Introduction

NGINX is a well-known free open source software (F/OSS) written by Igor Sysoev, a Russian engineer. NGINX has been used widely around the Internet by either large scale, or mid- to small sized companies. Typical scenario of implementing NGINX infrastructure, is to deploy it at the frontend part of a website. Very often it instantly enables a lot of additional performance and efficiency simply because of an ability of NGINX to fully utilize the potential of the underlying operating system and server hardware.

NGINX has been mostly famous by its outstanding webserver capabilities. However, these days it is already more than just a webserver. It's a powerful Internet infrastructure platform which includes an ultra-fast webserver for static and dynamic content, Layer 7 load-balancer, reverse proxy for HTTP, SMTP, IMAP and POP3, caching engine and SSL offloading.

With that said, we tend to think, it would be beneficial for the reader to be familiar with the brief history of NGINX development.

Igor Sysoev, the author of NGINX, spent 10 years working for Rambler — a big Russian media company and a search engine. Igor joined Rambler in 1999 and after 3 years of maintaining and developing frontend web infrastructure, he realized that he had spent already quite a lot of time and efforts in trying to make traditionally architected webservers work better. He was applying all the known and secret techniques to make webservers more efficient, he tuned the operating systems to the maximum, but one main problem remained, and it was the problem of handling concurrent connections without wasting memory and CPU on the servers. Ensuring high concurrency and low latency responses are vitally important for the majority of online services today, so naturally Igor was distracted by the inability of then existing webservers to do it efficiently.

Concurrent connections are essentially produced by the clients who stay connected to a webserver during a period of time. If we take a peak hour concurrency it would mean the number of clients hanging in a connected state at the peak of the service usage. It can be evenings for residential clients, or noons for SMB and enterprise — but there are always peak hours where many clients connect to the web service simultaneously, stay connected for a while and are superseded by the other clients.

A big number of concurrent connections would mean lots of memory consumed on the frontend servers, because the web software Rambler was using, served each connection with a separate process or thread, which is largely a comparable thing, as userland threads are generally mapped into kernel threads.

A separate process or thread requires creating additional writable memory datasets, such as stack areas and possibly more heap, which means the webserver is allocating quite a lot of memory just to serve each new connection. In absolute values it's something in the range of at least few MBs to hundreds of MBs in extreme cases, when inefficient embedded scripting is used along with the webserver. Accordingly, MBs of memory are needed for serving up every user. Obviously this kind of an architecture doesn't scale very well. In addition, every new process or thread leads to an extra overhead in terms of CPU usage because more context switching is required.

It is also very interesting what was the main cause for many concurrent connections. A curious reader who still recalls how the largest parts of the Internet worked in the 90's and at the beginning of 00's, would probably have already guessed the answer. Lots of concurrent connections are caused by the slow clients, that is by the clients using low-bandwidth and high-delay access to the Internet, such as dial-up, GRPS/EDGE/3G/HSDPA, older/slower variants of Wi-Fi, ADSL and other similar connectivity, or just being located far away from the server they connect to.

Nowadays, concurrency is also caused by a combination of the mobile clients (who are still largely slow) and by the newer application architectures which are typically based on maintaining a persistent connection allowing the client to be quickly updated with the news, tweets, friend feeds and basically everything which is consumed today from the modern online services.

Another important factor additionally contributing to increased concurrency is also the changed behavior of modern browsers, which would open 4-6 simultaneous connections to a website to increase the speed of page downloads.

Imagine a simple webserver which produces a relatively short 100KB answer (a modern web page with text or an image). It can be just a fraction of second to generate or retrieve such file, but then it takes 12.5s to transmit these 100KB to a client with a bandwidth of 64 Kbps (8KB/s). Now imagine that you have 1,000 simultaneously connected clients who just

have requested the same content. It would mean that the webserver quickly pulls 100KB from disk or an backend application, and then it takes 12.5s to output it to all of the clients and free all of the connections. In turn, if only 1MB of additional writable memory was allocated per connection (i.e. per one client), it means that 1000MB or 1GB or memory was occupied by serving just 1,000 clients with a 100KB content. And 64 Kbps is still widely the effective speed of communications over GPRS/EDGE and even 3G networks. In reality, a typical webserver based on an older software also allocates more than 1MB of additional memory per connection.

So, Igor was trying to overcome the barriers of being unable to scale the web infrastructure he was maintaining. Because there weren't a better software available for him to use, he quite naturally — yes, it was natural back then <g>, decided to improve things himself. This is how Igor started to implement the first version of NGINX.

