TEMA 1: INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE Y PARALELA

1.¿QUÉ ES LA PROGRAMACIÓN CONCURRENTE?

Programación concurrente: estudio y desarrollo de programas que pueden realizar varias tareas al mismo tiempo.

Surgen cuando los procesadores pueden ejecutar varios procesos de forma concurrente:

- Entorno interactivo a múltiples usuarios
- Aprovechar el procesador cuando un proceso espera por una operación E/S.

Al principio solo se usaba la concurrencia en SO (programas simultáneos), más tarde evolucionan los lenguajes de programación y librerías para que los programas sean concurrentes (tareas simultáneas).

Programa secuencial: declaraciones de datos e instrucciones ejecutables siguiendo una secuencia determinada por un algoritmo.

Proceso: ejecución de un programa secuencial en un sistema informático.

- Se pueden iniciar por segunda vez sin antes haber finalizado la ejecución previa y permite varios procesos de un mismo programa secuencial.
- Propio espacio de memoria (heap) y pila de ejecución (stack)
- Se comunican entre sí (mensajes) pero son autónomos e independientes
- Gestionados por el SO

HZ: veces por segundo que pinta información de nuevo

Proceso: proceso en el Administrador de tareas cuando se arranca un fichero

Programas concurrentes: formados por varios procesos colaborando entre sí

- Linux se comunican usando pipes (tuberías), es costoso, poco ágil
- Más importante que los procesos consuman menos recursos y compartan memoria

Hilo (threads): procesos ligeros que se ejecutan en el mismo espacio de memoria

- Tienen una propia pila de ejecución (stack)
- Consumen menos recursos que los procesos pesados
- Son menos robustos: un fallo afecta a todo el programa y lleva a la finalización inesperada

Niveles de concurrencia:

- **Nivel SO**
    - Programa secuencial: fichero ejecutable
    - Proceso: proceso que aparece cuando se ejecuta un fichero ejecutable
    - Permiten concurrencia de procesos
- **Nivel Programa**

- Programa secuencial: fragmento de código de un programa
    - Sentencias de un método con llamadas a otros métodos
- **Proceso** (hilo): ejecución independiente del fragmento/método
- Permiten concurrencia de hilos

Robustez: procesos más robustos que hilos

- Cada proceso tiene su propio espacio de direcciones y su propia pila de ejecución
    - Si uno falla los otros continúan
- Los hilos comparten el espacio de direcciones y el contexto de ejecución con el resto de hilos y el proceso padre
    - Si uno falla afecta al resto

Tiempo de inicialización: los procesos tardan más tiempo en inicializarse que los hilos

- Un proceso requiere asignación de recursos del SO
- Un hilo comparte los recursos del proceso padre

Consumo de recursos: procesos suelen consumir más recursos que los hilos

- Los hilos comparten recursos con el proceso padre: no requieren tanta memoria
- Contexto de ejecución: info para recuperar el estado de cuando se ejecutaba por última vez

Comunicación: entre procesos es más compleja y requiere más recursos que entre hilos

- Los procesos tienen espacios de direcciones separados, se comunican con pipes/sockets
- Los hilos comparten el mismo espacio de direcciones

Procesos concurrentes: si la primera instrucción de uno de ellos se ejecuta entre la primera y la última instrucción del otro.

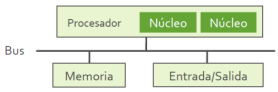
Programa concurrente (multi-hilo): varios programas secuenciales, cuyos procesos pueden ejecutarse concurrentemente en un sistema informático.

2.¿DÓNDE SE USA LA PROGRAMACIÓN CONCURRENTE?

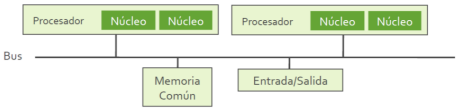
Sistemas Monoprocesador: un único core en un chip (procesador mononúcleo)

Sistemas Multiprocesador:

- **Muy acoplados**: varios cores en un único chip (procesador multinúcleo)



  - La mayoría de procesadores actuales (PC, smartphones…)
  - Multiprocesamiento simétrico: varios procesadores en la misma máquina para servidores y estaciones de trabajo



- **Poco acoplados**: memoria distribuida
  - Dispositivos conectados en red para comunicarse (actúan como un único sistema)
- **Híbridos**: varios procesadores con varios núcleos conectados en red

Tensor Core: cálculos matriciales y optimizado para realizar deep learning

Ley de Moore: cada año se duplica el nº de transistores en un circuito integrado. Cambia a 18 meses

- Ya no se puede aumentar más la frecuencia del reloj debido a la potencia y la disipación del calor, para aumentar la potencia de cómputo se necesitan incluir varios cores.

Clusters de ordenadores

- **Homogéneo**: varios equipos con el mismo SO y HW
- **Heterogéneo**: distintos SO y/o HW

Cloud gaming: tipo de juego en línea que permite la transmisión directa bajo demanda mediante el uso de un cliente

- El juego se almacena en el servidor y es transmitido a las computadoras a través del cliente

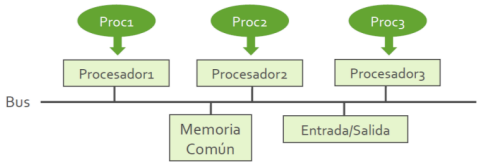
Modelos de concurrencia: hay dos grandes bloques dependiendo de la arquitectura física

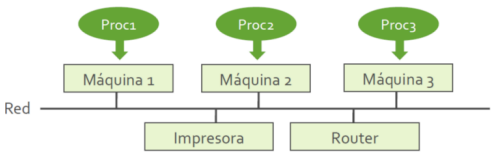
- **Memoria compartida**: variables compartidas
- **Paso de mensajes**: un proceso envía y otro recibe

Arquitecturas de Sistemas Concurrentes	Modelos de Concurrencia: Paso de Mensajes	Modelos de Concurrencia: Memoria Compartida
Monoprocesador	√	√
Multiprocesador Muy Acoplado	√	√
Multiprocesador Poco Acoplado (Red)	√	?*

Asignación de procesos a procesadores

- **Mononúcleo**: un proceso a la vez
- **Multiproceso**: cada proceso se ejecuta en su propio procesador con memoria compartida

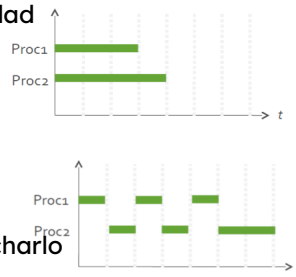




- **Procesamiento distribuido**: cada proceso se ejecuta en un procesador dentro de cada máquina de una red
- **Multiprogramación**: varios procesos en el mismo procesador
  - Los procesos comparten el tiempo

Paralelismos

- **Paralelismo real**: hay un núcleo por cada proceso, se aumenta la velocidad
  - Multiproceso
  - Procesamiento Distribuido
- **Paralelismo simulado**: varios procesos comparten el mismo procesador
  - Sensación de paralelismo real
  - Cuando los procesos esperan por E/S otro proceso puede aprovecharlo
  - Multiprogramación



Abstracciones de la Programación Concurrente

- **1º**: se considera que cada proceso se ejecuta en su propio procesador
  - Tiene en cuenta las interacciones entre procesos
- **2º**: se ignoran las velocidades relativas de cada proceso
  - Considerar sólo las secuencias de instrucciones que se ejecutan

3.¿PARA QUÉ SE USA LA PROGRAMACIÓN CONCURRENTE?

