UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

SINCRONIZACIÓN DE EVENTOS EN JUEGOS MULTI-USUARIO DISTRIBUIDOS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

FELIPE XAVIER LEMA SALAS

PROFESOR GUÍA:
SR. JOSÉ PIQUER GARDNER

MIEMBROS DE LA COMISIÓN:
SR. NELSON BALOIAN TATARYAN
SR. ANDRÉS FARÍAS RIQUELME

SANTIAGO DE CHILE
SEPTIEMBRE 2009

Resumen

Este trabajo consiste en el desarrollo de una biblioteca que permita jugar en línea con emuladores para la plataforma Windows. Se analiza y contrasta la emulación de juegos que no poseen una vía para jugarlos en línea con la sincronización y consistencia en sistemas distribuidos.

Por un lado, el distribuir la emulación de un hardware requiere una consistencia fuerte, sacrificando tiempo de reacción. Mientras que en los juegos en línea, se desea un tiempo mínimo entre entrada y reacción. Análisis para ambos casos son presentados, los que debieron ser aplicados de forma equilibrada. Para ello se define un enfoque y un aborde de sincronización controlando la entrada a la máquina emulada.

Se presenta una solución inicial rápida para ver los problemas específicos a este particular caso. Lo primero fue la conciencia de que el estado de una máquina debe ser mantenido no por el tiempo de ejecución del emulador, sino que por el tiempo virtual de la máquina emulada.

Ya con esto establecido, vino un ajuste del retardo para mejorar la interactividad con el usuario y un protocolo que se ajustara al comportamiento reacio del emulador al intentar ajustar la velocidad de emulación. Se vió que una pausa por un tiempo fijo sería compensado por la emulación.

Después de esto, frente a una buena consistencia entre los nodos se mejoró nuevamente la interactividad permitiendo que la entrada del jugador pudiese tener reintentos de ser inyectada. Esto entrega una solución satisfactoria para usuarios jugadores.

Finalmente se discuten posibles problemas no abarcados, maneras de solucionarlos y detalles de la implementación resultante.

A mi Sol

Agradecimientos

Agradezco a mi profesor guía Jo por su apoyo y orientación a lo largo de este trabajo. Nahid Akbar (a.k.a. Killer Civilian) por sus opiniones y conociemientos con *Kaillera p2p*. `iq_132` y los usuarios de los foros de FinalBurn Alpha por su ayuda con el código del emulador. A Joaquín, Sebastián y Haníbal por sus *feedbacks*.

Felipe Lema S.

Índice General

Índice de figuras

Capítulo 1

Introducción

Desde la aparición de la conexión de banda ancha a internet, el desarrollo de juegos multiusuario síncronos en línea ha aumentado. [11] En paralelo, el mundo de la emulación (ver más adelante) iba reuniendo más seguidores, tanto jugadores como desarrolladores. El enlace ocurrió con la aparición de un *middleware* para la sincronización entre emuladores, llamado *Kaillera*, desarrollado durante los años 2001-2003. Siendo una opción gratis y simple de usar con su API de cinco funciones, se veía como una buena opción para aplicar el juego en línea sobre juegos de esta categoría. Sin embargo, nunca se liberó el código y a medida que los jugadores se volvían más exigentes, junto con la moda de las aplicaciones *peer-to-peer* una implementación de *Kaillera* con este modelo era inminente. [4]

Varias implementaciones han aparecido, tanto de código cerrado como abierto. Entre ellas están:

- *Kaillera* (original): Primera implementación con un modelo cliente servidor de código cerrado. En esta se define una API para su uso.
- *Kaillera p2p*: Implementación abierta y estable de la misma API definida por *Kaillera* original con un modelo p2p entre solo dos jugadores.
- Emulinker: Implementación del servidor de *Kaillera* con arreglos a problemas de seguridad presentes en el servidor *Kaillera* original.
- *Unix Kaillera* client: Implementación inconclusa para unix/linux del cliente.
- Mystiq's Client: Parche para interfaz en castellano de *Kaillera*.

Todas estas implementaciones siguen la misma API presentada en *Kaillera* original y, por lo tanto, son compatibles entre sí.

El hecho de que *Kaillera* original fuese en código cerrado restringió su desarrollo al autor original. Este último fue dejando de lado el proyecto y al no poder ser retomado por más personas, el proyecto quedó estancado. En la mayoría de estas implementaciones presentan bugs: desincronización e inconsistencia entre instancias de juego, mal cálculo de tiempo de respuesta asumiendo y ocupando un ping estático, fallas de seguridad o incluso una funcionalidad incompleta.

Es por esto que se propone en esta memoria el desarrollar un nuevo sincronizador de códgigo abierto, basándose en múltiples estudios tanto en el área de juegos en línea como en posibles aplicaciones a *streaming* multimedia en tiempo real.

1.1. Motivación

Si bien muchas perspectivas han sido aplicadas al estudio de desarrollo de juegos en línea, la mayoría afirma que se debe incluir algún sistema de predicción y corrección. [3, 10] Otras propuestas y enfoques son hechas teniendo en mente un FPS[1] que también son aplicables en juegos en general: cada cliente no necesita la información de todo el juego, solo la zona, ítemes y jugadores involucrados; [10] [3] búsqueda rápida de recursos en un sistema distribuido peer-2-peer[2]; [3] tomar una perspectiva conservadora y ofrecer tolerancia a la pérdida y/o desorden de paquetes en la red y escalamiento en ambientes de bajo ancho de banda. [10]

Nuestro gran problema surge al tratar con juegos emulados. Estos juegos no fueron pensados para ser distribuidos a través de una red, por lo que no podemos aplicar esas ideas. Al menos, no directamente debido a que no contamos con el código fuente de estos juegos. Se presenta en la figura 1.1 cómo se desea que los clientes interactúen con la máquina emulada: Todos sienten que interactúan con la misma máquina.[3] Se piensa que los conceptos desarrollados para *streaming* de video en tiempo real a múltiples destinos están muy relacionados con nuestro problema, puesto a que ambos se basan en mantener un balance entre un ordenamiento consistente de mensajes (o frames, en este caso) entre clientes y un retardo mínimo entre envío y arribo de mensajes.

Actualmente existe una implementación de *Kaillera*, *Open Kaillera* u *okai*, bastante es-

[1] *First-Person Shooter*

[2] Al proponer un sistema sin servidor centralizado, el preguntar quién está en un cuarto no es sencillo de responder. Además debe tener una respuesta rápida

[3] Esto es lo que perciben los jugadores, pero más adelante se detalla que no es precisamente así

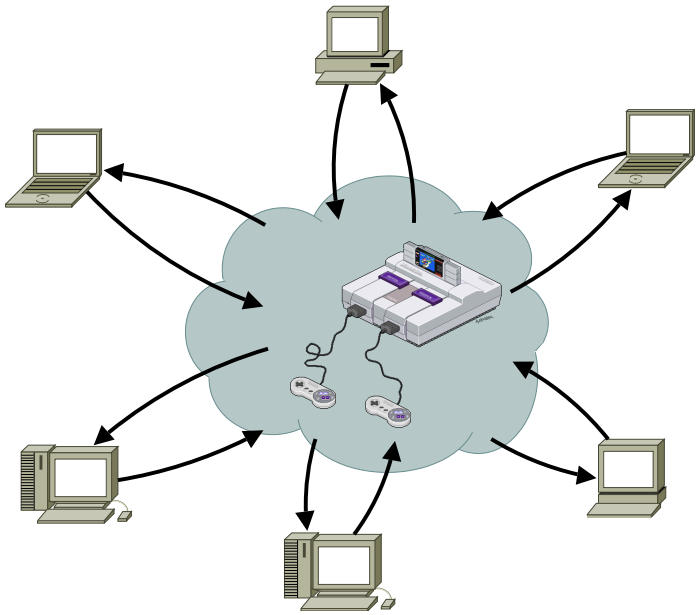


Figura 1.1: Múltiples clientes creyendo que interactúan con una sola máquina

table para redes peer-2-peer, la favorita de los desarrolladores y jugadores, ocupa la misma API de cinco funciones que el *Kaillera* original, lo que la hace compatible con emuladores desarrollados sobre *Kaillera* original, pero está limitada a dos jugadores. Nuestro trabajo tratará de mantener esta misma compatibilidad.

Por otro lado, las consolas portátiles más populares [4] llevan un par de años con juegos multiusuarios con redes ad-hoc inalámbricas. El tema de esta memoria es de interés para grandes compañías desarrolladoras de videojuegos. Se estudió también cómo resuelven ellos estos problemas, aunque estos juegos han sido diseñados especialmente para estos ambientes.

1.2. Objetivos

1.2.1. Objetivo General

Diseño e implementación de una biblioteca de código abierto y extensible que resuelva la sincronización del estado de una máquina emulada entre varios jugadores, manteniendo un balance entre tiempo de respuesta y consistencia. Esto incluye una implementación práctica e identificar los distintos problemas que puedan ir surgiendo, proponiendo una solución práctica y aceptabe para cada uno de ellos, pero también ofrecer la extensibilidad de redefinir estas soluciones para quienes quieran usarla en sus emuladores. Entre estos problemas (para-

[4] como la PSP de Sony o la Nintendo DS

metrizables) se incluyen: cálculo de *nearest neighbor*; ordenamiento y descarte de paquetes; método de actualización de eventos y predicción de retardo.

Es importante que sea código abierto porque es muy difícil para una persona abarcar y resolver todos los problemas que puedan surgir de los distintos comportamientos en la red. Aún cuando el aporte no sea definitivo, el trabajo hecho podrá ser retomado y complementado.

1.2.2. Objetivos Específicos

- Desarrollar un protocolo nuevo de mensajes entre emuladores que permita el juego en línea eficiente.
- controlar la emulación y coordinar las entradas de los jugadores de modo que todos perciban los mismos eventos consecuentes de y en la partida.
- Generar una solución escalabale que permita jugar entre cuatro participantes.
- Desarrollar un prototipo de implementación, en código abierto para que pueda ser extendido por personas interesadas en ello.
- Desarrollar una aplicación de prueba que funcione sobre esta biblioteca.

1.3. Organización

En el capítulo 2 se presenta la base de conocimientos para el trabajo. En el capítulo 3 se introduce y explica el primer ambiente armado. Luego, en el capítulo 4 se explican problemas encontrados, ajustes hechos y justificación para las iteraciones. Después se hace una especificación sobre la implementación resultante en el capítulo 5 y, finalmente, las conclusiones y discusión se presentan en el capítulo 6.

Capítulo 2

Antecedentes

2.1. Emulación y juegos

El software más antiguo tiende a ser muy dependiente del hardware en el que corre. Alguien puede adquirir un software propietario, o en este caso, juego, y debe tener el hardware apropiado para correrlo. Esto se torna un problema cuando el hardware se vuelve obsoleto y/o falla, puesto que el software se vuelve inútil al no contar con el código fuente y por su alta dependencia del hardware mencionado. La emulación brinda una solución para no perder el uso de este software, simulando el hardware y haciendo creer al software que está corriendo en él.

También hay otras dos razones derivadas del problema recién señalada por la cual se quisiese emular un hardware específico: comodidad a la hora de probar cómo correrá un software sobre una plataforma como PDAs; conseguir portabilidad de software y nostalgia. Si bien la emulación de hardware en un computador no es un tema nuevo, había sido solo abordado en grupos pequeños debido a la alta capacidad de proceso que requiere. Esto último no es solo porque basta con "correr" la máquina: normalmente se requiere que el emulador entregue de manera amigable la salida y entrada de la máquina. Para esto se requiere implementar adaptadores que funcionan entre la entrada y salida de la máquina real, y la entrada y salida de la emulada.

Por ejemplo, para emular una consola casera, debemos adaptar la entrada de teclado a un control (pudiendo diferir su tasa de refresco, compensando con capacidad de proceso) y adaptar la salida de TV en formato NTSC a la pantalla VGA del PC que corre el emulador. Hay otras veces que se pueden aprovechar recursos, en la figura 2.1 podemos ver el uso de un hardware de aceleración gráfica en un emulador, brindando una mayor resolución para la

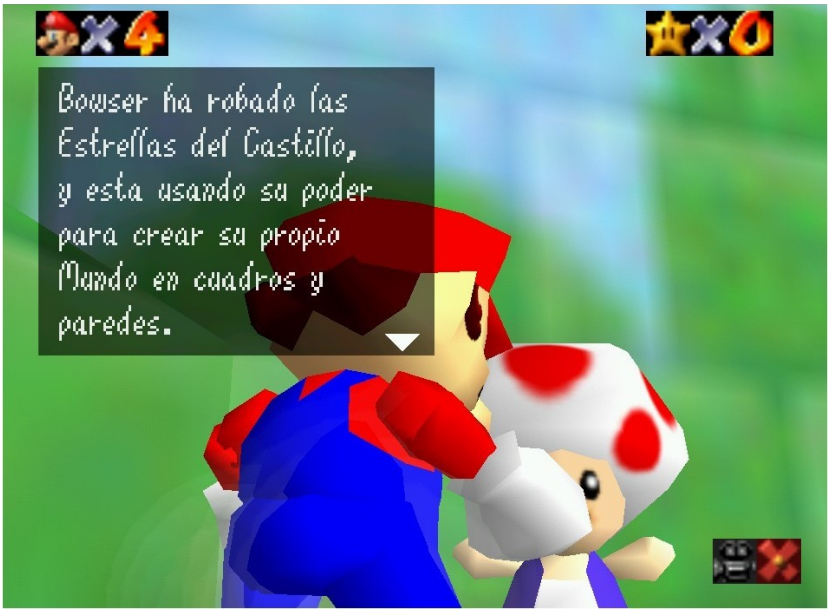


Figura 2.1: *Mario64*, de la consola Nintendo 64 corriendo por emulación en un PC

que fue hecho el juego. De esta imagen también se puede rescatar que la emulación puede no ser perfecta viendo los problemas con las transparencias con los ítemes en el HUD,[1] como el número de estrellas y de vidas.

Los emuladores de consola y máquinas arcade[2] se fueron haciendo populares cerca de 1995, principalmente porque por fin el poder de computación permitía no solo emular el hardware sino también poder usarlo como una aplicación en un computador personal, brindando a los jugadores nostálgicos el poder revivir aquellos juegos que jugaban cuando eran más jóvenes o de poder jugar aquellos juegos cuya plataforma ya no está disponible.

Con la masificación de Internet y el poder de cálculo de los computadores actuales, se ha vuelto posible el jugar de a varios jugadores distribuidos a lo largo del mundo, como si fueran multi-usuarios de la misma máquina emulada. Esto presenta varios problemas de sincronización, retardos y visualización que antes no existían. Esta memoria busca estudiar y resolver en mejor forma estos últimos problemas. [2, 5]

[1] Heads-up Display: Presentación en pantalla de información relacionada con la partida, generalmente compuesta por íconos y números

[2] No se restringe a estas dos plataformas, también se aprecia que existen emuladores para otras plataformas, como MS-DOS/PC, por ejemplo.

2.2. Acerca de sincronización

Lo que se hace normalmente en juegos en línea es ocupar un modelo cliente-servidor. El lado bueno de este modelo es que provee consistencia: el servidor recibe los mensajes de los clientes, hace un ordenamiento de eventos y envía las notificaciones o *updates* a los clientes.

El problema con este enfoque es que, en caso de congestión, un cliente puede empezar a perder updates o que sus mensajes lleguen tarde al servidor[3], haciendo que éste los procese como si fuesen eventos que ocurrieron más tarde de lo que realmente ocurrieron. Consecuentemente, el usuario experimenta una sensación de déjà vu producida por la corrección fuerte del estado en la aplicación cliente, barriendo acciones que se hayan producido durante el retardo de la actualización.

Un modelo en que los clientes se envían mensajes entre sí permite alivianar la carga y una comunicación directa entre los clientes sin pasar por el servidor. Esto no sale gratis, pues surgen complicaciones que deben ser tratadas, como el ordenamiento de los eventos.

El tiempo en un esquema distribuido no centralizado es ambiguo: no se obtiene la misma respuesta cuando dos procesos en máquinas distintas preguntan la hora. Esto es debido a que cada máquina tiene su propio reloj, por lo que no hay un tiempo absoluto, preciso ni común a todos los nodos. Es por esto que debe haber una sincronización entre relojes si se desea determinar el orden de eventos por su *timestamp*. Para mayor complicación, el envío de paquetes por redes tiene un retardo **variable** asociado. Esto complica la sincronización, puesto que se debe tomar una decisión de cómo medir y predecir este retardo, aún quedando incertidumbre sobre la exactitud de este valor nominal.

Si bien hay técnicas para mantener consistencia en un sistema distribuido, mientras mayor sea la estrictez, más mensajes se requieren para mantenerla. Esto implica una mayor demora entre acción y reacción. Dentro de las mencionadas en [12, 6, 8] la que más se apega al modelo de los videojuegos es la *consistencia causal*, una de las más estrictas. Esta consistencia puede ser relajada un poco, como se explica más adelante, pero aún así se necesita que sea bastante estricta. Debido a la alta precisión necesaria en los juegos mencionados es necesario encontrar un equilibrio entre consistencia y tiempo de respuesta, según lo que vaya siendo más aceptable por los usuarios.

[3] A este comportamiento de la red se le conoce como *jitter*

2.3. Nuestro enfoque

Para el desarrollo del trabajo se tiene como prioridad la consistencia entre los pares: las acciones de un nodo deben verse reflejada de la misma manera en todos los nodos. También, la *jugabilidad*: preservar el desempeño del emulador y no disminuir su capacidad de emular un juego en tiempo real; mantener el intervalo de acción (presionar una tecla) y reacción (visualización del efecto en el mismo nodo) en mínimo.

Abordamos la interacción en red con la sincronización de la entrada a la máquina emulada. [4]. Es la misma propuesta que *Kaillera* y se justifica a continuación.

Más adelante, en 2.3.2, se discute la relación entre consistencia y estado de la máquina emulada por medio de la sincronización de la entrada a la máquina.

Cabe mencionar que hablar de un nodo i es una referencia al computador en el que está el jugador i y viceversa.