Initially Igor started to learn all of the modern techniques for improving the network and disk input/output performance. It is worth to note another interesting detail which is still applicable today for many of the web setups. It's not a secret that most of the software enabling Internet today was architected in the 80's and in the 90's. Back then, lots of historical limitations of the operating systems, stemming from BSD and System V, were preventing additional performance optimizations — for the web software as well. However, situation began to change by the end of the 90's and in just few years many new programming interfaces appeared that allowed applications to provide hints and clues to the operating systems in regards to performance. Many of them are still underutilized by the majority of popular infrastructure products, though.

In order to make NGINX most efficient, Igor has paid lots of attention to kqueue and epoll development in FreeBSD and Linux respectively, which were continuously being improved across the kernel developer communities. Igor was testing different approaches to use such new mechanisms and provided useful feedback to kernel developers as well. He had also explored thoroughly all of the aspects of architecting a new generation web software. Eventually Igor chose to implement the architecture that can be briefly described as modular, event-driven, asynchronous and non-blocking. This is what became the foundation of NGINX code.

While many concepts that NGINX uses, were inspired by Apache, it is not true that NGINX was forked from Apache code or is even using portions of Apache code. NGINX codebase is

original and was written from scratch during already almost 10 years of development. NGINX is written entirely in C programming language and has been ported to numerous architectures and operating systems, including Linux, FreeBSD, Solaris, Mac OS X and even Microsoft Windows.

Igor wrote the first lines of NGINX code in 2002 and by 2004 the project was ready to be launched to the public. In 2004 Igor opened NGINX to the users, and many people jumped on the idea and started to use NGINX to scale their quickly growing web infrastructures. They were also actively providing the necessary reports about their experiences with NGINX, noticing bugs, offering suggestions and requesting more functionality. The help from the user community has been enormous.

It was the dawn of the new era of the ubiquitous social media, mobile applications, permanently connected and demanding users of all ages, and instantaneous communications. In a sense, Igor has implemented what had to become a very important enabling mechanism for big and small companies worldwide. Today NGINX is powering four out of ten World's Most Valuable Startups (Business Insider 2011 Top 100 list), including Facebook, Groupon, LivingSocial and Dropbox. But it is also being run by other 25% of the top 1000 busiest websites and overall by 45,000,000 of websites on the Internet.

Overview of NGINX architecture

NGINX differs from the older webserver software primarily by its architecture. First of all, we may say that NGINX is clever in its behavior and it doesn't try to outthink a well-engineered modern operating system while handling the network and disk load with ultimate efficiency in memory and CPU. NGINX elaborately optimizes the usage of the operating system and the hardware resources with its event-driven, asynchronous, non-blocking, modular architecture.

NGINX is using 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.

<Diagram depicting NGINX architecture>

NGINX runs three types of processes in memory. There is a single master process and several worker processes. There are also a couple of special purpose processes, namely — cache loader and cache manager. All processes are single-threaded in version 1.x of NGINX. All processes primarily use shared memory mechanisms for IPC. Master process is run as root. Cache loader, cache manager and the workers run unprivileged.

The master process is responsible for the following tasks:
- reading and validating the configuration;
- creating, opening, binding and closing sockets;
- starting, terminating and maintaining the configured number of worker processes;
- reconfiguring without service interruption;
- controling non-stop binary upgrades (starting the new binary and rolling back if necessary);
- maintaining log files;
- compiling embedded Perl scripts.

The cache loader process is responsible for checking the on-disk cache items and populating NGINX in-memory database with the cache metadata. Essentially, cache loader is preparing NGINX instance to work with the 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 tidy and ready for operations.

The cache manager is mostly responsible for the cache expiration and invalidation. It stays in memory during normal NGINX operations and it is restarted by the master process in the case of failure.

Cache in NGINX is implemented in the form of an hierarchical data storage on a filesystem. Cash keys are configurable and different request specific parameters can be used to control what gets into cache. The cache keys and cache metadata are stored in the shared memory segments — which cache loader, cache manager and the workers access. Currently there isn't any in-memory caching of files, other than optimizations implied by the VFS cache.

Each cached response is placed in a different file on the filesystem. The hierarchy (levels

and naming details) are controlled through NGINX configuration file directives. When a response is written to the cache directory structure, the path and the name of the file are derived from MD5 hash of the proxied URL.

