Advanced Computer Networks Lab

Dr Sudipta Saha
Associate Professor
Dept of Computer Science & Engineering
Indian Institute of Technology Bhubaneswar

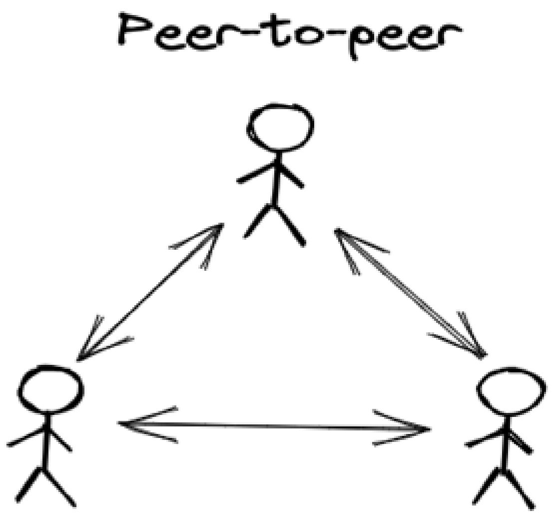
Tutorial-2

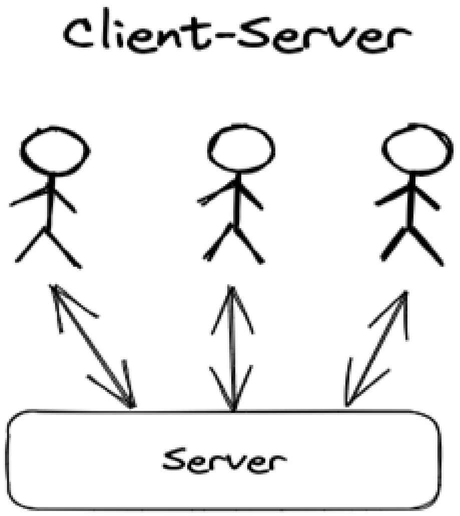
Teaching Assistant: Nitin

p2p Networks

A decentralized network model where all devices, or "peers," have equal status. Each computer can act as both a client (requesting resources) and a server (sharing resources) directly with others, without needing a central authority.

- Client-Server: Centralized. One powerful server provides services to many clients.
- P2P: Distributed. All peers are equal, sharing resources directly among themselves.



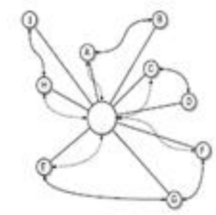


p2p Unstructured Topologies

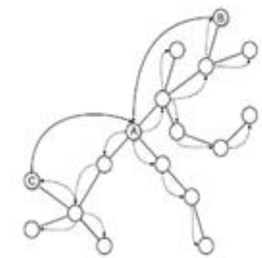
A decentralized network design, common in P2P systems, where connections between peers are established arbitrarily and without a specific organizational pattern. There is no predefined rule or map for how the network is formed.

Key Characteristics:

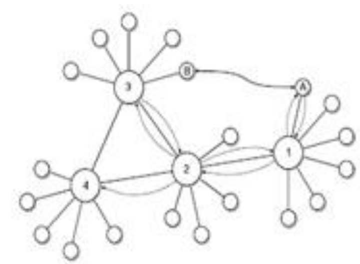
- Random Connections: Peers connect to whichever neighbors they discover, often forming random, ad-hoc links.
- Search by Flooding: To find a file or resource, queries are broadcast (flooded) to all connected neighbors, which is simple but can generate significant network traffic.
- Resilient & Easy to Join: Highly robust to peers frequently joining/leaving (churn) and easy for new peers to integrate.
- Inefficient Search: There is no guarantee of locating a resource, even if it exists in the network.



(a) Napster & BitTorrent



(b) Gnutella



(c) Gnutella/Overnet/eDonkey2000

Common Use: Early file-sharing networks

(e.g., Gnutella), Napster, bittorrent.

