Instituto Puig Castellar
Ciclo formativo: DAM
Grado: CFGS
Crédito de Síntesis / Proyecto intermodular

PuigCraft

Servidor multijugador de Minecraft

Autor/a/es: Alessandro Nadal
Tutor: Nombre y Apellidos
Curso: 2025-2026
Fecha de entrega: 15/05/2026

Resumen del proyecto

PuigCraft es un servidor de Minecraft personalizable, desarrollado en Python utilizando la librería Twisted, la cual permite la programación de servicios en red de manera asíncrona. El proyecto sigue una arquitectura cliente-servidor y se comunica con el cliente oficial de Minecraft de Mojang.

El objetivo principal es desarrollar un servidor funcional que permita la conexión simultánea de múltiples jugadores, gestionando sus interacciones y recursos de forma eficiente mediante técnicas asíncronas. Además, se incorpora la generación procedural del mundo como parte del sistema.

La metodología se basa en el estudio del protocolo de red de Minecraft y en la comprensión del funcionamiento de Twisted para gestionar múltiples conexiones sin bloqueos. El desarrollo comienza con una implementación básica del servidor que permite la conexión de clientes, y posteriormente se amplía mediante la incorporación progresiva de módulos (stages) que añaden nuevas funcionalidades.

En conclusión, el proyecto demuestra que es posible desarrollar un servidor de Minecraft en Python, a pesar de sus limitaciones de rendimiento, siempre que se optimice el uso de recursos y se empleen herramientas adecuadas para la programación asíncrona.

Palabras clave

Minecraft, Juego, Bloques, Python, Flask, Twisted

Abstract

PuigCraft is a customizable Minecraft server implemented in Python using the Twisted library, which is used to program asynchronous network services. The project is based on the client–server model and network communication using Mojang's official Minecraft client.

The main objective is to develop a functional server that allows players to connect while managing resources and player interactions efficiently and asynchronously, also including procedural world generation.

The methodology is based on studying the Minecraft server network protocol and learning how the Twisted library works to manage multiple connections without blocking. Development starts by programming the basic foundations of a server to allow client connections, and then progressively expanding and improving the different modules (stages).

In conclusion, the project demonstrates that it is possible to create a Minecraft server using Python, despite it being one of the slower programming languages, as long as resources and libraries are handled correctly.

Keywords
Minecraft, Videogames, Blocks, Python, Flask, Twisted

Índice

1. Presentación del proyecto

1.1 Introducción

El proyecto PuigCraft consiste en el desarrollo de un servidor personalizable de Minecraft utilizando el lenguaje de programación Python. Este servidor sigue una arquitectura cliente-servidor y se comunica con el cliente oficial de Minecraft mediante el protocolo de red definido por Mojang.

Los servidores de videojuegos son una parte fundamental en la experiencia multijugador, ya que permiten la conexión de varios usuarios y gestionan sus interacciones dentro de un mismo entorno virtual. En el caso de Minecraft, estos servidores ofrecen además la posibilidad de ser modificados y adaptados, lo que permite crear experiencias de juego personalizadas.

El objetivo principal del proyecto es crear un servidor funcional capaz de gestionar la conexión simultánea de múltiples jugadores y sus interacciones dentro del juego. La idea es construir una base sólida que permita implementar diferentes funcionalidades de forma progresiva, partiendo de un sistema básico e incorporando mejoras a lo largo del desarrollo.

A lo largo de esta memoria se describe el proceso de desarrollo del proyecto, comenzando por el contexto en el que se sitúa, seguido de la justificación de su realización. Posteriormente, se explican los aspectos técnicos más relevantes, las decisiones tomadas y las funcionalidades implementadas en la versión final del servidor.

1.2 Contexto

El desarrollo de servidores para videojuegos es un ámbito importante dentro del software de red, especialmente en juegos multijugador como Minecraft, donde el servidor es el encargado de gestionar la conexión de los jugadores, el estado del mundo y las interacciones entre usuarios. Este tipo de sistemas requiere manejar múltiples conexiones simultáneas de

forma eficiente, lo que lo convierte en un problema interesante desde el punto de vista técnico.