What happens when NGINX is going to place the content in the cache is the following. When NGINX reads the response from the backend, the content is first being written to a temporary file outside of cache directory structure. When NGINX finishes request processing it renames the temporary file into the cache directory file. If the temporary files directory for proxying is on another file system, the file will be copied. Thus it's recommended to keep both temp directory 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 3rd party extensions for NGINX available which make it possible to control cached content remotely, and more work is planned to integrate such functionality in the main branch of code.

Last but not least, the worker processes accept, handle and process connections from the clients, provide reverse proxying and filtering functionality and do everything else that NGINX is capable of. Within each worker NGINX can handle many thousands of concurrent connections and requests per second. In regards to monitoring the behavior of NGINX instance, a system administrator should keep an eye on workers as they are the processes reflecting the actual day-to-day operations of a webserver. A number of parameters can be effectively checked for an NGINX instance and the work is ongoing to make NGINX monitoring even more functional.

Now it's time to talk a bit more about how NGINX is trying to achieve maximum efficiency in handling the workloads we see on the Internet today.

Traditional process-based or thread-based model of handling concurrent connections involves processing each connection with a separate process or thread, blocking on network and input/output operations until completion. Depending on the application it can be extremely inefficient in terms of memory and CPU consumption.

Spawning a separate process or thread requires preparation of new runtime environment, including creating stack memory areas, and preparing new execution context. Additional CPU time is also spent on creation.

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 arbitration or distributing connections from the master processes to the workers. The only thing the master process does with sockets is creating the initial set of listening sockets. All workers then can do an accept() and the task of synchronization is done by the workers, not the master. A optional mechanism exists — allowing workers to avoid spinning unnecessarily on accept(), and it's a typical mutex which is raised by the worker to check if there's a new network event notification for accept().

The run-loop is the most complicated part of NGINX worker code where most of the work of processing the requests is done. It includes comprehensive inner calls and is relying heavily on the idea of asynchronous handling of tasks. Asynchronous operations are implemented through modularity, event notifications, extensive use of callback functions and fine-tuned timers. Overall, the key principle here is to be as much non-blocking as possible. The only situation where NGINX can still block is when there's no enough storage performance for a worker.

Because NGINX is free from forking a process or thread per connection, the memory usage is very conservative and extremely efficient in the vast majority of cases. Examples of praises NGINX has been enjoying include something like "It's a tiny thing with flat usage of resources .. A great improvement over the previous software in doing massively input/output bound operations .. Allows to get the most performance out of the least amount of hardware .. Able to continuously take more connections without consuming additional resources".

NGINX conserves CPU cycles as well because there's no ongoing create/destroy pattern for processes or threads. What NGINX is doing is checking the state of network and storage, allocating new connection, adding it to the run-loop, processing asynchronously until completion and de-allocating connection from the run-loop. Combined with the economical use of syscalls and accurate implementation of supporting interfaces like pool and slab memory allocators, it typically leads to a relatively low CPU usage even under extreme workloads.

To fully enable performance and scalability while handling a variety of consequent 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 enablements in Linux, Solaris and BSD-based operating systems. To put it simple, it is all about providing hints to the operating system and getting a timely feedback in regards to when expect an inbound or outbound traffic, when check disk operation, when refresh content and so on. Currently, NGINX is able to utilize pretty much every modern notification mechanism invented for an operating system, like for instance kqueue, epoll or event ports.

With the model of running several workers to process connections, NGINX scales quite well across multiple cores. Typical scenario is to allocate one worker per core. Depending on the load patterns, disk and CPU utilization on the server, it might be beneficial to increase the number of workers to 1.5 or 2 times the number of cores. A separate worker per core allows full utilization of multicore architectures, prevents excessive context switching and thread locking. 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 generally allows to avoid blocking on disk I/O. Overall, the hardware server resources are also utilized more efficiently with the workload shared across several workers.

There are certain drawbacks of the existing implementation as well. One major problem, the developers of NGINX will be solving in the next versions of the product, is how to avoid most of the situations with blocking on disk I/O. At this time, if there's no enough storage performance to serve disk operations generated by a particular worker, such worker may block on reading/writing from disk which may potentially affect thousands of connections served by this worker.

A number of mechanisms and control knobs 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. However, the mileage may vary, depending of the data set, the amount of memory available for NGINX, the underlying storage architecture and other specifics of a particular NGINX installation.