Un programa concurrente es mejor ya que es más fácil de implementar, más rápido en ejecutarse y consume menos recursos

Imprescindible:

- Servidores que atienden varios usuarios/peticiones simultáneas
  - Servidores web, de Discord, de juegos online
- Aplicaciones interactivas (tareas en segundo plano)

Conveniente:

- Aumentar la velocidad con E/S con un único procesador
- Aumentar la velocidad de los programas usando procesadores en paralelo
  - Los procesadores ya no aumenta la velocidad pero si hay cada vez más

4.¿CÓMO SE PROGRAMA CONCURRENTEMENTE?

Modelos de concurrencia: modelos teóricos que luego se concretan de formas diferentes en los lenguajes y librerías

- Modelo de memoria compartida y Modelo de paso de mensajes

Nivel de abstracción:

- **Bajo**: cercanos a la funcionalidad del HW
- **Alto**: inspirados en modelos matemáticos

Elección del modelo: tipo de programa a implementar, lenguaje de programación y librerías y arquitectura física.

Hilos y cerrojos: modelo de memoria compartida

- De más bajo nivel y disponible en lenguajes imperativos
- Más usado, más potente y más complejo

Actores: modelo de paso de mensajes

- Actores: primitivas concurrentes de procesamiento que se comunican con mensajes
- Se puede: crear más actores, enviar mensajes y cambiar su estado

Comunicación de Procesos Secuenciales: modelo de paso de mensajes

- Basado en álgebra o cálculo de procesos
- Los mensajes se envían a través de canales

Programación Funcional: modelo de memoria compartida

- La info no se modifica y se puede compartir en varios hilos sin problemas
- Procesamiento mediante funciones con valores de salida y parámetros de entrada

Modelo Software Transaccional: modelo de memoria compartida

- Transacciones de bases de datos a la memoria compartida
- Se puede leer y escribir y ocurre de forma atómica

TEMA 2: MODELOS DE CONCURRENCIA

TEMA 2.1: INTRODUCCIÓN

1.¿QUÉ ES .NET?

Plataforma creada por Microsoft para fusionar el catálogo de productos.

Arquitectura:

- Lenguajes
- Common Language Runtime
- Biblioteca de clase base

Funcionamiento:

- Escribimos un programa
- El código se compila a un lenguaje intermedio
- El código pasa al compilador JIT que lo compila a lenguaje máquina

JIT: realiza optimizaciones del código específicas para el HW

- Proporciona una capa de abstracción para aislar al desarrollador del sistema HW

2. C# PARA DESARROLLADORES JAVA

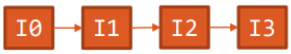
Guía de estilo:

- Clases y métodos PascalCase
- Interfaces con el prefijo I
- Llaves comienza en la siguiente línea
- Variables: camelCase

TEMA 2.2: MEMORIA COMPARTIDA

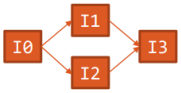
1.INTERCALACIÓN DE INSTRUCCIONES: INDETERMINISMO

Programa secuencial: instrucciones ordenadas



- Relación de precedencia (→)
- Determinismo: Al ejecutar con los mismos datos de entrada salen los mismos resultados

Programa concurrente: sentencias en orden diferentes



- No hay determinismo

1º Abstracción: se considera que cada proceso se ejecuta en su propio procesador

2º Abstracción: se ignoran las velocidades relativas de cada proceso, permite considerar solo las secuencias de instrucciones que se ejecutan

Instrucción atómica: aquella cuya ejecución es indivisible. Se ejecuta completamente sin interferencias y otros procesos no pueden interferir

Intercalación: considerar que todas las sentencias de todos los procesos se intercalan en una única secuencia

- **3º Abstracción**: considerar que las secuencias de ejecución de instrucciones atómicas se intercalan en una única secuencia
  - No hay solapamientos
  - Ejecución de instrucciones atómicas en paralelo igual a secuencial
- **Multiprogramación**: instrucciones se ejecutan de forma intercalada (solo 1 procesador)
- **Multiproceso**: si dos instrucciones compiten por un recurso se ejecutan de forma secuencial

Abstracción de la Programación Concurrente: estudio de las secuencias de ejecución intercalada de las instrucciones atómicas de los procesos secuenciales.

Indeterminación: cuando un programa obtiene resultados diferentes dependiendo de la ejecución concreta

- Observar efectos: ejecutar múltiples incrementos y decrementos
- Main es un hilo, y ejecutará Console.WriteLine() sin esperar a que los hilos inc y dec finalizan.
  - Thread.Join(): para que Main espere a todos los hilos

2.VARIABLES COMPARTIDAS

La lectura y escritura de atributos compartidos de tipo simple son instrucciones atómicas.
Si un proceso lee sobre una variable compartida y simultáneamente otro escribe sobre ella, el resultado nunca será un valor intermedio.

3. SINCRONIZACIÓN CON ESPERA ACTIVA

Procesos interactuando:

- **Comunicación**: intercambio de información
- **Sincronización**: impone restricciones a la ejecución de sentencias
  - **Sincronización condicional**: los procesos esperan a que se cumpla una condición establecida por otro de los procesos
  - **Exclusión mutua**: varios procesos compiten por un recurso común de acceso exclusivo. Solo uno accede a la vez.