Actualmente, la mayoría de servidores de Minecraft están desarrollados en Java, ya que es el lenguaje original del juego. Existen implementaciones muy extendidas como Spigot o Paper, que ofrecen un alto rendimiento y una gran cantidad de funcionalidades. Además, también existen APIs y frameworks que facilitan la creación de plugins y la personalización del servidor sin necesidad de modificar su núcleo.

Sin embargo, estas soluciones suelen ser complejas y están pensadas para entornos ya establecidos, lo que puede dificultar la comprensión de cómo funciona internamente un servidor desde cero. Además, al estar muy optimizadas y abstraídas, no siempre permiten experimentar fácilmente con aspectos más básicos como la gestión directa del protocolo de red o el control completo del sistema.

En los últimos años, también ha aumentado el interés por explorar el desarrollo de servidores utilizando otros lenguajes de programación, como Python, especialmente en contextos educativos o de aprendizaje. Esto permite centrarse en entender conceptos como la comunicación en red, la asincronía o la gestión de múltiples clientes, sin depender completamente de herramientas ya construidas.

Por este motivo, el desarrollo de un servidor propio desde cero resulta relevante, ya que permite profundizar en el funcionamiento interno de este tipo de sistemas y explorar nuevas formas de implementarlos, aportando una visión más completa del desarrollo de aplicaciones de red dentro del ámbito de los videojuegos.

1.3 Justificación

El proyecto PuigCraft tiene sentido por varias razones. Primero, permite aprender cómo funciona un servidor de Minecraft desde cero y entender el protocolo de red del juego, algo que no se consigue usando servidores ya existentes como Spigot o Paper.

Además, desarrollar el servidor desde cero ayuda a practicar conceptos importantes de programación de redes, asincronía y gestión de múltiples conexiones de jugadores, ofreciendo un aprendizaje práctico y útil.

El proyecto también puede servir como base para crear servidores personalizados o experimentar con funcionalidades como la generación procedural de mundos, aportando flexibilidad y posibilidades de ampliación.

Por último, a nivel personal, siempre me ha interesado Minecraft y quería aprender cómo funciona su comunicación en red. Esto hace que el proyecto sea motivador y al mismo tiempo formativo.

En resumen, PuigCraft combina aprendizaje técnico, experimentación y motivación personal, y por eso es un proyecto relevante y útil.

1.4 Objetivos

Objetivo general

Desarrollar un servidor de Minecraft desde cero en Python, capaz de gestionar la conexión de múltiples jugadores y su interacción dentro del juego de forma funcional y organizada.

Objetivos específicos

- Comprender el funcionamiento del protocolo de red de Minecraft.
- Implementar una arquitectura cliente-servidor básica que permita la conexión de jugadores.
- Gestionar múltiples conexiones de forma eficiente utilizando programación asíncrona.
- Desarrollar un sistema modular que permita añadir funcionalidades de forma progresiva.
- Implementar la gestión de jugadores dentro del servidor (conexión, estado, interacción).
- Crear un sistema básico de mundo, incluyendo generación procedural.

- Analizar las limitaciones y el rendimiento del uso de Python en este tipo de aplicaciones.

2. Estrategia y planificación

2.1 Estrategia de desarrollo y viabilidad

El desarrollo del proyecto se ha planteado de forma incremental, implementando primero la estructura base del servidor y añadiendo posteriormente las distintas funcionalidades del protocolo de Minecraft. La estrategia seguida ha consistido en dividir el proyecto en bloques funcionales independientes, comenzando por la gestión de conexiones y el sistema de estados, y continuando con la implementación progresiva de los paquetes y comportamientos del protocolo. Este enfoque ha permitido disponer de versiones funcionales desde fases tempranas del desarrollo y facilitar la detección de errores durante cada etapa.

A nivel técnico, el proyecto es viable gracias al uso de Python y de librerías especializadas como Twisted, que proporciona herramientas para gestionar conexiones de red asíncronas de forma eficiente. La documentación pública del protocolo de Minecraft Java Edition también ha permitido comprender la estructura de los paquetes y desarrollar una implementación propia adaptada a los objetivos del proyecto. Además, la arquitectura modular utilizada facilita el mantenimiento y la ampliación futura del sistema.