Another problem of the existing worker model is related to a severely limited support for embedded scripting. For one, only Perl embedded scripting support is implemented. There's a good reason for that, and the explanation is quite simple — the key problem here is a 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 in regards to make embedded scripting with NGINX simpler, more reliable and suitable for a broader range of applications like hosting.

NGINX architecture is modular and the functionality of NGINX can generally be extended without modifying the NGINX core architecture. NGINX modules come in slightly different incarnations, namely core modules, event modules, phase handlers, protocols, handlers of variables, filters, upstreams and load balancers. More detailed information about the roles of different modules can be find in "NGINX internals" section of this document.

Modules constitute most of the presentation and application layers functionality of NGINX. Modules read and write from the network and storage, transform content, do outbound filtering, apply server-side include actions and pass the requests to the backend servers if proxying is activated. 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.

At this time NGINX doesn't support dynamically loadable modules, that is — modules are compiled along with the core at the build stage. However, support for loadable modules and ABI is planned for the future major releases.

NGINX module development is something that has been quite problematic for the 3rd party developers, because previously there weren't any good developer documentation available, and this is primarily because of the lack of time for Igor to do that. Although, there are some very good explanations out there, describing NGINX module development

those are based on a huge effort in reverse engineering of NGINX code, and the implementation of NGINX modules is still more of a black art for many. Some good documents were produced by Evan Miller (http://www.evanmiller.org/http://www.evanmiller.org/), and parts of the present document were influenced by his hard work. Now, as the developers of NGINX have more time, resource and a strong intention to produce and maintain an adequate API documentation, the NGINX developer community will hopefully be growing in a more healthy way in the future.

A few words about the Windows version of NGINX. While NGINX works under Windows

environment, Windows version of NGINX is more like a proof-of-concept rather than a fully functional Windows port. There are certain limitations of both NGINX and Windows kernel architectures that don't mix quite well at this time. Known issues of NGINX version for Windows include the following: no parallel processing of requests by the workers (only one worker is doing the job at any given moment of time), up to 1024 connections per worker, no caching functionality (because there's no shared memory in Windows Vista and above). Future enhancements of NGINX version for Windows might include running NGINX as a service, a better performance, utilizing multiple threads in a single worker for better scalability.

NGINX configuration

Ideas that served as the foundation for NGINX configuration design were significantly inspired by Igor's Apache experience. What Igor has primarily learned from his work duties told him that a much more scalable configuration style is essential. The main problem was seen in maintaining large complicated configs with lots of virtual servers, directories, locations and everything. In a relatively large scale web setup it can turn into a big nightmare if not done properly both at the application level and by the system engineer himself.

So, from its early inception NGINX configuration was designed to simplify day-to-day operations and to be ultimately scalable.

The configuration file is initially read and verified by the master process. Its compiled read-only form is available for use by the worker processes as the latter are forked from the master process. Configuration structures are automatically shared by the usual operating system and virtual memory mechanisms.

NGINX configuration has several different contexts for Main, Http, Mail, Server, Upstream and Location blocks of directives. Contexts never overlap. For instance, there's no such thing as putting the location block in the main block of directives. Also, there isn't anything like the "global webserver" configuration. Legacy Apache-like style of configuration which allowed to have "global" or "main server" don't necessarily simplify configuration of a webserver, but rather often introduce an unnecessary ambiguity. In turn, NGINX configuration is clean and logical, allowing to avoid hassle, and maintain complicated config files that comprise thousands of directives. To quote Igor, "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 not implemented in NGINX".

While configuring NGINX, a few global settings which affect overall server behavior are put in the main context. Here the number of worker processes, paths to error log file, preferred mechanisms of event notifications, and similar directives are typically found. Then there're "http" and "mail" contexts ("mail" — because NGINX also includes SMTP/IMAP/POP3 proxying capabilities), where a number of common directive like timeouts, log formats, filter controls are configured.

Inside "http" or "mail" contexts a number of "server" blocks of directives are put, which describe individual web or mail servers and their respective configuration in regards to the names, listening ports, protocols and other important details. In the "http" context there might be also "upstream" parts, defining how NGINX will proxy the connections to the servers. For "http" context the server blocks of directives essentially define the virtual servers.

For a "server" context an arbitrary number of "locations" are typically configured. Here the configuration blocks can contain directives which are also applicable in the higher level contexts like "server" and "main". There are lots of directives which can only belong to the "location" context, though. There's also a mechanism in place to inherit the upper level configuration directives for both "location" and "server" context. In regards to the "upstream" context, it has a limited number of directives which aren't inherited from either "server" or "main".

