электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

**УДК 004.4**

# Серверные архитектуры и особенности их масштабирования

***Мялкин М.П.*** *студент*
*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,*
*кафедра «Системы обработки информации и управления»*

***Чернобровкин С.В.*** *студент*
*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,*
*кафедра «Системы обработки информации и управления»*

*Научный руководитель: Гапанюк Ю.Е., к.т.н., доцент,*
*Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана*
*gapyu@bmstu.ru*

## Введение

В наше время интернет стал неотъемлемой частью жизни, и практически никакая отрасль не обходится без своего интернет-ресурса. Интернет-проект может быть использован для разных целей, и, в зависимости от этого, может варьироваться его нагрузка. В качестве таких проектов могут выступать различные новостные ресурсы, форумы, блоги, магазины и т.д. Для каждого из них нужно продумывать архитектуру, которая будет отвечать необходимым требованиям. Она складывается из множества частей, основными из которых являются веб-сервера. Именно они отвечают за обработку запросов пользователей и формирование ответов на них. Каждый веб-сервер (ПО, установленное на машине) имеет собственное строение, которое напрямую отражается на его характеристиках. Существует множество типовых архитектур серверов, поэтому необходимо выбирать ту архитектуру, которая сможет максимально удовлетворить потребности проекта. Кроме того, некоторым проектам необходимо разрабатывать свое узкоспециализированное ПО, и здесь требуется понимание основных принципов построения веб-серверов. Этим архитектурам и наиболее популярным серверам, основанным на них, и посвящена статья.

## HTTP

HTTP – протокол прикладного уровня модели OSI. Он часто приводится, как пример идеального, простого решения – изначально созданный для передачи гипертекстовых документов, сегодня он является основой для многих интерактивных приложений. Возможно, иногда приходится искать обходные решения проблем, связанных с

ограничениями это протокола, и, может быть, в будущем его сменит другой протокол. Но на сегодня факт остается фактом: основываясь на нескольких методах запросов, кодах статусов и простых текстовых аргументах, HTTP является достаточно гибким и надежным. Задача, которая традиционно решается с помощью протокола HTTP — обмен данными между пользовательским приложением, осуществляющим доступ к веб-ресурсам (обычно это веб-браузер) и веб-сервером.

Процесс передачи/обработки HTTP:

- Клиент формирует и отправляет запрос. Запрос содержит заголовки и стартовую строку, по которой сервер понимает, что необходимо ответить. В зависимости от типа запроса он может содержать тело.
- Сервер принимает и разбирает запрос. По пришедшему URI либо берется файл, либо генерируется динамический контент.
- Сервер формирует ответ по протоколу HTTP из заголовков и тела и передает его клиенту.

Архитектура сервера влияет на то, как будет происходить прием и обработка запросов.

## Архитектуры веб-серверов

### Многопроцессная архитектура

В многопроцессной и многопоточной архитектурах используется один и тот же принцип: каждое соединение обрабатывается отдельной сущностью.

Многопроцессная архитектура использует модель, в которой для обработки соединения используется выделенный процесс. Эта архитектура была использована в самом первом HTTP-сервере. Из-за природы процессов различные запросы изолированы друг от друга, т.к. у них нет общей разделяемой памяти. Разные процессы могут общаться друг с другом с помощью глобальной памяти, сигналов или событий, семафоров, мьютексов, каналов, сокетов. Создание процесса очень дорогая операция, поэтому часто используют стратегию preforking.