En cuanto a la viabilidad temporal, el proyecto se ha ajustado a una planificación organizada por fases, priorizando primero las funcionalidades esenciales del servidor antes de abordar mejoras secundarias. Esto ha permitido asegurar el correcto funcionamiento del núcleo del sistema incluso en caso de limitaciones de tiempo. Respecto a los recursos, el proyecto no requiere infraestructura compleja ni hardware especializado, ya que el servidor puede ejecutarse en un equipo convencional de desarrollo, lo que hace viable su realización dentro del contexto académico del ciclo formativo.

2.2 Metodología de trabajo

Enfoque de trabajo

Se seguirá un enfoque progresivo e incremental. El objetivo es tener una versión funcional lo antes posible e ir mejorándola poco a poco.

El desarrollo será flexible, permitiendo adaptarse a problemas o cambios durante el proceso.

Organización del trabajo

El proyecto se dividirá en tareas relacionadas con las distintas partes del servidor:

- Sistema de conexión
- Gestión de jugadores
- Manejo del mundo
- Funcionalidades adicionales

Cada tarea se desarrollará de forma independiente y se comprobará su funcionamiento antes de pasar a la siguiente.

El progreso se revisará de forma continua, asegurando que cada parte funciona antes de integrarla con el resto.

Herramientas de seguimiento

GitHub: para el control de versiones del código y guardar el progreso del proyecto.

2.3 Planificación

Fases del proyecto

Fase 1 · Análisis y aprendizaje

Se estudia el protocolo de Minecraft y el funcionamiento de la programación asíncrona. Se define cómo se va a estructurar el servidor.

Fase 2 · Preparación del entorno

Se crea la estructura del proyecto, se configura el entorno de desarrollo y se inicializa el repositorio.

Fase 3 · Desarrollo del núcleo del servidor

Se implementa la conexión cliente-servidor y una primera versión funcional básica.

Fase 4 · Gestión de jugadores y mundo

Se añaden funcionalidades para gestionar jugadores y se implementa un sistema básico de mundo.

Fase 5 · Funcionalidades adicionales

Se incorporan mejoras como generación procedural o ampliaciones del sistema.

Fase 6 · Pruebas y mejoras finales

Se revisa el funcionamiento general, se corrigen errores y se prepara la entrega final.

3. Análisis

3.1 Casos de uso

El sistema tiene un actor principal: el jugador, que se conecta al servidor de Minecraft utilizando el cliente oficial. A continuación se describen los casos de uso más importantes.

CU-01 · Conectarse al servidor

- Actor: Jugador
- Descripción: El jugador inicia conexión con el servidor para entrar al juego.
- Flujo principal:
- El jugador introduce la IP del servidor en el cliente.
- El cliente inicia la conexión.
- El servidor recibe la solicitud y valida la conexión.
- El jugador entra al servidor.

Flujo alternativo:

- Si la conexión falla, el sistema muestra un error.
- Si el servidor no responde, el jugador no puede entrar.
- Resultado: El jugador queda conectado al servidor.

CU-02 · Interactuar dentro del mundo

Actor: Jugador

Descripción: El jugador interactúa con el entorno del juego.

Flujo principal:

- El jugador se mueve dentro del mundo.
- Realiza acciones (caminar, interactuar, etc.).
- El servidor recibe y procesa las acciones.
- El estado del mundo se actualiza.

Flujo alternativo:

- Si hay errores en la comunicación, la acción no se procesa correctamente.
- Resultado: Las acciones del jugador afectan al mundo del servidor.

CU-03 · Desconectarse del servidor

Actor: Jugador

Descripción: El jugador abandona el servidor.

Flujo principal:

- El jugador cierra la conexión.
- El servidor detecta la desconexión.
- Se eliminan los datos temporales del jugador.
- Resultado: El jugador deja de estar en el servidor.

3.2 Requisitos funcionales

El sistema debe cumplir las siguientes funcionalidades:

El sistema permitirá la conexión de jugadores al servidor.