2.3.1. Detalles de entrada

Compartir y mezclar el estado y toda la memoria de la máquina emulada enviándola completamente entre los nodos puede ser muy costoso considerando un ancho de banda actual promedio de a lo más 50Mbps. [5] [1] Seccionar la memoria tampoco es recomendable debido a que no sabemos qué partes estamos compartiendo, por lo que tenemos que garantizar un ordenamiento causal [6] sobre un número significativo de escrituras y lecturas, perdiendo la respuesta rápida del emulador y la fluidez del juego.

A priori, tampoco podemos detectar los eventos que deberían ser sincronizados, puesto que ello requeriría un trabajo de ingeniería reversa por cada uno de los juegos de manera separada.

[4]Esta entrada está, en la realidad, compuesta por palancas, botones. El emulador toma la entrada del computador como, por ejemplo, el teclado y la redirige a la máquina emulada.

[5]Para comparar, la máquina SEGA NAOMI, lanzada en 1998, tiene 32MByte de RAM, 16MByte de video y 8MByte de sonido

Dado esto, se decidió que cada nodo estuviese a cargo de la entrada del jugador que se le asigna. Este puede incluir otras entradas comunes como botón de servicio, de reset o de encendido. Eso sí, el correcto manejo de éstos últimos quedan a cargo del emulador que ocupe la biblioteca, ya que se asume que cada jugador o nodo envía una entrada que no se ve alterada por las que envían los demás.

También, como parte de la implementación, se reutiliza la entrega y retorno de entradas de jugadores. El emulador entrega de entrada un puntero a un arreglo de n bytes que representan la entrada del jugador al emulador mismo[6] y luego lee, del mismo puntero, un arreglo de tamaño $m * n$, donde m es el número de jugadores interactuando, y en que los primeros n bytes está la entrada del primer jugador, en los n siguientes la del segundo y así sucesivamente. Este arreglo retornado es el que se entrega a la máquina emulada. El valor de n se conoce en tiempo de ejecución y se mantiene constante por el resto del juego, puesto que representa una entrada física: botones, *joystick*, etc. Al emulador le es indiferente saber qué (índice de) jugador está frente al computador, de eso se encarga la biblioteca.

Esta operación de entrega y respuesta ocurre aproximadamente sesenta veces por segundo, correspondiendo a cada *frame* dibujado en pantalla según el estándar NTSC (59.94 fps), aunque su valor puede variar levemente según la máquina que se esté emulando. A la función que hace esta operación se le llama `sync`.

Esto último nos da una noción de la sensibilidad a la latencia en la red. Dado que `sync` ocurre aproximadamente cada **16ms**, en una red cerrada con una latencia menor a 8ms, por ejemplo no se debería sentir un desfase muy grande[7]: a lo más un solo frame entre entrada del usuario y llegada del mensaje a otro nodo. Más detalles de la importancia de estos 16ms por frame se ven en el siguiente capítulo.

2.3.2. Acerca del estado de máquina

Los juegos arcade normalmente tienen una pantalla o video de presentación, en que simplemente se autopublicita con una presentación y una demo corta cíclicas. Esta parte no contempla un gameplay por parte de ningún jugador, puesto que los controles son ignorados y es natural pensar que la parte interactiva no es afectada en este ciclo. Solo al insertar una

[6] no a la máquina emulada

[7] Esto en el supuesto que no hay pérdida de paquetes y que por cada mensaje hay un *ack*

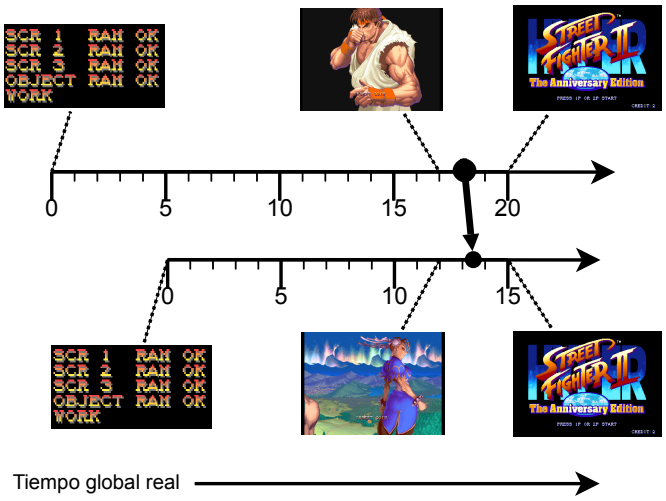


Figura 2.2: Los pantallazos son los mismos después de presionar "start", pero sus estados de máquina son distintos

ficha se puede sacar de este ciclo. Los juegos de consola normalmente también tiene este "modo demostración", solo que se escapa con el botón "start".

Cuando se activa la entrada de ficha, se pasa a una fase de interacción con el jugador. De hecho, al replicar esta inserción de ficha en el otro nodo, la misma pantalla aparece, suponiendo una especie de reinicio o cambio de estado de la máquina. En la figura 2.2 se puede apreciar esto gráficamente. Las líneas de tiempo de cada máquina (en unidades de frames procesados) están desfasadas. Los pantallazos representan el inicio de la máquina, un antes y después de presionar *start*, en distintos tiempos virtuales o "de emulación" pero mismos tiempos reales.

Luego, ambos jugadores escojen su personaje y entran a la pelea, pero en distinto escenario. Esto fue revelado en las primeras pruebas y mostró en evidencia que había un factor de generación de números pseudoaleatorios no visible que se estaba ignorando.

Esto fue crucial, debido a que hay muchos juegos que se ven influidos por este factor. Efectivamente se pudo comprobar esto después de jugar un par de veces, en que se notaba una diferencia de energía de un mismo personaje entre las dos máquinas tan grande que en una de ellas se marcaba terminado el round[8] en un nodo y en el otro no.

Este juego fue particularmente escogido para las primeras pruebas por tener un aspecto muy marcado de azar. Esto ha sido comprobado empíricamente y analizado. [9] Una de

[8]al acabarse la energía de un jugador

las consecuencias más notorias e interesantes es que un mismo poder marcado de la misma manera no es reconocido la mitad de las veces.[9] Todo esto se traduce en dos cosas. No podemos asumir que dos estados son equivalentes a menos de que hayan recibido la misma entrada en cada frame desde un mismo estado inicial. Afortunadamente, todas las máquinas emuladas parten del mismo estado. Lo único que tenemos que hacer es entregar la misma entrada en los mismos frames en todas ellas, sin márgenes de error.

[9]esto es hecho con un control programable marcado en distintos momentos de una o varias peleas.

Capítulo 3

Desarrollo inicial

Si bien el enfoque de este *middleware* es ser usado como biblioteca portable, se optó por escoger un emulador específico para partir. Debido a que se propone una API distinta, debe modificarse el código de los emuladores para compatibilizarlos. El hecho de que *Kaillera* fuese integrado en los emuladores años después de hacerse popular con un solo emulador, Project64, indica que habría un apoyo mínimo de los desarrolladores de emuladores a integrar esta biblioteca. El esfuerzo de integrarlo debía ser por parte nuestra.

Además, debido a que la distribución intencionada del *middleware* sería por medio de un archivo *dll*, para seguir la pauta de *Kaillera*, éste viviría en la memoria del emulador. Esto podía complicar el desarrollo puesto que podían haber restricciones implícitas[1] o incluso *bugs* propios de un emulador con los cuales habría que lidiar, sumado a aquellos de la biblioteca.

Junto con esto, el hecho de usar una API distinta a la de *Kaillera* obliga a que el código del emulador que se ocupe tenga que ser modificado. Esta necesidad, sumada a la complejidad del código al tener no sólo que simular el hardware en tiempo real y adaptar la entrada y salida de la máquina, sugieren que el código debe ser ordenado y entendible.

3.1. Emulador

El emulador elegido fue **Final Burn Alpha** o `fba`. Éste corre juegos para múltiples máquinas arcade cuyo enfoque es la jugabilidad. Se eligió un emulador multi-arcade porque éstos tienen un gran número de juegos soportados. En el repositorio más antiguo de emuladores, *Zophar's Domain* (`http://www.zophar.net/`), se enlistan 93 proyectos (17 activos) y para emuladores multi-arcade[2] contra 89 proyectos (9 activos) la consola NES de Nintendo.

[1] Por ejemplo, uso de recursos.

[2] Proyectos basados en el código de MAME y el de FinalBurn

[3] Aunque la consola NES tiene más proyectos, solo tiene 22 juegos para jugar con más de dos jugadores simultáneamente, mientras que los juegos soportados en los emuladores multi-arcade suman más de 70. De los proyectos multi-arcade, MAME, `fba` y Winkawaks son los más persistentes (como proyectos) y con emulación correcta. Winkawaks es de código cerrado, por lo que fue descartado inmediatamente.

En contraste con MAME y sus derivados, cuyo propósito es documentar tanto el hardware emulado como el emularlo correctamente, `fba` presenta un menor tiempo de respuesta entre la entrada del usuario en el procesamiento de la salida de la máquina emulada. Con la reciente salida de la emulación del sistema CPS3, fue fuertemente discutida en foros de la red el retardo de tres frames (cerca de 48ms) de entrada que hay presente en MAME **sin** considerar el juego en línea.

Si bien este detalle puede sonar como mínimo, es lo que lo hace el preferido entre jugadores fieles y exigentes debido a la cercanía que tiene la emulación de la experiencia con las máquinas reales, contrastado con la presición requerida para jugar estos juegos con seriedad y dedicación. Un ejemplo de cuán dedicadas y exigentes pueden ser las personas a los juegos es que en Japón está normado que las nuevas consolas no pueden ser lanzadas en días de semana pues se comprobó que esto baja la producción de todo el país.

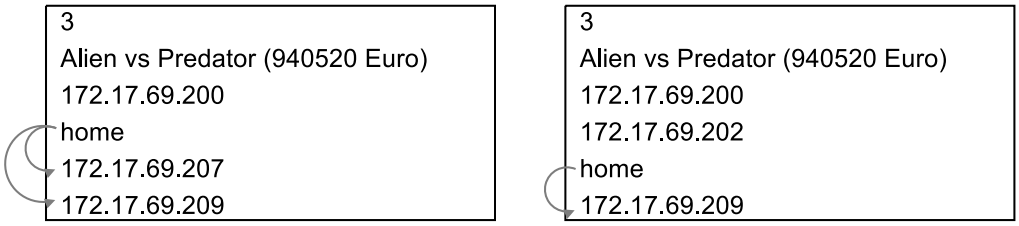
Además, es comentado entre gente del rubro que el código de `fba` se ha mantenido ordenado, aunque sin documentación. Pese a esto último se mantiene como buena opción debido a que, al igual que la mayoría de los emuladores, MAME no solo no está documentado, sino que está lleno de parches para funcionar.

3.2. Simplificaciones

Una simplificación fue la definición de participantes del juego. En ella cada jugador recibe una lista de IPs para conectar. En esta lista se reemplaza la IP de quien la recibe por la palabra "home", debido a que puede no ser simple especificar cuál es la dirección que ven los demás de uno por NAT usado en ahora comúnmente en routers hogareños. Además, es la manera más rápida y fácil de definir el orden de los jugadores que representan cada nodo.

El uso de librerías dinámicas requiere un trabajo extra por parte de quien las llame, debido a que por cada función que se se esté exportando, debe haber una función declarada y

[3]Estas dos secciones fueron las que más proyectos en total tienen en la página señalada



(a) Archivo de partida nodo 2 (IP 172.17.69.202)

(b) Archivo de partida nodo 3 (IP 172.17.69.207)

Figura 3.1: Archivos de una misma partida para distintos nodos

definida que la reciba, para no tener problemas con el *linker*. Para el caso nuestro, en que el diseño está orientado a objetos, es aún más complicado ya que hay que escapar código en *assembler*. Para no complicarse y tener una posible desviación, se optó por un enlace estático.

3.3. Configuración

Para iniciar la jugabilidad en un cliente en línea se procede de la siguiente manera:

- Se leen las direcciones participantes y el juego a jugar
- Se envía una invitación a todos los que estén bajo la dirección "home" en la lista.
- Se confirma la invitación, y la conexión se admite como establecida entre los participantes

Esto permite tener una conexión bidireccional entre los nodos de manera determinística. En la figura 3.1 se comparan dos archivos de configuración de una partida para dos nodos distintos. Las flechas grises indican la(s) invitaciones que se hacen para establecer una conexión: el nodo 2 debe iniciar la conexión con el tercer y cuarto nodo; el nodo 3 debe iniciar la conexión con el cuarto nodo.

3.4. Primer protocolo

Una vez que todas las conexiones han sido establecidas, se hace un llamado al emulador para que cargue el juego con el número de jugadores correspondientes. Una vez hecho esto, se transmiten mensajes desde cada nodo al resto cada vez que `sync` es llamado, especificando

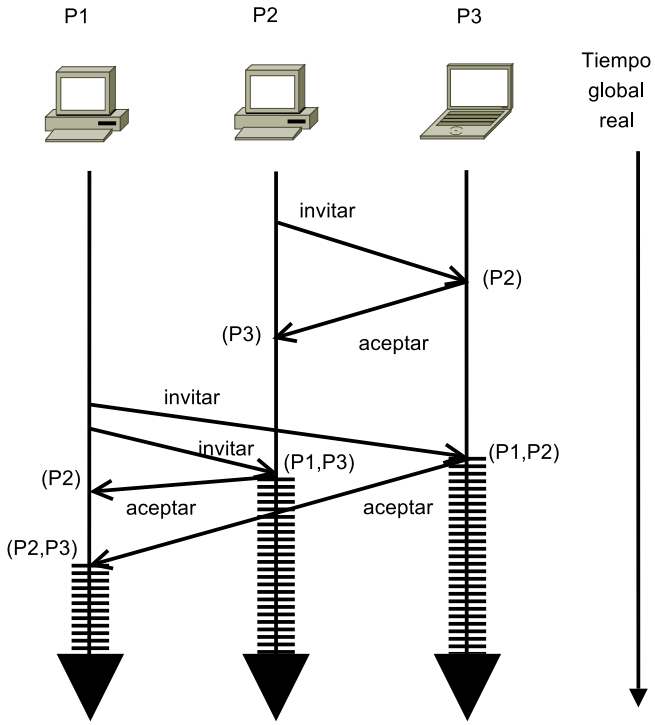


Figura 3.2: Creación de conexiones en el inicio de la partida con el primer protocolo

que el contenido del mensaje es el arreglo de entrada del jugador correspondiente al nodo que envía.

Asímismo, al recibir una entrada se almacena en el cache o réplica de la entrada del jugador que lo envía. Se asume un retraso no significativo, lo que permite que en cada frame los mensajes de input o datos lleguen a todos los nodos. En caso de llegar un mensaje atrasado, comparando su número de frame con el frame de la emulación, este es descartado. No hay confirmación ni ordenamiento de mensajes.

Junto con esto, `sync` automáticamente toma la entrada entregada por el emulador (usuario) y la almacena en el slot correspondiente (caché del jugador local).

En 3.2 se presenta el esquema para la creación de conexiones al iniciar la partida. Dentro de los paréntesis están las conexiones activas que ve cada nodo: el nodo 3 ve que se realizó la conexión del nodo 2 al recibir la invitación de este; el nodo 2 ve que la conexión está activa al recibir la respuesta y así sucesivamente.

Cada nodo inicia la emulación al ver que todas las conexiones con el resto de los nodos han sido establecidas. Aunque la emulación no se inicia al mismo tiempo, se pensó que los retardos de una red cerrada (1ms) no afectarían la sincronización en tiempos de frames (16ms).

3.5. Primeras pruebas y problemas

Las primeras pruebas fueron entre dos nodos en una red cerrada con el juego "Hyper Street Fighter II" con una latencia siempre menor a los 2ms. El primer problema que se notó fue que, al ser distintas máquinas en que se corría el emulador, demoraron tiempos distintos para cargar el juego en memoria. El emulador (y cualquiera, en general) no está pensado para sincronizar el inicio, por lo cual el coordinar la llamada a cargar el juego se vuelve inútil. Por esto, podía generarse fácilmente una incongrunecia al ingresar una entrada cuando el nodo atrasado aún no iniciaba la emulación.

Además de esto, aún cuando la entrada alcanzaba a llegar de un nodo a otro dentro del mismo frame, se producían incongruencias. Esto fue explicado en 2.3.2.

Capítulo 4

Sincronización y pérdidas

Para que cada entrada llegue en el tiempo correspondiente, se ideó ocupar el número de llamados a `sync` como reloj, equivalente al número de frames procesados. Esto es válido en la medida que no haya *frameskipping* o que el emulador garantice que el llamado a `sync` sea continuo. El frameskipping consiste en sacrificar los cálculos gráficos cuando los recursos para emular la máquina y el juego en tiempo real no son suficientes. El código de `fba` tiene mezclado el código de dibujo en pantalla con el reconocimiento y entrega de la entrada del usuario a la biblioteca. Al menos se puede desactivar el frameskipping en `fba`, pero es un aspecto a tener en consideración.

El transmitir en cada frame es el mismo método que ocupa la versión *Kaillera p2p* para dos jugadores. Esto permite tener una mejor latencia entre los nodos y, consecuentemente, poder coordinar mejor la emulación en tiempo real. Sin embargo, ya escalando el número de jugadores, el mantener una emulación alineada se hace más difícil. En el peor de los casos, podría haber deadlocks de sincronización por la distinta perspectiva que tiene cada nodo del resto. Ante esta problemática se optó por reducir el número de paquetes enviados y relajar la restricción de sincronización en tiempo real.