Preforking

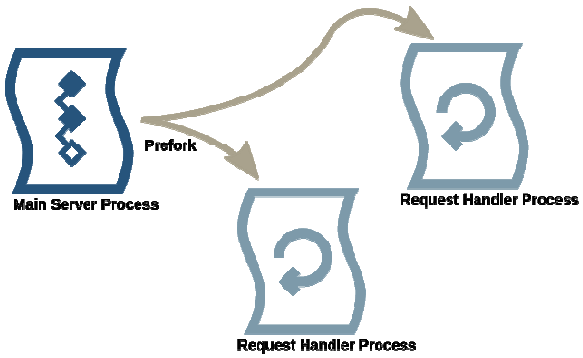


Рис. 1. Принцип работы модели preforking

Как показано на рисунке 1, при запуске сервера главный процесс в соответствии с конфигурацией и операционной системой создает несколько дочерних процессов, которые будут обрабатывать запросы. Каждый процесс-обработчик ждет нового соединения. Но существуют некоторые проблемы: при высокой нагрузке процессы могут не успевать обрабатывать запросы и те будут становиться в очередь. Чтобы исключить такую ситуацию, некоторые серверы измеряют нагрузку и при необходимости порождают дополнительные процессы-обработчики. Важно учесть, что эта архитектура накладывает ограничения на количество одновременных соединений, поэтому она не подходит для медленных клиентов или неактивных соединений (механизм long-pooling). Порождение нового процесса или потока влечёт за собой подготовку новой среды окружения с выделением в оперативной памяти кучи (heap), стека и нового контекста исполнения. ЦПУ тратит дополнительное время на создание этих объектов, что, в конечном итоге, может привести к снижению производительности из-за частого переключения контекста исполнения.

**Многопоточная архитектура**

Когда стали доступными библиотеки многопоточной обработки, появилась новая архитектура создания серверов. Тяжеловесные процессы были заменены на легковесные потоки. В данном случае для обработки соединения выделяется отдельный поток. Существуют некоторые отличия по сравнению с многопроцессной архитектурой:

- Множество потоков разделяют общее адресное пространство и, следовательно, общие переменные и состояния. Это позволяет реализовать некоторые возможности для всех обработчиков запросов. Например, наличие общего кэша запросов. Однако разработчику нужно обеспечить синхронизацию потоков.

- Легковесность потоков достигается за счет использования меньшего объема памяти. По сравнению с процессами, поток имеет ограниченный объем памяти (thread stack). Также потоки требуют меньше ресурсов на порождение/уничтожение.

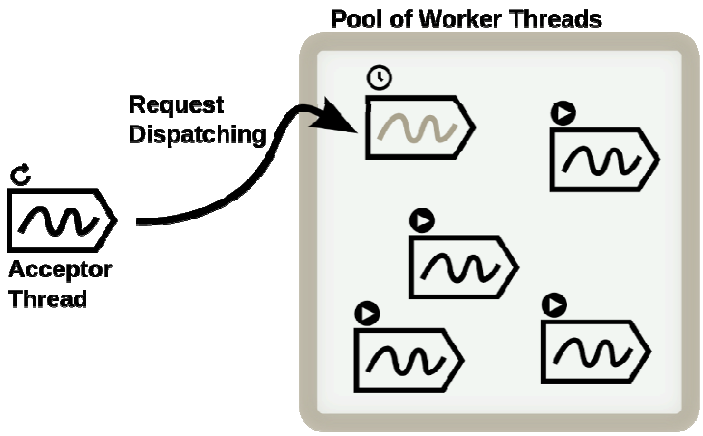


Рис. 2. Принцип работы многопоточной модели.

На практике в этой архитектуре размещается поток-диспетчер перед пулом потоков-обработчиков, как показано на рисунке 2. Пул потоков используется для того, чтобы ограничить максимальное количество потоков внутри сервера. Поток-диспетчер принимает соединения и кладет их в очередь. Потоки из пула забирают соединения из очереди, выполняют запросы и ждут новых соединений. Если на сервер поступает огромное количество запросов, то пул потоков может не успевать обрабатывать их все, и очередь будет постоянно увеличиваться. Но мы можем её ограничить. В этом случае при переполнении очереди новые запросы будут отклонены. Хотя это и ограничивает параллелизм, но зато мы получаем предсказуемые задержки и предотвращаем полную перегрузку сервера.

**Масштабируемость многопоточных архитектур**

Многопоточные серверы легко реализовать. Обычно используются синхронные блокирующие I/O операции. Операционная система выполняет несколько потоков одновременно благодаря запланированному переключению контекста. Операционная система сама распределяет потоки и процессы по всем доступным ядрам процессора.

При высоких нагрузках многопоточный веб-сервер потребляет огромный объем памяти (для каждого соединения выделяется стек), и константное время переключения между потоками приводит к потерям процессорного времени. Также это увеличивает

шанс cache-miss'ов. Уменьшая общее количество потоков, мы увеличиваем производительность каждого потока, но и ограничиваем количество одновременных подключений.