Configuration syntax, formatting and definitions follow a so-called C-style convention. This particular approach for making configuration files had been already a proven recipe for a variety of open source and commercial software and many engineers liked it. By its design, C-style configuration is well suitable for nested configuration and for being logical and easy to read, change and maintain. Igor preferred it over an alternative XML-style of configuration which he perceived to be appropriate mostly for use in an automated machine-controlled environment. While XML configuration provides certain advantages for automation, it is hard to maintain by hand and the vast majority of NGINX configurations out there are still controlled by the humans. Besides, C-style configuration can be easily automated as well.

With the notion of C-style configs being quite flexible and sometimes loose in terms of formatting, it would be probably useful to note that NGINX developers advocate for clear indentation and for having an opening curly bracket "{" for a block in the directive's line, like "server {".

To simplify the configuration, make it more scalable and less cluttered an engineer can utilize the "include" feature which is applicable to any context and block of directives. However, NGINX configuration doesn't currently support macro expansion, which is something that is also planned for the future releases.

It makes a lot of sense to describe the behavior of "location" context in more detail now.

Whenever NGINX receives a new connection and reads the header, it then matches the request against a suitable server. The next step is looking up a location inside the server definition that matches the request URI. A connection always falls in one location, there's no conflict in the end. Location match doesn't include the query strings — only the URI part is being matched for a particular request.

Depending on the configuration style, NGINX administrator might keep in mind certain subtleties which can contribute to the request processing and virtual server behavior. First of all, locations can be defined with either prefix character strings or by using regular expressions. When searching for a matching location, NGINX first checks the ones defined with the prefix strings (prefix locations). If any of the prefix locations are found, the **most** specific match is picked. By using a special operator type of "=" it is also possible to define the "**exact match**" for the request URI and a location. If the exact match is found, the search terminates **immediately**. It can be useful, for instance, to speed up the processing of something like a frequent "/" URI by utilizing the definition of "location = /".

Then the locations with **regex**'es (if any) are checked, **in order of their appearance** in the configuration file. This search terminates on the **first match**, and the corresponding configuration is used. If **no match** with a regular expression location is found, NGINX will use a configuration with the **most specific** prefix location.

Location configurations without regex'es are generally more scalable and work faster — that's because of the best match being always quickly applied without unnecessary comparison cycles. With the regular expressions the best match isn't that easy and next to

impossible to algorithmically implement in a reliable manner. To reiterate, locations with **regex'es** are tracked by NGINX **sequentially**, and the **first match** is picked. In the case of complicated configuration which contains tens or hundreds of locations, verifying such configuration might be a daunting experience for an engineer.

Location definitions can be nested, with a couple of exceptions. One would be a special type of location which is the so-called "named" location — which are specified by "@" prefix, and can be on the server level only. Named locations are used for redirecting the requests into an upstream connection. Another example is the exact match locations which also can't be nested.

NGINX configuration has a support for rewrite rules and this particular feature was also inspired by Apache. Rewrites are generally not scalable and typically produce lots of complications in regards to maintenance and debugging. In general an NGINX administrator should avoid the use of any rewrite rules if possible, and only implement those which are absolutely necessary. With the particular rewrite definition it might be a good idea to refer first to the documentation and a variety of "howto" publications available on the web.

NGINX supports conditional configuration directives inside the "if" statements. The "if" statements are implemented as part of the rewrite engine and while perceived by many system administrators as a powerful instrument for an NGINX tuning, "if" definitions can be in fact very tricky to properly implement and debug because of a couple of architectural nuances of the existing implementation. It's not an exaggeration to say that "if" statements (and rewrites) are the dark side of NGINX. They work very well with simple returns, though.

A much better and scalable alternative to "if" statements is the use of "try_files" directive. This directive was initially meant to be gradually replacing conditional "if" statements in a proper way, and it was designed to quickly and efficiently try/match against different URI-to-content mapping. Overall "try_files" definition works great and can be extremely efficient and useful. It's recommended for a reader to thoroughly check "try_files" directive and adopt its use whenever applicable. The only exception when it's probably not that much flexible is handling POST requests.

