

part3:-

```
bool isWpaSupplicantRunning(const std::string& iface) {  
    // Check if the wpa_supplicant control socket exists for this interface  
    std::string ctrl_path = "/var/run/wpa_supplicant/" + iface;  
    return (access(ctrl_path.c_str(), F_OK) == 0);  
}
```

```
bool startWpaSupplicant(const std::string& iface) {  
    // Use -B for background, -Dnl80211 for modern drivers  
    std::string command = "wpa_supplicant -B -Dnl80211 -i " + iface + " -c  
/etc/wpa_supplicant.conf";  
    std::cout << "[*] Starting wpa_supplicant: " << command << std::endl;  
    return (system(command.c_str()) == 0);  
}
```

```
bool sendP2PFindCommand(const std::string& iface) {  
    std::string ctrl_path = "/var/run/wpa_supplicant/" + iface;  
    std::string client_path = "/tmp/wpa_ctrl_" + iface + "_" + std::to_string(getpid());
```

```
    int sock = socket(AF_UNIX, SOCK_DGRAM, 0);  
    if (sock < 0) {  
        perror("[ERROR] socket creation");  
        return false;  
    }  
  
    // Set a timeout for recv  
    struct timeval tv = {3, 0};  
    setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof(tv));  
  
    // Bind to client path  
    struct sockaddr_un local = {};  
    local.sun_family = AF_UNIX;  
    strncpy(local.sun_path, client_path.c_str(), sizeof(local.sun_path) - 1);  
    unlink(client_path.c_str());  
    if (bind(sock, (struct sockaddr*)&local, sizeof(local)) < 0) {  
        perror("[ERROR] bind");  
        close(sock);  
        return false;  
    }  
  
    // Destination address (wpa_supplicant control socket)  
    struct sockaddr_un remote = {};  
    remote.sun_family = AF_UNIX;  
    strncpy(remote.sun_path, ctrl_path.c_str(), sizeof(remote.sun_path) - 1);  
  
    const char *cmd = "P2P_FIND";
```

```

std::cout << "[DEBUG] Sending command: " << cmd << std::endl;
if (sendto(sock, cmd, strlen(cmd), 0, (struct sockaddr*)&remote, sizeof(remote)) < 0) {
    perror("[ERROR] sendto");
    close(sock);
    unlink(client_path.c_str());
    return false;
}

char buf[4096] = {};
socklen_t remote_len = sizeof(remote);
int len = recvfrom(sock, buf, sizeof(buf) - 1, 0, (struct sockaddr*)&remote, &remote_len);
if (len < 0) {
    perror("[ERROR] recvfrom");
    close(sock);
    unlink(client_path.c_str());
    return false;
}

buf[len] = '\0';
std::cout << "[P2P_FIND response] " << buf << std::endl;

close(sock);
unlink(client_path.c_str());
return true;

```

```

}

```

```

int main() {
    auto wirelessIfs = getWirelessInterfaces();
    if (wirelessIfs.empty()) {
        std::cerr << "[X] No wireless interfaces found.\n";
        return 1;
    }
    std::string baseIface = wirelessIfs[0];
    std::string p2pIface = "p2p-" + baseIface + "-0";
    std::string useIface = p2pIface;

```

```

std::cout << "[*] Using base wireless interface: " << baseIface << std::endl;

bool p2pCreated = false;
if (!interfaceExists(p2pIface)) {
    std::cout << "[*] Attempting to create P2P interface...\n";
    p2pCreated = createP2PInterface(baseIface, p2pIface);
    if (!p2pCreated) {
        std::cout << "[!] Could not create P2P interface. Falling back to base interface\n";
        useIface = baseIface;
    }
} else {
    p2pCreated = true;
}

if (!isInterfaceUp(useIface)) {
    std::cout << "[*] Interface " << useIface << " is down. Bringing it up...\n";