Событийно-ориентированная архитектура

Альтернативой синхронного блокирующего I/O является событийно-ориентированный подход. Благодаря асинхронным/неблокирующим вызовам каждый поток обслуживает несколько соединений. Когда к серверу пришел запрос, он ставит его на обработку, при этом, не дожидаясь её окончания, берется за другой запрос.

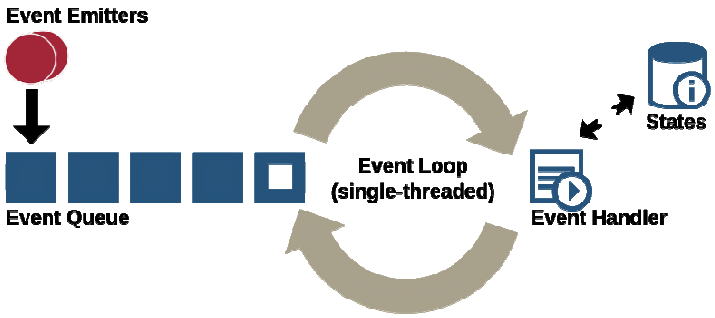


Рис. 3. Принцип работы event-driven подхода

Для каждого соединения выполняется асинхронная I/O операция. По завершении этой операции возникает соответствующее событие. Все события становятся в очередь (Event queue), которую обрабатывает единственный поток, как видно из рисунка 3. Данный поток выполняет так называемый цикл событий - извлечение события из очереди, вызов соответствующего обработчика события (Event handler) и последовательного выполнение инструкции этого обработчика. После этого поток берёт новое событие из очереди если оно есть, либо находится в ожидании. Вместо последовательных операций событийно-ориентированная модель использует асинхронные вызовы и callback'и, вызываемые по событиям. Из-за этого поток управления становится не таким очевидным, и отладка программы усложняется.

Использование этой архитектуры стало возможным благодаря асинхронным/неблокирующим операциям на уровне ОС и подходящим, высокопроизводительным интерфейсам уведомлений (epoll и kqueue).

В качестве веб-серверов данной архитектуры можно привести nginx, lighttpd, Tornado.

Масштабируемость Event-driven архитектур.

Если мы не связываем соединение с потоком, как в многопоточном варианте, то значительно уменьшается количество потоков на сервере. Как минимум у нас может быть только поток, выполняющий цикл событий, и несколько потоков в ядре системы для I/O операций. Таким образом, происходит уменьшение используемой памяти и затрат процессорного времени для переключения контекста между потоками.

В то время как многопоточный подход покрывает сразу параллелизм, основанный на I/O и CPU, то event-driven подход покрывает только параллелизм I/O. Для того, чтобы использовать несколько ЦП или ядер, необходимо дополнительно адаптировать сервер.

Самым очевидным подходом является запуск нескольких серверных процессов на одной машине. Он также называется “Подход N-копий для использования N экземпляров на хосте с N ЦП/ядрами”. В этом случае на машине будет запущено N экземпляров сервера, и балансировщик нагрузки будет распределять все пришедшие запросы на эти сервера.

Другим способом является разделение единого сокета между несколькими экземплярами, но это требует некой координации между всеми экземплярами. Это решение доступно в Node.js благодаря кластерному модулю, который порождает несколько экземпляров приложения и разделяет между ними единый сокет.

## Apache

Apache имеет многозадачную архитектуру, однако, это порождает некоторые ограничения. В Apache принята модель Preforking. Для распределения запросов между дочерними процессами используются мьютексы.

Архитектура Apache основана на пуле задач. Во время старта порождается пул задач, его контролирует главный процесс (Master server process), как показано на рисунке 4.

При поступлении нового запроса устанавливается коннект, управление которым передается процессу/потоку (Child Server) из пула. После завершения обработки Child Server закрывает коннект и засыпает до поступления нового запроса.