Lat but not least, NGINX configuration also supports variables. Implementation of

variables in NGINX is unique, and it was developed to provide additional, even more powerful mechanism to control run-time configuration of a webserver. Variables offer another great means for a system administrator to influence a number of things during the run-time phase. Variables are optimized for quick evaluation and are internally pre-compiled to indices. Evaluation is done on-demand, that is — variable values are typically calculated only once and cached during the request processing. Because variables are calculated in the run-time, they cannot be used as something like macro expansion. Although, variables were added somewhere in the middle of NGINX code development, at this time still not all of the directives and modules fully support them. This is another thing NGINX developers have been continuously working on.

NGINX internals

In this section we will make an attempt to describe some of the internals of NGINX. It can be important for a system administrator to understand a general picture of how NGINX works inside, and what is the relationship between different parts of NGINX code.

As it was mentioned before, NGINX code consists of a core code and a number of modules. The core of NGINX is responsible for providing the foundation of the webserver, web- and mail reverse proxy functionality — 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 is processing connections through a pipeline — or chain of modules. In other words for every operation there's a module which is doing the relevant work like compression, modifying the content, executing server-side include, communicating to the backend application servers through FastCGI or WSGI protocols, or talking to a memcache backend.

There are a couple of NGINX modules which 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. What is implemented inside these in-between modules, is primarily the handling of the sequence of events associated with a respective application layer protocol like HTTP, SMTP or IMAP. In combination with NGINX core, these upper level modules are responsible for

maintaining the right order of calls to the respective "functional" modules.

The functional modules can be divided into event modules, phase handlers, outbound filters, handlers of variables, protocols, upstreams and load balancers. Most of these modules are used to implement HTTP-based functionality of NGINX. Event modules and protocols are also used for "mail". Event modules provide a particular event-notification mechanism like kqueue or epoll. Depending on the operating system capabilities and the build configuration, NGINX uses either this or that event module. Protocol modules bring to work HTTPS, TLS/SSL, SMTP, POP3, IMAP. On the other hand, HTTP protocol is currently implemented as part of "http" module and not as a separate protocol module. In the future this can be further separated into a module too, due to a necessity to support other protocols like SPDY (http://www.chromium.org/spdyhttp://www.chromium.org/spdy).

A typical HTTP request processing cycle looks like the following:
1. Client sends HTTP request
2. NGINX chooses the appropriate phase handler based on the location configuration
3. (If applicable) load balancer picks a backend server
4. 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 sent to the client

NGINX module invocation is extremely customizable. Invocation is performed through a series of callbacks using pointers to the executable functions. Caveat emptor, it may place a big burden on the programmers who'd like to write their own modules, because it has to be defined exactly how and when the module should run. However, there are certain ongoing activities to make both NGINX API and developers' documentation better and generally more available too.

Examples of where a module can attach include:
- Before the configuration file is read and processed
- For every 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 headers and the body
- When picking, initiating and re-initiating a request to a backend server
- When processing the response from a backend server
- When finishing an interaction with a backend server

Inside the worker, the run-loop where the requests are being processed, looks like the following:
1. Begin ngx_worker_process_cycle()
2. Process events with an OS specific mechanisms (such as epoll or kqueue)
3. Accept events and dispatch the relevant actions
4. Process/proxy request headers, body
5. Generate response content (headers, body)
6. Finalize request, send response
7. Re-initialize timers and events and continue the loop

A more detailed view of an HTTP request processing might look like this:
a) Init request processing
b) Process headers
c) Process body
d) Call the associated handler
e) Run through the processing phases

Which brings us to the phases. When NGINX is handling an HTTP request, it passes it through a number of processing phases. At each phase there's a handler 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. There's always one handler attached to a location.

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 the headers.

When the HTTP request headers are read, NGINX does a lookup of the associated virtual server configuration. If the virtual server is found, the request goes through:

1. Server rewrite phase
2. Location phase
3. Location rewrite phase (which can bring the request back to step 2)
4. Access control phase
5. Try-files phase

What NGINX does next is passing the request for processing by a suitable content phase handler. In other words, NGINX makes an attempt to generate the necessary content as a response to the request. The first handlers to try are — perl, proxypass, flv and mp4. If the request doesn't match the above content handlers, it goes further to be picked by one of the following handlers — in this exact order of appearance — random index, index, autoindex, gzip static, static.