```

```

        if (!bringInterfaceUp(useIface)) {
            std::cerr << "[X] Failed to bring up interface.\n";
            return 1;
        }
    }

    std::cout << "[*] Checking for wpa_supplicant on " << useIface << "... \n";
    if (!isWpaSupplicantRunning(useIface)) {
        std::cout << "[!] wpa_supplicant not running. Starting... \n";
        if (!startWpaSupplicant(useIface)) {
            std::cerr << "[X] Failed to start wpa_supplicant.\n";
            return 1;
        }
        // Wait a moment for wpa_supplicant to create the control socket
        sleep(2);
    }

    std::cout << "[*] Sending P2P_FIND command to " << useIface << "... \n";
    if (!sendP2PFindCommand(useIface)) {
        std::cerr << "[X] Failed to send P2P_FIND command.\n";
        return 1;
    }

    std::cout << "[✓] P2P_FIND command sent successfully on " << useIface << ". \n";
    monitorWpaSupplicantEvents(useIface, 30); // Monitor for 30 seconds
    return 0;
}

```

}

thankyou

also please explain the other lib,bin internal componets of kernel

what thier role is ?

how sockets and other are used to interact?

flow diagram of the code

Absolutely! Here's a **comprehensive breakdown** of your code's design, the role of kernel/userland components, socket interactions, and a **flow diagram**.

1. Code Design & Function Relationships

Your program is modular, with each function handling a specific aspect of the P2P (Wi-Fi Direct) setup and monitoring process. Here's how the main parts relate:

Function Map

- **main()**
 - Calls `getWirelessInterfaces()` to find a usable Wi-Fi interface.
 - Tries to create a P2P interface with `createP2PInterface()`.
 - Falls back to base interface if P2P-device creation fails.
 - Ensures the chosen interface is up with `isInterfaceUp()` / `bringInterfaceUp()`.

- Checks if `wpa_supplicant` is running with `isWpaSupplicantRunning()`.
 - If not, starts it with `startWpaSupplicant()`.
- Sends a `P2P_FIND` command using `sendP2PFindCommand()`.
- Monitors for P2P events with `monitorWpaSupplicantEvents()`.
- **getWirelessInterfaces()**: Scans system interfaces for wireless capability.
- **interfaceExists()**: Checks if a network interface exists.
- **isInterfaceUp()**: Checks if an interface is up.
- **bringInterfaceUp()**: Brings an interface up using `ioctl`.
- **createP2PInterface()**: Uses `libnl` (Netlink) to ask the kernel to create a P2P interface.
- **isWpaSupplicantRunning()**: Checks for the existence of the `wpa_supplicant` control socket.
- **startWpaSupplicant()**: Starts `wpa_supplicant` via a shell command.
- **sendP2PFindCommand()**: Sends the `P2P_FIND` command to `wpa_supplicant`'s control socket.
- **monitorWpaSupplicantEvents()**: Attaches to the control socket and prints real-time events.

2. Linux Kernel, Libraries, and Binaries Involved

Kernel Components

- **nl80211**: The main kernel subsystem for Wi-Fi management, exposing netlink interfaces for advanced wireless operations (including P2P).
- **Network interfaces**: e.g., `wlp2s0` managed by drivers like `iwlwifi` (Intel), `ath9k` (Atheros), etc.

Libraries

- **libnl-3, libnl-genl-3**: C libraries for communicating with the kernel's netlink interface, used for creating interfaces, querying wireless capabilities, etc.

Userland Binaries

- **wpa_supplicant**: User-space daemon that manages Wi-Fi authentication, association, and P2P operations.
- **wpa_cli**: Command-line client for interacting with `wpa_supplicant` via its control socket.
- **iw**: Utility for configuring wireless devices via `nl80211` (not directly used here, but relevant).

