Criação dos 4 servidores:

Inicialmente tive dificuldade com a criação dos 4 servidores e como eu armazenaria os pokémons neles.

```
typedef struct{
    struct sockaddr_in sockaddr_in;
    Pokemon pokemonsInBase[TOTAL_DEFENSIVE_POKEMONS];
    int totalPokemons;
} Server;
```

A maneira que encontrei para solucionar esse problema foi a criação de uma struct que, além de armazenar as informações do sockaddr_in que são necessárias para criação do servidor, armazena os pokémons e o total de pokémons existentes no próprio servidor.

Lógica de turnos:

Outra dificuldade encontrada foi o retorno do comando “getturn”. Não tinha certeza se seria possível acessar turnos aleatórios ou só de maneira sequencial. O enunciado do trabalho não especifíca isso. Em função disso, implementei a funcionalidade de só permitir acesso de turnos de maneira sequencial.

```
int createNewTurn(){
    if (actualTurn == atoi(receivedCommands[1])){
        for (int actualPokemon = 0; actualPokemon < totalEnemyPokemonsWasCreated; actualPokemon++){
        makesEnemyWalkFoward(actualPokemon);
        enemyPokemons[actualPokemon].hits = 0;
        }
        for (int i = 0; i < 4; i++){
            int createEnemy = rand() % 100;
            if (verifyIfHasSpaceToCreateNewEnemyPokemon() && createEnemy < 40){
                generateNewRandomEnemyPokemon();
            }
        }
        actualTurn++;
        printf("New turn!\n");
        return TRUE;
    }
```

No código acima, podemos ver que o turno só é executado se o *actualTurn* (variável que representa o turno atual), for igual ao *atoi(receivedCommands[1])* (variável que representa o valor inserido do turno, convertido para inteiro).

Surgimentos de Pokémons inimigos:

O surgimento de pokémons inimigos em posições aleatórias e de tipos aleatórios se mostrou um problema durante o desenvolvimento.
Para resolver, primeiramente fiz com que cada espaço disponível só tivesse 40% de chance de aparecer um Pokémon. Além disso, só é permitido que os pokémons apareçam no primeiro servidor, que representa a Localização de defesa física 1, também só é permitido que seja criado um inimigo em um espaço vazio. Implementação na imagem abaixo.

```
for (int i = 0; i < 4; i++){
    int createEnemy = rand() % 100;
    if (verifyIfHasSpaceToCreateNewEnemyPokemon() && createEnemy < 40){
        generateNewRandomEnemyPokemon();
    }
}
```

Dentro dos 40% de chance de aparecimento de um Pokémon inimigo, cada tipo de Pokémon possui sua respectiva porcentagem. O Zubat tem 20% de chance de surgimento (3/6 de 40%). O Lugia tem 13,3% de chance de surgimento (2/6 de 40%). E o Mewtwo tem 6,7% de chance de surgimento (⅙ de 40%).

```c
void generateNewRandomEnemyPokemon(){
    enemyPokemons[totalEnemyPokemonsWasCreated].id = generateID();
    enemyPokemons[totalEnemyPokemonsWasCreated].hits = 0;
    int randomNumber = rand() % 6;
    char name[POKENAME_MAX_SIZE] = "";
    if (randomNumber < 3){
        strcpy(enemyPokemons[totalEnemyPokemonsWasCreated].name, POKENAME_1);
        enemyPokemons[totalEnemyPokemonsWasCreated].hitsToDie = 1;
    }
    else if (randomNumber < 5){
        strcpy(enemyPokemons[totalEnemyPokemonsWasCreated].name, POKENAME_2);
        enemyPokemons[totalEnemyPokemonsWasCreated].hitsToDie = 2;
    }
    else{
        strcpy(enemyPokemons[totalEnemyPokemonsWasCreated].name, POKENAME_3);
        enemyPokemons[totalEnemyPokemonsWasCreated].hitsToDie = 3;
    }
    enemyPokemons[totalEnemyPokemonsWasCreated].pokemon = returnValidPositionToEnemyPokemon();
    printf("Enemy created!\n");
    totalEnemyPokemonsWasCreated++;
}
```

