Network i/o = read/write to a socket
Disk i/o = read/write to a db/storage

BASIC web server:

The basic sequential steps for serving a request for static content are illustrated in Figure 1, and consist of the following:

Accept client connection - accept an incoming connection from a client by performing an accept operation on the server's listen socket. This creates a new socket associated with the client connection.

Read request - read the HTTP request header from the client connection's socket and parse the header for the requested URL and options.

Find file - check the server filesystem to see if the requested content file exists and the client has appropriate permissions. The file's size and last modification time are obtained for inclusion in the response header.

Send response header - transmit the HTTP response header on the client connection's socket.

Read file - read the file data (or part of it, for larger files) from the filesystem.

Send data - transmit the requested content (or part of it) on the client connection's socket. For larger files, the "Read file" and "Send data" steps are repeated until all of the requested content is transmitted.

ISSUE:

All of these steps involve operations that can potentially block. Operations that read data or accept connections from a socket may block if the expected data has not yet arrived from the client. Operations that write to a socket may block if the TCP send buffers are full due to limited network capacity. Operations that test a file's validity (using stat()) or open the file (using open()) can block until any necessary disk accesses complete. Likewise, reading a file (using read()) or accessing data from a memory-mapped file region can block while data is read from disk.

Therefore, a high-performance Web server must interleave the sequential steps associated with the serving of multiple requests in order to overlap CPU processing with disk accesses and network communication. The server's architecture determines what strategy is used to achieve this interleaving.
In addition to its architecture, the performance of a Web server implementation is also influenced by various optimizations, such as caching. In Section 5, we discuss specific optimizations used in the Flash Web server

APPROACH: **Single-process event-driven**

The approach is quite simple: you simply wait for something (i.e., an "event") to occur; when it does, you check what type of event it is and do the small amount of work it requires (which may include issuing I/O requests, or scheduling other events for future handling, etc.). That's it!

A SPED server can be thought of as a state machine that performs **one BASIC** step associated with the serving of an HTTP request at a time, thus interleaving the processing steps associated with many HTTP requests. In each iteration, the server performs a select()/epoll() to check for completed I/O events (new connection arrivals, completed file operations, client sockets that have received data or have space in their send buffers.) When an I/O event is ready, it completes the corresponding **BASIC** step and initiates the next step associated with the HTTP request, if appropriate.

Ex.

```
while (1) {
        events = getEvents();
        for (e in events)
                processEvent(e); #event handler
}
```

Imp: when a handler processes an event, it is the only activity taking place in the system; thus, deciding which event to handle next is equivalent to scheduling. This explicit control

over scheduling is one of the fundamental advantages of the event-based approach.

Blocking (or synchronous) interfaces do all of their work before returning to the caller;
Non-Blocking (or **Asynchronous**) interfaces begin some work but return immediately, thus letting whatever work that needs to be done get done in the background.

Non-Blocking vs Asynchronous
Differs in their return value
ex . in case of read
Non-blocking = returns with the data that can be read w/o blocking
Asynchronous = returns two values: 0 for success (i.e all the data is read) and EINPROGRESS for if NOT all the data is read and initiates the read in the background.

Main Adv: **Event-based servers enable fine-grained control over the scheduling of tasks(NO over head no context switch and scheduling of threads and NO problems related to threads as in critical section(solved by Locks) and no conditional variables required)**
BUT

ISSUE: if event handler wants to perform **NON-BASIC STEP(generally I/O operations)** that will take some time to execute. (Ex. Disk i/o, Network i/o, select a backend server etc)

"However, to maintain such control, **no call that blocks** the execution of the caller can ever be made; failing to obey this design tip will result in a blocked event-based server, frustrated clients, and serious questions as to whether you ever read this part of the book".
This is **NOT** an issue with threads: With a thread-based server, while the thread issuing the I/O request suspends (waiting for the I/O to complete), other threads can run, thus enabling the server to make progress.