4.1. Manejo de entrada

El siguiente paso fue implementar un caché y enviar solo los cambios de entrada para reducir el número de paquetes enviados. Para este propósito fue necesario diferenciar dos caches para el jugador local: caché de envío (equivalente a aquel que entrega `fba` a la biblioteca) y el caché de lectura (aquel que se retorna a `fba` para ser procesado por la máquina

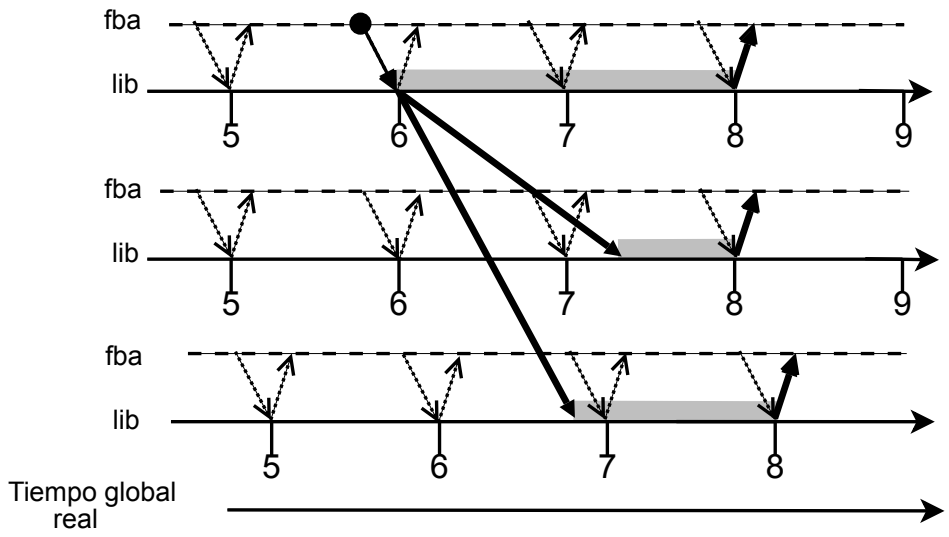


Figura 4.1: El segundo jugador activa un botón para el frame 6, pero se entrega este cambio al emulador dos frames más adelante. El resto recibe el cambio momentos antes de procesar el frame 8, haciendo efectivo el cambio a los emuladores

emulada).

El primero es actualizado cuando el usuario ingresa una entrada distinta. Cuando esto pasa, también se encola este cambio para el caché de lectura, para que se refleje[1] en unos frames más. Este retardo inducido corresponde al tiempo suficiente para que el cambio alcance a llegar al resto de los clientes. En una primera instancia, el retardo estaba fijo en cincuenta frames (aproximadamente 800ms).

Este método se asemeja a aquel usado en [7]. En ambos se desea que, al haber un evento, todos los estados de los juegos sean actualizados al mismo tiempo, aún cuando se reciba el mensaje que contiene el evento antes. A esto se le llama *imparcialidad en la actualización de estados* en la misma publicación. La diferencia es que en esta fase no se garantiza esta imparcialidad puesto a que los nodos toman distinto tiempo en partir, entonces los eventos ocurren en distinto tiempo real (pero en el mismo tiempo de emulación).

Las réplicas en cada cliente solo tienen caché de lectura que se actualizan con un mensaje de otro cliente (el cliente que tiene a ese jugador como usuario). Como en la vida real, un jugador no puede, o al menos no debería poder intervenir en la interacción con la máquina de los demás.

[1] la biblioteca lo entregue para ser procesado por la máquina emulada

Para tener una mejor respuesta y jugabilidad, había que disminuir este retardo fijo de entrada, por lo que se implementó un cálculo de retardo entre los clientes para poder mantener este retardo inducido en un mínimo.

4.2. Manejo de retardo

Debido a que el intentar obtener un retardo mínimo entre los clientes implicaba una serie de intercambios de mensajes, fue necesario también coordinar la partida de la emulación, una vez establecidos los retardos y cargados los datos de la emulación (cartucho ROM, desencriptación del mismo). Gracias a que la función `sync` es llamada al dibujar cada frame en pantalla podemos detener la emulación en el primer frame en cada nodo hasta que todos estén en la misma situación para poder resincronizar una nueva partida.

Para el cálculo del retardo se ocupó un esquema de pings y pongs. En cada paquete de ping se incluye el máximo retardo local almacenado. Cada nodo almacena el máximo retardo recibido o calculado con la salida del paquete de ping y la llegada del paquete de pong, asumiendo que el retardo de un nodo a otro es la mitad del tiempo entre un ping y su pong correspondiente. Esta operación se repite varias veces en cada nodo para que el mismo máximo retardo alcance a ser internalizado por todos los nodos. Este máximo es el que se ocupa para el retardo de la entrada en frames. Cada 16ms implican un frame de retardo.

El cálculo de retardo se efectúa una sola vez. Se asume que el mayor retardo se alcanza a registrar mientras todos estén en esta misma etapa y que se mantendrá constante por el resto de la partida. Se asume también que todo paquete alcanza a llegar dentro de ese retardo, al menos por esta etapa.

Para el manejo de entradas, cada nodo tiene un contador del último frame procesado. Al recibir la entrada del jugador local por parte del emulador, se admite esa entrada para ser procesada “al menos, en el siguiente ciclo”:

```
1 p2pSync::sync(char *playerData, int size){
2   //currentFrame = 19
3   //...
4   //recibir para la 20a entrada
5   unsigned char frame2store = currentFrame+1;  //frame2store = 20
6   frame2store += frameDelay;
7   replicas[ localPlayerNo ]->store(playerData, size, frame2store;
```

```
8     //...
9
10    //retornar datos
11    int di=0;
12    for(int i=0; i< nPlayers; i++)
13      char *input = playerReplicas[i]->getInput(currentFrame+1); //
          obtener n20
14      for(int j=0; j<size; j++,di++)
15      playerData[di] = input[j];
16
17    // entrada 20 procesada
18    currentFrame++;
19  }
```

Esta condición de borde es crucial debido a que distingue qué entradas pueden o no ser inducidas. Por esto mismo, las líneas 11 hasta la 18 son manejadas como sección crítica.

El problema que surgió de este enfoque es que habían mensajes que no lograban llegar antes del frame al que estaban designados a ser accionados. Estos mensajes, al no poder inducirlos, simplemente eran descartados. Esto, sin embargo, no pasaba en el nodo de origen pues se asumía implícitamente que todo mensaje alcanzaba a llegar.

Todo esto ocurrió aún con la resincronización original después del cálculo de retardo. Este problema es abarcado con mayor detalle en 4.4. Luego de varios intentos de enmendar esta desincronización, se optó por enfocarse en una mejor consistencia. Para esto se optó por un sistema de two-phase commit para cada mensaje.

4.3. Llegada tardía y two-phase commit

El modelo que se implementó estaba basado en un two-phase commit en el que se envían los datos, se recolectan las respuestas de los nodos receptores y se envía una confirmación una vez se hayan juntado todos. Este enfoque fue usado porque cada cambio en una entrada debe ser primero consultado con el resto de los nodos. Si estos cambios fuesen inducidos solo en algunos nodos habría problemas de inconsistencias que enmendar, como se comentó anteriormente. Por otro lado, al no tener un stream de paquetes, es más difícil prever cuándo un input va a llegar, lo que hace más complicado revertir los cambios. Este enfoque también permite preparar a los nodos para estos posibles conflictos.

La implementación fue optimista, puesto que no contemplaba mensajes para cancelar la inducción. Cada cambio encolado en el caché vendría también con un estado: confirmado o no confirmado. Por defecto, el estado era no confirmado, lo que permitía naturalmente que los mensajes de datos que no alcanzaran a llegar antes del frame objetivo[2] fuesen descartados. Los nodos que no pudieron inducir esta entrada no contestarían al nodo enviador y, finalmente, esta entrada no sería confirmada en el resto de los nodos. Al final de este ciclo esta entrada habrá sido ignorada.

Luego, con las pruebas, se vería que habían *commits* que no alcanzaban a llegar a tiempo. Esto ocurría al asumir que si alguien respondía con un *ack*, alcanzaba a recibir el *commit* a tiempo. Esto se traducía en un cambio ignorado en el nodo con el atraso, mientras que en otros nodos el cambio sí era confirmado. Aunque esto pasara unas pocas veces en medio de muchas acciones de réplica por segundo, era fatal por lo explicado anteriormente en cuanto a la sensibilidad. Los ajustes para mantener una consistencia aún no habían terminado.

4.4. Autoevaluación en los nodos

Aún con la sincronización inicial, se observó muchas veces (aunque no todas) una diferencia de frames de hasta tres o cuatro frames de adelanto por parte de un nodo. Esto resulta extraño teniendo en cuenta que se trabajó en una red cerrada con retardos de a lo más 2ms en comparación a los 64ms que representan este desfase. Si bien se mantenía consistente, sólo el o los nodos que fuesen adelantados podían transmitir al resto de ellos. Esto ocurría porque al enviar un cambio de un nodo atrasado éste no lograba llegar antes del frame objetivo en el nodo adelantado, aún con el retardo previamente calculado en base a los retardos entre los nodos. Revisando el registro se certificó que estos paquetes no se habían perdido.

Por un lado, se implementó un *abort* por parte de los recipientes al enviador. Al recibir un mensaje de cambio, se compara el frame objetivo con el último frame procesado. Esta diferencia se compara con el tiempo necesario para que este mensaje alcance a ser confirmado, equivalente a la demora estimada del arrivo del (*no-*)*ack* al enviador y el *commit* de vuelta. [3]. En caso de que se estime que el tiempo necesario antes de ser procesado el frame objetivo

[2]frame en el que se produce el cambio en el caché

[3]Todo esto asumiendo que cada frame equivalen 16ms

```
MessageManager::receiveData(char *buf){
  //...
  int frameDifference =
    // lastProcessedFrame - receivedFrame
    compareWithCurrentFrame( receivedFrame );

  int neededMS = senderNodeInfo->estimatedDelay*2
    - frameDifference;
  //...
}
```

Figura 4.2: Estimación de inducción

no fuese suficiente, se retorna un *no-ack* y la emulación se detiene por unos milisegundos de reajuste. Éstos corresponden al tiempo necesario para que el enviador "alcance" al recibidor, sincronizando el primero con el segundo.

Un detalle relevante es que la espera debe ser hecha en el thread de emulación, por lo que al recibir el paquete en el thread de conexiones y estimar la espera, se debe **encolar** para que lo tome en el thread correcto el próximo llamado a `sync`.

Este procedimiento de inducción también fue optimizado al relajar la condición sobre si puede ser confirmada la entrada recién recibida, solo preguntando si el frame de ésta no ha sido aún procesado. Esto requirió una pequeña modificación a `sync` requiriendo que cada entrada no confirmada debe ser esperada. Para esto se ocuparon objetos de sincronización específicos a threads en Windows para (por un lado) esperar y (por otro lado) avisar la llegada de la confirmación. Esto nos libera de tener que estimar correctamente la respuesta del enviador.

Esta espera fuerza a los nodos a mantenerse sincronizados, ya que al esperar la confirmación de la entrada vuelven a una sincronización en la que ninguno necesitaría esperar (o incluso descartar completamente) una entrada nuevamente.

Las pruebas que siguieron mostraron una buena consistencia, pero un déficit muy grande en la jugabilidad. En variadas ocasiones, los controles "quedaban pegados": requería de reinsistencia manual para marcar una posición de la palanca. El problema radicaba en que las entradas no alcanzaban a ser inducidas porque al recibir una entrada con un número de frame atrasado o "ya procesado", se descartaba.

Para estos casos de descarte, también se incluía una detención del emulador correspon-

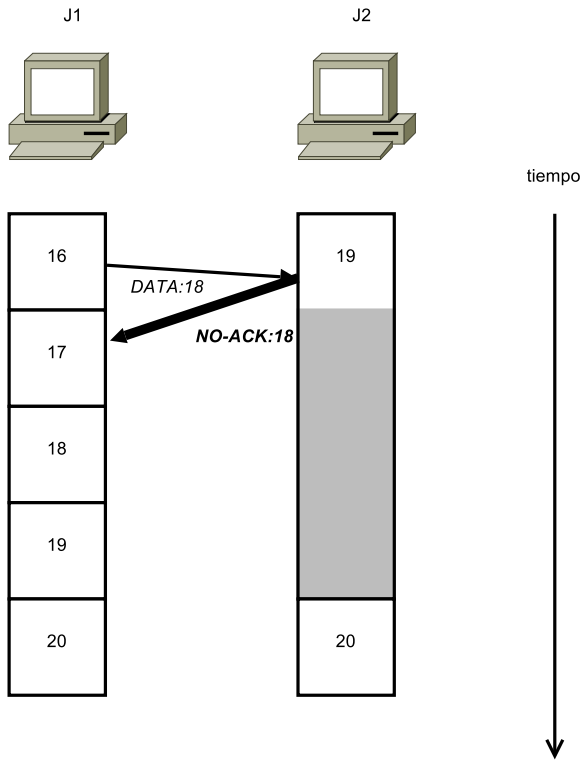


Figura 4.3: Alineación

dientes a los que necesitaba el nodo recibidor para que se esté sincronizado con el enviador para futuros envíos de entradas.

En la figura 4.3 se muestra un ejemplo para dos jugadores en el que el segundo va adelantado. Éste avisa al nodo 1 que descarte la entrada para el frame 18 y detiene su emulación para que el primero lo alcance. Finalmente, ambos nodos están alineados en el mismo frame.

El tiempo señalado anteriormente en 4.2 también contempla esto. El problema es que, aún midiendo y confirmando el tiempo de alto en el emulador, el tiempo de desfase permanecía constante como si no hubiese ocurrido la detención. Hay sospechas no confirmadas de que el sistema de emulación intente compensar el tiempo de pausa para mantener una emulación en tiempo real.

Aunque, por otra parte, unas pruebas particulares en las que se ocupó una proporción mayor al tiempo de pausa calculado mostraron que había una etapa de ajuste por parte del nodo atrasado, pero que el o los otros nodos ahora irían atrasados con respecto al recién ajustado.

```
1  MessageManager::receiveData(char *buf){
2    // ...
3    int frameDifference =
4      compareWithCurrentFrame( receivedFrame );
5    if( frameDifference > 0 ){
6      answerAck( outBuf );
7      // currentFrame++
8      otherThreadCallBack->receivedData( dataPackage );
9    }
10   // ...
11 }
```

Figura 4.4: Mal manejo de concurrencia

4.5. Optimizaciones

En esta etapa del trabajo se hicieron varias optimizaciones en el código en base a los errores identificados en casos específicos. Estos casos eran muy difíciles de reproducir, debido al paralelismo por el uso de redes sumado al paralelismo por el uso de threads en cada nodo. El usar un registro de manera minucioso ayudó mucho a identificar estos problemas. Algunas de estas optimizaciones, sobre todo los cambios en herramientas, fueron pensadas para un mejor manejo de recursos permitiendo, por lo tanto, menores restricciones para portar el código a otra plataforma.

4.5.1. Atomicidad y manejo de threads

Un problema de concurrencia que debió ser verificado fue la atomicidad de consultas y acciones. En el siguiente código se muestra cómo se recibían los paquetes de una entrada nueva:

Entre la comparación con el último frame procesado (líneas 3 y 4 del código 4.4) y el encolamiento para ser procesado (línea 8), se alcanza a procesar un nuevo frame algunas veces. Esto da problemas cuando los datos son para el siguiente frame a procesar, por lo que se confirma un dato al nodo original, pero no alcanza a ser administrado por el thread de emulación, conllevando a una inconsistencia.

Aún cuando no debiese ser labor de la clase manejadora de entradas, la comparación y estimación de si la entrada recibida alcanza a ser procesada ha quedado ahí debido a este problema.

```
1  MessageManager::receiveData(char *buf){
2    // ...
3    int frameDifference =
4    otherThreadCallBack->compareAndReceive( receivedFrame, dataPackage
         );
5    if( frameDifference > 0 ){
6      answerAck( outBuf );
7    }
8    // ...
9  }
```

Figura 4.5: Correcto manejo de concurrencia

En la plataforma Windows no hay un equivalente a `pthread_cond_wait`.[4] Esta función permite atómicamente dejar una sección crítica y esperar una señal de otro thread. Esto era necesario para el procesamiento de entradas en la parte de entrega desde las almacenadas al emulador. Antes de entregar la entrada de un jugador, se chequea en caso de tener una nueva, si ésta ha sido confirmada. Por otro lado, se debe recibir la confirmación esperada. En los códigos 4.6 y 4.7 se presenta el primer aborde a este problema

La pregunta que se hace en la línea 11 del código 4.6 es si, en caso de tener un cambio de entrada, este ha sido confirmado. En caso de que no haya sido así, se debe esperar a que haya sido confirmado el cambio. No se puede esperar la señal sin haber liberado el semáforo debido a que quedaría en un deadlock. El problema surge en que entre las líneas 18 y 21 se ejecuta todo el código 4.7, por lo que la señal enviada en la línea 13 es ignorada. Una corrección fue cambiar la línea 11 del código 4.6 a:

```
11     while( !playerCache[i]->inputReady( currentFrame ) ){
```

Alternativamente, se puede configurar el objeto para que mantenga la señal activa hasta que otro thread espere por una señal. Sin embargo, ninguna de estas opciones son recomendadas, pues la entrada de un jugador con índice menor puede llegar, ser almacenada al tener un número de frame mayor, pero no procesada, lo que llevaría a una inconsistencia.

Por ejemplo, mientras se espera la confirmación de la entrada del frame 13 del jugador 3, puede llegar la entrada del jugador para el mismo frame, pero del jugador 2. Por comparación

[4]función para control de concurrencia definida en threads de POSIX suspende el thread que llama hasta que otro thread notifique su liberación

```
/* thread de emulaci'on */
p2pSync::synchronize(char *data, int size){
  // En data viene la entrada del jugador y
  // dejamos la entrada a ser procesada

  // ...

  // entrar a secci'on cr'itica
  waitForSingleObject( semHandle, INFINITE );
  for( int i=0; i< totalPlayers; i++){
    if( !playerCache[i]->inputReady( currentFrame ) ){
      // hay entrada nueva, pero no confirmada

      // estamos esperando esta entrada del jugador/nodo
      unconfirmed.add( make_pair( currentFrame, i) );

      // salir de secci'on cr'itica
      releaseSemaphore( semHandle, 1 );

      // esperar se~nal de otro thread
      waitForSingleObject( eventHandle );

      // entrar a secci'on cr'itica
      waitForSingleObject( semHandle, INFINITE );
    }
  }
  // ...
}
```

Figura 4.6: Proceso de entrada

```
/* thread de mensajer'ia */
p2pSync::receivedConfirm( int playerIndex, int frame ){
  // entrar a secci'on cr'itica
  waitForSingleObject( semHandle, INFINITE );
  // ...
  if( unconfirmed.find( make_pair( frame, playerIndex) ) ){
    //esta es (al menos) una entrada no confirmada
    unconfirmed.erase( make_pair( frame, playerIndex );

    //era la 'ultima entrada sin confirmar
    if( unconfirmed.size() == 0 )
      //enviar se~nal para continuar
      SetEvent( eventHandle );
  }
  // salir de secci'on cr'itica
  releaseSemaphore( semHandle );
}
```

Figura 4.7: Recibimiento de confirmaciones

del último frame procesado (12), sería almacenada, pero ya se habría procesado la entrada del frame 13 del jugador 2.

