По сути существуют 3 модели работы сервера:

Простой последовательный сервер

Сервер открывает слушающий сокет и ждет, когда появится соединение (во время ожидания он находится в заблокированном состоянии). Когда приходит соединение, сервер обрабатывает его в том же контексте, закрывает соединение и снова ждет соединения. Очевидно, это далеко не самый лучший способ, особенно когда работа с клиентом ведется достаточно долго и подключений много. Кроме того, у последовательной модели есть еще много недостатков (например, невозможность использования нескольких процессоров), и в реальных условиях она практически не используется.

Это самая простая и понятная форма сервера: приняв запрос он его обрабатывает, и до завершения обработки недоступен для новых запросов. Обычно используется как эталон для сравнения: он имеет минимальное время реакции, так как не затрачивается время на порождение каких-либо механизмов параллелизма.

Многопроцессная (многопоточная). Сервер открывает слушающий сокет. Когда приходит соединение, он принимает его, после чего создает (или берет из пула заранее созданных) новый процесс или поток, который может сколь угодно долго работать с соединением, а по окончании работы завершиться или вернуться в пул. Главный поток тем временем готов принять новое соединение. Это наиболее популярная модель, потому что она относительно просто реализуется, позволяет выполнять сложные и долгие вычисления для каждого клиента и использовать все доступные процессоры. Однако у этого подхода есть и недостатки: при большом количестве одновременных подключений создается очень много потоков (или, что еще хуже, процессов), и операционная система тратит много ресурсов на переключения контекста. Особенно плохо, когда клиенты очень медленно принимают контент. Получаются сотни потоков или процессов, занятых только отправкой данных медленным клиентам, что создает дополнительную нагрузку на планировщик ОС, увеличивает число прерываний и потребляет достаточно много памяти.

Классический параллельный сервер

Классическая реализация параллельного сервера: по поступлению запроса порождается (вызовом fork()) новый обслуживающий процесс (клон родительского процесса). Родительский процесс при этом возвращается в режим прослушивания следующих поступающих запросов. После получения запроса от клиента и порождения отдельного процесса, он закрывает свою копию прослушивающего сокета, производит ретрансляцию через соединённый сокет, завершает соединение и завершается сам. Родительский же процесс закрывает свою копию соединённого сокета и продолжает прослушивание канала.

Сервер с предварительным созданием копий процесса

Для серверов, работающих на высоко интенсивных потоках запросов, традиционный fork-метод может оказаться затратным. Поэтом можно создать заранее некоторый пул обслуживающих процессов, каждый из которых до прихода клиентского запроса будет заблокирован. А после отработки клиентского запроса заблаговременно создать новый обслуживающий процесс. Эта техника известна как «предварительный fork» или pre-fork.

Параллельный сервер, создающий потоки по запросам

Классический параллельный сервер, но вместо параллельных клонов процессов теперь будем порождать параллельные потоки в том же адресном пространстве

Аналогично **– сервер с предварительным созданием потоков.**

Неблокируемые сокеты/конечный автомат. Сервер работает в рамках одного потока, но использует неблокируемые сокеты и механизм поллинга. Т.е. сервер на каждой итерации бесконечного цикла выбирает из всех сокетов тот, что готов для приема/отправки данных с помощью вызова select(). После того, как сокет выбран, сервер отправляет на него данные или читает их, но не ждет подтверждения, а переходит в начальное состояние и ждет события на другом сокете или же обрабатывает следующий, в котором событие произошло во время

обработки предыдущего. Данная модель очень эффективно использует процессор и память, но достаточно сложна в реализации. Кроме того, в рамках этой модели обработка события на сокете должна происходить очень быстро – иначе в очереди будет скапливаться много событий, и в конце концов она переполнится. Именно по такой модели работает nginx. Кроме того, он позволяет запускать несколько рабочих процессов (так называемых workers), т.е. может использовать несколько процессоров.