**RETO02**

**Ejercicio de modelado y diseño de aplicaciones distribuidas**

**“Triqui Distribuido”**

**Tópicos Especiales en Telemática**

**Alumnos:**

**Esteban Arango Medina**

**Sebastián Duque Jaramillo**

**Daniel J. Duque Tirado**

**Profesor:**

**Edwin Montoya Múnera**

**Fecha:**

**6 Marzo de 2012**

**UNIVERSIDAD EAFIT**

**MEDELLIN**

**2012**

[Análisis y Diseño 2](#_Toc318746913)

[1.1. Arquitectura 2](#_Toc318746914)

[1.2. Modelo/Middleware sistema distribuido 2](#_Toc318746915)

[1.3. Interacción asimétrica y sincrónica 3](#_Toc318746916)

[1.4. Manejo de sesión y estado 4](#_Toc318746917)

[1.5. Modelo de manejo de fallos 4](#_Toc318746918)

[1.6. Modelo de seguridad 4](#_Toc318746919)

[1.7. Niveles de transparencia 4](#_Toc318746920)

[1.7.1. Transparencia de acceso: 5](#_Toc318746921)

[1.7.2. Transparencia de ubicación: 5](#_Toc318746922)

[1.7.3. Transparencia de concurrencia: 5](#_Toc318746923)

[1.7.4. Transparencia frente a fallos 5](#_Toc318746924)

[1.7.5. Transparencia al escalado: 5](#_Toc318746925)

[1.8. Multiusuario 5](#_Toc318746926)

[1.9. Criterios o atributos de calidad 5](#_Toc318746927)

[1.9.1. Concurrencia: 6](#_Toc318746928)

[1.9.2. Extensibilidad: 6](#_Toc318746929)

[1.9.3. Escalabilidad: 6](#_Toc318746930)

# Análisis y Diseño

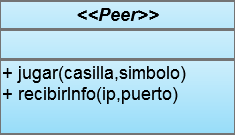
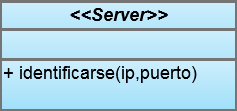
* 1. Arquitectura

Para el juego triqui distribuido se plantea una arquitectura Peer-to-Peer Hibrida “server-based” (servidor no es un peer). El servidor brinda la funcionalidad de identificación y emparejamiento de los peers que desean disputar una partida, pero no genera la conexión entre ellos. Esta es iniciada entre los dos peers que fueron “emparejados” por el servidor.

* 1. Modelo/Middleware sistema distribuido

Se utilizará el protocolo de invocación remota RPC (Remote procedure call), en donde tanto el servidor como cada peer exponen una interfaz remota y un STUB encargados de dar un manejo apropiado de los parámetros a los procedimientos, empaquetándolos en un mensaje que es enviado a la entidad que actúa como “servidor”, para que este los desempaquete, realice el procedimiento y genere el *return*. (El servidor es aquel que está proveyendo el servicio para una llamada específica). Cada entidad peer como la entidad servidor tienen un *demon* escuchando, además de *portMappers* para permitir la transparencia de localización (servicio obtenido inherentemente por la utilización de middleware de invocación remota, cada implementación depende del middleware escogido, ej: RMI, SUN, dbrRuby).

Las interfaces se describen en el siguiente diagrama UML:



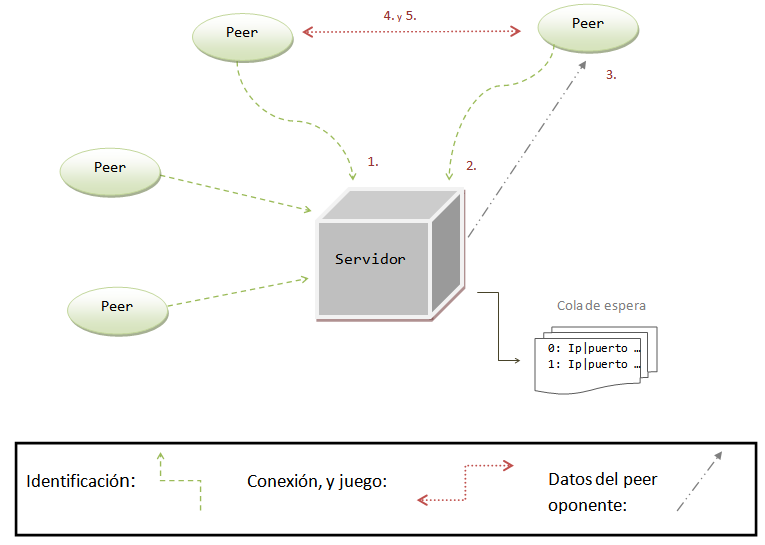
**Interfaz Peer**:

+ jugar(casilla, símbolo):Mediante este método se indica en que casilla se desea realizar la jugada y con que símbolo se marcará, un peer invoca este método en el peer contrincante para llevar a cabo la partida.+

+ recibirInfo(IP, puerto):El peer recibe una IP y puerto, si son mi IP y puerto espero a que un contrincante invoque el método de jugar. Si son otra IP y puerto me conecto e invoco el método de jugar allí.