"With an event-based approach, however, there are no other threads to run: just the main event loop. And this implies that if an event handler issues a call that blocks, the entire server will do just that: block until the call completes. When the event loop blocks, the system sits idle and thus is a huge potential waste of resources. **We thus have a rule that must be obeyed in event-based systems: NO blocking calls are allowed.**"

SOLUTION: **Asynchronous I/O**

Meaning: These interfaces enable an application to issue an I/O request and return control immediately to the caller, before the I/O has completed; additional interfaces enable an application to determine whether various I/Os have completed.(As mentioned above)

Done by API in UNIX explained here => link

BUT:
However, a problem associated with SPED servers is that many current operating systems do **NOT** provide suitable support for asynchronous **disk** operations.
=> However, in these operating systems that do not have asynchronous disk operations, **non-blocking** read and write operations work as expected on **network sockets and pipes**, but may actually **block** when used on **disk** files. As a result, supposedly non-blocking read operations on files may still block the caller while disk I/O is in progress.

And the ones who do like many UNIX systems provide alternate APIs(as mentioned above in the link) that implement true asynchronous disk I/O, but these APIs are generally not integrated with the select/epoll operation. This makes it difficult or impossible to simultaneously check for completion of network and disk I/O events in an efficient manner.

Moreover, operations such as open and stat on file descriptors may still be blocking.
For these reasons, existing SPED servers do not use these special asynchronous disk interfaces.
As a result, file read operations that do not hit in the file cache may
cause the main server thread to block, causing some loss in concurrency and performance.

=> **Existing SPED servers uses non-blocking systems calls to perform asynchronous I/O operations.=> Ex. non-blocking read/write for n/w I/O that performs exactly like asynchronous I/O.**

TO resolve everything we use: AMED(Asymmetric Multi-Process Event-Driven)

link :
For example, let us examine the interface provided on a Mac (other systems have similar APIs). The APIs revolve around a basic structure, the struct aiocb or AIO control block in common terminology. A simplified version of the structure looks like this (see the manual pages for more information):

```
struct aiocb {
        int aio_fildes; // File descriptor
        off_t aio_offset; // File offset
        volatile void *aio_buf; // Location of buffer
        size_t aio_nbytes; // Length of transfer
};
```

To issue an asynchronous read to a file, an application should first fill in this structure with the relevant information: the file descriptor of the file to be read (aio_fildes), the offset within the file (aio_offset) as well as the length of the request (aio_nbytes), and finally the target memory location into which the results of the read should be copied (aio_buf).
After this structure is filled in, the application must issue the asynchronous call to read the file; on a Mac, this API is simply the asynchronous read API:

```
int aio_read(struct aiocb *aiocbp);
```

This call tries to issue the I/O; if successful, it simply returns right away and the application (i.e., the event-based server) can continue with its work.
There is one last piece of the puzzle wemust solve, however.
Question: How can we tell when an I/O is complete, and thus that the buffer (pointed to by asynchronous _i/o structure) now has the requested data within it?

Solution:
1. Polling: check regularly after some time if the buffer is full or not through an api call(Ex. aio_error)

```
int aio_error(const struct aiocb *aiocbp);
```

Prob:if a program has tens or hundreds of I/Os issued at a given point in time, should it simply keep checking each of them repeatedly.

2. Interrupt: To remedy this issue, some systems provide an approach based on the **interrupt**. This method uses UNIX **signals** to inform applications when an asynchronous I/O completes, thus removing the need to repeatedly ask the system

IMPLEMENTING: **Non-Blocking read/write for n/w I/O that performs exactly like asynchronous I/O.**

UNDERSTAND:

 # Level-Triggered and Edge-Triggered Notification

Level-triggered notification: A file descriptor is considered to be ready if it is possible to perform an I/O system call without blocking.

Edge-triggered notification: edge-triggered model delivers events only when **changes** occur on the monitored file descriptor.

Notification is provided if there is I/O activity (e.g., new input) on a file descriptor since it was last monitored.

An edge is not "somebody wrote more data". An edge is "there was no data, now there is data".

And a level triggered event is *also* not "somebody wrote more data". A level-triggered signal is simply "there is data".

Notice how neither edge nor level are about "more data". One is about the edge of "no data" -> "some data", and the other is just a "data is available".