Esto obligó a primero confirmar todas las entradas en un frame antes de procesarlas (ambas acciones atómicamente) como se muestra en el código 4.8.

Este patrón de prechequeo y esperar cíclicamente se repite en varias partes del código.

Otro punto que vale la pena recalcar es que la API de Windows ofrece herramientas para

```
  bool allReady=false;
  while( !allReady ){
    allReady = true;
    for( int i=0; i< totalPlayers; i++){
      if( !playerCache[i]->inputReady( currentFrame ) )
        allReady = false;
    }
    if( !allReady ){
      //agregar a la cola y esperar se~nal
    }
  }
```

Figura 4.8: Pre-chequeo atómico

exclusión mutua llamadas *Critical Section Objects*. En la documentación se señala que funcionan similarmente a un objeto de exclusión mutua, pero que tienen un mecanismo más eficiente y más rápido para la sincronización. En la práctica, su uso trajo retardos de más de 900ms al ingresar a una sección crítica.[5] Ante esto se mantuvo el uso de semáforos.

4.5.2. Estimación de retardo

Durante el ajuste de los tiempos para el cálculo de retardo se percibieron pequeñas anomalías debido a que varias veces los paquetes de ping llegaban en desorden. Por esto, se agregó un identificador a cada paquete de ping y pong, guardando el último ping en la información de cada nodo. Se asume un entorno de rápida respuesta. De no ser así, al recibir un pong distinto al último ping esperado de un cierto nodo, se aumenta el tiempo entre pings para ese nodo, adaptando el cálculo de retardo al entorno.

Se agregaron, también ponderadores para retardos calculados para dar peso mayor a los retardos más grandes. La idea detrás de esto es no sacrificar la jugabilidad por "hipos" o atrasos aislados al mantener un retardo máximo fijo. Así mismo, su peso mayor hace no desmerecerlos; sólo un buen promedio puede mantener bajo el retardo estimado.

4.6. Reinsistencia

Para la última fase se diseñó e implementó un algoritmo de reinsistencia sobre el esquema anterior. Pensando en el último problema en que se desea que toda entrada llegue a ser procesada, el recibidor puede sugerir un frame distinto para la entrada recién recibida en caso de que ésta no pueda ser inducida en ese nodo. La mejor sugerencia de frame para cada nodo que desee rebotar una entrada es su siguiente frame a ser procesado. A esta acción se le llamó "bouncing" o "rebote" por la analogía al *salto*[6] de frame.

Una vez recolectadas todas las respuestas, el enviador toma el mayor frame de rebote recibido y confirma a todos que deben ocupar este frame. Tanto quienes confirmaron como los que ofrecieron quedan esperando ya sea al frame original o a su propia propuesta de frame de rebote.

[5] Esto fue notado solo en Windows Vista
[6] retardo inducido.

```
receivedNewData(frameNo, playerNo, data)
    player[playerNo][frameNo] = newInput(data)
    answerMsg = ... // Datos de respuesta
    if frameNo >= currentFrame
        player[playerNo].bounceFromTo(frameNo, currentFrame)
        answerMsg.TYPE = "bounce"
    else
        answerMsg.TYPE = "correct"
```

Figura 4.9: Creación de rebote (recibidor)

```
receivedAnyAck(originalFrame, bounceFrame, playerNo, data)
    storedInputs[originalFrame] =
    if storedInputs[originalFrame].allNodesCollected
        highestBounceFrame = storedInputs[originalFrame]
        timeToWait = (highestBounceFrame − currentFrame) ∗ 16
        for each node N in nodes
            deltaT = timeToWait − N.estimatedDelay
            confirmBounceMsg = newMsg(highestBounceFrame, originalFrame)
            sendLater(confirmBounceMsg, N, deltaT)

    else
        ... // Confirmar a todos inmediatamente
```

Figura 4.10: Recepción de noACK con rebote (enviador)

Aprovechando el hecho de que quienes van adelantados están esperando, el enviador se preocupa de enviar la respuesta para que llegue en el momento apropiado para que todos se mantengan sincronizados.

Se muestra el procedimiento de mensajería en el código 4.9 (por parte del recibidor) y 4.10 (por parte del enviador). Es difícil ver las consecuencias solo con esto, por lo que es explicado a continuación con un ejemplo práctico en el diagrama 4.11.

Los cuadros representan el estado simplificado de cada nodo. En cada uno hay un número que señala el último frame procesado. Los tres nodos están desincronizados al inicio. El nodo (o jugador) 1 recibe una entrada nueva para ser procesada teniendo como último frame procesado el nº 15. Teniendo un retardo estimado de dos frames (+2) y que se desea que esta entrada nueva sea procesada *después* del último frame procesado (+1) se envía para que sea procesada en el frame 18 al resto. El nodo 3 lo recibe teniendo el nº 17 como último frame procesado, por lo que responde afirmativamente que puede inducir la entrada nueva y ésta es encolada para ser procesada localmente.

Sin embargo, el segundo nodo no puede inducir esta nueva entrada puesto que ya ha sido procesada, pero ofrece inducirlo en su siguiente frame a procesar. Así mismo, el nodo 2 encola una entrada de rebote para un proceso local. El primer nodo recopila las respuestas y encola las confirmaciones para ser enviadas en un lapso en base al tiempo que lleven de adelanto para que queden sincronizados. Se estima que se debe enviar inmediatamente la respuesta al nodo 3 ya que originalmente el retardo está pensado para que todos alcancen a responder y a recibir la confirmación.

Aunque el nodo 3 pudo inducir la entrada, no estaba sincronizado con el nodo 1, por lo que hay un tiempo de espera para la confirmación de la entrada (marcado en gris) esperando a la confirmación de la entrada nº 18. Esta espera hace que este nodo se resincronice con el primero. La confirmación recibida anula la entrada nueva que estaba siendo esperada (nº 18), por lo que el proceso de entrada continúa normalmente. La entrada nueva es procesada luego en el frame nº 20.

Finalmente, el primer nodo envía la confirmación a la entrada nueva al segundo nodo. Ésta es procesada inmediatamente y quedan los tres nodos sincronizados.

Esta transacción se puede ejecutar en paralelo con otras, ya que cada nodo se resincroniza

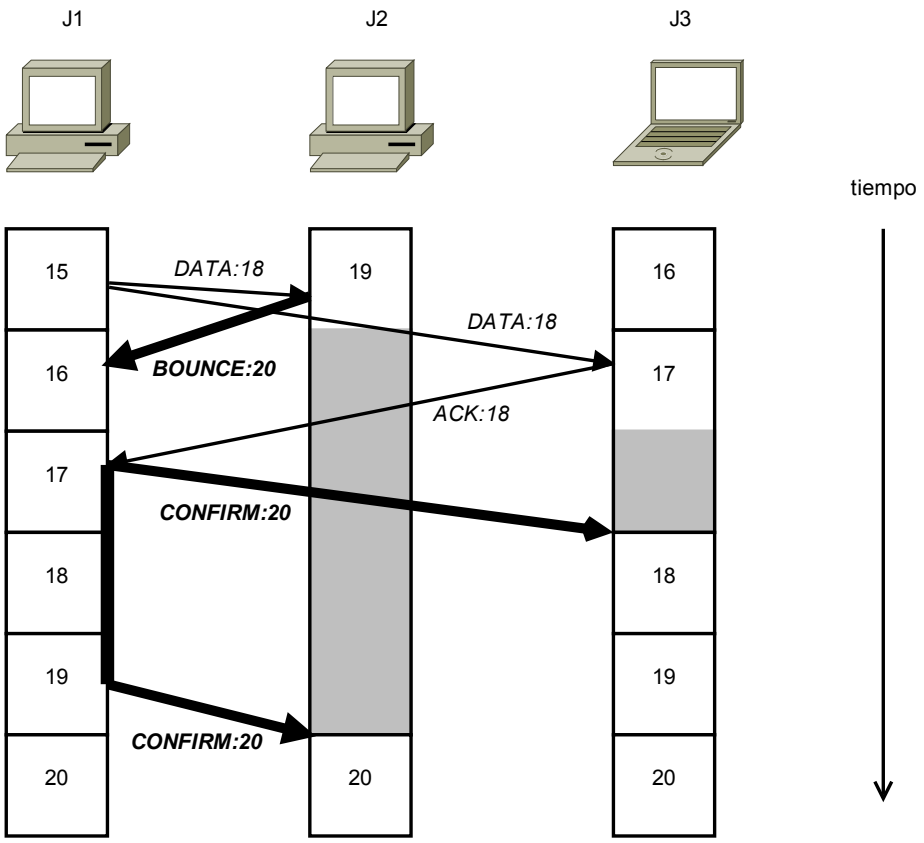


Figura 4.11: Esquema de "rebote"

con quien tenga una cuenta menor de frames procesados. Si llegasen a haber casos aislados de mensajes que lleguen con retraso mayor al esperado, se resincronizaría con los mensajes que siguen.

Todos estos cambios implicaron una revisión y modificación masiva del código, principalmente por parte del manejo de entradas. No solo había que preferir las entradas no rebotadas sobre las rebotadas, sino que también comparar las rebotadas entre sí. Para esto, para cada entrada nueva, se mantiene el frame original. Inicialmente, todo mensaje que incluyera un número de frame lo traía empaquetado en un byte. Se implementó una función que comparara sin *overflow*.[7] Mientras se implementó el manejo de entradas de rebotes se estimó que podría complicarse al comparar frames originales, debido a que no sería trivial la comparación porque podría no tenerse un contexto claro.

Para resolver, se ocuparon tres bytes de un entero. Asumiendo partidas de cuatro horas (864.000 frames) tres bytes serían suficientes. No se ocuparon todos los bytes a modo de

[7]La comparación con bytes admite un contexto. Por ejemplo, se sabría que si llega un mensaje considerado para el frame 4 estando en el 253, se diría que el mensaje alcanzó a llegar.

mantener el tamaño de paquetes en mínimo. Con esto se sacrificaría el posible uso de la biblioteca entre distintas plataformas al mismo tiempo, debido a la distinta representación de los bytes en bits o *endianess*.

Las pruebas mostraron una buena aprobación por parte de los jugadores que participaron. Éstas fueron hechas en una red cerrada con un tráfico cargado (con otro computador haciendo descargas de la web) y tráfico liviano (solo los computadores participantes conectados entre sí). En ambas instancias hubo pequeños retrasos de mensajes y, consecuentemente, ajustes por parte de los emuladores a la espera de estos mensajes. Se apreció una buena jugabilidad con hasta cuatro jugadores simultáneos.

Pruebas hechas por la web fueron simuladas aumentando el retardo entre los nodos. Si bien el juego respondió correctamente a las entradas, con el aumento de retardo se volvía "muy complicado para jugar" debido a que el jugador tiene que compensar el inmenso retardo entre entrada y reacción.[8] Esto fue notado ocupando sobre 300ms de retardo.

[8]Esta opinión fue dada y explicada por los jugadores con quienes se hicieron las pruebas

Capítulo 5

Detalles implementación final

5.1. Componentes

El código se divide en tres partes principales. Ellas cuentan con una alta cohesión por lo que un buen número de variables y funciones en cada una de las clases van orientadas a la comunicación entre ellas. Por esto mismo el término "componentes" no estaría bien aplicado. La biblioteca implementada fue llamada *p2pSync*.

El código de estas tres partes importantes está en el apéndice.

5.1.1. Interfaz con emulador

Esta parte se encarga de recibir la entrada del jugador y gestionarla. Específicamente:

- Gestiona cuándo un frame de la entrada del jugador debe ser replicada en el resto de los nodos.
- Controla y gestiona parcialmente el avance del emulador. Por ahora, solo en los casos de rebote se gestiona el avance o pausa.
- Gestiona el almacenamiento de entradas nuevas y los casos que se relacionen: detener la emulación porque el número de frames es muy avanzado con respecto a uno o más nodos;
- Ofrece herramientas para guardar y cargar estados de máquina por medio de peticiones encoladas. El encolamiento está pensado para que la acción alcance a ser avisada en todos los nodos, cosa de que todos puedan reproducirla.

- Funciona en base a los retardos calculados y mensajes recibidos de la parte de mensajería y a los datos almacenados en la parte de réplicas.
- Carga datos para realizar conexiones e iniciar la partida.

En particular, la implementación hecha lee los datos de la partida desde un archivo.

La comunicación con `fba` es hecha por medio de cuatro funciones y el constructor. En el último, se señalan de qué archivo se cargan los datos. Dos de las funciones son de chat (envío y recepción de mensaje), pero esta característica no fue implementada.

Las últimas dos funciones son `gameCallBack` y `sync`. La primera, `gameCallback` es ofrecida por `fba` y usada por la biblioteca para pedir la carga de máquina y ROM y comenzar la emulación. La segunda, `p2pSync::sync(...)`, es llamada desde el thread de emulación de `fba` y es acá donde está la mayor interacción con el emulador. Es aquí donde el emulador cede la entrada del jugador para ser gestionada. De hecho, hay un modo de reproducción de partida en *Kaillera* que lee la entrada desde un archivo. También podemos detener la emulación puesto a que los emuladores esperan a que la función `sync` retorne.

La clase `p2pSync` implementa a la clase abstracta `CalleeBack` para hacer *callbacks* desde otra clase.[1] La clase `FileP2pSync` implementa a la clase abstracta `p2pSync` para cargar los datos de la partida y conexiones por medio de la lectura de un archivo.

También se pueden proveer, por parte de los usuarios de esta biblioteca, *callbacks* a funciones de:

- Guardado de estado
- Carga de estado
- Pausa de juego
- Avance de juego (mientras se esté en pausa)

Las funciones de pausa y avance de juego están pensadas para permitir una jugabilididad más agradable para el usuario final. Normalmente esto está ausente y, consecuentemente, la espera de un mensaje se manifiesta como un mal funcionamiento del emulador.

Las funciones de carga y guardado de estado son discutidas en 5.2

[1] Esta es la manera correcta de hacerlo con clases en C++. Debido a la cantidad grande de funciones, es mejor pasar la clase que cada una de las funciones

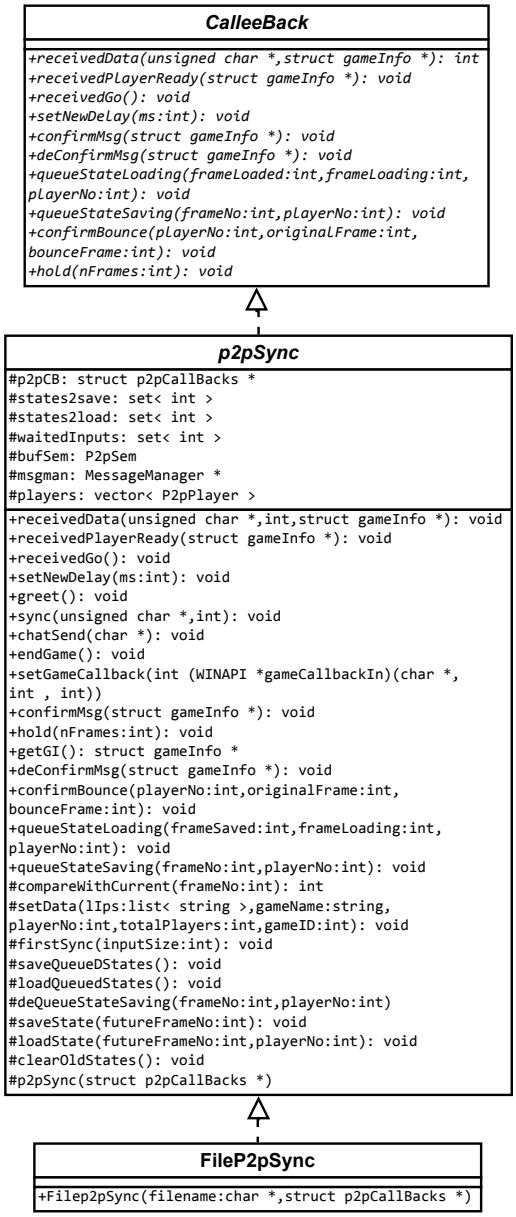


Figura 5.1: Diagrama de clase: Parte de interfaz

5.1.2. Mensajería

La parte de mensajería se encarga de gestionar la estimación del retardo de los mensajes y de comunicar y gestionar las acciones e informaciones entre nodos. Todas las funciones de mensajería son no-bloqueantes.

- Transmitir y recibir datos
- Avisar y recibir que el emulador en un nodo ha cargado el hardware y que está listo para correr.
- Avisar recibir el inicio de la partida.
- Estimar el retardo de los mensajes. Se calculan independientemente para cada nodo, pero se entrega a la parte de interfaz el máximo retardo.
- Realizar las conexiones iniciales entre los nodos
- Entregar a la parte de interfaz datos nuevos o avisos sobre eventos que vayan llegando: conexiones iniciales realizadas; avance de emulación; confirmación de todos los nodos de un cambio local de entrada; rebote de un cambio de entrada.
- Decidir el tiempo de las pausas en la emulación

Para portar el código a otra plataforma se deben proveer (implementar) las funciones de transmisión y recepción de datos, almacenamiento de información de nodos y una función para enviar datos con un retardo definido. Se optó que este almacenamiento nodos en memoria fuese específico a la plataforma usada, debido a que pueden haber datos extra que deben ser almacenados.[2] Para esto, se debe también extender la clase de información de nodos `P2pNode`. Las direcciones son manejadas como strings, así que se debe implícitamente implementar funciones para transformar las direcciones nativas[3] a strings y viceversa. En el caso de la implementación hecha, se almacena también los mensajes que deben ser enviados con retardo en la información de cada nodo.