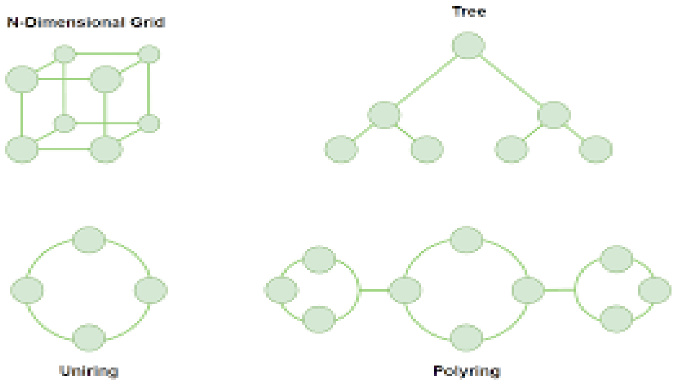
p2p Structured Topologies

A controlled P2P network architecture where connections between peers are organized according to a specific, deterministic rule set—most commonly a Distributed Hash Table (DHT). This creates a predictable global structure that allows efficient resource location.

p2p Structured Topologies

Key Characteristics:

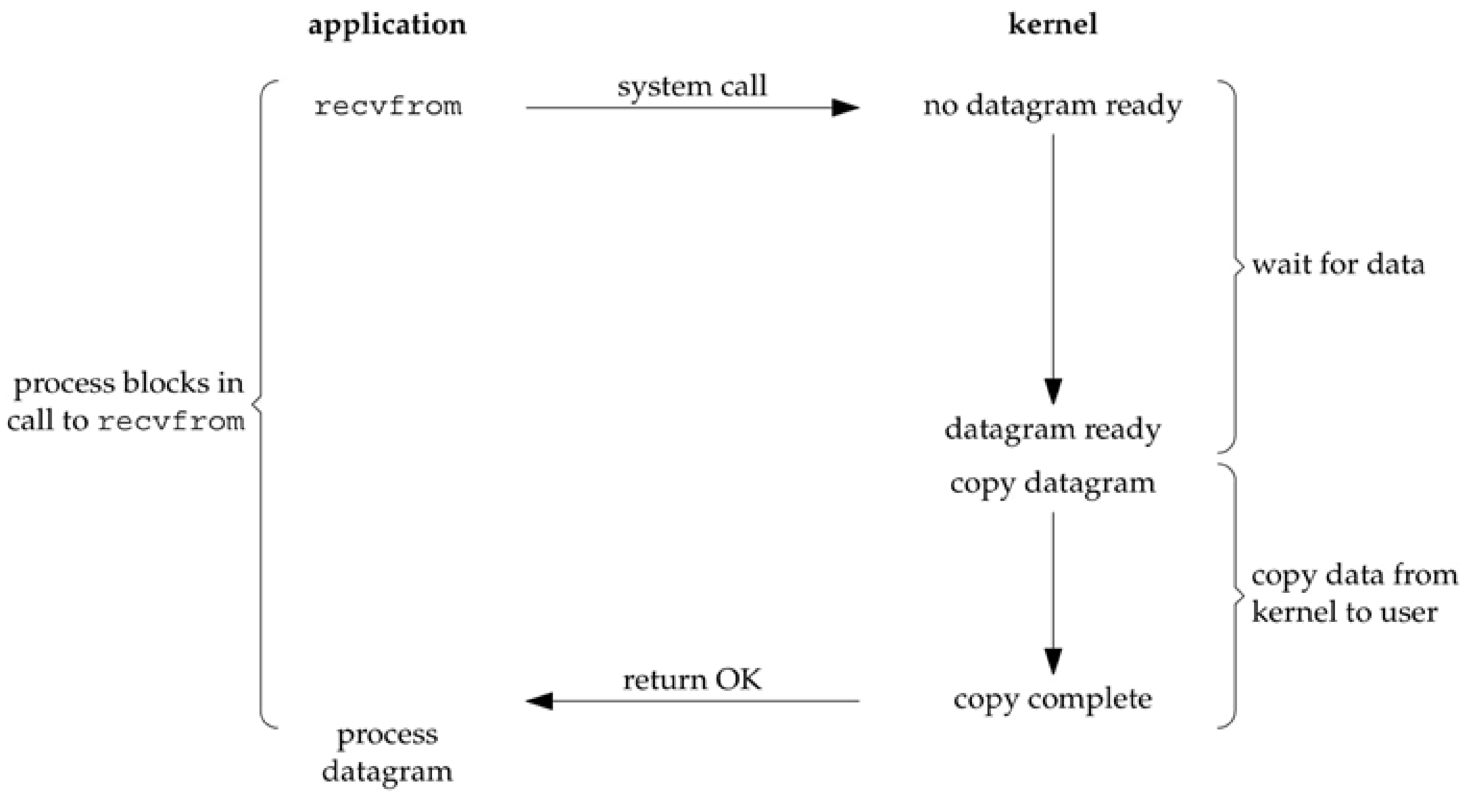
- Organized & Predictable: Peers and the resources they hold are mapped to specific, logical positions within the network using unique identifiers (like keys).
- Efficient Lookup: Any resource can be reliably located within a small, predictable number of steps (usually O(log n)), without flooding the network.
- Scalable: Provides guaranteed discovery and scales efficiently to very large networks.
- Maintenance Overhead: Requires more complex protocols to maintain the structure as peers join and leave (churn).