Comunicación con memoria compartida: se utilizan variables compartidas

Sincronización con memoria compartida:

- **Espera activa**: utiliza solo variables compartidas
- **Herramientas de sincronización**: semáforos, monitores…

Sincronización condicional con espera activa: proceso espera a que se cumpla una condición

- Declarar un atributo compartido: public static volatile

Proceso A	Proceso B	continuar
`Console.Write("PA1 ");`		false
	`Console.Write("PB1 ");`	false
`continuar = true;`		true
`Console.Write("PA2 ");`		true
	`while (!continuar) ;`	true
	`Console.Write("PB2 ");`	true

Salida: `PA1 PB1 PA2 PB2`

Proceso A	Proceso B	continuar
`Console.Write("PA1 ");`		false
`continuar = true;`		true
`Console.Write("PA2 ");`		true
	`Console.Write("PB1 ");`	true
	`while (!continuar) ;`	true
	`Console.Write("PB2 ");`	true

Salida: `PA1 PA2 PB1 PB2`

Exclusión mutua con espera activa: cuando compiten por el acceso a un recurso exclusivo

- Sección bajo exclusión mutua: cuando se usa el recurso de acceso exclusivo
  - Las instrucciones se deben ejecutar sin intercalaciones (todas seguidas)
- **Usando una variable compartida**: variable booleana que indica si hay algún proceso ejecutando su sección de EM
  - Problema: si varios procesos entran a la vez las instrucciones se intercalan
  - Los dos miran y después entran pudiendo entrar dos a la vez
- **Usando dos variables compartidas**: antes de comprobar si hay alguien dentro (bool 1), pide ejecutar bajo exclusión mutua (bool 2)
  - Problema: si piden a la vez se quedan esperando los dos (interbloqueo)
- **Con un mutex**: Mutex mutex = new Mutex(); + mutex.WaitOne(); + mut.ReleaseMutex();

Instrucciones atómicas: dos procesos que trabajan con una misma variable

- **Grano fino**: instrucciones máquina del procesador
- **Grano grueso**: sentencias que ejecuta un proceso sin interferencias

Exclusión mutua en una instrucción atómica de grano grueso:

- **Indivisible**: ningún proceso puede interferir en el recurso compartido
- **Divisible**: se pueden intercalar las instrucciones de la sección no crítica

4. PROPIEDADES DE CORRECCIÓN

Condición de carrera: cuando el resultado de un programa depende del orden en que se ejecutan las operaciones en múltiples hilos de ejecución.

- Error: cuando alguno de los comportamientos es indeseable

Propiedad de corrección: propiedades que todo programa concurrente debe de cumplir para ser correcto. Se deben cumplir en cualquier intercalación de las instrucciones atómicas.

- **Exclusión mutua**: para evitar la corrupción de datos compartidos
- **Ausencia de interbloqueos**:
    - **Activo** (livelock): instrucciones que no producen un avance en el programa. Sirven para sincronizarse entre sí
    - **Pasivo** (deadlock): procesos esperan indefinidamente
    - **Ausencia de retrasos innecesarios**: procesos deben progresar en su ejecución
    - **Ausencia de Inanición**: todo proceso que quiera acceder a un recurso compartido deberá poder hacerlo

Tipos de propiedades de corrección:

- **Seguridad**: si se incumple en alguna ocasión, el programa se comportará de forma errónea.
    - Exclusión mutua y ausencia de interbloqueo pasivo
- **Vida**: si se incumple, el programa se comportará de forma correcta, pero será más lento y desaprovechará los recursos.
    - ausencia de retrasos innecesarios, inanición y de interbloqueo activo

Justicia:

- **FIFO**: first in first out, cola del pan
- **Espera lineal**: acceder antes de un proceso que lo haga más de una vez. Se cuela una vez, no dos
- **Aleatorio**: no es justa pero si eficiente

5. ESPERA ACTIVA VS HERRAMIENTAS DE SINCRONIZACIÓN

Problemas de la espera activa:

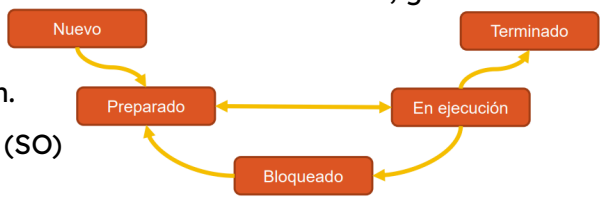
- **Multiprogramación**: procesos que esperan malgastan la CPU
- **Multiproceso**: ejecutar instrucciones consume energía y produce calor, si son inútiles debería no hacerlas.

Multiprogramación:

- **Planificación**: SO decide que proceso se ejecuta a continuación
  - Políticas: FIFO, Lineal, Prioridades o Aleatoria
- **Despacho**: se carga y se inicia la ejecución de un proceso
  - Guarda la configuración del procesador para que el proceso se ejecute de nuevo
  - Descriptor de proceso: PID, estado, entorno …

Estados:

- **Nuevo**: el proceso está siendo creado.
- **Preparado**: el proceso está a la espera de que le asignen alguna unidad de procesamiento.
- **En ejecución**: el proceso está ejecutando operaciones o instrucciones.
- **Bloqueado**: el proceso está a la espera de que ocurra un determinado evento, generalmente la finalización de una operación de E/S.
- **Terminado**: el proceso ha finalizado su ejecución.



El planificador es el encargado de despertar a los hilos (SO)

6. INTRODUCCIÓN A LOS SEMÁFOROS

Semáforos: clase formada por un contador de permisos y un conjunto de permisos bloqueados

- **Creación**: SemaphoreSlim (int permisos)
- **Wait**(): decrementa el número de permisos si hay disponibles
- **Release**(): aumenta el número de permisos

Hacer exclusión mútua: un permiso

Hacer sincronización condicional: cero permisos para frenar hasta que se cumpla

Tipos:

- Según el número de permisos:
  - **Binarios**: solo un permiso
  - **Generales**: varios permisos
- Según la política de desbloqueo:
  - **FIFO**: en orden de llegada
  - **Aleatorio**

7. SINCRONIZACIÓN AVANZADA

Exclusión mutua generalizada: más de un proceso puede ejecutar la sección de EM

- Semáforo con valor inicial de K

Comunicación con buffer: adaptar las diferentes velocidades entre productores y consumidores

- Permite insertar info aunque no esté preparado el proceso de consumir
- Los consumidores deben de bloquearse cuando no tengan datos que consumir (array vacío).
- Los productores deben bloquearse cuando no puedan insertar más datos (array lleno).