El envío retardado es necesario para poder controlar bien las pausas de los emuladores. Esto fue discutido en 4.6 y en 4.2.

[2]Por ejemplo, en la implementación hecha no se almacena la conexión a y de un nodo sino que sólo su dirección

[3]En este caso `sockaddr_in`

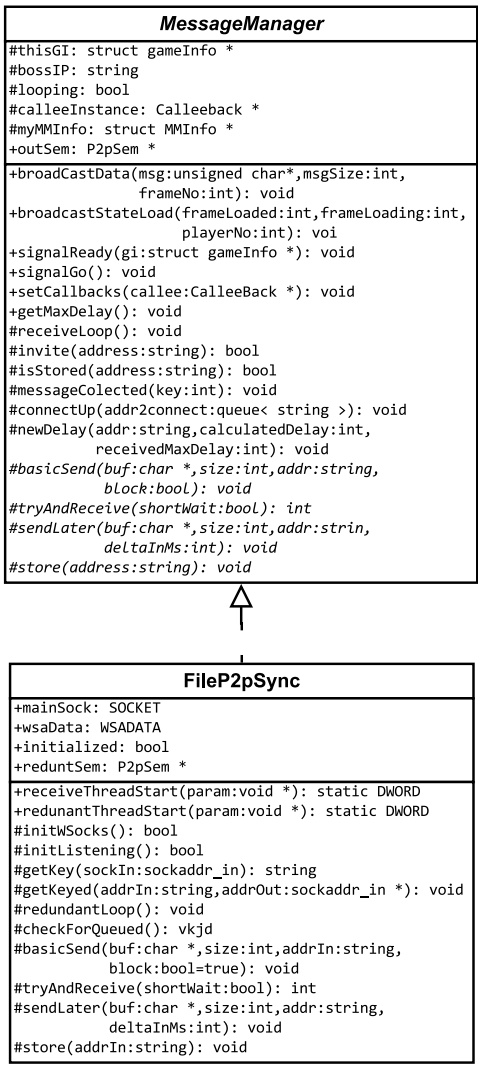


Figura 5.2: Diagrama de clase: Parte de interfaz

5.1.3. Réplicas

Esta parte administra las réplicas locales del resto de los nodos. Su objetivo principal es abstraer el manejo de entradas y de caches. Se le llamó `P2pPlayer` debido a que se ven como un jugador a quien se le pide la entrada. Para el acceso concurrente es manejado afuera por medio de `p2pSync` debido a que se deben hacer gestiones (que involucran otro manejo de concurrencia) con respecto al estado de la partida o a los mensajes que se hayan recibido. Esta parte ofrece:

- Obtener la entrada de un jugador para un cierto frame
- Verificar si la entrada de un cierto frame está disponible. En los casos de que haya una nueva entrada o una entrada rebotada hasta ese frame que no hayan sido confirmadas se dice que no está disponible.
- Confirmar una entrada nueva o rebotada
- Rebotar una entrada

Para obtener la entrada, primero se verifica si hay una entrada nueva o rebotada en el frame señalado. En caso de que haya más de una entrada rebotada, se comparan sus frames originales. En caso de que en un cierto frame no hubiese una entrada nueva o rebotada, se retorna los mismos datos de la última procesada. Para ello se mantiene un caché con el último frame **original** procesado. Debe ser el original para un manejo correcto de las entradas rebotadas.

5.2. Discusión

El código fue hecho para correr en Windows, pero está pensado para que sea reimplementado en otra plataforma. El único requerimiento es que esta debe proveer la Standard Temple Library de C++, aunque no se alienta a mezclar distintos tipos tipos de plataformas o procesadores entre los nodos en una partida debido a que su comportamiento podría ser impredecible. Esto es debido a la distinta representación del tipo `int` en bits. En cada mensaje va empaquetado el número de frame y este podría ser leído equivocadamente.

En los códigos 4.6 y 4.7 se aprecian funciones específicas a la API de Windows. Esto era a modo de referencia, puesto a que estas están abstraidas en la clase `P2pSem`.

Para la reimplementación en otra plataforma solo se deben instanciar las clases `p2pSync` y `MessageManager` y reimplementar `P2pSem`.[4]. Para `p2pSync` se debe proveer el método para conseguir los datos de la partida, mientras que para `MessageManager` las funciones para enviar, recibir y enviar con retraso. En el trabajo hecho, se leen los datos de la partida de un archivo, pero se puede usar otro proceso para obtenerlos como, por ejemplo, una conexión ad-hoc bluetooth, aunque se deben mantener los mismos pares $(jugador, nodo)$ en todos los nodos.

El código permite que sea extendido, pero con una mediana complicación. Este percance ocurrió porque, al discutir el inicio del trabajo, se concluyó que era mejor partir con una aplicación tan básica como sea posible, lo que fue dificultando la modularización y el orden del código posteriormente. Aunque se tenía contemplado un pre-diseño pensado para problemas comunes encontrados a la hora de hacer juegos distribuidos o sistemas de respuesta mínima, no se tenían en cuenta los problemas prácticos que eventualmente surgirían. El desarrollo fue haciéndose en base a esta primera implementación rápida y principalmente con parches.

De no ser por una dedicación exhaustiva a la modularización y orden del código, probablemente habría resultado otra biblioteca estática como *Kaillera*.

5.3. Trabajo a futuro

Entre las cosas que se puedan mejorar o hacer, las más importante son la resistencia a pérdida de paquetes y el retardo asociado a las etapas del *two-phase commit*.

Si bien es mejor no tener una fase de confirmación para las entradas nuevas, esto requiere que estén sincronizados los nodos. Las pruebas mostraron que no solamente la mayoría de las veces esto **no** es cierto, sino también que los nodos son reacios a mantener una resincronización. Se plantea que para un trabajo posterior se ocupe este mismo procedimiento de respuesta-confirmación pero con una mensajería de **control**. Esto mantendría una mejor

[4]Esta clase tiene cuatro funciones

medida para la latencia entre las conexiones y una mejor gestión del retardo y de las pausas en el emulador.

En caso de pérdida de paquetes, podría combinarse con la ya implementada funcionalidad para salvar y cargar estados. En cada corte consistente se almacenaría en memoria el estado de una máquina. Así cualquier nodo puede invocar el cargado de ese estado a modo de *rollback* de manera sincronizada.

A un más corto plazo, se puede mantener el protocolo actual y pedir el re-envío de una confirmación o respuesta. En el caso de que se pierda un mensaje de solicitud de nueva entrada (local), se debe tener un tiempo de espera máximo para recibir las respuestas del resto de los nodos. Pasado este tiempo, se rebotaría la entrada y re-enviaría las nuevas solicitudes.

Estos *timeouts* se reflejarían con una pausa leve en el emulador, por lo que esta propuesta sería viable en una red cerrada, pero no necesariamente sobre la web.

Otra mejora sería el implementar una mejor interfaz para iniciar la partida de los jugadores. Se propone un cliente externo al emulador y a la biblioteca que transe con un servidor, éste genere y envíe los archivos a cada nodo y que la aplicación local cargue el emulador señalando el archivo con los datos.

Capítulo 6

Conclusiones

Se diseñó, desarrolló, una biblioteca de código abierto[1] que abstrae al usuario para la sincronización del estado de una máquina emulada entre varios jugadores, manteniendo un balance satisfactorio entre tiempo de respuesta y consistencia. En contraste con *Kaillera* y *Kaillera p2p*, el código resultante es extensible y puede ser modificado por programadores de emuladores a la medida. Además se plantean maneras de abordar problemas en otros contextos usando como base el trabajo presentado.

También se logró, en específico:

- Diseño y desarrollo de un protocolo de mensajes que permita el juego en línea que sea satisfactoria para un usuario final.
- Ordenamiento de mensajes y eventos que mantuviesen una imparcialidad para y entre los jugadores. Tanto como en las acciones hechas por los mismos como en la actualización de estados.
- Una solución escalable que permite jugar entre cuatro jugadores.
- Diseño de la aplicación para que sea extensible.
- Modificación de un emulador para probar la funcionalidad de la biblioteca.

En cuanto al número de jugadores, se tendría que reimplementar otro jugador para adaptar a la API de la biblioteca construida para poder probar más de cuatro clientes, debido a que ninguno de los juegos soportados por `fba` son de más de cuatro jugadores. Si bien, hay una prueba de entradas, esta no certifica que su jugabilidad sea satisfactoria.

[1] código disponible en `http://anakena.dcc.uchile.c/~felema/fbaSync.tgz`

Aparte de esto, se puede concluir que se logró el objetivo general y una realización satisfactoria de los objetivos específicos.

Como trabajo a futuro se plantea lo siguiente:

- Modificar el protocolo de mensajería, ocupando el protocolo de *two-phase commit* para control.
- Combinar el punto anterior con el guardado y salvado de estados de máquina para *rollbacks*
- Una mejor interfaz para cargar o recibir datos de la partida.

Referencias

[1] R. D. Atkinson, D. K. Correa, and J. A. Hedlund. Explaining international broadband leadership. Technical report, The Information Technology and Innovation Foundation, may 2008.

[2] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament 2003. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 144–151, New York, NY, USA, 2004. ACM Press.

[3] A. R. Bharambe, J. Pang, and S. Seshan. Colyseus: A distributed architecture for online multiplayer games. In *NSDI*. USENIX, 2006.

[4] K. Forums. "kaillera should go peer to peer". `http://www.kaillera.com/forums/viewtopic.php?f=2&t=3444`. Retrieved on August 10th, 2009.

[5] L. Gautier and C. Diot. Mimaze, a multiuser game on the internet.

[6] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[7] Y. Lin, K. Guo, and S. Paul. Sync-ms: Synchronized messaging service for real-time multi-player distributed games, 2002.

[8] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[9] A. Mkhikian. Sf2 engine randomness rundown. `http://sonichurricane.com/articles/sf2randomness.html`. Retrieved on August 10th, 2009.

[10] J. Pang, F. Uyeda, and J. R. Lorch. 6th international workshop on peer-to-peer systems. In *Scaling Peer-to-Peer Games in Low-Bandwidth Environments*, february 2007.

[11] M. Roberts. A study of the massively multiplayer online business model within the interactive entertainment industry, december 2005.

[12] A. S. Tanenbaum and M. van Steen. *Distributed Systems. Principles and Paradigms.* Prentice Hall International, 2nd rev. ed. edition, October 2006.

Apéndices

A . Código clase p2pSync

```
#include "p2pSync.h"
#include "WinMessageManager.h"

#include <assert.h>

#ifdef __MINGW32__
  #define DPRINT if(FBA_DEBUG)bprintf
  extern int (__cdecl *bprintf) (int nStatus, TCHAR* szFormat, ...);
#else
  #define DPRINT if(FBA_DEBUG)bprintfPS
  void bprintfPS(int nStatus, TCHAR* szFormat, ...){ }
#endif

using namespace std;


#define INIT_TH 1 //coordinate after INIT_TH first sync
#define MIN_DELAY 3
#define MAX_STATES 3 //max states per player slot


int (WINAPI *gameCallback)(char *game, int player, int numplayers);

extern int clk2ms(clock_t in);


//is left equal or greater than right?

int msKey(int frameNo, int playerNo){
  return frameNo*100 + playerNo;
}

int msKeyedFrame(int key){
  return key/100;

}
int msKeyedPlayer(int key){
  return key%10;
}


