the client terminates prematurely)
- Finalizing the request when nginx finishes reading from the upstream server
- Trimming the response body (e.g. removing a trailer)

Load balancer modules attach to the `proxy_pass` handler to provide the ability to choose an upstream server when more than one upstream server is eligible. A load balancer registers an enabling configuration file directive, provides additional upstream initialization functions (to resolve upstream names in DNS, etc.), initializes the connection structures, decides where to route the requests, and updates stats information. Currently nginx supports two standard disciplines for load balancing to upstream servers: round-robin and ip-hash.

Upstream and load balancing handling mechanisms include algorithms to detect failed upstream servers and to re-route new requests to the remaining ones—

though a lot of additional work is planned to enhance this functionality. In general, more work on load balancers is planned, and in the next versions of nginx the mechanisms for distributing the load across different upstream servers as well as health checks will be greatly improved.

There are also a couple of other interesting modules which provide an additional set of variables for use in the configuration file. While the variables in nginx are created and updated across different modules, there are two modules that are entirely dedicated to variables: `geo` and `map`. The `geo` module is used to facilitate tracking of clients based on their IP addresses. This module can create arbitrary variables that depend on the client's IP address. The other module, `map`, allows for the creation of variables from other variables, essentially providing the ability to do flexible mappings of hostnames and other run-time variables. This kind of module may be called the variable handler.

Memory allocation mechanisms implemented inside a single nginx `worker` were, to some extent, inspired by Apache. A high-level description of nginx memory management would be the following: For each connection, the necessary memory buffers are dynamically allocated, linked, used for storing and manipulating the header and body of the request and the response, and then freed upon connection release. It is very important to note that nginx tries to avoid copying data in memory as much as possible and most of the data is passed along by pointer values, not by calling `memcpy`.

Going a bit deeper, when the response is generated by a module, the retrieved content is put in a memory buffer which is then added to a buffer chain link. Subsequent processing works with this buffer chain link as well. Buffer chains are quite complicated in nginx because there are several processing scenarios which differ depending on the module type. For instance, it can be quite tricky to manage the buffers precisely while implementing a body filter module. Such a module can only operate on one buffer (chain link) at a time and it must decide whether to overwrite the input buffer, replace the buffer with a newly allocated buffer, or insert a new buffer before or after the buffer in question. To complicate things, sometimes a module will receive several buffers so that it has an incomplete buffer chain that it must operate on. However, at this time nginx provides only a low-level API for manipulating buffer chains, so before doing any actual implementation a third-party module developer should become really fluent with this arcane part of nginx.

A note on the above approach is that there are memory buffers allocated for the entire life of a connection, thus for long-lived connections some extra memory is kept. At the same time, on an idle keepalive connection, nginx spends just 550 bytes of memory. A possible optimization for future releases of nginx would be to reuse and share memory buffers for long-lived connections.

The task of managing memory allocation is done by the nginx pool allocator. Shared memory areas are used to accept mutex, cache metadata, the SSL session cache and the information associated with bandwidth policing and management (limits). There is a slab allocator implemented in nginx to manage shared memory allocation. To allow simultaneous safe use of shared memory, a number of locking mechanisms are available (mutexes and semaphores). In order to organize complex data structures, nginx also provides a red-black tree implementation. Red-black trees are used to keep cache metadata in shared memory, track non-regex location definitions and for a couple of other tasks.

Unfortunately, all of the above was never described in a consistent and simple manner, making the job of developing third-party extensions for nginx quite complicated. Although some good documents on nginx internals exist—for instance, those produced by Evan Miller—such documents required a huge reverse engineering effort, and the implementation of nginx modules is still a black art for many.

Despite certain difficulties associated with third-party module development, the nginx user community recently saw a lot of useful third-party modules. There is, for instance, an embedded Lua interpreter module for nginx, additional modules for load balancing, full WebDAV support, advanced cache control and other interesting third-party work that the authors of this chapter encourage and will support in the future.

14.5. Lessons Learned

When Igor Sysoev started to write nginx, most of the software enabling the Internet already existed, and the architecture of such software typically followed definitions of legacy server and network hardware, operating systems, and old Internet architecture in general. However, this didn't prevent Igor from thinking he might be able to improve things in the web servers area. So, while the first lesson might seem obvious, it is this: there is always room for improvement.

With the idea of better web software in mind, Igor spent a lot of time developing the initial code structure and studying different ways of optimizing the code for a variety of operating systems. Ten years later he is developing a prototype of nginx version 2.0, taking into account the years of active development on version 1. It is clear that the initial prototype of a new architecture, and the initial code structure, are vitally important for the future of a software product.

Another point worth mentioning is that development should be focused. The Windows version of nginx is probably a good example of how it is worth avoiding the dilution of development efforts on something that is neither the developer's core competence or the target application. It is equally applicable to the rewrite engine that appeared during several attempts to enhance nginx with more features for backward compatibility with the existing legacy setups.

Last but not least, it is worth mentioning that despite the fact that the nginx developer community is not very large, third-party modules and extensions for nginx have always been a very important part of its popularity. The work done by Evan Miller, Piotr Sikora, Valery Kholodkov, Zhang Yichun (agentzh) and other talented software engineers has been much appreciated by the nginx user community and its original developers.

Back to top
Back to *The Architecture of Open Source Applications.*