BLOCKING MODEL

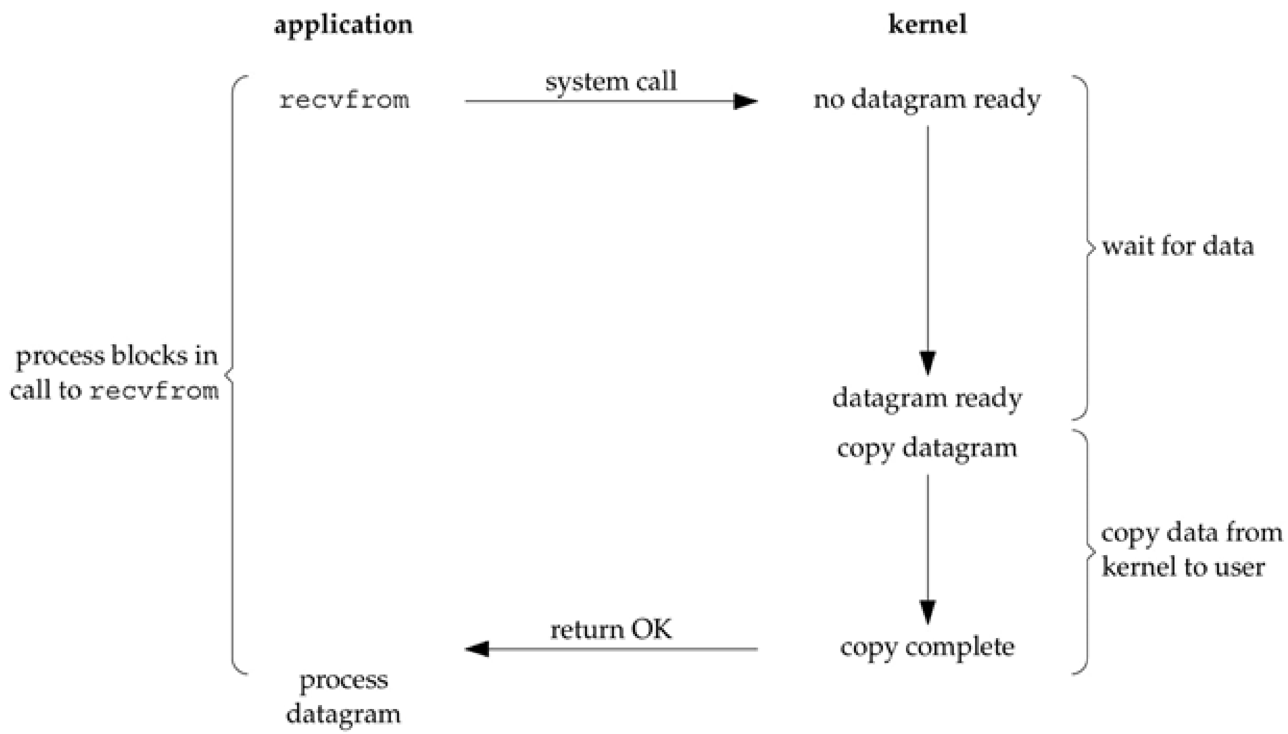
A default mode of operation in socket programming where a function call (like `send()` or `recv()`) halts the execution of the program until the requested network operation is fully complete or an error occurs.