LEVEL-TRIGGERED:

When we employ level-triggered notification, we can check the readiness of a file descriptor at any time. This means that when we determine that a file descriptor is ready (e.g., it has input available), we can perform some I/O on the descriptor, and then repeat the monitoring operation to check if the descriptor is still ready (e.g., it still has more input available), in which case we can perform more I/O, and so on.

In other words, because the level-triggered model allows us to repeat the I/O monitoring operation at any time, it is **not necessary** to perform as much I/O as possible (e.g., read as many bytes as possible) on the file descriptor (or even perform any I/O at all) each time we are notified that a file descriptor is ready.

EDGE-TRIGGERED:

when we employ edge-triggered notification, we receive notification only when an I/O event occurs. We don't receive any further notification until another I/O event occurs.

Ex. of different between level vs edge

Suppose that we are using epoll to monitor a socket for input (EPOLLIN), and the following steps occur:

1. Input arrives on the socket.
2. We perform an epoll_wait(). This call will tell us that the socket is ready, regardless

of whether we are employing level-triggered or edge-triggered notification.
3. We perform a second call to epoll_wait().
If we are employing level-triggered notification, then the second epoll_wait() call will inform us that the socket is ready. If we are employing edge-triggered notification, then the second call to epoll_wait() will block, because no new input has arrived since the previous call to epoll_wait().

Ex2: Difference between level vs edge triggered for EPOLLOUT
In level-triggered mode, the EPOLLOUT event will be generated whenever a socket is ready for writing. This means that if you call epoll_wait with EPOLLOUT set, and a socket is available for writing, the event will be returned immediately. If you don't write to the socket, the EPOLLOUT event will continue to be returned by epoll_wait.

In edge-triggered mode, the EPOLLOUT event will only be generated ONCE, when the socket transitions from not-ready-for-writing to ready-for-writing. Even though the socket may already be available for writing, the application may not be aware of it(because we monitor it for only incoming data), so the EPOLLOUT event is necessary to ensure that the application is notified when the socket is ready. You will only receive an EPOLLOUT event when the socket becomes available for writing again after having previously been unavailable.
Ex. if we modify to EPOLLOUT from earlier EPOLLIN => this will lead to an event generation because socket state changes from non-writable(i.e since buffer has data that needs to be read or buffer might be free as we have read all the data but the program is not aware about it since we were only mintoring for incoming data) to writable(i.e buffer is free to send data)

Furthermore, when an I/O event is notified for a file descriptor, we usually don't know how much I/O is possible (e.g., how many bytes are available for reading). Therefore, programs that employ edge-triggered notification are usually designed according to the following rules:

1. After notification of an I/O event, the program should—at some point—perform as much I/O as possible (e.g., read as many bytes as possible) on the corresponding file descriptor. If the program fails to do this, then it might miss the opportunity to perform some I/O, because it would not be aware of the need to operate on the file descriptor until another I/O event occurred. This could lead to spurious data loss or blockages in a program. We said **"at some point,"** because sometimes it may not be desirable to perform all of the I/O immediately after we determine that the file descriptor is ready. The **problem** is that we may starve other file descriptors of attention if we perform a large amount of I/O on one file descriptor known as **STARVATION of File Descriptor.**

2. **If the program employs a loop to perform as much I/O as possible on the file**

descriptor, and the descriptor is marked as blocking, then eventually an I/O system call will block when no more I/O is possible. For this reason, each monitored file descriptor is normally placed in nonblocking mode, and after notification of an I/O event, I/O operations are performed repeatedly until the relevant system call (e.g., read() or write()) fails with the error EAGAIN or EWOULDBLOCK.

Thus, the general framework forusing edge-triggered epoll notification is as follows:
1. Make all file descriptors that are to be monitored nonblocking.
2. Build the epoll interest list using epoll_ctl().
3. Handle I/O events using the following loop:
a) Retrieve a list of ready descriptors using epoll_wait().
b) For each file descriptor that is ready, process I/O until the relevant system call (e.g., read(), write(), recv(), send(), or accept()) returns with the error EAGAIN or EWOULDBLOCK.