El sistema gestionará múltiples conexiones simultáneas.

El sistema procesará los paquetes del protocolo de Minecraft.

El sistema mantendrá el estado de los jugadores conectados.

El sistema permitirá la interacción básica dentro del mundo.

El sistema gestionará la desconexión de jugadores correctamente.

El sistema generará un mundo básico para los jugadores.

El sistema permitirá ampliar funcionalidades mediante módulos.

3.3 Requisitos no funcionales

Además de las funcionalidades, el sistema debe cumplir ciertas condiciones:

Rendimiento: El servidor debe gestionar varias conexiones sin bloquearse.

Usabilidad: La conexión debe realizarse sin configuración compleja para el usuario.

Compatibilidad: Debe funcionar con el cliente oficial de Minecraft.

Mantenibilidad: El código debe estar organizado para facilitar futuras mejoras.

Escalabilidad: El sistema debe permitir añadir nuevas funcionalidades sin rehacer el código.

Disponibilidad: El servidor debe poder mantenerse activo durante la ejecución sin fallos críticos.

3.4 Análisis de alternativas tecnológicas

Lenguaje de programación

Se valoraron varias opciones:

Java: Es el lenguaje oficial de Minecraft y el más utilizado en servidores, con buen rendimiento.

Python: Más sencillo y fácil de desarrollar, pero menos eficiente.

C++: Muy eficiente, pero más complejo de implementar.

Se eligió Python porque permite centrarse en el aprendizaje del protocolo y la lógica del servidor sin añadir demasiada complejidad.

4. Diseño

4.1 Arquitectura del sistema

El proyecto sigue una arquitectura cliente-servidor, donde el cliente es el juego oficial de Minecraft y el servidor es la aplicación desarrollada en Python.

El sistema está dividido en varios componentes principales que se encargan de diferentes responsabilidades:

Cliente (Minecraft)

Es la aplicación que utiliza el jugador. Se encarga de enviar las acciones del usuario al servidor y mostrar la información recibida.

Servidor principal

Es el núcleo del sistema. Gestiona las conexiones de los jugadores, recibe los datos del cliente y coordina el funcionamiento general del servidor.

Módulo de red (Twisted)

Se encarga de gestionar las conexiones de red de forma asíncrona. Permite que varios jugadores se conecten al mismo tiempo sin bloquear el sistema.

Gestión de jugadores

Controla el estado de cada jugador (conectado, posición, acciones, etc.).

Sistema de mundo

Gestiona el entorno del juego, incluyendo la generación y actualización del mundo.

La comunicación sigue este flujo

- El cliente envía información al servidor (acciones del jugador).
- El servidor procesa esa información.

- El servidor actualiza el estado del mundo o del jugador.
- El servidor envía la respuesta al cliente.

Se ha elegido esta arquitectura porque es la utilizada en el propio juego y permite separar claramente las responsabilidades. Además, el uso de módulos facilita ampliar el sistema sin tener que modificar toda la estructura.

4.2 Modelo de datos

El proyecto no utiliza una base de datos tradicional, pero sí maneja estructuras de datos internas para gestionar la información del sistema.

Jugador

Contiene la información de cada jugador conectado, como su identificador, estado de conexión y datos dentro del juego (posición, acciones, etc.).

Mundo

Representa el entorno del juego. Incluye la información necesaria para generar y mantener el estado del mundo.

Conexión

Representa cada conexión activa entre un cliente y el servidor. Se encarga de gestionar el intercambio de datos.

La relación entre estas entidades es la siguiente:

- Un jugador está asociado a una conexión.
- El jugador interactúa con el mundo.
- El servidor coordina la relación entre todos ellos.

Esta estructura permite gestionar múltiples jugadores de forma independiente y mantener el estado del sistema de forma organizada.

4.3 Diseño de interfaz

Al tratarse de un servidor, el proyecto no incluye una interfaz gráfica propia. La interfaz de usuario es el propio cliente oficial de Minecraft.

Desde el punto de vista del usuario, la interacción es la siguiente:

- El jugador introduce la dirección del servidor en el cliente.
- Se conecta al servidor.
- Una vez dentro, interactúa con el mundo de forma normal (movimiento, acciones, etc.).