Outro ponto importante é a criação de um struct para representar os Pokémons inimigos. Nele, além da posição já armazenada nos Pokémon defensores. Está sendo armazenado também o nome, o número de hits recebidos na rodada atual e o número total de hits necessários para derrotá-lo.

```c
typedef struct{
    Pokemon pokemon;
    int id;
    char name[POKENAME_MAX_SIZE];
    int hits;
    int hitsToDie;
} EnemyPokemon;
```

Score do Gameover:

A mensagem de gameover, deve retornar um score contendo o número de Pokémons adversários atingidos e destruídos pelos defensores, o número de Pokémons adversários que chegaram na Pokédex e o tempo total em segundos utilizados pelo programa cliente para completar o jogo. Foi bastante trabalhoso resolver essas dificuldades.

```c
void clearEnemyPokemonsInTableByID(int id){
    EnemyPokemon auxEnemyPokemons[20];
    int x = 0;
    for (int actualPokemon = 0; actualPokemon < totalEnemyPokemonsWasCreated; actualPokemon++){
        if (enemyPokemons[actualPokemon].id != id){
            auxEnemyPokemons[actualPokemon-x] = enemyPokemons[actualPokemon];
        }
        else{
            x++;
            printf("%s was killed\n", enemyPokemons[actualPokemon].name);
            totalEnemiesKilled++;
        }
    }
    clearEnemyPokemonsInTable();
    totalEnemyPokemonsWasCreated--;
    for (int actualPokemon = 0; actualPokemon < totalEnemyPokemonsWasCreated; actualPokemon++){
        enemyPokemons[actualPokemon] = auxEnemyPokemons[actualPokemon];
    }
}
```

Na questão de derrotar o Pokémon adversário e somar pontos, criei a variável global *totalEnemiesKilled*. Essa função simula a derrota de um Pokémon inimigo, apagando ele do vetor que armazena todos os pokémon inimigos existentes e somando 1 no valor da variável.

Para resolver a questão dos pokémons que chegaram na Pokédex, a lógica é exatamente a mesma. O Pokémon é apagado e soma-se 1 na variável *enemiesThatReachedPokedex*. É considerado que o inimigo chegou na Pokédex quando ele se encontra no servidor 4 e permanece vivo no turno.

Já sobre o tempo total em segundos, foi criada uma variável, também global que armazena inicialmente o tempo atual.

```
void createInitialConfigurations(char** argv){
    start = time(NULL);
```

No momento de envio da mensagem de GameOver, é novamente calculado o tempo e também se calcula a diferença entre esses dois tempos, como mostrado abaixo:

```
void sendGameOverMessage(char s[]){
    time_t diff = time(NULL) - start;
```

A variável *diff* é retornada no score.

Reinício do jogo:

Essa foi a questão que mais levou tempo para ser resolvida. De acordo com meu entendimento do trabalho, caso o *client* envie um comando inválido, o jogo retorna a mensagem de *gameover*, porém o *client* pode iniciar uma nova rodada. Para isso, foi criada a função *waitCommandAfterGameOver*, para que após envio da mensagem de *gameover*, fosse possível reiniciar o jogo enviando uma mensagem *start*.

```
while (strcmp(status,"1") == 0){
    openCommunicationWithClient();
    if (strcmp(status,"3") != 0)
        waitCommandAfterGameOver();
}
```

Além disso, o comando *quit* foi de difícil implementação. De acordo, novamente, com meu entendimento do trabalho, caso o *client* enviasse o comando quit durante um jogo, o *gameover* deve ser retornado e a conexão com o servidor encerrada. Porém, caso não estivesse com um jogo ativo, o comando devia apenas encerrar a conexão, sem retornar mensagem de *gameover*.

```
void waitCommandAfterGameOver(){
    if (strcmp(buffer,"quit") == 0){
        return;
    }
    receiveMessageFromClient();
    if (strcmp(buffer,"start") == 0){
        strcpy(status,"1");
        printGameStarted();
    }
    else{
        strcpy(status,"0");
        sendMessageToClient("end");
    }
}
```

A solução encontrada foi, dentro da função *waitCommandAfterGameOver*, apenas retornar um valor vazio, fazendo com que a variável status ficasse diferente de “1”, fazendo assim com que o programa saia do while de comunicação com o *client*.

Considerações finais:

A retransmissão de pacotes pelo cliente e a conexão IPV6 não foram implementadas.