**Interfaz Server**:

+ identificarse(IP, puerto): Los peers invocan este método del servidor para registrarse y este se encarga de ponerlos en la cola de espera o indicarles contra quien jugar mediante el método recibirInfo().



**Flujo de Sincronización***:*

1. Un peer se conecta con el servidor y se identifica enviándole su IP y puerto, y es puesto en la cola de espera.
2. Otro peer ingresa y realiza la misma acción (identificarse), este también es puesto en la cola de espera.
   * El servidor se da cuenta de que tiene dos peers en la pila de espera.
3. El servidor envía la información del primer peer al segundo, invocando el método recibirInfo().
4. Se realiza la conexión entre peers
5. Los pares realizan jugadas y sincronizan su juego invocando el método jugar().

**Detalles del diseño***:*

* ¿Qué ocurre si uno de los peer’s durante un juego pierde la conexión?

Si uno de los peer’s se desconecta, la partida será anulada inmediatamente y el peer que todavía está activo se registrará de nuevo con el servidor para esperar un nuevo emparejamiento, en el cliente habrá una lógica que anuncie este suceso (despliegue de un mensaje de aviso).

* ¿Qué ocurre si un peer se registra y pierde la conexión sin haber sido emparejado con otro peer (se encuentra en la cola de espera del servidor, esperando otro peer)?

Cuando otro peer llegue y el servidor empareje este peer activo con el peer inactivo, el peer activo, el cual acaba de llegar, recibe la información del peer inactivo por parte del servidor. A partir de ello buscará la conexión y si después de n intentos (definibles en implementación) no ha logrado conectarse, se registrará de nuevo en el servidor.

* ¿Qué ocurre si un peer se registra y otro peer activo llega a la cola y el servidor le indica conectarse con el primero, sacándolos a ambos de la cola de espera, pero nunca se genera la conexión debido a que el peer con la información ha caído?

Para solucionar esta problemática los peer’s que están en la cola de espera deberán registrarse de nuevo cada cierto tiempo (definido en implementación) y en el servidor se realiza la lógica de validación para identificar las IP’s que se encuentren en la cola y aquellas que provienen de la invocación del método identificarse().Previniendo así que existan peer’s duplicados en la cola.

Si el caso planteado en la pregunta ocurre, el peer afectado pasará un tiempo ocioso mientras el tiempo para identificarse de nuevo expira. Esto hace que el peer se identifique de nuevo en el servidor debido a que fue afectado por otro peer sin conexión cuando ya había sido emparejado.

* 1. Interacción asimétrica y sincrónica

Interacción Simétrica: Existe invocación remota de peer a servidor, de servidor a peer y entre peer’s. Todas las entidades invocan procedimientos entre sí. Los peers invocan el procedimiento *Identificarse()* en el servidor, el servidor invoca el procedimiento *recibirInfo ()* en el peer, y además, los peers invocan entre sí los métodos *jugar()* y *conectarse().*

Interacción Sincrónica: En todas las posibles comunicaciones la entidad que solicita es bloqueante mientras el proceso termina. Interacción inherentemente sincrónica.

* 1. Manejo de sesión y estado

No se tiene un manejo de estado ni una sesión de aplicación, no existe persistencia tampoco. Cada peer al conectarse brinda su IP y su puerto al servidor y espera el llamado para invocar la conexión y comenzar el juego en otro peer. Cuando un peer se desconecte no se guardará nada acerca de su última conexión y así mismo, una petición que realice no se verá afectada por otra que haga anteriormente. Las movidas se transfieren a una matriz que cada peer modifica cuando recibe una invocación del método *jugar(),* y así se va construyendo la secuencia de participación de ambos peers.

* 1. Modelo de manejo de fallos

El servidor contiene gran parte de la lógica del servicio, que es encargada de identificar y emparejar los peer’s y la solución se plantea con un solo servidor, convirtiendo el manejo de fallos en un tópico delicado. No se tiene planeación de carga ni servidores de soporte, así que si el servidor cae, el servicio caerá para nuevos peer’s que desean jugar y no conocen información de otros peer’s, pero los peer’s que están jugando actualmente o que ya conocen una información previa de otro peer seguirán ejecutándose sin problemas, debido a que no se verán afectados por la caída del servidor. Se podría plantear un segundo servidor con la misma lógica, y crear comunicación entre los peers y él, realizando rutinas de chequeo del servicio, si este no se encuentra disponible, chequear por el servidor de soporte.