```

```
42  p2pSync::p2pSync( struct p2pCallBacks* pCB){
43
44
45    myGI = (struct gameInfo *)malloc( sizeof(struct gameInfo));
46
47    p2pCB = (struct p2pCallBacks *)malloc( sizeof(struct p2pCallBacks) );
48    p2pCB->stateSavePtr = pCB->stateSavePtr;
49    p2pCB->stateLoadPtr = pCB->stateLoadPtr;
50    p2pCB->pauseGamePtr = pCB->pauseGamePtr;
51    p2pCB->unPauseGamePtr = pCB->unPauseGamePtr;
52    if(   !p2pCB->pauseGamePtr ||
53          !p2pCB->unPauseGamePtr)
54      DPRINT(0,"Wopa... esto no pasa!!!\n");
55    if(   !p2pCB->stateSavePtr ||
56          !p2pCB->stateLoadPtr){
57      DPRINT(0,"Saving/loading disabled\n");
58      p2pCB->stateSavePtr = NULL;
59      p2pCB->stateLoadPtr = NULL;
60    }
61    bufSem = new P2pSem();
62
63    //players = (P2PSplayer *)malloc(sizeof(P2PSplayer)*MAX_PLAYERS);
64    for(int i=0;i<MAX_PLAYERS;i++)
65      players.push_back( new P2pPlayer( i+1, myGI) );
66
67
68
69    //lastFirstSync = (clock_t)-1;
70
71  }
72  void p2pSync::setData(list< string > lIpsIn,
73                  string gameNameIn, int playerNoIn,
74                  int totalPlayersIn, int gameIDIn){
75
76    myGI->frameNo        = 0; //if you change this, you have to change it in
          P2pPlayer
77    myGI->playerNo      = playerNoIn;
78    myGI->gameID        = gameIDIn;
79
80    myGI->firstSynced   = false;
81    myGI->totalPlayers  = totalPlayersIn;
82    myGI->inputSize     = -1;
83    myGI->greenLighted = false;
84    myGI->syncDelay  = 50;
85    myGI->have2hold = 0;
86    myGI->lastSync = clock();
87    gameName = gameNameIn;
88
89
90    msgman = new WinMessageManager(lIpsIn,this);
91
92
93  }
94  int p2pSync::compareWithCurrent(int frameNo){
95    return frameNo - myGI->frameNo;
96  }
```

```
97
98  int p2pSync::receivedData( unsigned char *inBuf, int inSize, struct
        gameInfo *inGameInfo){
99
100   DPRINT(0,"p2p::rD, sync:%dp%d while @%u\n",
101       inGameInfo->frameNo, inGameInfo->playerNo, myGI->frameNo);
102
103   bufSem->enterCritical();
104   int inFrameNo = inGameInfo->frameNo;
105   int hisPlayerIndex=inGameInfo->playerNo-1;
106
107
108
109   if( !(myGI->firstSynced) ){
110     DPRINT(0,"...too bad we haven't started syncing\n");
111     bufSem->exitCritical();
112     return -1;
113   }
114
115
116   //if it's equal it's okay
117   //myGI->frameNo points to the frame to be processed, not being proc'd
118
119   assert( myGI->inputSize == inSize);
120   //assert( inFrameNo < (unsigned char)UCHAR_IDXS);
121
122   int frameDiff = compareWithCurrent( inFrameNo );
123
124
125
126   //Everything OK, copying to cache
127   players[hisPlayerIndex]->setInput( (unsigned char *)inBuf, inSize,
          inFrameNo);
128
129   if( frameDiff <= 0 ){
130
131     int bounceFrameNo = myGI->frameNo+1;//process it on next "process",
          please
132
133     //offer new bounce
134     players[hisPlayerIndex]->bounceInput( inFrameNo, bounceFrameNo);
135
136     //trick for stateSaving
137     inFrameNo = bounceFrameNo;
138
139     //return bounce frame
140     inGameInfo->frameNo = bounceFrameNo;
141   }
142
143
144   queueStateSaving( inFrameNo, inGameInfo->playerNo);
145
146   bufSem->exitCritical();
147
148   return frameDiff;
149 }
```

```
150 void p2pSync::confirmBounce(int playerNo, int originalFrame, int
      bounceFrame){
151   DPRINT(0,"ps: confBouncing p%d's %d to %d\n", playerNo, originalFrame,
        bounceFrame);
152   //correct input if necessary
153   bufSem->enterCritical();
154   players[playerNo -1]->confirmBounce(originalFrame, bounceFrame);
155   int key = msKey(originalFrame, playerNo);
156   if( waitedInputs.size() > 0){//somebody was (probably) waiting for this?
157     waitedInputs.erase( key );
158     if( waitedInputs.size() == 0 )//we've just inputed the last waited
159       bufSem->greenLight();
160   }
161   bufSem->exitCritical();
162
163 }
164 void p2pSync::receivedPlayerReady(struct gameInfo *inGameInfo){
165   int hisPlayerI = inGameInfo->playerNo-1;
166
167   players[hisPlayerI]->ready = true;
168   DPRINT(0,"Player %d says he's ready @%u\n",
169       inGameInfo->playerNo, clock() );
170
171
172
173   if(myGI->playerNo != 1)
174     DPRINT(0,"How come I recvd player ready and I'm not boss?\n");
175
176   bool allReady=true;
177   for(int i=0;i<myGI->totalPlayers;i++)
178     if( !(players[i]->ready) ) //if one is not ready
179       allReady=false; //none of us is
180   if(allReady){
181     DPRINT(0,"\nGO!(%u)\n", clock() );
182     msgman->signalGo();
183     bufSem->greenLight();
184   }
185 }
186 void p2pSync::receivedGo(){
187
188   myGI->greenLighted = true;
189   bufSem->greenLight();
190
191 }
192
193
194 p2pSync::~p2pSync ()
195 {
196   DPRINT(0,"deleting p2pSync Object\n");
197   delete msgman;
198   delete bufSem;
199 }
200
201 void p2pSync::greet(){
202   msgman->greet();//send and receive greet
203 }
```

```
204
205 void p2pSync::chatSend(char *msg){
206
207 }
208 void p2pSync::sync(unsigned char *data, int size){
209   int pI = myGI->playerNo -1; //player index
210
211
212
213
214   //this section is to give the emu some 'breathing' at drawing window
215
216
217   if(!myGI->firstSynced){
218     if(myGI->frameNo < INIT_TH){
219       myGI->frameNo++;
220       return;
221     }
222
223
224     firstSync(size);
225   }
226
227   if(myGI->have2hold){
228     clock_t timeBefore = clock();
229     int frames = myGI->have2hold/MS_PER_SYNC;
230     if(myGI->have2hold%MS_PER_SYNC)
231       frames++;
232
233     //p2pCB->pauseGamePtr();
234     Sleep(frames*MS_PER_SYNC);
235     //p2pCB->unPauseGamePtr();
236     DPRINT(0,"Holded for %ums\n",
237       clk2ms(clock()-timeBefore));
238     myGI->have2hold = 0;
239   }
240
241   //assert(myGI->inputSize == size);
242
243   //translate his input into our slot (paste it in the future)
244   int futureFrameNo = myGI->frameNo +1 //frameNo == "last processed frame
        ", s +1 = "frame in process"
245                                 + myGI->syncDelay;
246   bool shouldBC = players[pI]->setInput( data, size, futureFrameNo);
247
248   //assert(futureFrameNo < UCHAR_IDXS);
249
250
251   //assert(size <= BUF_SIZE);
252
253   if( shouldBC ){
254     queueStateSaving( futureFrameNo, myGI->playerNo );
255   }
256   saveQueuedStates();
257   loadQueuedStates();
258   clearOldStates();
```

```
259     //let's pack only our stuff for broadcast
260
261
262     //broadcast this input
263     if( shouldBC ){
264       clock_t bcastcp = clock();
265       DPRINT(0,"Broadcasting new input from %d to %d\n", myGI->frameNo,
              futureFrameNo);
266       msgman->broadcastData( data ,size, futureFrameNo);
267       DPRINT(0,"Broadcasted took %dms\n",clk2ms(clock()-bcastcp) );
268     }
269
270     //check if all slots are ready
271     bool allReady = false;
272     int nextFrame = myGI->frameNo +1; //candidate for processing is the next
           one
273
274     bufSem->enterCritical();
275
276     while( !allReady ){
277       int originalFrameNotReady = -1;
278       allReady = true;
279       for(int i=0; i< myGI->totalPlayers; i++){
280         //bool printedOut = false;
281         originalFrameNotReady = players[i]->inputReady(nextFrame);
282         if( originalFrameNotReady >= 0){
283           waitedInputs.insert( msKey(originalFrameNotReady, i+1) );
284           DPRINT(0,"Input %up%d isn't (de)confirmed\n",
285             nextFrame, i+1);
286           allReady = false;
287         }
288       }
289       if( !allReady ){
290         DPRINT(0,"Waiting unconfirmed inputs on %d while @%d\n",
291                 nextFrame, myGI->frameNo);
292         bufSem->exitCritical();
293         bufSem->waitForGreenLight(16);
294         bufSem->enterCritical();
295
296         //originalFrameNotReady = players[i]->inputReady(myGI->frameNo);
297       }
298
299     }
300
301     // all inputs ready... let's copy our slots to his
302     int di = 0;
303     unsigned char *input;
304     input = new unsigned char[size];
305     for(int i=0;i < myGI->totalPlayers; i++){
306       if( players[i]->getInput( (unsigned char *)input, nextFrame ) <= 0)
307         DPRINT(0,"Read -1 from getInput at %u\n", nextFrame);
308       for(int j=0;j<size;j++,di++)
309         data[di] = input[j];
310
311     }
312     //break point
```

```
313     myGI->frameNo++;//frame has been processed... move on
314
315     delete[] input;
316
317     //getReady for next loop
318
319
320     //the following can be parallel, since we've used this frame
321     bufSem->exitCritical();
322
323     //this frame is clear
324     for(int i=0;i<myGI->totalPlayers;i++){
325       players[i]->clearInput( myGI->frameNo ); // "last processed frame is
             clear"
326     }
327
328
329   }
330
331   void p2pSync::setGameCallback(int (WINAPI *gameCallbackIn)(char *game, int
          player, int numplayers)){
332
333     gameCallback = gameCallbackIn;
334     char gameNameC_STR[257];
335     strcpy(gameNameC_STR,gameName.c_str());
336     DPRINT(0,"p2p: Running \"%s\", I'm player %d/%d\n", gameNameC_STR,
337         myGI->playerNo, myGI->totalPlayers);
338     gameCallback(gameNameC_STR, myGI->playerNo, myGI->totalPlayers);
339     DPRINT(0,"p2p: Callback ended\n");
340   }
341   void p2pSync::endGame(){
342
343     DPRINT(0, "p2p: game should've ended\n");
344   }
345
346   void p2pSync::firstSync(int size){
347
348     myGI->inputSize = size;
349     int pI = myGI->playerNo -1;
350
351     players[pI]->ready = true; // I'm ready
352
353     if( myGI->playerNo == 1){
354       DPRINT(0,"Waiting for other players...");
355       bufSem->waitForGreenLight(-1);
356       msgman->signalGo();
357     } else {
358       //loop into signaling until we receive go
359       while( bufSem->waitForGreenLight(2) && !myGI->greenLighted)// timeout
             and not green lighted in between
360         msgman->signalReady(myGI);
361
362     }
363
364     myGI->firstSynced = true;
365     DPRINT(0,"Exited first sync @%u (sync:%u)\n", clock(), myGI->frameNo );
```

```
366
367
368
369   }
370   void p2pSync::setNewDelay(int ms){
371
372     if(ms < 0)
373       return; //ignore
374
375     bufSem->enterCritical();
376     myGI->syncDelay  = ms/MS_PER_SYNC;
377
378     if( ms%MS_PER_SYNC )
379       myGI->syncDelay++;
380
381     if(myGI->syncDelay < MIN_DELAY)
382       myGI->syncDelay = MIN_DELAY; //testing found out that it was 2
383     bufSem->exitCritical();
384     DPRINT(0,"New sync delay:%d frames\n", myGI->syncDelay);
385
386
387   }
388   void p2pSync::deConfirmMsg(struct gameInfo *gi){
389     DPRINT(0,"Deconfirming %dp%d\n", gi->frameNo, gi->playerNo);
390     bufSem->enterCritical();
391
392     players[ gi-> playerNo -1 ]->deConfirmMsg( gi->frameNo );
393     deQueueStateSaving( gi->frameNo, gi->playerNo );
394
395     if( waitedInputs.size() > 0){//somebody was waiting for this?
396         waitedInputs.erase( msKey( gi->frameNo, gi->playerNo ) );
397         if( waitedInputs.size() == 0 )//we've just inputed the last waited
398           bufSem->greenLight();
399     }
400     bufSem->exitCritical();
401     DPRINT(0,"Deconfirmed %up%d\n", gi->frameNo, gi->playerNo);
402   }
403
404   void p2pSync::confirmMsg( struct gameInfo *gi ){
405
406     DPRINT(0,"p2p: confirmed %u while @%u\n", gi->frameNo, myGI->frameNo );
407     int key = msKey( gi->frameNo, gi->playerNo );
408     bufSem->enterCritical();
409     if( gi->frameNo > myGI->frameNo
410         || waitedInputs.find( key ) != waitedInputs.end() ){
411       players[ gi-> playerNo -1 ]->confirmMsg( gi->frameNo );
412
413       if( waitedInputs.size() > 0){//somebody was waiting for this?
414         waitedInputs.erase( key );
415         if( waitedInputs.size() == 0 )//we've just inputed the last waited
416           bufSem->greenLight();
417       }
418     }
419     else {
420       //broadcast state load
421       unsigned char syncLoading = myGI->frameNo + myGI->syncDelay;
```

```
422       DPRINT(0,"Queueing state loading from %u on %u\n", gi->frameNo, gi->
              frameNo + myGI->syncDelay );
423       msgman->broadcastStateLoad( gi->frameNo, syncLoading, gi->playerNo);
424       queueStateLoading( gi->frameNo, syncLoading, gi->playerNo);
425
426
427     }
428     bufSem->exitCritical();
429
430   }
431
432   void p2pSync::saveState(int futureFrameNo, int guiltyPlayerNo){
433     if( !p2pCB->stateSavePtr ) //disabled
434       return;
435     int key = msKey( futureFrameNo, guiltyPlayerNo);
436
437     mStates[key] = (struct machineState *)malloc(sizeof(machineState));
438     mStates[key]->data = NULL;
439     mStates[key]->size = 0;
440     mStates[key]->frameNo = futureFrameNo;
441     mStates[key]->guiltyPlayerNo = guiltyPlayerNo;
442     mStates[key]->time2kill = clock()+ (clock_t)4*CLOCKS_PER_SEC;
443
444     p2pCB->stateSavePtr( &(mStates[key]->data),&(mStates[key]->size) );
445     DPRINT(0,"Saved state to %u\n", futureFrameNo);
446     return;
447   }
448
449   void p2pSync::loadState(int frameNo, int playerNo){
450     if( !p2pCB->stateLoadPtr )
451       return;
452     int key = msKey( frameNo,playerNo);
453     if( mStates.find(key) == mStates.end() ){
454       DPRINT(0,"Can't load state %d\n", key);
455       return;
456     }
457
458
459     p2pCB->stateLoadPtr( mStates[key]->data, mStates[key]->size );
460     /*
461     free(mStates[key]->data);
462     free(mStates[key]); //struct machineState
463     mStates.erase(key);
464     */
465   }
466
467   void p2pSync::queueStateSaving(int frameNo, int playerNo){
468     states2save.insert( msKey(frameNo, playerNo) );
469   }
470   void p2pSync::deQueueStateSaving(int frameNo, int playerNo){
471     states2load.erase( msKey(frameNo, playerNo) );
472   }
473   void p2pSync::saveQueuedStates(){
474     if( states2save.empty() )
475       return;
476
```

```
477     set<int>::const_iterator iter = states2save.begin();
478     while(iter != states2save.end()){
479       if( msKeyedFrame(*iter) == myGI->frameNo ){
480         saveState( msKeyedFrame(*iter), msKeyedPlayer(*iter) );
481         states2save.erase( iter++ );
482       } else
483         ++iter;
484 
485     }
486   }
487   void p2pSync::queueStateLoading(int frameSaved, int frameLoading, int
        playerNo){
488     states2load[ frameLoading ] = msKey(frameSaved, playerNo);
489   }
490 
491   void p2pSync::loadQueuedStates(){
492     if( states2load.empty() )
493       return;
494 
495     map<int, int>::iterator result =
496       states2load.find(myGI->frameNo);
497     if( result != states2load.end() ){
498       int key = states2load[myGI->frameNo];
499       loadState( msKeyedFrame(key), msKeyedPlayer(key) );
500       states2load.erase(result);
501     }
502 
503 
504   }
505 
506 
507 
508   void p2pSync::clearOldStates(){
509     if( states2load.empty() )
510       return; //nothing to clear
511     map<int, struct machineState *>::iterator iter = mStates.begin();
512     while( iter != mStates.end() ){
513       machineState *curr = iter->second;
514       if( curr->time2kill >= clock() ){
515         //destroy
516         DPRINT(0,"Destroying mstate %d:%d\n", curr->frameNo, curr->
              guiltyPlayerNo);
517         free( curr->data );
518         free( curr ); //struct mState
519         mStates.erase( iter++ );// first ++, then erase
520       } else
521         iter++;
522 
523     }
524   }
525 
526   void p2pSync::hold(int ms){
527     myGI->have2hold = ms;
528   }
529 
530   struct gameInfo *p2pSync::getGI(){
```

```
531     return myGI;
532   }
```