TEMA 2.3: PASO DE MENSAJES

1.INTRODUCCIÓN

En programas concurrentes se usa memoria compartida o paso de mensajes. En sistemas distribuidos (también se puede en muy acoplados)

Primitivas básicas:

- **Send**: envía info
- **Receive**: recibe info
- **Explícitas:** usadas directamente por el programador
- **Implícitas**: se ocultan dentro de otras primitivas de alto nivel

Canal: mecanismo de comunicación que permite la transferencia de los datos

Paso de mensajes depende de:

- **Direccionamiento**: identificación de procesos
- **Sincronización**: envío y recepción bloqueantes o no
- **Características** del canal

2.DIRECCIONAMIENTO

Forma en la que el emisor indica el proceso receptor y viceversa

Comunicación directa simétrica: emisor y receptor se conocen (indican id)

- Desventajas: poco flexible a ampliaciones

Comunicación directa asimétrica: el receptor no sabe el emisor

- Ventajas: flexible y usado en cliente-servidor

Comunicación directa: ambos conectados a la vez, procesos se identifican entre ellos

Comunicación indirecta: emisores envían a almacenes intermedios, receptores recogen de ahí

- Almacenes: buzones o colas
- Diversos modelos:
    - Varios procesos reciben mensajes de una misma cola o esta es exclusiva
    - Mensaje puede ser recibido por un proceso (unicast) o varios (multicast)
    - Un proceso puede enviar mensajes a varias colas
    - Creación de colas estática o dinámica (tiempo de ejecución)

3.SINCRONIZACIÓN

Comunicación síncrona: emisor y receptor conectados a la vez

Comunicación asíncrona: emisor y receptor no coinciden en el tiempo

Bloqueo de envío y recepción:

- **Asíncrona** (cita simple):
    - Emisor bloqueado en envío hasta que el receptor reciba

- Receptor bloqueado en la recepción
- **Cita extendida**: receptor envía mensaje cuando termina

- **Síncrona** (tiempo de espera)
  - A veces se especifica un timeout
  - Mensajes del emisor se guardan en un buffer, emisor no queda bloqueado
    - El emisor se bloquea si el buffer está lleno
  - Receptor recibe sin bloquearse
    - El receptor se bloquea si no hay mensajes

4.CANAL DE COMUNICACIÓN

Características:

- **Flujo de datos**: bidireccional o unidireccional
- **Capacidad el canal**: limitada o ilimitada dependiendo del HW
- **Tipo y tamaño** de los mensajes
- **Ordenación y fiabilidad del envío**: paquetes ordenados o duplicados

5.PROGRAMACIÓN CON SOCKETS

Socket: mecanismo que permite la transferencia de datos formado por par de direcciones IP y par de números de puerto

API Sockets de Berkley: no accedemos a la tarjeta de red, se crea un descriptor de archivo (socket)

- Se almacenan en una estructura de datos dentro del SO
- **AddressFamily**: servicio de direcciones, IPv4 o IPv6 (InterNetwork)
- **SocketType**: datagramas o streams (secuencias de bytes)
- **ProtocolType**: UDP (datagramas) o TCP

Programación de un servidor:

- **Servidor**: asociar el socket a una instancia de IPEndPoint
- **ReceiveFrom** es bloqueante

Cierre de conexión: Shutdown() para garantizar que los mensajes se envían y se reciben y Close() para cerrar y liberar recursos en un try-catch (try-finally)

Múltiples clientes: el servidor crea un hilo por cada persona que se esté comunicando (hay un límite)

UDP: no existen clientes o servidores.

- Es de mínimo esfuerzo, es decir, no tiene garantía de entrega
- El cliente no necesita escuchar un puerto
- No almacena cada mensaje en un buffer, si se va muy rápido se pierden datagramas

TCP: si hay buffer interno

TEMA 3: CONCURRENCIA EN LOS LENGUAJES

1.INTRODUCCIÓN

Plataforma de desarrollo: Sistema Operativo + Lenguaje + Librerías

- Lenguaje
    - Dinámico: definir en tiempo de compilación
- **Librerías**: del sistema operativo, de la plataforma de desarrollo o de terceros

2. CONCURRENCIA EN JAVA

Java: lenguaje Java + Máquina virtual de Java + librería estándar (API)

Modelo de concurrencia: memoria compartida (crea hilos)

- Aparecen herramientas de alto nivel en la librería estándar

Concurrencia: integrada en el propio lenguaje

- Definido como se comparte la memoria entre hilos en el Modelo de Memoria de Java
- **Synchronized**: palabras reservadas para zonas de EM (semáforos)
- **Volatile**: atributos compartidos entre hilos

Trabaja con objetos de la clase java.lang.Thread en la librería estándar

3.CONCURRENCIA EN C/C++

Familia de lenguajes de programación de bajo nivel, para el acceso al HW y el rendimiento

Hay diversos estándares y muchos compiladores (gcc, VisualC++...)

Tres enfoques para la concurrencia:

- **En el lenguaje**: palabras reservadas para paralelizar el código automáticamente
    - **OpenMP**: trabaja con esquema fork/join
        - #pragma omp parallel for reduction(+:c) : paralelizar iteraciones y reducir los valores de cada hilo con una operación de suma
    - **Intel Cilk Plus**: usa tres nuevas palabras clave para paralelizar sobre arrays
        - int x = cilk_spawn fib(n-1) : fib(n-1) se puede ejecutar en un nuevo hilo
        - cilk_sync: el hilo se bloquea hasta que han terminado los otros hilos
- **Con librerías de terceros**
    - Windows API y POSIX permiten concurrencia de un modelo de memoria compartida
    - Librerías multiplataforma: herramientas de alto nivel (Boost)
- **Con librerías estándar:** llega para C++11 (un poco tarde)

Existen extensiones de compiladores que amplían con nuevas palabras reservadas para que el código sea ejecutado en diferentes hilos.

4. CONCURRENCIA EN JAVASCRIPT

Diseñado para ejecutarse en el contexto de una página web dentro de un navegador.

Muchos aspectos se definen bajo el estándar HTML5

Concurrencia en primeras versiones:

- Una página web se suele implementar con un único hilo de ejecución. Al principio no podía haber dos funciones ejecutándose de forma concurrente
- Implementa un modelo de programación asíncrono: operaciones E/S no son bloqueantes
    - En vez de esperar a que llegue el resultado, se define qué se ejecutará cuando llegue la respuesta

Web Workers: permite usar un modelo concurrente de paso de mensajes en JavaScript

- Web Workers of workers a los hilos en segundo plano
    - No usan memoria compartida, se comunican con paso de mensajes
    - Ejecuta código ante la llegada de un mensaje desde el script principal u otro worker

Se está popularizando el uso de JavaScript fuera del navegador, en el servidor web **node.js**

TEMA 4: ASPECTOS AVANZADOS

TEMA 4.1: EXPRESIONES LAMBDA

1. USANDO CÓDIGO COMO PARÁMETRO

Ejemplo: obtener los elementos dentro de una lista que cumplen con un determinado filtro.

```
private static bool FindEvenNumbers(int number) {
      return number % 2 == 0; }
public static void Main() {
      List list = new List() { 1, 2, 3, 4, 5, 6 };
      Predicate predicate = FindEvenNumbers;
      List evenNumbers = list.FindAll(predicate);
```

El código es demasiado largo

Expresiones lambda: una forma compacta de pasar a un método código como parámetro

- `List evenNumbers = list.FindAll((number) => number % 2 == 0);`

2. EXPRESIONES LAMBDA

Dos formas:

- **Lambda de expresión**: (input-parameters) => expression
- **Lambda de instrucción**: (input-parameters) => {<sequence-of-statements> }

Los tipos de parámetro de entrada deben ser todos explícitos o todos implícitos; de lo contrario, se produce un error del compilador.