EX.

```
# Non-blocking mode means it gives error when the call is suppose to block the program
but because it generates an error we use that error to exit from the INFINITE loop
# Set the socket to nonblocking mode
server_socket.setblocking(0)
```

```
for fileno, event in events:
            # Handle new connections
            if fileno == server_socket.fileno():
                # onaccept(function) and choosing server and create connection to the
selected server
                while True:
                    try:
                        client_connection, client_address = server_socket.accept()
                    except socket.error as e:
                         if e.errno == (errno.EWOULDBLOCK | errno.EAGAIN):
                            break
                         else:
                            print("Can't establish connection with client:
{}".format(e))
                            raise


                    client_connection.setblocking(0)

                    client_fd = client_connection.fileno()
                    epoll.register(client_fd, select.EPOLLIN | select.EPOLLET )
```

```
                    connections[client_fd] = client_connection
                    print("New Client connection from {} on socket:
{}".format(client_address,client_fd))
```

Preventing file-descriptor **starvation** when using edge-triggered notification
Starvation:
Suppose that we are monitoring multiple file descriptors using edge-triggered notification, and that a ready file descriptor has a large amount (perhaps an endless stream) of input available. If, after detecting that this file descriptor is ready, we attempt to consume all of the input using nonblocking reads, then we risk starving the other file descriptors of attention (i.e., it may be a long time before we again check them for readiness and perform I/O on them).

One **solution** to this problem is for the application to maintain a list of file descriptors that have been notified as being ready, and execute a loop that continuously performs the following actions:

1. Monitor the file descriptors using epoll_wait() and add ready descriptors to the application list. If any file descriptors are already registered as being ready in the application list, then the timeout for this monitoring step should be small or 0, so that if no new file descriptors are ready, the application can quickly proceed to the next step and service any file descriptors that are already known to be ready.
2. Perform a limited amount of I/O on those file descriptors registered as being ready in the application list (perhaps cycling through them in round-robin fashion, rather than always starting from the beginning of the list after each call to epoll_wait()). A file descriptor can be removed from the application list when the relevant nonblocking I/O system call fails with the EAGAIN or EWOULDBLOCK error.
Although it requires extra programming work, this approach offers other benefits in addition to preventing file-descriptor starvation. For example, we can include other steps in the above loop, such as handling timers and accepting signals with sigwaitinfo() (or similar).

Starvation considerations can also apply when using signal-driven I/O, since it also presents an edge-triggered notification mechanism. By **contrast**, starvation considerations **don't** necessarily apply in applications employing **a level-triggered notification** mechanism. This is because we can employ blocking file descriptors

with level-triggered notification and use a loop that continuously checks descriptors for readiness, and then performs **some** I/O on the ready descriptors before once more checking for ready file descriptors.
Ex. In case of read from socket [i.e n/w I/O] set the MAX_BUFFER size

In depth understanding of file descriptors = A closer look at epoll Semantics[Pg 1363]
 # Understand epoll API from book
Reasons why epoll is better than select and poll = Pg 1365

SUMMARY:-

Event based used to have fine-grained control over scheduling of tasks.
We monitor sockets for various events.

We achieve our goal of getting control over the scheduling of tasks mainly using HOW events are generated(i.e event triggering type) and WHAT type of events are generated(i.e event state). We manage the scheduling of multiple connections(i.e execution of multiple threads if we implement it through a multi-threaded program) by deciding which event to handle next (i.e equivalent to scheduling) and deciding which event to handle next is basically how events are generated since they are added automatically to the total event list(ready list of epoll). Generation of events is basically setting the triggering type(i.e level, edge, oneshot)[note1]. We set the right event triggering type for the sockets we are <b monitoring\b>(i.e edge_oneshot for new connection sockets and level for listening_socket) and then we handle these events by identifying their event state(i.e this basically tells us what type of event we are dealing with....incoming data or outgoing data) and if need be also modify the event state of a monitored file decsiptor[note2] and then take actions accordingly....and thus achieving the goal that Event-based servers enable fine-grained control over the scheduling of tasks.

