PERSONA 1: Arquitectura de Despliegue en AWS (KERRY)

(Proyectar: Diagrama de la arquitectura o Consola de AWS)

"Buenos días. Nuestro proyecto no se limitó a una comunicación local (localhost). Llevamos la implementación a un entorno de producción real utilizando **Amazon Web Services (AWS)**.

Desplegamos una instancia **EC2** (con sistema operativo Ubuntu/Linux) que actúa como nuestro Servidor Central.

El reto principal aquí no fue solo el código en C, sino la red. Al trabajar en la nube, el servidor tiene dos direcciones IP:

1. **IP Privada:** Donde el código C 'escucha'.
2. **IP Pública:** A donde los clientes remotos (nuestras laptops) apuntan.

Nuestro objetivo fue lograr que nuestros clientes locales atravesaran internet para comunicarse con el proceso en la nube."

PERSONA 2: Configuración de Seguridad y Firewall (Security Groups) (ARIANNA)

(Proyectar: Captura de pantalla de los "Inbound Rules" en AWS)

"Antes de tocar una línea de código, tuvimos que configurar la capa de seguridad.

Por defecto, AWS bloquea todo el tráfico externo. Para que nuestros sockets funcionaran, configuramos el **Security Group** (el firewall virtual de AWS).

Tuvimos que abrir explícitamente dos puertos en las 'Reglas de Entrada' (Inbound Rules):

1. **Puerto 8080 (TCP):** Para la conexión fiable.
2. **Puerto 9000 (UDP):** Para la comunicación rápida.

Sin esta configuración en la consola de AWS, el handshake TCP hubiera fallado con un 'Connection Timed Out' antes de siquiera tocar nuestro programa en C."

PERSONA 3: Código TCP Servidor (Compilado en Linux) (ARIEL)

(Proyectar código: server_tcp.c)

"Entrando al código del servidor, que compilamos directamente dentro de la instancia EC2 usando gcc.

Aquí hay un detalle crítico para que funcione en AWS: la constante **INADDR_ANY**."

C

```
/* server_tcp.c - Ejecutándose en AWS EC2 */
int main() {
   int server_fd, new_socket;
   struct sockaddr_in address;

   server_fd = socket(AF_INET, SOCK_STREAM, 0);

   address.sin_family = AF_INET;
   // CRÍTICO PARA AWS:
   address.sin_addr.s_addr = INADDR_ANY;
   address.sin_port = htons(8080);

   bind(server_fd, (struct sockaddr *)&address, sizeof(address));
   listen(server_fd, 3);

   printf("Servidor en AWS esperando conexiones en puerto 8080...\n");

   // ... lógica de accept() y read() ...
}
```

"Si hubiéramos puesto una IP específica en el bind, habría fallado.

En la nube, usamos INADDR_ANY (que equivale a 0.0.0.0) para decirle al sistema operativo Linux: 'Acepta conexiones por **cualquiera** de tus interfaces de red', asegurando que la petición que viene de internet sea procesada correctamente."

PERSONA 4: Código TCP Cliente (Conectando a la IP Pública) (CARRION)

(Proyectar código: client_tcp.c)

"Ahora, el cliente. Este código se ejecuta en nuestra máquina local.

La diferencia fundamental con un ejemplo de libro es que en la función inet_pton (Presentación a Red), no usamos 'localhost'. Tuvimos que ingresar la **IP Pública Elástica** que AWS nos asignó."

C

```
/* client_tcp.c - Ejecutándose en nuestra PC */

int main() {

   int sock = 0;

   struct sockaddr_in serv_addr;


   sock = socket(AF_INET, SOCK_STREAM, 0);

   serv_addr.sin_family = AF_INET;

   serv_addr.sin_port = htons(8080);


   // Aquí va la IP Pública de la instancia AWS

   if(inet_pton(AF_INET, "54.23.11.98", &serv_addr.sin_addr) <= 0) {

      printf("\nDirección inválida / No soportada \n");

      return -1;

   }


   if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) {

      printf("\nError: No se pudo conectar al servidor AWS (Revise Security Group)\n");

      return -1;

   }

   // ... send() y read() ...

}
```

"Al ejecutar connect, el paquete viaja por internet, pasa el firewall de AWS y llega a nuestro servidor C, estableciendo el túnel TCP."