Método sin parámetros: `() => Console.WriteLine("Hola, mundo!");`

Varias sentencias:
```
() => { Random rnd = new Random();
        return rnd.Next(1, 100); };
```

Pueden tratarse como datos y almacenarse en variables denominadas delegados.

3. DELEGADOS

Representa referencias a métodos con una lista de parámetros y un tipo de valor devuelto

- <access modifier> delegate <return type> <delegate_name>(<parameters>)

Hay que asignar el delegado a un método compatible: `Del handler = DelegateMethod;`

Multidifusión: delegado puede llamar a más de un método

Propiedades:

- Similares a punteros a funciones en C++
- Permiten pasar métodos como parámetros
- Se usa para definir callbacks
- Delegado puede llamar a más de un método
- Orientados a objetos
- Type-safe

Delegados genéricos:

- **Action<...>:** cuando es necesario realizar una acción
  - No tiene retorno
  -
```
Action holaNombre =
      (nombre) => Console.WriteLine("Hola " + nombre);
```
- **Predicate<...>:** cuando es necesario comprobar si un argumento se cumple
  - Siempre devuelve un bool
  - `Predicate esMayusc = s => s.Equals(s.ToUpper());`
- **Comparison<T>:** cuando es necesario comparar dos elementos
  - Retorno de un int
  - Para hacer ordenaciones en estructuras de datos
  -
```
Comparison comparer =
      (par1, par2) => par1.CompareTo(par2);
```

- **Func<...,Out>:** cuando se necesita transformar los argumentos de un delegado
  - Tiene un tipo de retorno (último parámetro)
  - `Func randomRange = (min, max) => new Random.Next(min, max);`

Delegados anónimos:

- Instrucción que se puede emplear siempre que se espera un tipo delegado

```
Print printDel = delegate (int num) { Console.WriteLine("Num: {0}", num);
};
```

Son incompatibles entre sí, son iguales si apuntan al mismo método

- `D2 d2b = d1.Invoke;`

4.EVENTOS

Dos **roles principales**:

- **Emisor** (broadcaster): contiene un delegado y decide cuándo emitir, invocándolo
- **Suscriptor** (subscriber): destinatario. Decide cuando empezar y dejar de escuchar

Eventos: manera para que un objeto difunda que algo ha sucedido (invocación privada)

- Otros componentes se suscriben para recibir la notificación
- <access modifier> event <delegate> <name>: `public event Action OnEnemyDeath;`
- Al definirlo se declara un delegado (manejador de eventos) que especifica el tipo de método

TEMA 4.2: GESTIÓN DE HILOS EN C#

2. CICLO DE VIDA DE UN PROGRAMA CON HILOS

Un programa finaliza su ejecución cuando:

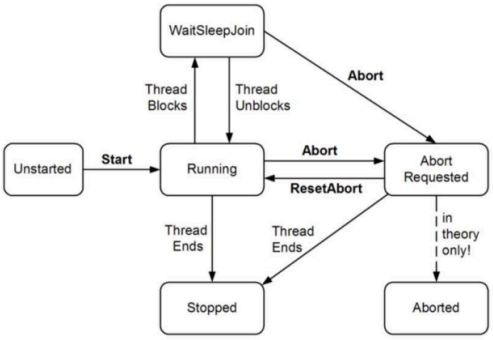
- Todos sus hilos han finalizado su ejecución
- Se ejecuta Environment.Exit()

Un hilo finaliza su ejecución

- Cuando se han ejecutado todas sus sentencias
- Cuando se eleva una excepción no chequeada

Estados de un hilo en ThreadState:

- **UNSTARTED**: definido pero no iniciado
- **RUNNING**: cuando se ejecuta Thread.Start()
- **WAITSLEEPJOIN**: hilo bloqueado
- **STOPPED**: finaliza el hilo



3.FINALIZACIÓN DE UN HILO

Razones por las que **finalizar un hilo**:

- **Cancelación del usuario**: A través de GUI o por red
- **Actividades limitadas en el tiempo**
- **Eventos** que ocurren en otros hilos
- **Errores** en la aplicación

Si tuviéramos un método **Abort**() la tarea se quedaría a medias y los objetos en estado inconsistente

Debería preguntar si otro hilo ha interrumpido e ir liberando recursos, cerrando conexiones y dejar los objetos en estado estable.

Si interrumpe hay que poner un **Join**() para que se espera a que acabe.

Los hilos que se bloquean a la espera de algo levantan la excepción **ThreadInterruptedException**

- Cuando otro hilo interrumpe con **Thread.Interrupt()**
- Finalizar el hilo de forma controlada

5. DATOS COMPARTIDOS ENTRE HILOS

Hay que marcar los atributos compartidos que pueden cambiar de valor:

- Volatile
- Usar alguna herramienta de exclusión mutua

Memory Barriers: sistema que asegura que un hilo lea el último valor de un atributo compartido

Si no está marcado el sistema de runtime reorganiza las lecturas y escrituras por rendimiento

6. OBJETOS COMPARTIDOS ENTRE HILOS

Un **objeto se puede compartir** sin problemas:

- Si se accede bajo exclusión mutua
- La clase está preparada para la ejecución concurrente
- Los objetos no se modifican mientras se comparten
- La clase es inmutable (no cambia de estado)

Clase thread-safe: si los objetos se pueden compartir sin necesidad de una EM

- Las clases inmutables son thread-safe
- Los mecanismos para hacer una clase thread-safe hace que sea menos eficiente

Objeto inmutable: aquel cuyo estado sólo puede ser establecido en la inicialización

Mejor usar una clase thread-safe que una EM, reduce el paralelismo

Crear clase thread-safe: asegurarse que sus métodos se pueden ejecutar concurrentemente sin condiciones de carrera

- Hacerla inmutable, sincronizar los hilos o con atributos thread-safe

TEMA 4.3: SINCRONIZACIÓN DE HILOS

2.EXCLUSIÓN MUTUA

Instrucción lock(): construye bloques de código que ponen bajo exclusión mutua

- **Reentrante**: si llama a otro método con ese lock (dentro de la EM) no se queda bloqueado porque ya lo tiene
- El bloqueo se libera aunque se produzca una excepción en el cuerpo del lock()
- El objeto que se mete dentro del lock debe ser inmutable
- **Limitaciones**
    - La EM no se puede adquirir en un método y liberar en otro
    - No se puede especificar un tiempo de espera máximo
    - No se puede crear una EM generalizada

3. SINCRONIZACIÓN CONDICIONAL

Monitores: originalmente era un tipo abstracto de datos (TAD)

- **Exclusión mutua** automática en los **procedimientos**
- **Sincronización condicional** con **variables de condición** (condition)
    - Como semáforo inicializado a 0

Clase Barrier(): sincronización de barrera cíclica

- Se indica el número de hilos que tienen que llegar a la barrera
- Se puede pasar un código (delegado Action<T>) para ejecutar cuando llegan todos

Clase CountdownEvent(): implementar cuenta atrás

- CountdownEvent.**Wait():** hilos bloqueados y continúan cuando la cuenta llega a cero
- CountdownEvent.**Signal():** salida de los hilos
- Usos:
    - Varios hilos esperan y uno desbloquea a los demás
    - Un hilo espera y varios hilos invocan .Signal(). Cuando todos los hilos han dado la señal, el hilo que espera se desbloquea.