* 1. Modelo de seguridad

La seguridad e integridad en la aplicación se ve estructurada en que los peer’s comparten información de la partida con su peer oponente únicamente. La información de un peer específico podrá eventualmente ser vista por n peer’s que jueguen contra él, y así conocerla, como por ejemplo su IP y puerto, pero los datos de la partida son vistos por quien los solicita solamente. Los datos de la partida son confiables, ningún otro usuario peer podrá acceder a ellos, o en su defecto atentar contra su integridad.

* 1. Niveles de transparencia

La transparencia se identifica especialmente en que aunque el servicio tiene una tendencia publicador/suscriptor, al adicionar el servidor, estos pierden comunicación directa, así, las fuentes y los clientes no se conocen, pero realizan sus comunicaciones indirectas. Para el usuario son transparentes las comunicaciones.

* + 1. Transparencia de acceso:En este servicio los clientes(peer’s) acceden a una partida como una llamada local, que es después transformada y codificada para así realizar una llamada remota, haciendo el proceso transparente para el usuario.
    2. Transparencia de ubicación:Debido a la arquitectura peer to peer hibrida *sever-based* se hace necesario que cada peer conozca la ubicación en la red del servidor, ya que este es el encargado de establecer los emparejamientos entre los peers. (El servidor es el encargado de la asignación de los rivales dinámicamente). Se utiliza además *portMappers* en la utilización de middleware de invocación remota que permite a los peer’s comunicarse entre sí y con el servidor.
    3. Transparencia frente a fallos***:*** Frente a los fallos, por su arquitectura basada en un servidor y la ausencia de herramientas como balanceo de cargas y replicación de servidores, si el servidor cae, el servicio de identificación y emparejamiento caerá. Pero, si un cliente falla, la ejecución del servicio continua eficientementey otras parejas podrán disputar partidas.
    4. Transparencia al escalado:Debido a la diferenciación de entidades, y la arquitectura P2P, el servicio puede incrementarse sin cambiar la estructura del sistema en general, nuevas funcionalidades pueden ser agregadas en cada uno de los componentes sin reemplazar la lógica de comunicación.
  1. Multiusuario

La aplicación es multiusuario porque permite que múltiples parejas de peer’s pueden jugar en un instante de tiempo, y así puedan intercambiar datos en un ambiente multijugador.

* 1. Criterios o atributos de calidad
     1. Concurrencia: Como se comentaba anteriormente el servicio no tiene problemas de concurrencia, múltiples peers pueden acceder al servidor para buscar un juego y después generar un juego con su similar. Múltiples partidas pueden estar ejecutándose al mismo tiempo sin ningún problema de concurrencia, debido a que los datos fluirán entre los dos peers mediante invocaciones remotas entre ellos, y no necesitarán pasar por el servidor.
     2. Extensibilidad: La implementación permite agregar nuevas funcionalidades sin cambios en la arquitectura. Al identificar entidades y mapearlas a clases, se permite que nuevos procedimientos puedan ser agregados sin cambiar las especificaciones de comunicación y relación soportadas en la arquitectura. En el servidor se podría agregar más lógica y funcionalidades como por ejemplo guardar estadísticas de partida. Para esto se necesita agregar un modelo de sesión a la aplicación. Estas nuevas funciones se podrán agregar cargando la lógica del cliente tanto como la del servidor, pero no cambiarán la arquitectura de la solución.
     3. Escalabilidad: La escalabilidad de la aplicación es alta debido al hecho de que los peer’s juegan entre sí, manteniendo la conexión y la gestión de la partida con su pareja de juego, y así el servidor sólo mantiene una cola en donde los peers que desean jugar se registran mediante la invocación del método *identificarse()*. Su función es despachar las partidas pero no realiza ninguna conexión, sólo invoca el método *recibirInfo()* del cliente para guiarlo hacia el otro peer. Esta tabla esta constantemente variando a medida que los peers son despachados y otros son recibidos. (La idea es que cuando se tengan dos peers en al cola, inmediatamente el servidor brinde la información al primero para que este genere la conexión con el segundo, y así sean removidos de la cola). Por todo esto la aplicación es muy escalable porque no hay mucha carga en el servidor, y además este no debe actualizar constantemente todos los peers de la red, si no que estos son los que se reportan a él cuando desean iniciar una partida.