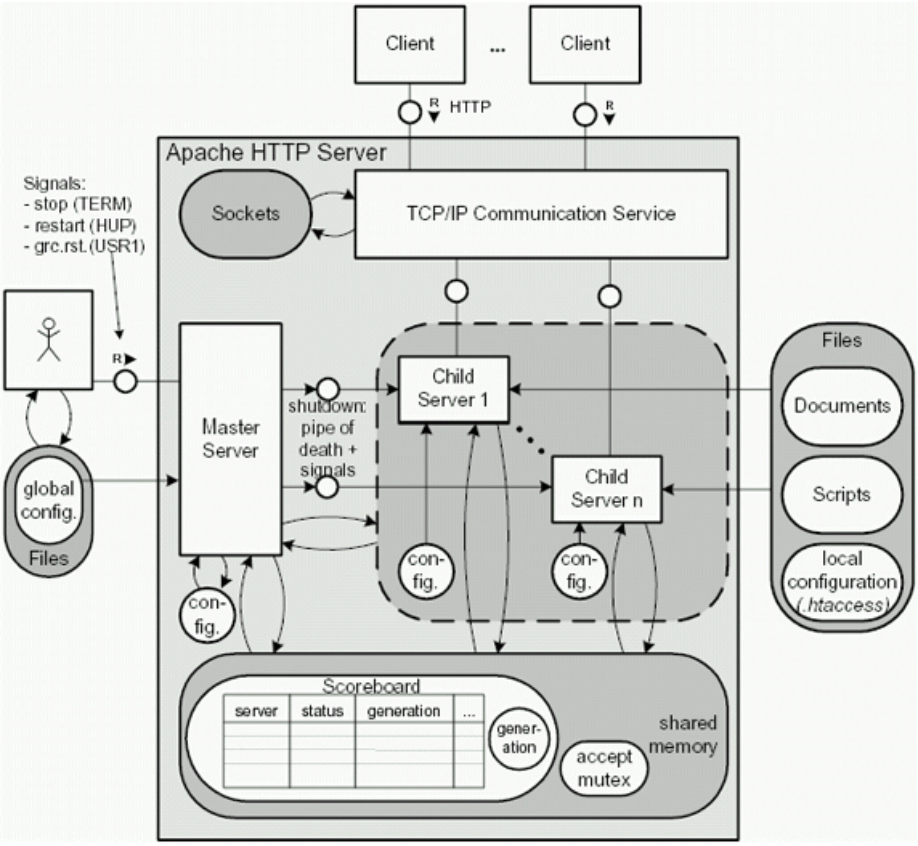


Рис. 4. Архитектура веб-сервера Apache.

Apache использует accept mutex для распределения поступающих клиентских запросов между дочерними процессами. Этот мьютекс делает активным в текущий момент на прослушке коннектов только один дочерний процесс с помощью системного вызова accept() и представляет собой контроль доступа к TCP/IP. После того, как дочерний процесс получает коннект, он начинает обрабатывать клиентский запрос, а на прослушивание становится другой дочерний процесс — в теории это называется Leader-Follower pattern.

Т.к. Child Server является форком главного процесса, он имеет глобальные настройки (global config.), заданные для главного процесса, а также локальные настройки, которые могут быть определены независимо от других процессов (см. рис. 4).

Взаимодействие главного и дочерних процессов происходит через общую память (shared memory), которая содержит accept mutex и scoreboard.

При создании дочернего процесса, информация о нем записывается в scoreboard. При изменении состояния дочерний процесс изменяет свою запись в scoreboard’е. Благодаря этому происходит обмен информацией между процессами.

При изменении администратором глобальных настроек необходимо перезапустить сервер, при этом все дочерние процесса должны быть перезапущены.

Для управления используются сигналы:

1. SIGTERM — остановка сервера (shutdown_pending).
2. SIGHUP — перезапуск сервера (restart_pending и graceful_mode).
3. SIGUSR1 — gracefull рестарт.

Gracefull означает, что дочерние процессы, обрабатывающие запрос в данный момент, не будут убиты, а закончат обработку.

Apache хорошо и легко использовать там, где нужно выполнение какого-то приложения.

В данный момент Apache является одним из наиболее используемых серверов и представляет собой яркий пример многопроцессного/многопоточного сервера. Ввиду своей архитектуры, он не используется как главный сервер в высоконагруженных системах, а выполняет бизнес-логику. Для обработки большого числа параллельных клиентских соединений часто используется nginx.