Aunque no hay una interfaz propia, el diseño del sistema está orientado a que la experiencia sea transparente para el usuario, es decir, que funcione como cualquier servidor de Minecraft.

A nivel interno, el sistema está diseñado para responder correctamente a las acciones del jugador y mantener una comunicación fluida con el cliente.

5. Desarrollo

5.1 Estructura del proyecto

El proyecto está organizado de forma clara separando el punto de entrada del sistema y la lógica interna del servidor. La mayor parte del código se encuentra dentro del directorio `src/`, que agrupa todos los componentes del servidor.

La estructura general es la siguiente:

/proyecto

├── main.py

└── src/

  ├── serverprocess.py

  ├── stages/

  └── ...

El archivo `main.py` actúa como punto de entrada del sistema. Su única responsabilidad es iniciar el servidor llamando al método `start()` definido en el módulo `serverprocess`. Esta decisión permite mantener el arranque del sistema desacoplado de la lógica interna del servidor.

Dentro del directorio `src/` se encuentra el núcleo del proyecto:

- **serverprocess.py** contiene las clases principales que gestionan el servidor:
  - `ServerFactory`: actúa como proceso principal del servidor. Se encarga de aceptar conexiones y crear un proceso independiente para cada jugador.
- **stages/** agrupa las distintas clases que representan los estados en los que puede encontrarse un jugador. Esta organización sigue un criterio basado en comportamiento, separando la lógica según el estado actual del usuario.

El criterio seguido en esta estructura es separar claramente:

- El arranque del sistema
- La gestión de conexiones
- La lógica específica de cada usuario
- La lógica dependiente del estado del jugador

Esto facilita el mantenimiento y permite ampliar el sistema añadiendo nuevos estados o comportamientos sin modificar la estructura principal.

5.2 Implementación de funcionalidades

5.2.1 Gestión de conexiones y procesos

El servidor está diseñado para gestionar múltiples jugadores de forma concurrente. Para ello, se utiliza una arquitectura basada en procesos.

La clase `ServerFactory` actúa como núcleo del sistema, encargándose de escuchar nuevas conexiones y crear un nuevo `ServerProcess` por cada jugador que se conecta. Cada proceso funciona de forma independiente, lo que permite aislar la ejecución de cada usuario y evitar interferencias entre ellos.

Se eligió este enfoque en lugar de un modelo basado en hilos o en un único proceso para mejorar la escalabilidad y la robustez del sistema, ya que un fallo en un proceso no afecta al resto.

5.2.2 Gestión de peticiones por jugador

Cada jugador conectado es gestionado por una instancia de `ServerProcess`. Esta clase es responsable de recibir, interpretar y responder a todas las peticiones del cliente.

El diseño se basa en encapsular toda la lógica del usuario dentro de su propio proceso, lo que simplifica el control del flujo de ejecución y evita la necesidad de sincronización compleja entre usuarios.

Esta decisión permite que cada jugador tenga su propio contexto de ejecución, facilitando tanto el desarrollo como la depuración del sistema.

5.2.3 Sistema de estados (Stages)

Una de las partes clave del diseño es el uso de un sistema basado en estados para gestionar el comportamiento del jugador.

Cada `ServerProcess` contiene un atributo de tipo `Stage`, que representa el estado actual del jugador dentro del sistema.
Dependiendo del estado, las peticiones del usuario se gestionan de forma diferente.

Los estados están definidos dentro del directorio `stages/`, donde cada clase implementa la lógica correspondiente a un estado concreto (por ejemplo, autenticación, juego, espera, etc.).

Este enfoque sigue el patrón de máquina de estados, permitiendo:

- Separar claramente la lógica según el contexto
- Evitar condicionales complejos dentro de una única clase
- Facilitar la ampliación del sistema añadiendo nuevos estados

Se eligió este diseño frente a una implementación monolítica porque mejora la modularidad y hace el sistema más fácil de mantener.