3. How Sockets and Netlink Are Used

Netlink Sockets

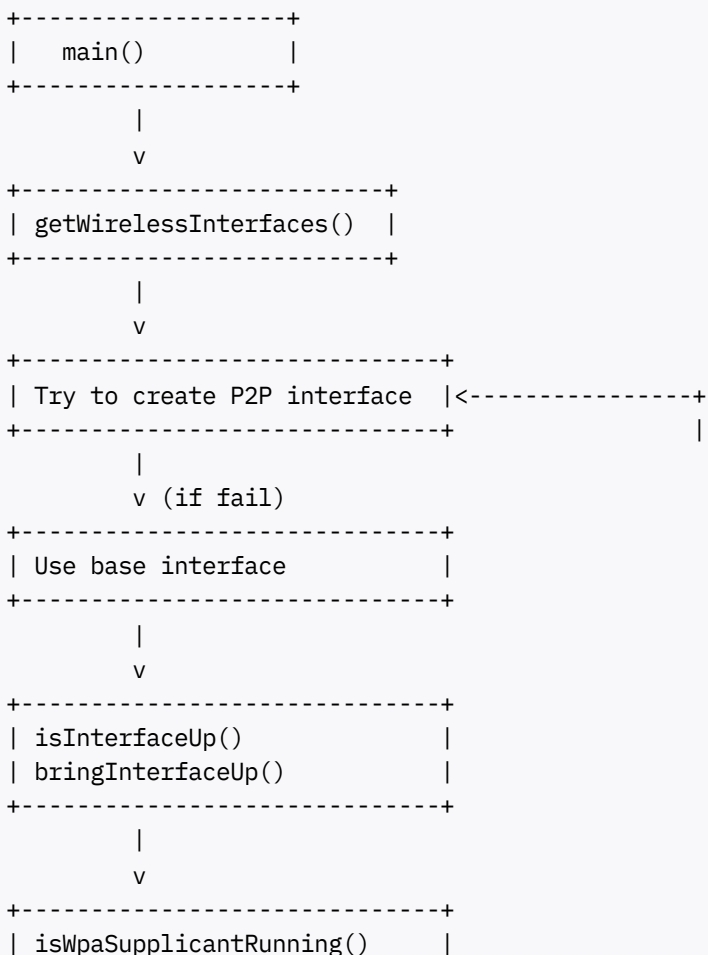
- Used by `libnl` in your `createP2PInterface()` function to communicate with the kernel (`nl80211` family).
- Allows user-space to send commands like "create new interface" directly to the kernel's networking stack.

UNIX Domain Sockets

- Used for communication between your application (and `wpa_cli`) and the `wpa_supplicant` daemon.
- The control socket (e.g., `/var/run/wpa_supplicant/wlp2s0`) is a UNIX domain socket.
- Your code sends commands (`P2P_FIND`, `ATTACH`, etc.) to `wpa_supplicant` and receives responses/events.

4. Flow Diagram

Here's a high-level flow diagram of your program's logic and interactions:





- **Dashed lines** represent communication via sockets (netlink or UNIX domain).
- **Arrows** represent function calls or process flow.

5. Component Roles and Interactions

Component	Role
Kernel (nl80211)	Handles wireless interface management, P2P interface creation, etc.
libnl	Provides C API to communicate with kernel netlink (nl80211)
wpa_supplicant	Manages Wi-Fi/P2P connections, exposes control socket for commands
Your Program	Orchestrates interface setup, P2P commands, and event monitoring
UNIX Sockets	Used for control channel between your program and wpa_supplicant
Netlink Sockets	Used for advanced kernel networking operations (via libnl)