## Nginx

Nginx использует событийно-ориентированную архитектуру и может выполнять следующие функции:

- Акселерирование

В этом случае nginx является front-end сервером. Front-end сервер принимает клиентские запросы и отправляет их на back-end сервер, на котором происходит обработка запроса. Back-end сервер генерирует ответ на запрос, который имеет определенный размер. В настоящее время не все клиенты используют быстрое соединение, и передача ответа может занимать продолжительное время. Если бы ответ передавался back-end сервером, построенным, например, на многопоточной архитектуре, то это привело бы к деградации производительности из-за блокирования обрабатывающего процесса/потока. Поэтому ответы с back-end’а быстро передаются на nginx, который и отправляет их медленным клиентам. Это не влияет на его производительность, т.к. он позволяет поддерживать огромное число одновременных соединений в силу своей архитектуры.

- Балансировка нагрузки

В этом случае nginx ставится перед множеством backend’ов и распределяет нагрузку равномерно между ними.

- Отдача статических файлов (html, css, jpg, mp3 и т.д.)

**Устройство nginx**

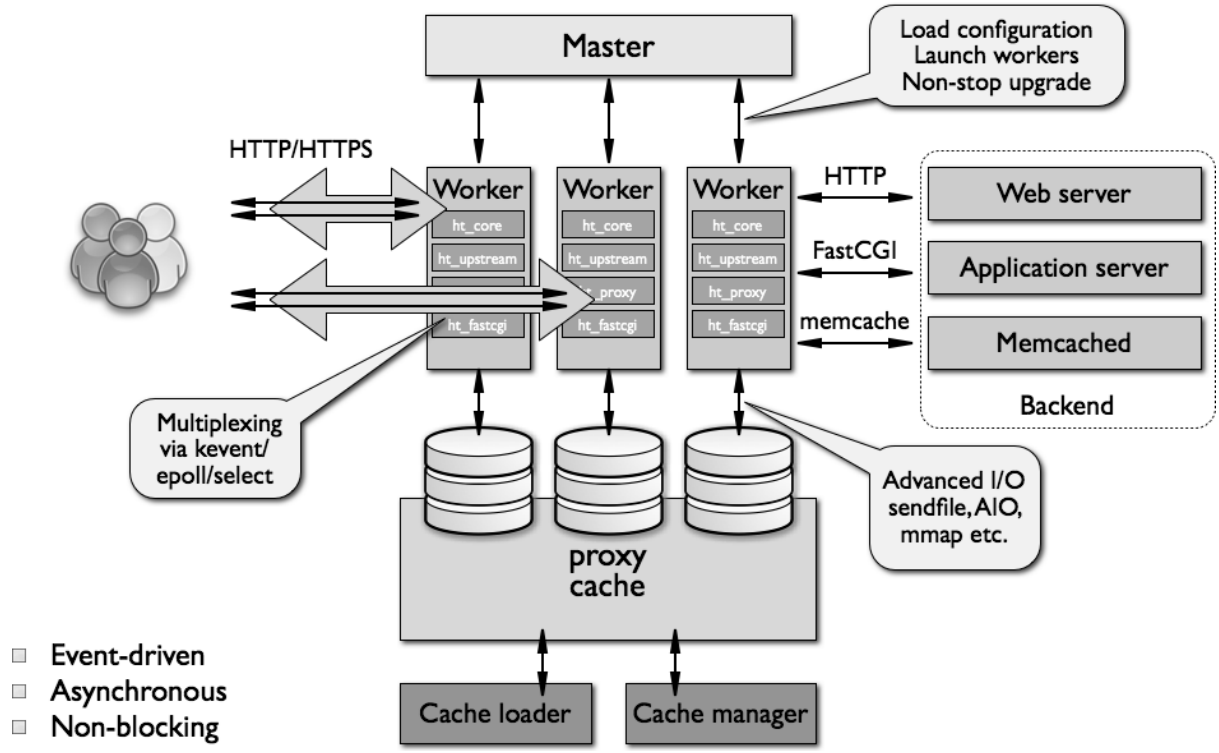


Рис. 5. Архитектура веб-сервера nginx.

Nginx делится на 2 части: ядро (реализует базовый функционал сервера) и некоторое количество модулей, отвечающих за функции для работы со специфичными протоколами и приложениями.