5.2.4 Flujo general del sistema

El funcionamiento global del sistema sigue el siguiente flujo:

1. El servidor se inicia desde `main.py`.
2. `ServerFactory` comienza a escuchar conexiones.
3. Cuando un jugador se conecta, se crea un nuevo `ServerProcess`.
4. Cada `ServerProcess` asigna un estado inicial (`Stage`).
5. Las peticiones del jugador se procesan según su estado actual.
6. El estado puede cambiar en función de las acciones del usuario.

Este flujo permite gestionar múltiples jugadores de forma simultánea manteniendo una estructura clara y organizada.

5.3 Pruebas

Las pruebas del sistema se han centrado en verificar el correcto funcionamiento del servidor en situaciones reales de uso, especialmente en la gestión concurrente de múltiples jugadores.

Se han realizado principalmente pruebas funcionales y de integración.

Pruebas funcionales

Se ha comprobado que cada componente del sistema cumple su función:

- Inicio correcto del servidor desde `main.py`
- Creación de procesos al conectarse nuevos jugadores
- Gestión independiente de cada jugador
- Correcto funcionamiento del sistema de estados

También se han probado casos límite, como:

- Conexiones simultáneas de varios usuarios
- Cambios de estado consecutivos
- Peticiones inválidas o inesperadas

6. Conclusiones

6.1 Conclusiones generales

El resultado del proyecto es un servidor funcional capaz de gestionar múltiples jugadores de forma concurrente mediante un sistema basado en procesos independientes y control de estados. El sistema cumple con el objetivo principal de manejar conexiones simultáneas de forma organizada y escalable, manteniendo separada la lógica de cada usuario.

A nivel técnico, el proyecto ha permitido profundizar en conceptos que van más allá de los contenidos básicos del ciclo, como la gestión de concurrencia, la arquitectura de servidores y el diseño basado en estados. Uno de los aspectos más relevantes ha sido entender cómo estructurar un sistema para que sea escalable y mantenible, en lugar de simplemente funcional.

La parte más lograda del proyecto es la arquitectura basada en procesos y el uso del sistema de estados (`Stage`), que permite organizar la lógica de forma clara y modular. Esta decisión ha facilitado el desarrollo y ha evitado estructuras complejas difíciles de mantener.

Como aspecto menos logrado, el proyecto podría haberse beneficiado de una mayor profundización en algunas funcionalidades o de una capa adicional de validaciones y control de errores. Aunque el sistema funciona correctamente, hay margen de mejora en términos de robustez y optimización.

En conjunto, el proyecto no solo ha dado como resultado un sistema funcional, sino que ha supuesto un aprendizaje importante en diseño de software y toma de decisiones técnicas.

6.2 Consecución de objetivos

El objetivo general del proyecto, desarrollar un servidor capaz de gestionar múltiples usuarios de forma concurrente, se ha alcanzado de forma satisfactoria. El sistema es capaz de aceptar conexiones, crear procesos independientes y gestionar las peticiones de cada jugador sin interferencias.

En cuanto a los objetivos específicos:

- La creación de un sistema de gestión de conexiones mediante un proceso principal (`ServerFactory`) se ha cumplido completamente. Este componente actúa correctamente como núcleo del servidor.
- La implementación de procesos independientes por jugador (`ServerProcess`) también se ha alcanzado. Cada usuario es gestionado de forma aislada, lo que mejora la estabilidad del sistema.
- El diseño e implementación del sistema de estados (`Stage`) se ha completado con éxito. Este sistema permite adaptar el comportamiento del servidor según el contexto del jugador, cumpliendo su función de forma clara.

Algunos objetivos secundarios, como una mayor optimización o ampliación de funcionalidades, se han cumplido parcialmente o han quedado fuera del alcance debido a limitaciones de tiempo. Sin embargo, estas limitaciones no afectan al funcionamiento principal del sistema.

En general, los objetivos planteados se han alcanzado en un grado alto, cumpliendo con las expectativas iniciales del proyecto.

6.3 Valoración de la metodología y planificación

La planificación inicial se ha seguido de forma general, aunque con algunas desviaciones durante la fase de desarrollo. La implementación de la arquitectura del servidor y la gestión de procesos requirió más tiempo del previsto, especialmente al tratarse de conceptos que implicaban investigación previa.