Blocking (Synchronous): "Wait here until this specific I/O operation is done."



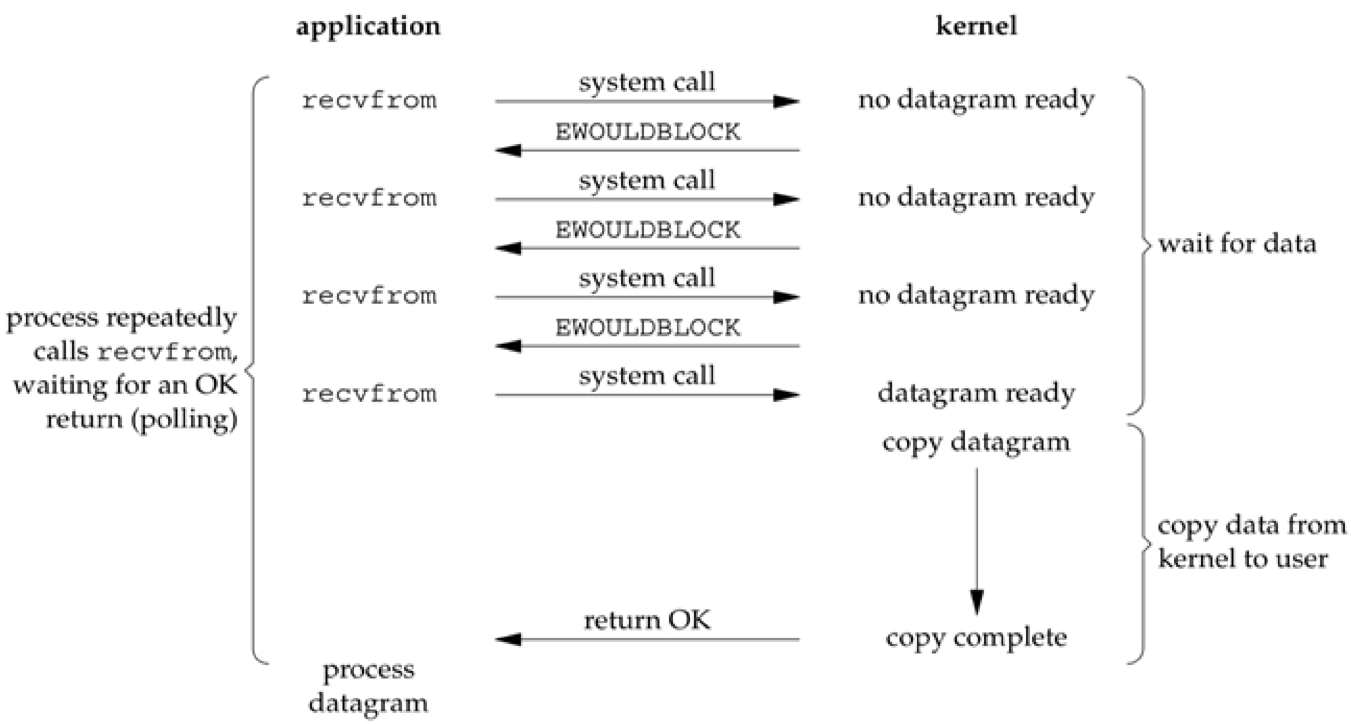
BLOCKING MODEL

- `accept()` - Sleeps until someone connects
- `connect()` - Sleeps until connection succeeds/fails
- `recv()` - Sleeps until data arrives
- `send()` - Sleeps (rarely) if buffer is full