Для обработки соединений nginx использует каналы (pipeline), цепочки команд (chain) и модули.

Таким образом, для каждой операции (сжатие, преобразование данных, выполнение серверных сценариев, взаимодействие с вышестоящими серверами приложений) используется соответствующий модуль.

Функциональные модули можно разделить на следующие типы:

- Модули обработки событий. Зависят от операционной системы, на которой запущен сервер. Реализуют зависимые от операционной системы механизмы извещения о событиях, например, kqueue или epoll
- Модули обработки фазы
- Выходные фильтры
- Модули обработки именованных переменных
- Модули реализации протоколов
- Модули взаимодействия с вышестоящими серверами
- Балансировщики нагрузки

Стадии процесса обработки HTTP-запроса сервером NGINX:

1. Инициализация процесса обработки
2. Обработка заголовка
3. Обработка данных запроса
4. Вызов соответствующего обработчика
5. Переход от фазы к фазе при обработке

Nginx обрабатывает запрос в несколько фаз. В каждой фазе могут быть вызваны обработчики, формирующие соответствующий ответ, который передается на фильтр. Фильтры дополнительно обрабатывают сгенерированный поток. Все фильтры, как поставляемые вместе с nginx, так и сторонние, могут быть настроены на этапе сборки, а порядок запуска определяется на стадии компиляции.

Обработка данных фильтрами происходит по цепочке до тех пор, пока не будет вызван последний фильтр. После этого nginx завершает подготовку ответа. Следующему фильтру не обязательно ждать, пока предыдущий завершит работу. Он может начинать работу сразу, как только появится порция обработанных данных предыдущим фильтром. Ответ, генерируемый последним фильтром, передается клиенту.

Для мультиплексирования и извещения о событиях в nginx использует главный процесс, а исполнение конкретных задач назначается подчиненным процессам, как показано на рисунке 5.

Циклы обработки событий помещены в потоки-воркеры. В них выполняется обработка соединений. Количество воркеров ограничено, однако каждый воркер может обрабатывать тысячи подключений одновременно.

Воркер состоит из ядра и функциональных модулей (см. рис. 5). Ядро отвечает за поддержание работы цикла обработки событий и выполнение соответствующих модулей на каждом из этапов обработки запроса.

Для обработки запросов используются механизмы уведомления о событиях. Механизм выбирается в зависимости от операционной системы (например, kqueue, epoll или event ports).

Так как запросы обрабатываются несколькими воркерами, nginx хорошо масштабируется на несколько ядер ЦП. Кроме воркеров в nginx запускается также несколько специализированных процессов, например, диспетчер кэша и загрузчик.

Мастер-процесс запускает от суперпользователя, тогда как другие процессы запускаются от непривилегированного пользователя.

В настоящее время nginx является одним из самых популярных серверов, его используют такие крупные высоконагруженные проекты как mail.ru, yandex, vkontakte, rambler.

**Заключение**

В статье были рассмотрены различные архитектуры серверов. Можно сделать вывод, что наилучшей и выдерживающей большие нагрузки является Event-Driven архитектура. Она позволяет обрабатывать множество запросов одновременно, независимо от количества запущенных процессов и потоков. Также были описаны принципы реализации наиболее популярных серверов различных архитектур (Apache, nginx), рассмотрены режимы работы и функции. Лучшим выбором для высоконагруженных проектов является совокупность обоих серверов. Это позволяет ускорить работу сервиса, благодаря возможности поддержания большого количества параллельных соединений, распределить нагрузку на несколько серверов, выполняющих бизнес-логику. Однако в проектах без высокой нагрузки достаточно использовать лишь Apache, что упрощает настройку и поддержание.

**Список литературы**


1. У истоков Apache. Режим доступа: http://www.ibm.com/developerworks/ru/library/os-apache_3/index.html (дата обращения 25.09.14).
2. Интервью с создателем NGINX Игорем Сысоевым Режим доступа: http://xakep.ru/58122 (дата обращения 27.09.14).
3. Concurrent Programming for Scalable Web Architectures. Available at: http://berb.github.io/diploma-thesis/original/042_serverarch.html, accessed 03.10.14.