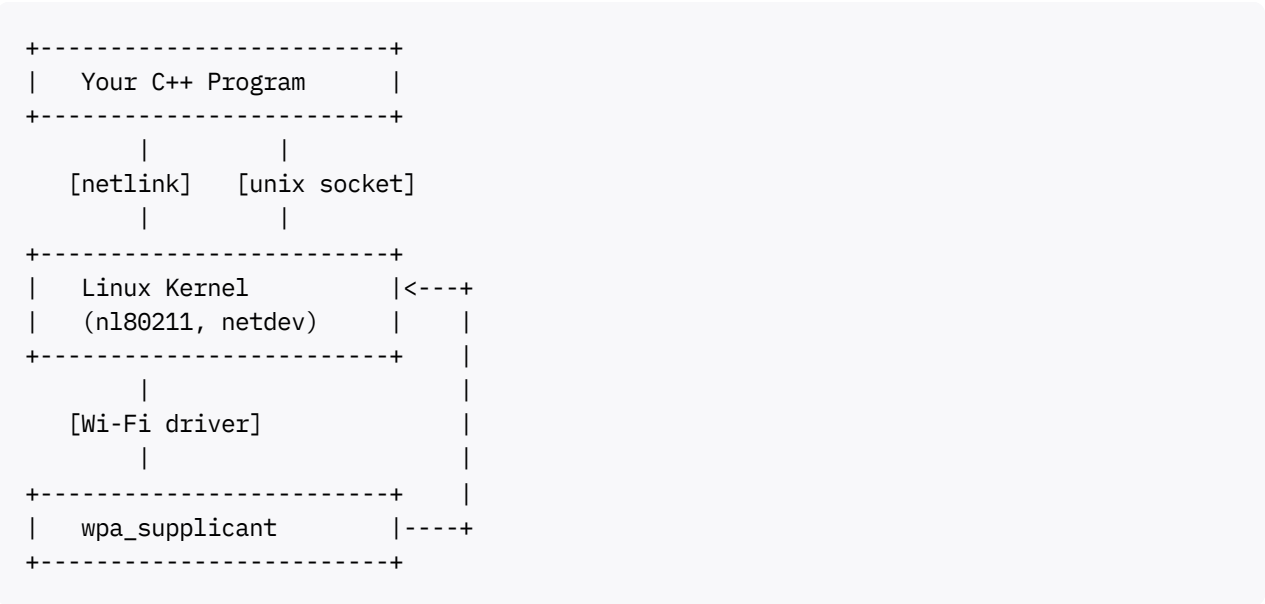
6. Typical Data Flows

- **Interface Creation:**
Your program → libnl/netlink → Kernel (nl80211) → Interface created
- **P2P Command:**
Your program → UNIX socket → wpa_supplicant → Wi-Fi hardware
- **Event Monitoring:**
wpa_supplicant → UNIX socket → Your program (events like P2P-DEVICE-FOUND)

7. Summary Table

Step	Library/API	Kernel/Userland Component	Socket Type
Interface discovery	C stdlib/sys	Kernel netdev	-
P2P interface create	libnl	Kernel nl80211	Netlink
Interface up	ioctl	Kernel	-
wpa_supplicant start	system()	Userland	-
P2P_FIND command	UNIX socket	wpa_supplicant	UNIX domain socket
Event monitoring	UNIX socket	wpa_supplicant	UNIX domain socket

8. Visual: Kernel/Userland/Socket Interactions



9. How Sockets Are Used

- **Netlink sockets:** For advanced networking commands to the kernel (interface creation, etc.).
- **UNIX domain datagram sockets:** For sending commands and receiving events from wpa_supplicant.

In Summary

- Your code orchestrates Wi-Fi Direct setup and monitoring by leveraging kernel interfaces (via libnl/netlink) and userland daemons (wpa_supplicant).
- Sockets (netlink and UNIX domain) are the glue for communication between your code, the kernel, and wpa_supplicant.

- The flow is: **detect interface** → **(try) create P2P interface** → **ensure interface is up** → **ensure wpa_supplicant is running** → **send P2P_FIND** → **monitor for events**.
- Each function is responsible for a clear, well-defined task, making your design modular and maintainable.

If you want a more detailed diagram, or have specific questions about any component, function, or flow, just ask!