PERSONA 5: Implementación UDP en la Nube (FABRIZIO)

(Diagrama visual de paquetes UDP viajando)

"Para la segunda parte, implementamos UDP.

En un entorno local, UDP casi nunca falla. Pero en un entorno de nube real como AWS, UDP presenta un desafío interesante: **la volatilidad**.

Al no haber conexión mantenida, si el firewall de AWS cerrara el puerto momentáneamente o si hubiera congestión en la red, los paquetes simplemente desaparecerían.

Aun así, elegimos UDP por su baja latencia, ideal para enviar comandos rápidos al servidor sin esperar la burocracia del protocolo TCP."

PERSONA 6: Código UDP Servidor (Stateless en AWS) (YO)

(Proyectar código: server_udp.c)

"En el servidor UDP dentro de EC2, usamos recvfrom.

Observen que no necesitamos 'aceptar' a nadie."

C

```
/* server_udp.c - En AWS */
int main() {
   int sockfd;
   struct sockaddr_in servaddr, cliaddr;

   // Creación del socket DGRAM (Datagrama)
   sockfd = socket(AF_INET, SOCK_DGRAM, 0);

   servaddr.sin_family = AF_INET;
   servaddr.sin_addr.s_addr = INADDR_ANY; // Escucha en todas las interfaces
   servaddr.sin_port = htons(9000);      // Puerto UDP 9000
```

```
  bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr));

  printf("Servidor UDP en Nube activo...\n");

  // Bucle infinito para recibir de cualquier IP del mundo
  while(1) {
    recvfrom(sockfd, (char *)buffer, MAX, MSG_WAITALL,
        (struct sockaddr *) &cliaddr, &len);
    // ... procesar ...
  }
}
```

"Este código es capaz de recibir paquetes de clientes en Ecuador, Japón o Estados Unidos simultáneamente en el mismo bucle while, ya que no mantiene una sesión exclusiva con ninguno."

PERSONA 7: Código UDP Cliente (Apuntando a la Nube) (LEANDRO)

(Proyectar código: client_udp.c)

"El cliente UDP tiene una particularidad técnica importante cuando hablamos con la nube."

C

```
/* client_udp.c */
int main() {
  int sockfd;
  struct sockaddr_in servaddr;

  sockfd = socket(AF_INET, SOCK_DGRAM, 0);
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(9000); // Puerto 9000
```

```
    inet_pton(AF_INET, "54.23.11.98", &servaddr.sin_addr); // IP Pública AWS


    // sendto: Dispara el paquete a internet
    sendto(sockfd, (const char *)msg, strlen(msg),
        MSG_CONFIRM, (const struct sockaddr *) &servaddr, sizeof(servaddr));


    printf("Datagrama enviado a la nube.\n");
    // ... close ...
}
```

"Usamos sendto. A diferencia de TCP, si la dirección IP de AWS estuviera mal o el servidor estuviera apagado, **este programa NO daría error**. Simplemente enviaría el paquete al vacío y terminaría exitosamente. Esa es la naturaleza 'best-effort' de UDP."

PERSONA 8: Compilación, Ejecución y Conclusión (GABRIEL)

(Proyectar comandos de terminal)

"Para finalizar, todo esto requirió un manejo fluido de la terminal Linux.

Para poner en marcha el sistema, seguimos estos pasos operativos:

1. Conexión SSH a la instancia: ssh -i clave.pem ubuntu@ec2-ip...
2. Compilación con GCC: gcc server_tcp.c -o servidor_tcp
3. Ejecución con permisos (por los puertos): sudo ./servidor_tcp

Conclusión:

Este proyecto nos permitió entender que programar Sockets en C es solo la mitad del trabajo. La otra mitad es comprender la infraestructura de red: IPs públicas vs privadas, NAT, firewalls de nube (Security Groups) y cómo los protocolos TCP y UDP se comportan en el 'mundo real' de internet frente a un entorno local."