TEMA 4.4: ESTRUCTURAS DE DATOS CONCURRENTES

2. OPERACIONES ATÓMICAS

Interlocked: permite realizar operaciones atómicas entre varios hilos

- Clase estática, no necesita volatile
- Más rápido que la instrucción lock (para contadores)

3. GENERICIDAD

System.Collections.Generic: estructuras de datos genéricas (thread-safe)

Genéricos: mecanismo utilizado en lenguajes de programación con tipado estático para implementar funcionalidades independientemente del tipo de datos

- **Tipo genérico <T>:** se usa al implementar una clase y se especifica cuando se usa

4. ESTRUCTURAS DE DATOS EN C#

Interfaces: definen la funcionalidad,

- Permiten manipular colecciones sin tener en cuenta la implementación

Tipos de colecciones:

- **IDictionary**<K,V>: es un mapa, guarda elementos asociados a una clave
- **IList**<T>: mantiene el orden y puede tener duplicados
- **ISet**<T>: no mantiene el orden y no puede tener duplicados
- **IEnumerable**<T>: para recorrer una colección
- **IEnumerator**<T>: acceder a los elementos de manera secuencial

Clase List<T>: array que crece de forma dinámica

- Rápido: inserción de elementos al final sin llegar a la capacidad máxima
- Lento: inserción de elementos en el límite de capacidad
    - Inserción de elementos al principio (desplazar)
- **Recorrer**: con foreach (más legible) o con IEnumerator.

Clase HashSet<T>: es la más eficiente buscando elementos (viene de ISet)

- Inserción es más costosa que en las listas
- **Recorrer**: secuencialmente con foreach

Clase Queue<T>: similar a una lista pero con un orden específico de procesado de elementos

Ordenación de elementos:

- Con IComparer<T>()
- Con expresión lambda que devuelve un entero positivo o negativo

Objetos iguales: si al comparar con .Equals() se obtiene true

Mismo objeto: si al comparar con == se obtiene true

Si una clase no tiene los métodos Equals() y GetHashCode():

- No existirán dos objetos iguales
- Si se intenta insertar el mismo objeto dos veces, no tendrá efecto

5. ESTRUCTURAS DE DATOS CONCURRENTES

System.Collections.Concurrent: diseñadas para compartirse entre hilos (thread-safe)

Consideraciones:

- Son menos eficientes que las convencionales salvo en casos específicos
- No garantiza que el código sea thread-safe
- No existe un equivalente a List<T> (listas enlazadas)

ConcurrentDictionary<K,V>: operaciones atómicas de grano grueso en forma de métodos

- Escritura protegida por locks (la lectura no)
- Están optimizadas para un acceso de datos eficiente (O(1))

Dirty reads: iteraciones sobre una EDD que se modifica dinámicamente

- Se pueden obtener tanto valores nuevos como antiguos

Snapshot: se hace una imagen instantánea de los pares clave-valor

ConcurrentQueue<T>: utiliza LinkedList (orden) y operaciones Interlocked para el principio y fin

Clase ConcurrentBag<T>: elementos no ordenados y que permite duplicados

- Útil cuando no hay orden (fairness)
- No útil cuando la bolsa deja de tener elementos (tiene que robar)

Clase BlockingCollection<T>: implementa el paradigma producto-consumidor en estructuras que implementan IProducerConsumerCollection<T>.

Clases inmutables: no modifican sus contenidos, crean copias de sí mismas con datos nuevos

TEMA 4.5: EJECUCIÓN DE TAREAS

1.INTRODUCCIÓN

Tareas: unidades de trabajo abstracta e independientes entres sí (bloques de ejecución)

Las tareas deben ser:

- **Independientes**: no deberían depender del estado/resultado de otra
- **No muy grandes**: favorece la escalabilidad y balanceo de carga

2. EJECUCIÓN SECUENCIAL DE TAREAS

Problema: el servidor no puede aceptar una nueva conexión mientras está procesando la actual

- Se desaprovechan recursos del sistema

3. EJECUCIÓN DE TAREAS EN UN NUEVO HILO

Problemas:

- La creación de un hilo es costosa
- Cada hilo consume memoria (pila de ejecución) y tiempo de CPU (cambios de contexto)
- Está limitado el número máximo de hilos

4. FRAMEWORK DE EJECUCIÓN DE TAREAS

Solución:

- Limitar el número máximo de hilos (antes de que afecten al sistema)
- Reutilizar los hilos que no ejecuten tareas (ahorrarse coste creación/destrucción)

Clase ThreadPool(): proporciona un grupo de subprocesos para ejecutar tareas

- Nos ahorra el overhead de crear hilos nuevos bajo demanda
- **Consideraciones**:
    - Depuración complicada: no se pueden asignar nombres a los hilos
    - Los hilos son siempre en segundo plano (el principal puede acabar sin terminar tareas)
    - Bloquear hilos puede provocar un peor rendimiento
- **Ventajas**:
    - Podemos especificar el nº máximo de hilos
    - Permite la reutilización de hilos
    - Podemos especificar el orden en el que se realizan las tareas

5. TAREAS EN C#

Limitaciones de hilos:

- No hay forma sencilla de devolver valores en los hilos (variables compartidas, EM…)
- Propagación de errores es complicado
- No puedes especificar que un hilo haga otra cosa cuando finaliza
- Muchos hilos con procesos de E/S degradan el sistema

System.Threading.Tasks: representan una operación concurrente

- Las Task.Run() creadas se ponen a la cola del **ThreadPool**
- Para tareas de larga duración se ponen fuera del ThreadPool con **Task.Factory**
- Se puede obtener el resultado (**Task.Result**), es un método bloqueante

TEMA 5: INTRODUCCIÓN A LOS SISTEMAS DISTRIBUIDOS PARA JUEGOS

1.INTRODUCCIÓN

Juego multijugador: modo de juego donde al menos dos jugadores participan al mismo tiempo