La decisión de estructurar el proyecto desde el principio en torno a procesos y estados ha resultado acertada, ya que ha evitado tener que realizar cambios importantes en fases avanzadas del desarrollo. Sin embargo, la estimación del tiempo necesario para implementar estas estructuras no fue del todo precisa.

La metodología de trabajo basada en desarrollar primero la estructura principal y después añadir funcionalidades progresivamente ha funcionado bien. Tener una base funcional desde etapas tempranas ha permitido probar el sistema continuamente y detectar errores antes de que se acumularan.

Como mejora de cara a futuros proyectos, sería recomendable:

- Reservar más tiempo para investigación técnica en fases iniciales
- Ajustar mejor la estimación de tareas complejas
- Planificar con más margen los imprevistos

En general, la metodología ha sido adecuada, aunque con margen de mejora en la planificación del tiempo.

6.4 Visión de futuro

El proyecto tiene varias líneas claras de mejora y ampliación.

A corto plazo, sería interesante reforzar el sistema con:

- Mejor gestión de errores y validaciones
- Registro de logs para facilitar la monitorización del servidor
- Optimización del rendimiento en escenarios con muchos usuarios

También se podría ampliar el sistema de estados añadiendo nuevos comportamientos o fases dentro del flujo del jugador, lo que permitiría hacer el sistema más completo y flexible.

A medio plazo, una mejora relevante sería incorporar:

- Persistencia de datos mediante base de datos
- Sistema de autenticación de usuarios
- Comunicación más avanzada entre cliente y servidor

A largo plazo, el proyecto podría evolucionar hacia un sistema más completo, como:

- Un servidor de juego real con lógica más compleja
- Integración con una interfaz gráfica o cliente dedicado
- Escalado a arquitecturas distribuidas

Estas posibles ampliaciones demuestran que el proyecto no es un sistema cerrado, sino una base sobre la que se pueden construir desarrollos más avanzados.

7. Glosario

A continuación se definen los principales términos técnicos y acrónimos utilizados en esta memoria.

Asincronía: Modelo de programación que permite ejecutar múltiples tareas de forma concurrente sin bloquear la ejecución principal del programa.

Cliente-servidor: Modelo de comunicación en red donde un cliente realiza peticiones y un servidor responde proporcionando servicios o datos.

Entrypoint: Punto de entrada principal de una aplicación desde el cual se inicia la ejecución del programa.

Estado (State): Representación de una fase concreta dentro de la conexión de un jugador con el servidor. Cada estado define qué paquetes pueden recibirse y cómo deben procesarse.

Java Edition Protocol: Protocolo de comunicación utilizado por Minecraft Java Edition para intercambiar datos entre cliente y servidor.

Paquete (Packet): Unidad de datos enviada entre el cliente y el servidor dentro del protocolo de Minecraft. Cada paquete contiene información específica, como movimientos, mensajes o datos de conexión.

Proceso: Instancia de ejecución independiente dentro del sistema operativo. En este proyecto se utiliza un proceso separado para gestionar cada jugador conectado.

Protocolo de red: Conjunto de reglas que define cómo se comunican dos sistemas a través de una red.

Serialización: Proceso de convertir datos o estructuras de memoria en un formato binario o textual para poder transmitirlos o almacenarlos.

Deserialización: Proceso inverso a la serialización, mediante el cual los datos recibidos se convierten nuevamente en estructuras utilizables por el programa.

Socket: Mecanismo de comunicación utilizado para intercambiar datos entre dispositivos a través de una red.

Stage: Clase encargada de gestionar el comportamiento del servidor según el estado actual del jugador dentro de la conexión.

Struct: Módulo estándar de Python utilizado para empaquetar y desempaquetar datos binarios siguiendo un formato concreto.

TCP (Transmission Control Protocol): Protocolo de comunicación orientado a conexión que garantiza la entrega ordenada y fiable de los datos transmitidos.

Twisted: Framework de Python orientado a programación asíncrona y aplicaciones de red, utilizado en este proyecto para gestionar conexiones concurrentes.