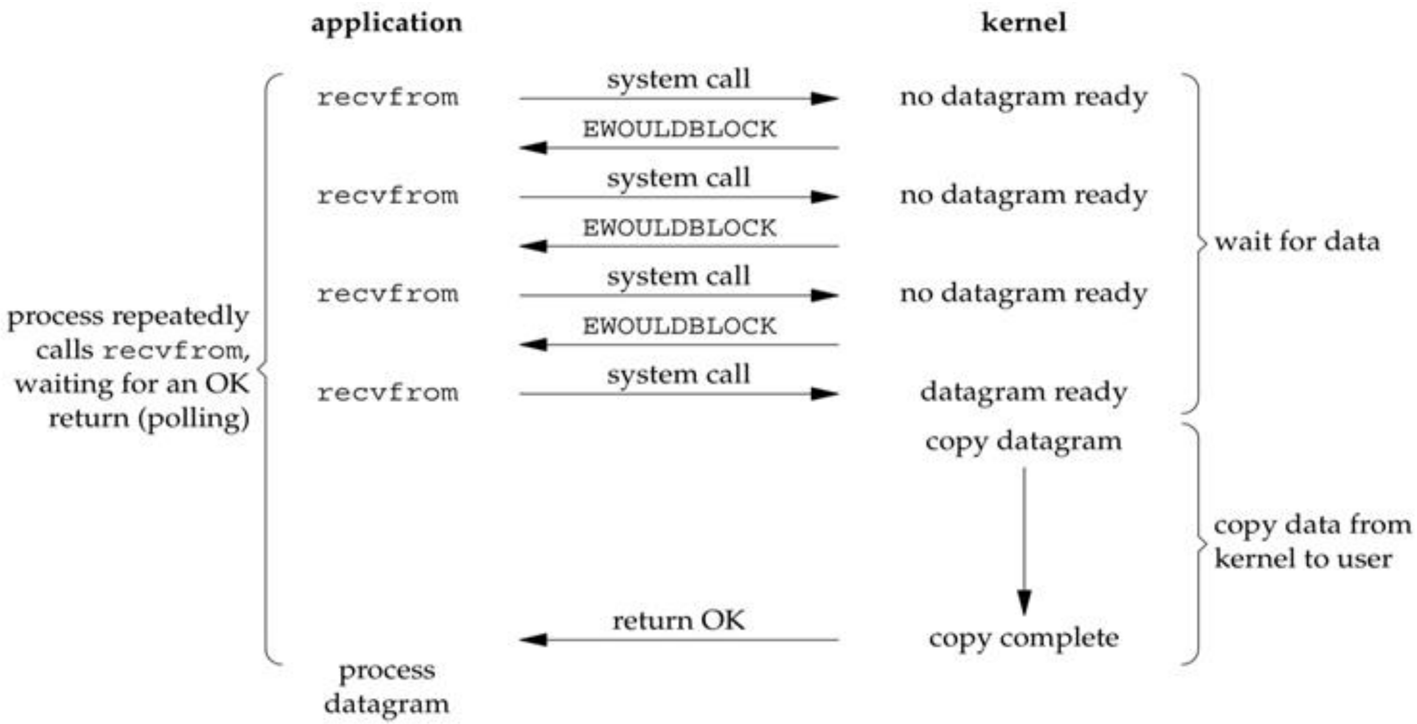
NON BLOCKING MODEL

An operational mode in socket programming where a function call (like send() or recv()) returns immediately, regardless of whether the operation could be completed. This allows the program to continue execution without waiting for the underlying network I/O to finish.