B . Código clase MessageManager

```
1   #include "MessageManager.h"
2   #include <time.h>
3   #include <assert.h>
4   using namespace std;
5
6
7
8   #ifdef __MINGW32__
9     //#define DPRINT(format, ...) if(1)bprintf(0, format, __VA_ARGS__ )
10    //#define DPRINT(format, args...) if(1)bprintf(0, ## args)
11    #define DPRINT if(FBA_DEBUG)bprintf
12    extern int (__cdecl *bprintf) (int nStatus, TCHAR* szFormat, ...);
13  #else
14    #define DPRINT(format, ...)
15  #endif
16
17
18
19  #define DATA_MSGS 3 //msgs needed to correctly transmit data: data, ack,
        ok
20  #define N_ESTIMATIONS 8
21  #define N_STRESS_LOAD 1000 //per player
22  /* ping types*/
23  #define TIME_PING 0
24  #define STRESS_PING 1
25
26
27  clock_t lastGreenLight=-1;
28  /*
29   * About byte packing:
30   * 640890 = 9C77A
31   * buf[0] = 7A
32   * buf[1] = C7
33   * buf[2] = 09
34   *
35   *
36   **/
37  void packInBytes(int toPack, int nBytes, char *dest){
38    memcpy(dest, &toPack, nBytes);
39  }
40  int getPackedInBytes(char *buf, int nBytes){
41    int out = 0;
42    memcpy(&out, buf, nBytes);
43    return out;
44  }
45
46  int clk2ms(clock_t in){
47    return (in*1000)/CLOCKS_PER_SEC ;
```

```
48  }
49
50  /*used for inputs to be confirmed (sync, playerNo)*/
51  int makeSyncAndPlayerNoKey(int frameNo, int playerNo){
52    return frameNo*100 + playerNo;
53  }
54  int getFrameNo(int key){
55    return key/100;
56  }
57  int getPlayerNo(int key){
58    return key%100;
59  }
60
61
62  void MessageManager::receiveLoop(){
63    outSockSem->enterCritical();
64    myMMInfo->looping = true;
65    outSockSem->exitCritical();
66
67    while(myMMInfo->looping){
68      tryAndReceive(true);
69    }
70    DPRINT(0,"MM:Finished receivelooping\n");
71  }
72
73  void MessageManager::retrialLoop(){
74    outSockSem->enterCritical();
75    myMMInfo->looping = true;
76    outSockSem->exitCritical();
77
78    while(myMMInfo->looping){
79      retrial();
80    }
81    DPRINT(0,"MM:Finished retrialLooping\n");
82  }
83
84  MessageManager::MessageManager(CalleeBack *callback){
85    calleeInstance = callback;
86    thisGI = callback->getGI();
87    outSockSem = new P2pSem();
88    myMMInfo = (struct MMInfo *)malloc( sizeof(struct MMInfo) );
89    memset(myMMInfo, 0, sizeof(struct MMInfo));
90    myMMInfo->maxDelay = -1;
91    myMMInfo->loWeight = 6;
92    myMMInfo->hiWeight = 1;
93    myMMInfo->looping = false;
94    myMMInfo->sentMsgCount = 0;
95    myMMInfo->allConnected = false;
96
97    //headerSize = iData;
98    dataSize = -1;
99
100
101 }
102
```

```
103 void MessageManager::connectUp(queue<string> IPsToConnectTo,unsigned int
      targetConns){
104
105   DPRINT(0,"Connecting people\n");
106
107   //while(lAddr.size() < targetConns){
108   while(myNI.size() < targetConns){
109     tryAndReceive(false);
110     if(IPsToConnectTo.size() > 0){
111       string addr = IPsToConnectTo.front();
112       IPsToConnectTo.pop();
113       if( !isStored(addr) ){
114         invite(addr);
115         IPsToConnectTo.push(addr); //go to the back of queue, we'll see if
              you connected
116       }
117     }
118   }
119
120   /*preserve this order: delay, stress*/
121   getMaxDelay();
122
123   string home("home");
124   if(bossIP == home){
125     stressAll();
126   }
127
128
129 }
130
131
132
133
134
135
136
137 bool MessageManager::invite(string addr){
138
139
140   DPRINT(0,"Inviting %s\n", addr.c_str());
141
142   char buf[headerSize];
143
144   memset( buf, 0, sizeof(buf));
145
146
147   buf[iMsgType] = mtOpenConn;
148
149   basicSend(buf, headerSize, addr);
150
151   return true;
152 }
153
154 void MessageManager::greet(){
155   char buf[headerSize + 2*INT_DIG];
156   for(  map<string, P2pNodeInfo*>::iterator lIter = myNI.begin();
```

```
157           lIter != myNI.end();
158           lIter++){
159
160       string currAdr = lIter->first;
161
162       memset( buf, 0, sizeof(buf));
163
164       buf[iMsgType] = mtPing;
165       buf[iPingType] = TIME_PING;
166       sprintf(buf+iData,"%d", myMMInfo->maxDelay);
167
168       int msTime = clk2ms(clock() - myNI[currAdr]->lastPingTime);
169
170       if( msTime > myNI[currAdr]->greetSpace ){
171         myNI[currAdr]->lastPingTime = clock();
172         sprintf(buf+iData + INT_DIG,"%u",myNI[currAdr]->lastPingTime);
173         DPRINT(0,"Pinging mr. %s having maxDelay of %d (lastPingTime of %u)\
              n",
174                currAdr.c_str(),
175                myMMInfo->maxDelay,
176                myNI[currAdr]->lastPingTime);
177        basicSend(buf, headerSize + 2*INT_DIG, currAdr);
178       }
179     }
180
181   }
182
183
184
185   bool MessageManager::isStored(string strin){
186     map< string, P2pNodeInfo *>::iterator result = myNI.find(strin);
187     return result != myNI.end();
188
189   }
190   /*
191   void MessageManager::store(string addrIn){
192     DPRINT(0,"If using WINMM, YOU'RE NOT SEEING THIS!");
193     myNI[addrIn] = new P2pNodeInfo();
194
195   }
196   */
197
198   void MessageManager::remove(string in){
199     map<string, P2pNodeInfo *>::iterator i = myNI.find(in);
200     if(i != myNI.end()){
201       P2pNodeInfo *hisNI = i->second;
202       myNI.erase(i);
203       delete hisNI;
204     } else
205       DPRINT(0,"You shouldn't erase what doesn't exist\n");
206
207
208   }
209
210
211
```

```
212
213
214 void MessageManager::broadcastData( unsigned char *msg,int msgSize, int
       frameNoIn){
215   if(dataSize < 0)
216     dataSize = msgSize;
217
218   char *buf = new char[headerSize + dataSize];
219   memset(buf, 0, sizeof(buf));
220
221  //header
222   buf[iMsgType] = mtData;
223   buf[iGameID] = thisGI->gameID;
224
225   packInBytes(frameNoIn, B_FRAME_NO, buf + iFrameNo);
226
227   buf[iPlayerNo] = thisGI->playerNo;
228   buf[iDataSize] = msgSize;
229
230   //data
231   memcpy(buf+iData, msg, msgSize);
232
233
234
235
236
237
238   //create message
239   //when is this cleared?
240   //outside critical section because no one will access this key
241
242   int key = makeSyncAndPlayerNoKey( frameNoIn, thisGI->playerNo);
243   msgInProcess[key] = new P2pMessage( msg, msgSize, thisGI->totalPlayers,
         frameNoIn);
244   msgInProcess[key]->playerConfirm( thisGI->playerNo );
245   DPRINT(0,"Created message on key:%d, size:%d and num2conf:%d\n",key ,
         msgSize, thisGI->totalPlayers);
246   //let this broadcast be as atomic as it can... hence the sem is outside
         the for
247   clock_t timer = clock();
248   DPRINT(0,"%d:Entering critical\n", clk2ms(clock() - timer));
249   outSockSem->enterCritical();
250   DPRINT(0,"%d:Entered critical\n", clk2ms(clock() - timer));
251
252   for(  map< string, P2pNodeInfo *>::iterator lIterAddr = myNI.begin();
253           lIterAddr != myNI.end();
254           lIterAddr++){
255     string out = lIterAddr->first;
256     basicSend(buf, headerSize + dataSize, out, false);//non-blocking
257     lIterAddr->second->lastSentTime = clock();
258   }
259   DPRINT(0,"%d:Exiting critical\n", clk2ms(clock() - timer));
260   outSockSem->exitCritical();
261   DPRINT(0,"%d:Exited critical\n", clk2ms(clock() - timer));
262   //DPRINT(0,"...broadcast done\n");
263   delete[] buf;
```

```
264 }
265
266 void MessageManager::broadcastStateLoad( int frameSaved, int frameLoading,
        int playerNo){
267   char *buf = new char[headerSize + 2];
268   memset(buf, 0, sizeof(buf));
269
270   //header
271   buf[iMsgType] = mtLoadState;
272   buf[iGameID] = thisGI->gameID;
273
274   packInBytes(frameSaved, B_FRAME_NO, buf + iFrameNo);
275   buf[iPlayerNo] = thisGI->playerNo;
276
277   //data
278   buf[iData] = playerNo;
279   packInBytes(frameLoading, B_FRAME_NO, buf + iData+1);
280
281
282
283
284   outSockSem->enterCritical();
285   for(  map< string, P2pNodeInfo *>::iterator lIterAddr = myNI.begin();
286           lIterAddr != myNI.end();
287           lIterAddr++){
288     string out = lIterAddr->first;
289     basicSend(buf, headerSize + 2, out, false);//non-blocking
290
291   }
292
293   outSockSem->exitCritical();
294   delete[] buf;
295 }
296
297
298 void MessageManager::setCallbacks(CalleeBack *callee ){
299   calleeInstance = callee;
300
301   DPRINT(0,"New callback\n");
302
303 }
304
305
306 MessageManager::~MessageManager()
307 {
308
309   looping = false;
310   delete outSockSem;
311   for( map<string, P2pNodeInfo *>::iterator iAddr = myNI.begin();
312       iAddr != myNI.end();
313       iAddr++)
314     delete iAddr->second;
315   myNI.clear();
316   ackedFromStress.clear();
317   free(myMMInfo);
318
```

```
319 }
320
321 void MessageManager::receivedOpenConnection(string addrIn){
322   char szBuffer[headerSize];
323   memset( szBuffer, 0, sizeof(szBuffer) );
324   szBuffer[iMsgType] = mtHandShake;
325
326   basicSend(szBuffer, headerSize, addrIn);
327
328   store(addrIn);
329   DPRINT(0,"added mr. %s\n",addrIn.c_str() );
330
331
332 }
333
334 void MessageManager::receivedHandShake(string addrIn){
335
336   if(!isStored(addrIn)){
337     store(addrIn);
338     DPRINT(0,"\nOh, didn't had him... no worry, I've added him\n");
339   }
340 }
341
342
343 void MessageManager::signalReady(struct gameInfo *giIn){
344   if(dataSize <0)
345     dataSize = thisGI->inputSize;
346
347   if( (clock() - lastGreenLight) > (clock_t)1*CLOCKS_PER_SEC){
348     DPRINT(0,"I (p%d) am ready!@%u\n", giIn->playerNo,clock());
349     lastGreenLight = clock();
350   } else {
351     return; //hasn't been 1sec since last GreenLight
352   }
353
354   char buf[headerSize];
355   memset(buf, 0, sizeof(buf));
356
357   //header
358   buf[iMsgType] = mtReady;
359   buf[iGameID] = giIn->gameID;
360   buf[iPlayerNo] = giIn->playerNo;
361
362   string home("home");
363   if(bossIP == home){
364     return; //I don't have to signal myself... do I?
365   }
366
367   for( map<string, P2pNodeInfo *>::iterator lIterAddr = myNI.begin();
368       lIterAddr != myNI.end();
369       lIterAddr++){
370     string addr = lIterAddr->first;
371     if(addr == bossIP){
372       basicSend(buf, headerSize, addr);
373       break;
374     }
```

```
375     }
376
377
378
379
380   }
381
382   void MessageManager::signalGo(){
383     if(dataSize <0)
384       dataSize = thisGI->inputSize;
385
386     char buf[headerSize + 2];
387     memset(buf, 0, sizeof(buf));
388     unsigned int localMaxDelay = getLocalMaxDelay();
389     DPRINT(0,"Signaling go to everybody...\n");
390     //header
391     buf[iMsgType] = mtGo;
392
393
394     outSockSem->enterCritical();
395
396
397     for(map<string, P2pNodeInfo *>::iterator lIterAddr = myNI.begin();
398         lIterAddr != myNI.end();
399         lIterAddr++){
400
401       string addr = lIterAddr->first;
402       unsigned int hisWaitTime = localMaxDelay - myNI[ addr ]->estDelay;
403
404       packInBytes(hisWaitTime, B_HOLD_TIME, buf + iData);
405       DPRINT(0,"mr %s should wait for %ums\n", addr.c_str(), hisWaitTime);
406       basicSend(buf, headerSize + 2, addr, false);
407     }
408     outSockSem->exitCritical();
409     DPRINT(0,"mr MYSELF should wait for %ums\n",localMaxDelay);
410     if(calleeInstance)
411       calleeInstance->hold( localMaxDelay );
412
413   }
414
415   void MessageManager::receivedPing(char *inBuf, string addrIn){
416     unsigned int hisLastPingTime =0;
417     char outType;
418     switch( inBuf[iPingType]){
419       case TIME_PING:
420         {
421         int hisMaxDelay = atoi(inBuf+iData);
422         hisLastPingTime = atoi(inBuf + iData + INT_DIG);
423         DPRINT(0,"%s says his maxDelay is %d\n", addrIn.c_str(), hisMaxDelay
              );
424         newDelay(addrIn, -1, hisMaxDelay); //haven't calculated delay
425         }
426         outType = TIME_PING;
427         break;
428       case STRESS_PING:
429         outType = STRESS_PING;
```

```
430         //just pong it
431         break;
432       default:
433         DPRINT(0,"This is not happening\n");
434         outType = (char)-1;
435         break;
436     }
437 
438 
439     char szBuffer[ headerSize + 2*INT_DIG];
440     memset( szBuffer, 0, sizeof(szBuffer) );
441 
442     szBuffer[iMsgType] = mtPong;
443     szBuffer[iPlayerNo] = thisGI->playerNo; //let's be kind
444     szBuffer[iPingType] = outType;
445     sprintf(szBuffer+iData, "%d", myMMInfo->maxDelay);
446     sprintf(szBuffer+iData+INT_DIG, "%u", hisLastPingTime);
447 
448 
449     //DPRINT(0,"Ponging %u\n", hisLastPingTime);
450     basicSend(szBuffer, headerSize + 2*INT_DIG, addrIn);
451 
452 
453   }
454 
455 
456   void MessageManager::receivedPong(char *inBuf, string addrIn){
457     if( inBuf[iPingType] == STRESS_PING ){
458       myNI[addrIn]->receivedPongs++;
459       return;
460     }
461     int hisMaxDelay = atoi(inBuf+iData);
462     clock_t hisLastPingTime = atoi(inBuf + iData + INT_DIG);
463     if(hisLastPingTime != myNI[addrIn]->lastPingTime){
464       DPRINT(0,"Discarding ping %u, expected %u\n",
465           hisLastPingTime, myNI[addrIn]->lastPingTime );
466       myNI[addrIn]->greetSpace *=3;
467       myNI[addrIn]->greetSpace /=2;
468       return;
469     }
470     myNI[addrIn]->lastPongTime = clock();
471 
472     int estDelay = (myNI[addrIn]->lastPongTime
473         - myNI[addrIn]->lastPingTime)
474         *1000/CLOCKS_PER_SEC
475         /2; //roundtrip
476 
477     newDelay( addrIn, estDelay, hisMaxDelay );
478     if( myNI[addrIn]->playerNo < 0 )
479       myNI[addrIn]->playerNo = inBuf[iPlayerNo];
480   }
481 
482   void MessageManager::receivedData(char *inBuf, string addrIn){
483     int frameNo = getPackedInBytes(inBuf + iFrameNo, B_FRAME_NO);
484     int playerNo = inBuf[iPlayerNo];
485     int recvDataSize = inBuf[iDataSize];
```

```
486
487
488     unsigned char *myBuf = new unsigned char[dataSize];
489     for(int i=0; i < recvDataSize; i++)
490       myBuf[i] = inBuf[iData+i]; //bits are the same, unless dealing with
            different endian machines
491     //store it and wait for confirmation
492
493
494     struct gameInfo *gi = (struct gameInfo *)malloc( sizeof(struct gameInfo)
            );
495     gi->frameNo = frameNo;
496     gi->playerNo = playerNo;
497
498     int syncDiff = calleeInstance->receivedData( myBuf, dataSize, gi );
499
500     int msNeeded = myNI[ addrIn ]->estDelay*2 //ack there, ok here... data
          already here
501                    - syncDiff*MS_PER_SYNC;
502     DPRINT(0,"I will need %dms to input this correctly (2x(%dms) - (%d
          frames))\n",
503       msNeeded,
504       myNI[ addrIn ]->estDelay,
505       syncDiff);
506     //spare ms == -(needed ms)
507
508     //let's ack him
509     char outBuf[headerSize + B_FRAME_NO];//spare bytes in case of ack
510     for(int i=0;i<headerSize + B_FRAME_NO;i++)
511       outBuf[i] = 0;
512     int outSize;
513     int key;
514     if( syncDiff > 0 ){ //can't be on "last processed frame" (==)
515       DPRINT(0,"Correct input:%d, acking\n", frameNo);
516
517       outSize = headerSize;
518       key = makeSyncAndPlayerNoKey( frameNo, playerNo);
519       outBuf[iMsgType] = mtAckData;
520     }
521     else {
522       //p2ps::rd packed it.. thanx
523       int bounceFrameNo = gi->frameNo;
524       DPRINT(0,"Bouncing %d to %d\n", frameNo, gi->frameNo);
525
526       outSize = headerSize + B_FRAME_NO;
527       key = makeSyncAndPlayerNoKey( bounceFrameNo, playerNo);
528       outBuf[iMsgType] = mtNoAckData;
529
530       //...and pack bounceFrame
531       packInBytes( bounceFrameNo, B_FRAME_NO, outBuf+iData);
532
533       //queueInTime(mensaje, gi->frameNo + syncDiff);
534     }
535     DPRINT(0,"Received data @synco:%u, creating msg (size:%d)on %d\n",
          thisGI->frameNo, dataSize, key);
536     //correct or bounce frame... create it either way
```

```
537     msgInProcess[key] = new P2pMessage( myBuf, dataSize, thisGI->
            totalPlayers, frameNo);
538
539     packInBytes(frameNo, B_FRAME_NO, outBuf + iFrameNo);
540     outBuf[iPlayerNo] = thisGI->playerNo;
541     //do NOT use other than outSize
542     basicSend( outBuf, outSize, addrIn);
543
544
545     delete[] myBuf;
546     free( gi );
547   }
548
549   void MessageManager::receivedClosedConnection(string addrIn){
550     remove(addrIn);
551   }
552
553   void MessageManager::receivedPlayerReady(char *inBuf){
554     struct gameInfo *gameInfo;
555     gameInfo = (struct gameInfo *)malloc(sizeof(struct gameInfo));
556     gameInfo->playerNo = inBuf[iPlayerNo];
557     gameInfo->gameID = inBuf[iGameID];
558
559     if(calleeInstance)
560       calleeInstance->receivedPlayerReady(gameInfo);
561     else
562       DPRINT(0,"No calleeInstance!!");
563     free(gameInfo);
564   }
565
566
567   void MessageManager::receivedGo(char *inBuf){
568     if( thisGI->greenLighted ){
569       DPRINT(0,"Discarding greenLight\n");
570       return;
571     }
572
573     int holdTime = 0;
574     holdTime = getPackedInBytes(inBuf + iData, B_FRAME_NO);
575
576
577     if(calleeInstance){
578       calleeInstance->hold( holdTime );
579       calleeInstance->receivedGo();
580     }
581   }
582
583
584   void MessageManager::receivedAckData( char *inBuf, string addrIn ){
585     int frameNo = getPackedInBytes(inBuf + iFrameNo, B_FRAME_NO);
586     int playerNo = inBuf[iPlayerNo];
587     int key = makeSyncAndPlayerNoKey( frameNo, thisGI->playerNo);
588
589
590     unsigned int deltaT = clk2ms(clock() - myNI[addrIn]->lastSentTime);
```

```
591     DPRINT(0,"Ack from %dp%d after %ums of waiting\n", frameNo, playerNo,
            deltaT);
592     if( msgInProcess.find(key) == msgInProcess.end() ){
593       DPRINT(0,"No such msg... did you deleted it? (maybe from a noAck)\n");
594       char *myBuf = new char[headerSize];
595 
596       myBuf[iMsgType] = mtKillData;
597       myBuf[iPlayerNo] = thisGI->playerNo;
598       packInBytes(frameNo, B_FRAME_NO, myBuf + iFrameNo );
599       basicSend(myBuf, headerSize, addrIn);
600 
601 
602 
603       return;
604     }
605 
606     msgInProcess[key]->playerConfirm( playerNo );
607 
608     if(msgInProcess[key]->isReady())
609       messageCollected(key);
610   }
611   void MessageManager::messageCollected(int key){
612     DPRINT(0,"Key %d ready\n",key);
613     int bounceFrame = msgInProcess[key]->getFrameBounce();
614 
615     char myBuf[headerSize+B_FRAME_NO];//TODO: resize this
616     for(int i=0; i<headerSize; i++)
617         myBuf[i] = 0;
618     myBuf[iPlayerNo]  =  thisGI->playerNo;
619 
620     struct gameInfo *gI = (struct gameInfo *)malloc( sizeof(struct gameInfo)
            );
621     int frameNo = getFrameNo(key);
622     if( bounceFrame < 0 ){
623       DPRINT(0, "MM: \"All\" confirmed for %d...", frameNo);
624       ++myMMInfo->sentMsgCount;
625       packInBytes(myMMInfo->sentMsgCount, B_FRAME_NO,  myBuf + iMsgCount);
626       myBuf[iMsgType] = mtOkData;
627 
628       packInBytes(frameNo, B_FRAME_NO, myBuf + iFrameNo);
629 
630       gI->frameNo = frameNo;
631       gI->playerNo = thisGI->playerNo; //someone confirmed MY message
632 
633       if( calleeInstance )
634         calleeInstance->confirmMsg( gI );
635 
636       outSockSem->enterCritical();
637       for(  map<string, P2pNodeInfo *>::iterator lIter = myNI.begin();
638           lIter != myNI.end();
639           lIter++){
640 
641         string out = lIter->first;
642         basicSend(myBuf, headerSize, out, false);
643       }
644       outSockSem->exitCritical();
```

```
645
646       //TODO:if it couldn't make it, don't count it
647     }
648     else{
649
650
651       myBuf[iMsgType]   = mtBounce;
652
653       packInBytes( frameNo, B_FRAME_NO, myBuf+iFrameNo);
654       packInBytes( bounceFrame, B_FRAME_NO, myBuf+iData);
655       //same frameNo
656
657       gI->frameNo = frameNo;
658       gI->playerNo = thisGI->playerNo; //someone UNconfirmed MY message
659
660       int time2wait = (bounceFrame - thisGI->frameNo)*MS_PER_SYNC;
661       //tell people
662       for(  map< string, P2pNodeInfo *>::iterator lIterAddr = myNI.begin();
663             lIterAddr != myNI.end();
664             lIterAddr++){
665         string out = lIterAddr->first;
666         int delta = time2wait - myNI[out]->estDelay;
667         if(delta < 0)
668           delta = 0;
669         sendLater(myBuf, headerSize+B_FRAME_NO, out, delta);
670       }
671       //tell here
672       calleeInstance->confirmBounce(thisGI->playerNo, frameNo, bounceFrame);
673       //calleeInstance->deConfirmMsg( gI );
674
675     }
676     free( gI );
677
678
679     //TODO:this below should be in critical sec?
680
681     delete msgInProcess[key];
682     msgInProcess.erase( msgInProcess.find(key) );
683
684   }
685   void MessageManager::receivedOkData(char *inBuf, string addrIn){
686
687     int frameNo = getPackedInBytes(inBuf + iFrameNo, B_FRAME_NO);
688     int playerNo = inBuf[iPlayerNo];
689     int key = makeSyncAndPlayerNoKey( frameNo, playerNo);
690
691     int hisMsgCount = getPackedInBytes(inBuf+iMsgCount, B_FRAME_NO);
692     if( msgInProcess.find(key) == msgInProcess.end() ){
693       DPRINT(0,"receivedOkData:No such msg... you should worry \n");
694       return;
695     }
696
697     myNI[addrIn]->recvMsgCount++;
698
699     DPRINT(0,"Ok on %dp%d, msg count for mr %s: %u(mine), %u(his)\n",
700         frameNo,
```

```
701         playerNo,
702         addrIn.c_str(),
703         myNI[addrIn]->recvMsgCount,
704         hisMsgCount);
705
706
707     struct gameInfo *gameInfo = (struct gameInfo *)malloc( sizeof(struct
            gameInfo) );
708     gameInfo->frameNo = frameNo;
709     gameInfo->playerNo = playerNo;
710
711
712
713
714     if( calleeInstance )
715       calleeInstance->confirmMsg( gameInfo );
716
717     //this input is no longer useful
718     delete msgInProcess[key];
719     msgInProcess.erase( msgInProcess.find(key) );
720
721     free(gameInfo);
722   }
723   void MessageManager::receivedBounce( char *inBuf ){
724
725     int originalFrame = getPackedInBytes( inBuf+iFrameNo,B_FRAME_NO);
726     int bounceFrame = getPackedInBytes(inBuf+iData, B_FRAME_NO);
727     int player = inBuf[iPlayerNo];
728
729     calleeInstance->confirmBounce(player, originalFrame, bounceFrame);
730
731   }
732
733   void MessageManager::receivedLoadState( char *inBuf ){
734     int frameSaved, frameLoading;
735     int hisPlayerNo, loadedPlayerNo;
736
737     //header
738     //inBuf[iPlayerNo];
739     frameSaved = getPackedInBytes(inBuf + iFrameNo, B_FRAME_NO);
740     //data
741     loadedPlayerNo = inBuf[iData];
742     frameLoading = getPackedInBytes(inBuf + iData +1, B_FRAME_NO);
743
744
745     calleeInstance->queueStateLoading(frameSaved, frameLoading,
            loadedPlayerNo);
746     //we're losing the player with delay!
747
748   }
749
750   void MessageManager::receivedNoAckData( char *inBuf, string addrIn){
751     int originalFrame = getPackedInBytes(inBuf + iFrameNo, B_FRAME_NO);
752     int playerNo = inBuf[iPlayerNo];
753     int key = makeSyncAndPlayerNoKey( originalFrame, thisGI->playerNo);
754
```

```
755     unsigned int deltaT = clk2ms(clock() - myNI[addrIn]->lastSentTime);
756     DPRINT(0,"noAck from %dp%d after %ums of waiting\n", originalFrame,
            playerNo, deltaT);
757     if( msgInProcess.find(key) == msgInProcess.end() ){
758       DPRINT(0,"No such msg... did you deleted it? (maybe from a noAck)\n");
759       return;
760     }
761
762     int bounceFrame = getPackedInBytes(inBuf+iData, B_FRAME_NO);
763
764     msgInProcess[key]->playerDeConfirm(playerNo, bounceFrame);
765
766     if( msgInProcess[key]->isReady() )
767       messageCollected(key);
768
769   }
770   void MessageManager::receivedKillData( char *inBuf ){
771     int frameNo = getPackedInBytes(inBuf + iFrameNo, B_FRAME_NO);
772     int playerNo = inBuf[iPlayerNo];
773     int key = makeSyncAndPlayerNoKey( frameNo, playerNo);
774
775     //unsigned int hisMsgCount = atoi(inBuf+iMsgCount);
776     DPRINT(0,"Kill on %up%d\n", frameNo,  playerNo);
777
778     struct gameInfo *gameInfo = (struct gameInfo *)malloc( sizeof(struct
            gameInfo) );
779     gameInfo->frameNo = frameNo;
780     gameInfo->playerNo = playerNo;
781
782
783     calleeInstance->deConfirmMsg( gameInfo );
784     if( msgInProcess.find(key) != msgInProcess.end() ){
785       delete msgInProcess[key];
786       msgInProcess.erase( msgInProcess.find(key) );
787     } else
788       DPRINT(0,"Couldn't kill %d... should I be worried?\n",key);
789
790     free(gameInfo);
791
792
793   }
794   void MessageManager::receivedStress(){
795     /*
796     stressConns();
797     */
798   }
799
800
801
802   int MessageManager::getLocalMaxDelay(){
803     int out = 0;
804     for( map<string, P2pNodeInfo *>::iterator iter = myNI.begin();
805             iter != myNI.end();
806             iter++){
807         if( ((*iter).second)->estDelay >= 0)
808           out = (out >((*iter).second)->estDelay)?out:
```

```
809             ((*iter).second)->estDelay;
810
811
812       }
813     return out;
814   }
815
816   void MessageManager::newDelay(string addrIn, int estDelay,int hisMaxDelay)
        {
817
818     if(estDelay == 0 )
819       estDelay = 1;
820     if(hisMaxDelay == 0)
821       hisMaxDelay = 1;//let's round it up
822
823     P2pNodeInfo *hisNI = myNI[ addrIn ];
824     int byWeight,maxDelay=-1;
825
826     outSockSem->enterCritical(); //two pings could arrive very close to each
          other
827
828     if(estDelay > 0){
829       //change one's delay
830       if( hisNI->estDelay < 0)
831         hisNI->estDelay = estDelay;
832       if( hisNI->estDelay > estDelay)
833         byWeight = myMMInfo->loWeight;
834       else
835         byWeight = myMMInfo->hiWeight;
836
837       hisNI->estDelay *= (byWeight-1);
838       hisNI->estDelay += estDelay;
839       hisNI->estDelay /= byWeight;
840       DPRINT(0,"Changin mr %s's delay to %u\n", addrIn.c_str(), hisNI->
            estDelay);
841
842       //recalculate our global delay
843       maxDelay = getLocalMaxDelay();
844     }
845
846     maxDelay = (maxDelay > hisMaxDelay)?maxDelay:hisMaxDelay;
847     myMMInfo->maxDelay = maxDelay;
848
849     if(calleeInstance)
850       calleeInstance->setNewDelay(maxDelay*3); //data, ack, ok
851
852
853
854
855
856     outSockSem->exitCritical();
857   }
858   void MessageManager::getMaxDelay(){
859     clock_t chkpt = clock();
860     DPRINT(0,"Request for new delay calculation @ %u\n",chkpt);
861
```

```
862     for(int i=0; i<N_ESTIMATIONS; i++){
863       chkpt = clock();
864       bool allClear = false;
865       while(!allClear){
866         greet();
867         if( !myMMInfo->looping)
868           tryAndReceive(false);
869         allClear = true;
870         for( map<string, P2pNodeInfo *>::iterator iter = myNI.begin();
871             iter != myNI.end();
872             iter++){
873           if( ((*iter).second)->lastPongTime < chkpt){
874             allClear = false;
875             break;
876           }
877         }
878       }
879     }
880     DPRINT(0,"delay calculated\n");
881   }
882   void MessageManager::receivedPkg(char *inBuf, string addrIn){
883     DPRINT(0,"mr %s ", addrIn.c_str());
884     if( myMMInfo->allConnected && myNI.find(addrIn) == myNI.end() ){
885       DPRINT(0,"invalid recvd pkg\n");
886       return; //invalid addr
887     }
888     switch( inBuf[iMsgType] ){
889       case mtOpenConn:
890         DPRINT(0,"is opening a connection\n");
891         receivedOpenConnection(addrIn);
892         break;
893
894       case mtHandShake:
895         DPRINT(0,"is confirming a connection\n");
896         receivedHandShake(addrIn);
897         break;
898
899       case mtPing:
900         //DPRINT(0,"just pinged me\n");
901         receivedPing(inBuf, addrIn);
902         break;
903
904       case mtPong:
905         //DPRINT(0,"just ponged me\n");
906         receivedPong(inBuf, addrIn);
907         break;
908
909       case mtData:
910         DPRINT(0,"sent me data\n");
911         receivedData(inBuf, addrIn);
912         break;
913
914       case mtLoadState:
915         DPRINT(0,"wants to load state\n");
916         receivedLoadState( inBuf );
917         break;
```

```
918
919       case mtCloseConn:
920         DPRINT(0,"is closing his connection\n");
921         receivedClosedConnection(addrIn);
922         break;
923
924       case mtReady:
925         DPRINT(0,"is ready\n");
926         receivedPlayerReady(inBuf);
927         break;
928
929       case mtGo:
930         DPRINT(0,"is saying we should start emu\n");
931         receivedGo(inBuf);
932         break;
933
934       case mtAckData:
935         DPRINT(0,"Just acked me\n");
936         receivedAckData(inBuf, addrIn);
937         break;
938
939       case mtOkData:
940         DPRINT(0,"Just OK'd me\n");
941         receivedOkData(inBuf, addrIn);
942         break;
943
944       case mtNoAckData:
945         DPRINT(0,"Just noAcK'd me\n");
946         receivedNoAckData( inBuf, addrIn);
947         break;
948
949       case mtKillData:
950         DPRINT(0,"Just KILL'd me\n");
951         receivedKillData( inBuf );
952         break;
953
954       case mtBounce:
955         DPRINT(0,"just confirmed bounce");
956         receivedBounce(inBuf);
957         break;
958
959       case mtStress:
960         DPRINT(0,"just told me to stress \n");
961         receivedStress();
962         break;
963
964       default:
965         DPRINT(0,"sent me an empty package\n");
966         //error
967         break;
968
969     }
970     //DPRINT(0,"...finished processing package\n");
971
972   }
973   void MessageManager::stressConns(){
```

```
974     int nNodes = myNI.size();
975     int totalBombing = N_STRESS_LOAD * nNodes;
976     DPRINT(0,"Stressing...\n");
977     //pkg
978     char buf[headerSize + 2*INT_DIG];
979     memset( buf, 0, sizeof(buf) );
980     buf[iMsgType] = mtPing;
981     buf[iPingType] = STRESS_PING;
982     map<string, P2pNodeInfo *>::iterator iAddr
983             = myNI.begin();
984
985     for(int i=0; i<totalBombing; i++, iAddr++){
986       if( iAddr == myNI.end() )
987         iAddr = myNI.begin();
988
989       basicSend( buf, headerSize, iAddr->first );
990       if( !myMMInfo->looping)
991           tryAndReceive(false);
992
993     }
994
995     DPRINT(0,"Number of received pongs:\n");
996     for( iAddr = myNI.begin(); iAddr != myNI.end(); iAddr++)
997       DPRINT(0,"%s:%d\n", iAddr->first.c_str(), iAddr->second->receivedPongs
           );
998
999     Sleep( getLocalMaxDelay() );
1000  }
1001
1002
1003  void MessageManager::stressAll(){
1004    //called by boss
1005    /*
1006    stressConns();
1007    basicSend(player)
1008    create msg
1009    waitfortest
1010    continue
1011    */
1012    DPRINT(0,"WARNING:No stress testing since haven't implemented passing
           msg\n");
1013  }
1014
1015  void MessageManager::retrial(){
1016
1017  }
```