The details about indexing modules can be found in the reference documentation (http://nginx.org/en/docs/http://nginx.org/en/docs/), but these are the modules which handle the requests with the trailing slash. If there's no a configuration for a location to pass the request to any specialized module like mp4 or autoindex, the content is considered to be just a file on disk (that is — static) and is served by "static" content phase handler. It automatically rewrites the URI so that the trailing slash is always there. After the rewrite NGINX does an internal redirect to serve the associated content from storage. Another use of internal redirect is when NGINX processes the request through X-Accel-Redirect feature.

After the phase handlers NGINX passes the content to filters. NGINX filters were designed from scratch, it is not something that was influenced by then existing webservers. Filters are also attached to to the 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 the compile time. For the out-of-the-box filters it's predefined and for a 3rd party development it can be configured at build stage. Currently there's no mechanism in place to write and attach filters to do content transformation on input. In the existing NGINX implementation filters can only do outbound changes. Input

filtering is something to appear in the future versions of NGINX.

Filters follow the design pattern which looks like the following — a filter gets called, starts working, calls the next filter until the final filter in the chain. After that NGINX finalizes the response. Filters don't have to wait for the previous filter to finish. Next filter in chain can start its own work as soon as the input from the previous one is available (much like in the Unix pipeline). Filters work with buffers which are usually 4KB long, though it is configurable. In turn, the output response being generated can be passed to the client before the entire response from the backend is received.

There are header filters and the 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 the body filters include:
- Server-side includes
- XSLT filtering
- Image filtering (for instance, resizing images on-the-fly)
- Charset modification
- Gzip
- Chunked encoding

After the filter chain the response is passed to the writer. Along with the writer there're a couple of additional special purpose filters, namely — the "copy" filter, and the "postpone" filter. The copy filter is responsible to fill in the memory buffers with the relevant response content which might be stored in a proxy temp directory. Postpone filter is used with subrequests.

Subrequests are a very important mechanism of 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 client has originally requested. Some web frameworks call this an "internal redirect." But NGINX goes further

— not only can filters perform multiple subrequests and combine the outputs into a single response, but subrequests can be also nested and hierarchical. A subrequest can perform their own sub-subrequest, and 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 scan the contents of the returned document, and then replaces "include" directives with the contents of the 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 to describe briefly here. Upstreams are used to implement what can be identified as a content handler which-is-a-reverse-proxy ("proxypass" handler). Upstream modules mostly do preparation of the request to be sent to a backend (or "upstream") server and gather the response from backend. There're no calls to output filters here, for instance. What an upstream module exactly does is setting callbacks to be invoked when the backend server is ready to be written to and read from. The following callbacks exist for that:
1. Crafting a request buffer (or a chain of them) to be sent to the backend
2. Re-initializing/Resetting the connection to the backend (which happens right before creating the request again)
3. Processing the first bits of a backend response, saving pointers to the payload received from the backend
4. Aborting the requests (which happens when the client terminates prematurely)
5. Finalizing the request when NGINX finishes to read from the backend
6. Trimming the response body (e.g. removing a trailer)

Load balancer modules attach to the proxypass handler to provide an ability to choose a backend server, when more than one backend server is eligible. What a load balancer modules does is the following. It 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, updates stats information. Currently NGINX supports just two standard disciplines for load balancing to backends — round-robin and ip-hash. More work on load balancers is planned, and in the next versions of NGINX the mechanisms of distributing the load across different

backends as well as the health checks will be greatly improved.

There are also a couple of other interesting modules which provide 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 — namely, "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 ip's. The other module which is "map" allows to create variables from variables, essentially providing an ability to do a flexible mapping of hostnames and other run-time variables. This kind of modules we may call the handlers of variables.

Memory allocation mechanisms implemented in NGINX were also 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 headers and body of the request and the response, and then freed upon connection release.

Going a bit deeper, when the response is generated by a module, the retrieved content is put to 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 depending on the module type. For instance, it can be quite tricky to manage the buffers precisely while implementing a body filter module. Such 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 the buffer chains, so before doing an actual implementation a 3rd party module developer should become really fluent with this obscure part of NGINX.

One drawback of the above approach is that for the entire life of a connection there are memory buffers allocated. So for long-lived connections some memory is wasted. At the same time for a keepalive connection NGINX currently maintains only 550 bytes of allocated memory. An optimization that would be possible in the next releases of NGINX is to re-use memory buffers for the long-lived connections.

The task of managing memory allocations is done by NGINX pool allocator. Shared memory areas are used for accept mutex, cache metadata, 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 allocations. 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 reb-black tree implementation.