- **Local**: jugadores comparten la misma máquina
- **En red**: jugadores juegan en máquinas separadas físicamente

Modalidades: simultáneo (misma pantalla), hot-seat (turnos) o pantalla dividida

2. ANTECEDENTES HISTÓRICOS

Primeros juegos eran en local: Tennis for Two, Spacewar! o Computer Space

Plataforma PLATO: primera comunidad de juegos online

- Los jugadores se conectaban a un mainframe desde un terminal

Multi-User Dungeons (MUDs): juegos basados en texto desarrollados sobre mundos virtuales

- Usaban una arquitectura cliente/servidor y eran Open-Source

Hosted Online Games: pay for play, pagar por ciclos de CPU

- **BBS**: servidor locales, te conectas a través de la línea de teléfono

Juegos en red: se usaba el puerto serie

3. HITOS DE JUEGOS MULTIJUGADOR

Doom: FPS de hasta 4 jugadores

- **Protocolo IPX** sobre LAN: arquitectura P2P (se conectaban todos a la misma red)
    - **Broadcasting**: enviaba los paquetes a todas las máquinas conectadas
    - **Problema**: había muchos paquetes llegando y se perdían algunos
- Se colapsaba la red cuando había más de 5 partidas. No lo probaron bien

Quake: habilitan servers 24h con IPs públicas → arquitectura Cliente-Servidor

- Los jugadores no necesitan verse mutuamente
- Aparece **latencia**: por las conexiones al servidor lentas y mucho ping
- **Ping time**: tiempo que tarda un paquete UDP en ir del host al servidor y volver
- Actualización **QuakeWorld**: optimiza comunicaciones
    - **Client-side prediction**: predice parcialmente la situación futura (fluidez)

Age of Empires: 8 jugadores, muchas unidades por jugador

- **Arquitectura P2P**: datos llegan más rápido
- Protocolo de datos fiable
- **Deterministic lockstep**: solo manda info de control de cada jugador
    - Cada equipo hace una simulación determinista para cada unidad
- **Sistema de turnos** (200ms de frame rate)
    - Se capta la información, se sincronizan los datos y se pasan los mensajes
    - 200ms se pulsan teclas, 200ms se manda la info y se garantiza la llegada, 200ms calcular físicas y renderizar
        - Se tardaba 600ms desde el click hasta que se veía en pantalla

Starsiege: Tribes: paquetes importantes con un protocolo seguro y los menos importantes con protocolo no fiable

- Arquitectura Cliente-Servidor
- **Ghost manager**: servidor prioriza la info que necesita el jugador

4. FACTORES TÉCNICOS

Latencia: suma de retardos en la propagación de un paquete dentro de la red (RTT)

Jitter: fluctuación de la latencia entre un paquete y el siguiente

- Información que llega desordenada
- Se utiliza un buffer que va guardando los paquetes y los entrega en orden

Pérdida de paquetes: es muy habitual (entre el 1 y 2.5%)

Fiabilidad de la red: no es fiable, hay que utilizar protocolo fiables

- No se puede garantizar entrega única
- No se puede garantizar el orden de entrega
    - Numeración de paquetes, colas de procesamiento con garantía, separar datos prioritarios…

Protocolo TCP: flujo bidireccional entre dos dispositivos

- Servidor espera a nuevas conexiones
- Cliente se conecta al puerto del servidor
- Los paquetes son más grandes porque tienen info de pérdida de paquetes

Protocolo UDP: no hay garantía ni orden

- Tiene menos latencia: no hay sistema de verificación

5. SISTEMAS DISTRIBUIDOS

Sistema distribuido: conjunto de componentes independientes que se ejecutan en diferentes máquinas, da la sensación de construir una única aplicación

Clusters: conjuntos de dispositivos para ofrecer un servicio, redes de alta velocidad

- Escalables y tolerantes a fallos

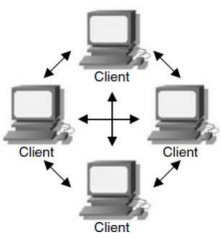
TEMA 6: ASPECTOS DE DISEÑO

1.ARQUITECTURAS DE JUEGOS

Juegos en red afrontan 3 limitaciones:

- Ancho de banda + latencia + capacidad de cómputo

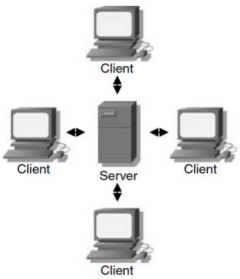
Topologías:



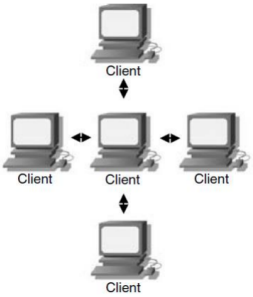
- **Peer-to-peer**: red entre pares, los usuarios se conectan entre sí sin mediadores
    - Cada jugador es responsable de algunos elementos concretos
    - Tipos de clientes:
        - **Heterogéneos**: hay algunos clientes que tienen más autoridad
        - **Homogéneos**: los clientes tienen unos recursos de los que son responsables

- **Ventajas**:
  - **Transmisión** rápida de datos: no tienen que llegar al servidor y volver
  - **Costes** económicos: no hay que implementar un servidor
  - Mayor **fiabilidad**: no hay un único punto de fallo
  - **Escalable**: cada nuevo jugador añade recursos
- **Inconvenientes**:
  - Menos **seguro**: sencillo hacer trampas
  - Difícil de **gestionar**: no hay punto con la info global
  - **Consistencia**: las correcciones se hacen en distintos puntos
  - Complejo de **desarrollar**

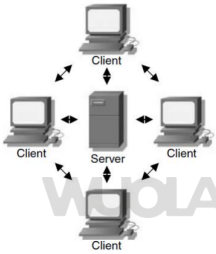
- **Cliente-servidor**: servidor centralizado que conecta a los usuarios (popular)
  - **Ventajas**:
    - Simplifica integración: hay un solo punto
    - Datos más fiables: vienen del servidor
    - Menor trabajo para el cliente: son más sencillos
    - Más sencillos de desarrollar



  - **Inconvenientes**:
    - Difícil de escalar: si el servidor no da para más no se puede hacer nada
    - Alto coste de mantenimiento
    - Punto único de fallo: si el servidor se cae se cae todo
- **Listen server**: cliente que actúa como servidor (host)
  - **Ventajas**:
    - Simplifica la integración
    - Más económico para los desarrolladores
    - Menor trabajo para cliente que no es host
    - Sencillo de desarrollar
    - Jugadores de la misma región tienen un lag bajo