note:
1. Be cautious in setting the triggering type of events...if it is set wrong it will lead to wrong event generation which will lead to handling of that event but since that event should be generated in the first place the handling of event will lead to disastorous outputs.
(Ex. Suppose an event of type EPOLLIN is generated => handling this event will lead to reading all the data from the buffer.
Ideally, next event generated should be of EPOLLOUT to send all the response data in the buffer but due to wrong triggering an EPOLLIN event is generated => handling this event will lead to reading an empty buffer)

IDEALLY
Recieved incoming data on socket: 8
Read Data: GET / HTTP/1.1
Host: 128.6.4.101:8085
Forwarding packets to socket: 9
Recieved incoming data on socket: 9
Read Data: HTTP/1.1 200 OK
Content-Type: text/html
Hello, world!
Forwarding packets to socket: 8
Closing Connection with client and backend server on socket 8 and 9

ERROR
Recieved incoming data on socket: 8
Read Data: GET / HTTP/1.1
Host: 128.6.4.101:8085
Forwarding packets to socket: 9
Recieved incoming data on socket: 9
Read Data: HTTP/1.1 200 OK
Content-Type: text/html
Hello, world!
<b Recieved incoming data on socket: 9
Read Data: \b>
Forwarding packets to socket: 8
Closing Connection with client and backend server on socket 8 and 9

References:
Server:
Flash: An efficient and portableWeb server
https://berb.github.io/diploma-thesis/community/042_serverarch.html

Event-based programming epoll:
[epoll code: https://android.googlesource.com/kernel/common/+/refs/heads/android-mainline/fs/eventpoll.c]
OSTEP
LINUX
https://suchprogramming.com/epoll-in-3-easy-steps/https://suchprogramming.com/epoll-in-3-easy-steps/
https://medium.com/vaidikkapoor/understanding-non-blocking-i-o-with-python-part-1-ec31a2e2db9b
example = http://scotdoyle.com/python-epoll-howto.html
https://copyconstruct.medium.com/nonblocking-i-o-99948ad7c957 [Non-Blocking i/o and async i/o]

https://www.cs.rutgers.edu/courses/416/classes/fall_2009_ganapathy/slides/io.pdf
https://eli.thegreenplace.net/2017/concurrent-servers-part-3-event-driven/
The **above** article includes event-based and libuv => node.js
Node.js related material
https://blog.bitsrc.io/event-based-asynchronous-programming-abb0447381eb
https://nodejs.org/en/docs/guides/blocking-vs-non-blocking [official non-blocking vs blocking]
http://docs.libuv.org/en/v1.x/design.html [official]

SCALING UP THE LOAD BALANCER using Epoll – MUST READ
https://idea.popcount.org/2017-02-20-epoll-is-fundamentally-broken-12/#fnref:2

LOAD BALANCER:
https://storage.googleapis.com/pub-tools-public-publication-data/pdf/44824.pdf
https://sre.google/workbook/managing-load/
https://sre.google/sre-book/load-balancing-datacenter/
https://medium.com/must-know-computer-science/system-design-load-balancing-1c2e7675fc27
https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/

Reactor Pattern

The Reactor pattern [Sch95] targets *synchronous, non-blocking I/O* handling and relies on an event notification interface. On startup, an application following this pattern registers a set of resources (e.g. a socket) and events (e.g. a new connection) it is interested in. For each resource event the application is interested in, an appropriate event handler must be provided--a callback or hook method. The core component of the Reactor pattern is a synchronous event demultiplexer, that awaits events of resources using a blocking event notification interface. Whenever the synchronous event demultiplexer receives an event (e.g. a new client connection), it notifies a dispatcher and awaits for the next event. The dispatcher processes the event by selecting the associated event handler and triggering the callback/hook execution.
The Reactor pattern thus decouples a general framework for event handling and multiplexing from the application-specific event handlers. The original pattern focuses on a single-threaded execution. This requires the event handlers to adhere to the non-blocking style of operations. Otherwise, a blocking operation can suspend the entire application. Other variants of the Reactor pattern use a thread pool for the event handlers. While this improves performance on multi-core platforms, an additional overhead for coordination and synchronization must be taken into account.