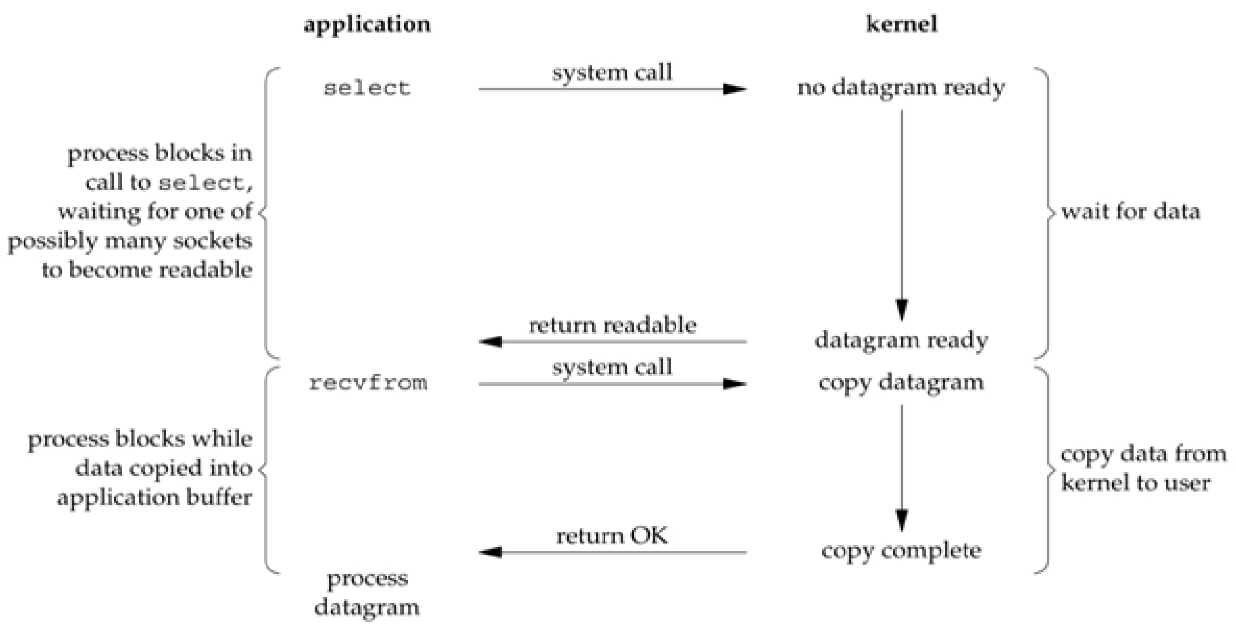
NON BLOCKING MODEL:POLLING

The technique of repeatedly checking the status of one or more non-blocking sockets to see if they are ready for a read or write operation. Instead of relying on the operating system to notify the program when data is available, the program actively and periodically inquires about each socket's state.



I/O MULTIPLEXING

A high-efficiency programming pattern that allows a single thread to monitor multiple socket descriptors simultaneously, and be notified when one or more become ready for I/O operations (e.g., readable or writable). It is the optimized implementation of polling for production systems.



select()

- A core function for I/O multiplexing that allows a program to monitor multiple file descriptors (typically sockets) simultaneously to see if any become ready for I/O operations. It's the original and most portable method for implementing event-driven network servers.
- The program passes three sets of file descriptors to select(): those it wants to monitor for read readiness, write readiness, and exceptional conditions. The call blocks until at least one descriptor in any set becomes ready or until a timeout expires.
- Descriptor Sets: Uses fixed-size bitmask arrays (fd_set) to represent sets of descriptors, limiting the maximum number of descriptors that can be monitored (typically 1024).
- Linear Scanning: On return, the program must iterate through all monitored descriptors to determine which ones are ready, which becomes inefficient with many connections.
- Destructive Call: The sets are modified by the call; they must be rebuilt before each subsequent select() invocation.

select()

*int select(int maxfdp1, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval *timeout);*

Returns: positive count of ready descriptors, 0 on timeout, −1 on error.

- **maxfdp1**:The maximum file descriptor value + 1 in all sets. Determines how much of the descriptor space the kernel scans.
- **readset**:Set of descriptors to monitor for read readiness (incoming data, connection requests, socket closure).
- **writeset**:Set of descriptors to monitor for write readiness (when data can be sent without blocking).
- **exceptset**:Set of descriptors to monitor for exceptional conditions (out-of-band data, socket errors).
- **timeout**:Maximum time to wait for activity. NULL = block forever, 0 = return immediately (poll), non-zero = wait specified time.

select()

- Each bit position represents a file descriptor number
- Example: Bit 5 = descriptor 5, Bit 32 = descriptor 32
- One fd_set can track hundreds of descriptors in just a few bytes.
- Defined by FD_SETSIZE (typically 1024 on most systems),Cannot monitor descriptors ≥ 1024 without recompiling kernel, Wastes memory if monitoring only a few descriptors
- After select() returns, must check all possible descriptors 0...maxfd
- Inefficient for checking a few active descriptors among many possible

*void FD_ZERO(fd_set *fdset); /* clear all bits in fdset */*

*void FD_SET(int fd, fd_set *fdset); /* turn on the bit for fd in fdset */*

*void FD_CLR(int fd, fd_set *fdset); /* turn off the bit for fd in fdset */*

*int FD_ISSET(int fd, fd_set *fdset); /* is the bit for fd on in fdset ? */*

poll()

An alternative I/O multiplexing function that overcomes the fixed descriptor limit of select(). Instead of bitmasks, poll() uses an array of structures to monitor file descriptors.