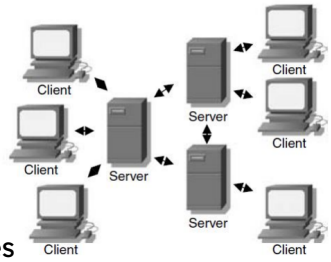


  - **Inconvenientes**:
    - Difícil de escalar: si el host no da para más no se puede hacer nada
    - Punto único de fallo
    - El host tiene ventaja respecto al resto
    - Hay mayor latencia y riesgo de pérdida de paquetes: depende de la red local del host que puede ser mala o estar congestionada
    - Seguridad
- **Híbrido**: combina P2P con cliente-servidor
  - Servidor almacena info global del juego y soluciona conflictos
  - La info no importante se mueve con Peer-to-Peer



- **Ventajas**:
    - **Latencia reducida**: cuando la latencia es crítica los clientes pueden comunicarse directamente entre sí con P2P
    - **Tolerancia a fallos**: si el servidor falla la red P2P puede mantenerlo
    - **Menos costoso**: no se depende tanto del servidor por lo tanto se puede traducir la inversión en él
- **Desventajas**:
    - **Complejidad**: desarrollar dos tipos de arquitectura a la vez
    - **Consistencia difícil**: entre nodos P2P y el servidor

- **Red de servidores**:
    - **Categorías**
        - El juego se divide en instancias completas (shards)
        - El juego se divide en regiones (juegos grandes)
    - **Ventajas**:
        - Escalable: integración de servidores permite más jugadores
        - Fiabilidad: servidores pueden ser backup de otros
    - **Inconvenientes**:
        - Jugadores aislados: usando shards se limita la interoperabilidad
        - Complejidad: por mantener consistencia entre servidores al cambiar de región
        - Costes: muchos servidores



2.RECURSOS Y LIMITACIONES

Ecuación del Principio de Información:

- Cantidad de recursos que necesita un juego: **Recursos = M x H x B x T x P**
    - **M**: mensajes transmitidos
    - **H**: receptores por mensaje
    - **B**: ancho de banda x paquete
    - **T**: puntualidad
    - **P**: ciclos CPU x paquete

Equilibrio entre consistencia y responsividad (puntualidad):

- **Consistencia**: que todos los nodos tengan la misma información
    - Depende de la arquitectura de control: centralizada, replicada o distribuida
- **Responsividad**: que tarde poco en llegar la información a todos los nodos

Escalabilidad: se debe de adaptar el sistema a las circunstancias

- Depende de la concurrencia del código y la capacidad de comunicación

3. TÉCNICAS DE COMPENSACIÓN

Optimización del protocolo: mandar mensajes tiene un coste de procesado

- **Comprimir mensajes**:
    - Se reduce el tamaño de bits del mensaje pero aumenta el coste computacional para procesarlo ya que hay que descomprimirlo
    - **Técnicas**:
        - Comprensión delta: evitar la información redundante (más usada)
        - Reducir el nivel de detalle de la info transmitida
        - Filtrar la info no relevante (tiene un coste añadido por el filtrado)
- **Reducir el número** de mensajes (agregación)
    - Reducir el tamaño de bits del mensaje y el número de enviados
    - **Técnicas**:
        - Timeout- based approach: se acumulan hasta cierto tiempo (tick)
        - Quorum-based approach: se pasa cuando llega a una cantidad de info
- Disminuimos la cantidad de mensajes, pero aumentamos el ancho de banda por paquete
- Reducimos la puntualidad pero aumentamos los ciclos de CPU por paquete

Interpolación en el cliente: reducir la frecuencia de comunicación entre los nodos

- Se interpola la info de los dos últimos ticks para generar frames intermedios
- El cliente se actualiza con un framerate superior a los ticks del servidor
- Juego más fluido pero no elimina el lag

Extrapolación en el cliente (dead reckoning): reducir la frecuencia de comunicación entre nodos

- Fases:
    - Predicción: estima la nueva situación de los datos conocidos
    - Convergencia: corrige con la realidad recibida del servidor

Predicción en el cliente: ejecutar órdenes en el cliente nada más recibir el input

- Info parecida a la del servidor si no hay alguna interacción externa
- Hay que resolver esa convergencia entre las dos informaciones
- Se duplica parte de la lógica del juego en el cliente
- No necesita una predicción determinista

Interest Manager: nodo no necesita conocer toda la info del juego

- El servidor gestiona los objetos de interés del jugador
- 3 **Game State**:
    - Del servidor: estado completo en servidores autoritativos
    - Publicable: información que se sincroniza en un momento
    - Del cliente: info del cliente

Simulación sincronizada: todos los jugadores tienen una copia completa del juego

- Las simulaciones son deterministas
- Sólo se comparte la info que cambia (no determinista)
- Necesitamos controles de consistencia y mecanismos de recuperación si algo falla

4. CRITERIOS DE DISEÑO

Tipos de objetos: inmutables, mutables, personajes jugadores (input), personajes no jugadores

Interacciones de personajes jugadores:

- Player Updates: solo al jugador
- Personaje-Personaje
- Personaje-Objeto
- Objeto-Objeto: objetos mutables

Replicación de copia primaria: para cada objeto hay una copia autoritativa (copia primaria)

- Las actualizaciones se realizan en la primaria primero

Remote Procedure Calls (RPC): sistema de comunicación punto a punto

- Pueden ser bloqueantes (void) y no bloqueantes (non-void)

5. PREVENCIÓN DE TRAMPAS

Cuatro objetivos generales en un juego multijugador:

- Usuario
- Servidor
- Cliente
- Comunicaciones

Violación de las reglas del juego:

- Depende de la arquitectura

Exposición de información: cuando el cliente conoce determinada info y la puede modificar

- Riesgo de Deterministic lockstep

Aumento de reflejos:

- Aimbot: dispara y se apunta solo
- Triggerbot: más difícil de detectar, dispara cuando la mirilla está sobre un enemigo
- Ejecutable sobre el cliente o packet tampering (modificar paquetes mientras viajan)
- Simular tener lag en momentos puntuales

Desconexiones: impacto moderado

- Hay que desarrollar un buen diseño de la lógica para penalizar

Bost para Grinding: repetir tareas repetitivas para obtener beneficios

- Expuesto a ataques de proxy o de cliente

Múltiples cuentas:

- Se suelen prohibir en los términos y condiciones del juego

Tipos de **ataques clásicos**:

- Ataques a la BD
- Keyloggers/troyanos/puertas traseras
- Robar código fuente
- DDos
- Phishing de contraseñas

Exploits: tomar ventajas de errores en la lógica del juego

- **Duping**: duplicar ilegalmente objetos
- **Chessing**: usar ataques básicos repetidamente