UUID (Universally Unique Identifier): Identificador único universal utilizado para distinguir de forma inequívoca a cada jugador dentro del sistema.

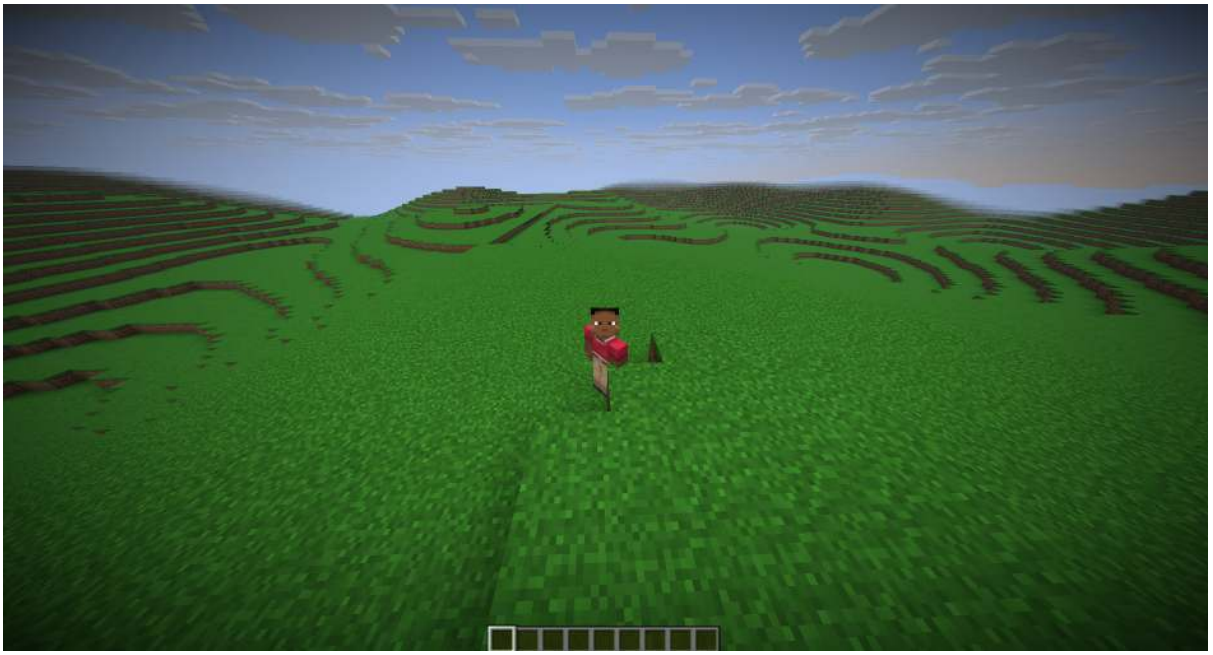
8. Bibliografía

[1] Minecraft Wiki contributors. (s.f.). *Java Edition protocol*. Minecraft Wiki. https://minecraft.wiki/w/Java_Edition_protocol

[2] Twisted Matrix Laboratories. (s.f.). *Twisted*. Twisted. https://twisted.org/

[3] Python Software Foundation. (s.f.). *struct — Interpret bytes as packed binary data*. Python Documentation. https://docs.python.org/3/library/struct.html

9. Anexos



Nota sobre el uso de inteligencia artificial

Durante el desarrollo de la memoria se han utilizado herramientas de inteligencia artificial únicamente como apoyo en tareas relacionadas con la redacción, la organización de contenidos y la mejora de algunas explicaciones técnicas. Su uso se ha limitado a la asistencia documental y a la revisión de textos, con el objetivo de mejorar la claridad y la estructura de la memoria.

La inteligencia artificial no ha sido utilizada en ningún momento para el desarrollo, implementación o generación del código del proyecto. Todas las decisiones técnicas, la arquitectura del sistema y la programación del servidor han sido realizadas íntegramente por el autor del trabajo.

Todo el contenido incluido en esta memoria ha sido revisado, comprendido y adaptado personalmente antes de su incorporación al documento final, garantizando así el conocimiento y la autoría del trabajo presentado.

Licencia