*int poll(struct pollfd *fds, nfds_t nfds, int timeout);*

Returns: count of ready descriptors, 0 on timeout, −1 on error

```
struct pollfd {
    int   fd;         /* file descriptor */
    short events;     /* events to watch for (input) */
    short revents;    /* events that occurred (output) */
};
```

- `nfds_t nfds` specifies the number of elements in the `fds` array that should be processed by `poll()`. It tells the kernel: "Check only the first `nfds` entries in this array."
- **timeout**:Maximum time to wait for activity. NULL = block forever, 0 = return immediately (poll), non-zero = wait specified time.

poll(): Common Event Flags

Flag	Meaning	When Used
`POLLIN`	Data available to read	Normal/priority data ready
`POLLPRI`	Urgent data available	Out-of-band data on TCP socket
`POLLOUT`	Writing will not block	Ready for output
`POLLERR`	Error condition	Always monitored automatically
`POLLHUP`	Hang up	Connection closed
`POLLNVAL`	Invalid request	Descriptor not open

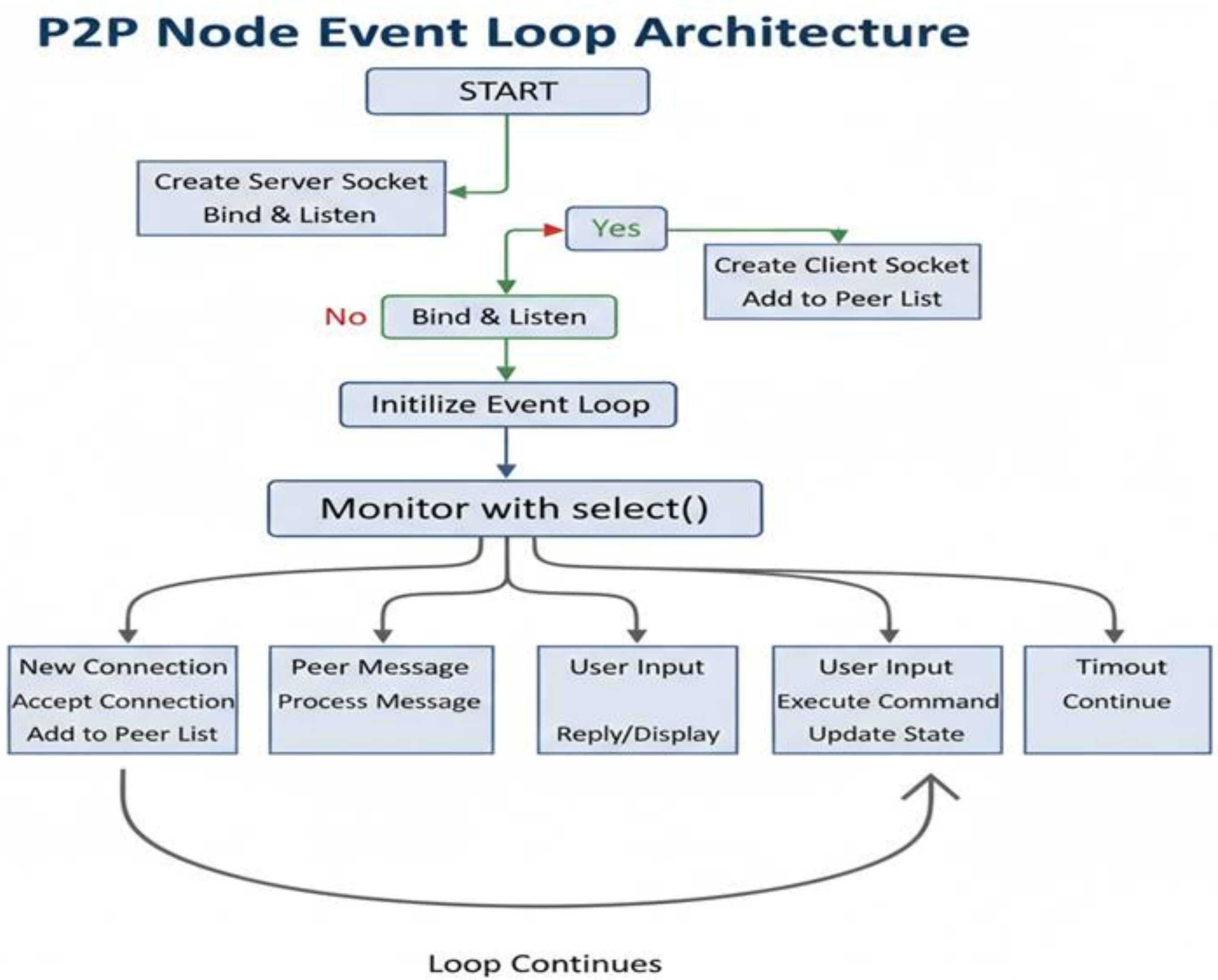
select() vs poll()

Feature	`select()`	`poll()`
Descriptor Limit	Fixed (FD_SETSIZE, typically 1024)	Dynamic (limited by system resources)
Event Separation	Input/Output sets modified on return	Separate `events` (input) and `revents` (output) fields
Portability	Widely portable	Almost as portable (some older Unix systems may not have it)
Performance	O(n) scan of all watched descriptors	O(n) scan of array elements

Writing a single threaded peer code

```
int main() {
    // 1. Initialize peer socket
    peer_socket = setup_peer();
    // 2. Add initial connections
    add_to_monitor(peer_socket, POLLIN);
    add_to_monitor(STDIN_FILENO, POLLIN);  // User input
    while (running) {
        // 3. Wait for ANY activity (connections, messages, input)
        int ready = poll(monitor_set, num_monitored, -1);
        // 4. Process all ready events
        for (each monitored descriptor) {
            if (descriptor == STDIN) {handle_user_command();}
            else if (descriptor == peer_socket) {
                handle_incoming_connection();}
            else if (descriptor is ready for reading) {  process_message(descriptor);}
            else if (descriptor is ready for writing) {end_pending_data(descriptor);}
        }
    }
}
```

Writing a single threaded peer code



QUERYING

In Peer-to-Peer (P2P) networks, there is no central server to manage data. Therefore, finding a specific file or resource requires decentralized search mechanisms. The efficiency of a query depends heavily on the network's **topology** (Structured vs. Unstructured).

Flooding	Unstructured	A node sends a query to all neighbors, who forward it until the TTL (Time-to-Live) expires.	**Pros:** Simple. **Cons:** High bandwidth overhead (broadcast storm).
Random Walk	Unstructured	A node sends a query to a limited number of random neighbors.	**Pros:** Lower traffic than flooding. **Cons:** Slower discovery time.
DHT Lookup	Structured	Uses a **Distributed Hash Table** (e.g., Chord, Kademlia) to map keys to specific nodes.	**Pros:** Extremely efficient (O(\log n) hops). **Cons:** High maintenance

Flooding

Flooding is the fundamental search algorithm used in unstructured P2P networks (e.g., Gnutella v0.4). It operates on a "broadcast" principle.

1. **Query Generation:** The source node (S) creates a query descriptor containing:
   - **Search Criteria:** Keywords or metadata.
   - **Message ID (GUID):** A unique identifier to track the query.
   - **TTL (Time-to-Live):** A counter (typically set to 5–7) to limit propagation range.
2. **Propagation (The Ripple Effect):**
   - Node S sends the query to *all* of its directly connected neighbors.
   - Each neighbor receives the query, decrements the TTL by 1, and forwards it to *all* of its own neighbors (excluding the one it received it from).
   - This continues until TTL = 0.
3. **Loop Detection:** Nodes track Message IDs they have seen recently. If a node receives a duplicate query (same ID), it discards it immediately to prevent infinite cycles.

References

- Beej’s Guide to Network Programming Using Internet Sockets Brian “Beej Jorgensen” Hall v3.3.0, Copyright © January 19, 2026

- Xuemin Shen · Heather Yu · John Buford · Mursalin Akon Editors Handbook of Peer-to-Peer Networking Springer