C . Código clase P2pPlayer

```
1  #include "P2pPlayer.h"
2  #include <assert.h>
3  #include <windows.h>
4  #ifdef __MINGW32__
```

```
5     //#define DPRINT(format, ...) if(1)bprintf(0, format, ## __VA_ARGS__ )
6     #define DPRINT if(1)bprintf
7     extern int (__cdecl *bprintf) (int nStatus, TCHAR* szFormat, ...);
8   #else
9     #define DPRINT(format, ...)
10  #endif
11
12  //all concurrent access are managed outside
13  using namespace std;
14
15  P2pPlayer::P2pPlayer(int playerNoIn, struct gameInfo *GIin){
16    playerNo = playerNoIn;
17    inputCache = NULL;
18    setInputCache = NULL;
19    ready = false;
20    myGI = GIin;
21  }
22
23  P2pPlayer::~P2pPlayer( ){
24    if(!inputCache)
25      free(inputCache);
26    //iterate over map to clean it
27  }
28
29
30  bool P2pPlayer::setInput( unsigned char *inBuf, int size, int frameNo){
31    if( inputCache == NULL || setInputCache == NULL ){
32      inputCache = new unsigned char[size];
33      setInputCache = new unsigned char[size];
34
35      for(int i=0; i< size; i++){
36        inputCache[i]    = 0;
37        setInputCache[i] = 0;
38      }
39
40
41    }
42
43    bool diff = false;
44    for(int i = 0; i < myGI->inputSize; i++)
45      if( inBuf[i] != setInputCache[i] ){
46        diff = true;
47        break;
48      }
49
50    //assert( inputCache != NULL );
51
52    if( !diff ) //if it isn't different, don't change anything
53      return false;
54
55    //create new p2pMsg from candidate
56    P2pMessage *msgIn = new P2pMessage( inBuf, size, myGI->totalPlayers,
        frameNo);
57
58    if( inputMsgCache.find(frameNo) != inputMsgCache.end()
59    ){
```

```
60        //let's take it out for comparison
61        map< int, P2pMessage *>::iterator iter =
62          inputMsgCache.find(frameNo);
63
64
65        //let's clear storage
66        delete inputMsgCache[frameNo];
67        inputMsgCache.erase( iter );
68
69        DPRINT(0,"pp:deleted previous input on %d\n", frameNo);
70      }
71      //let us hope that the number of players have been set
72      //assert( myGI->totalPlayers > 0);
73      //assert( myGI->totalPlayers <= 8);
74
75      DPRINT(0,"Creating message on %d\n", frameNo);
76      //copy here
77      inputMsgCache[frameNo] = msgIn;
78
79
80
81      //cache it
82      for(int i=0; i < size; i++)
83        setInputCache[i] = inBuf[i];
84
85
86      //if we had to cache it, then it was different
87      return diff;//true
88
89    }
90
91    int P2pPlayer::getInput( unsigned char *outBuf, int frameNo){
92      if( inputCache == NULL || setInputCache == NULL ){
93        inputCache = new unsigned char[myGI->inputSize];
94        setInputCache = new unsigned char[myGI->inputSize];
95
96        for(int i=0; i< myGI->inputSize; i++){
97          inputCache[i]    = 0;
98          setInputCache[i] = 0;
99        }
100     }
101
102     if( inputMsgCache.size() > 0 ){
103       DPRINT(0, "Player: frameNo:%d : Have %d on queue\n", frameNo,
              inputMsgCache.size() );
104     }
105     if( bounceInputMsgCache.size() > 0){
106       DPRINT(0, "Player: frameNo:%d : Have %d on BOUNCE queue\n", frameNo,
              bounceInputMsgCache.size() );
107     }
108
109     if( inputMsgCache.find( frameNo ) != inputMsgCache.end()
110         //&& inputMsgCache[frameNo]->isReady() ){
111         // above condition is checked separately
112       ){
113       //oh, let's update the output cache
```

```
114       DPRINT(0,"New input on %u\n", frameNo);
115       inputMsgCache[frameNo]->getInput( inputCache );
116       lastOriginalFrameProcessed = frameNo;
117     }
118     if( bounceInputMsgCache.find( frameNo ) != bounceInputMsgCache.end()  ){
119       // get range of elements under frameNo
120       pair< multimap<int, P2pMessage *>::iterator,
121             multimap<int, P2pMessage *>::iterator> range;
122       range = bounceInputMsgCache.equal_range(frameNo);
123       // get highest originalFrame and  reference
124       P2pMessage *selected = range.first->second;
125       int highestOriginalFrame=selected->getOriginalFrame();
126 
127       for( multimap<int, P2pMessage *>::iterator i = range.first;
128             i != range.second;
129             i++){
130         if( i->second->getOriginalFrame() > highestOriginalFrame ){
131           // select it
132           selected = i->second;
133           highestOriginalFrame = selected->getOriginalFrame();
134         }
135       }
136 
137       //input hasn't changed since original frame
138       if( selected->getOriginalFrame() > lastOriginalFrameProcessed ){
139 
140         DPRINT(0,"New bounce input on %u (original is %d)\n",
141               frameNo, selected->getOriginalFrame() );
142         selected->getInput( inputCache );
143         lastOriginalFrameProcessed =
144             selected->getOriginalFrame();
145       }
146     }
147     for(int i=0; i< myGI->inputSize; i++)
148       outBuf[i] = inputCache[i];
149 
150     return myGI->inputSize;
151   }
152 
153   int P2pPlayer::getSetCache( unsigned char *outBuf ){
154     if(myGI->inputSize <= 0)
155       return -1;
156     for(int i=0; i < myGI->inputSize; i++)
157       outBuf[i] = inputCache[i];
158 
159     return myGI->inputSize;
160   }
161 
162 
163   void P2pPlayer::clearInput(int frameNo){
164     //find on map
165     map<int, P2pMessage *>::iterator result = inputMsgCache.find( frameNo );
166     if( result != inputMsgCache.end() ){
167 
168       DPRINT(0,"Trying to delete frameNo:%u\n",frameNo);
169       //delete pp
```

```
170       delete inputMsgCache[frameNo];
171       //erase()
172       inputMsgCache.erase( result );
173     }
174
175
176     //bounce section
177
178     result = bounceInputMsgCache.find(frameNo);
179     if( result != bounceInputMsgCache.end() ){
180
181       DPRINT(0,"Trying to delete %d BOUNCEframeNos:%u\n",bounceInputMsgCache
              .count(frameNo),frameNo);
182       //delete pp
183       pair< multimap<int, P2pMessage *>::iterator,
184             multimap<int, P2pMessage *>::iterator> range;
185       range = bounceInputMsgCache.equal_range(frameNo);
186       for( multimap<int, P2pMessage *>::iterator i = range.first;
187             i != range.second;
188             i++){
189         delete i->second;
190       }
191       //erase()
192       bounceInputMsgCache.erase( range.first, range.second );
193     }
194   }
195
196   void P2pPlayer::confirmMsg(int frameNo){
197     if( inputMsgCache.find(frameNo) != inputMsgCache.end() ){
198       inputMsgCache[frameNo]->fullConfirm();
199       return;
200     } else {
201       DPRINT(0,"player:Whoa! %u\n", frameNo);
202     }
203
204
205   }
206   void P2pPlayer::confirmBounce( int originalFrame, int bFrameNo ){
207     bool bounced = false;
208
209
210     if( inputMsgCache.find( originalFrame ) != inputMsgCache.end() ){
211       DPRINT(0,"you should confirming, not bouncing... sigh\n");
212       bounceInput(originalFrame, bFrameNo);
213       confirmBounce(originalFrame, bFrameNo);
214       bounced = true;
215     }
216     multimap<int, P2pMessage *>::iterator iter;
217
218     if(!bounced){
219
220       for(iter = bounceInputMsgCache.begin(); iter != bounceInputMsgCache.
              end(); iter++){
221
222         if( (iter->second)->getOriginalFrame() == originalFrame ){
223           P2pMessage *curr = iter->second;
```

```
224           bounceInputMsgCache.erase(iter);//remove from previous position
225           bounceInputMsgCache.insert(
226               make_pair( bFrameNo, curr) );//move to new bFrame
227           curr->fullConfirm();
228           bounced = true;
229           break;
230         }
231
232       }
233
234     }
235
236     if(!bounced)
237       DPRINT(0,"pplay: ERROR! couldn't bounce");
238     else
239       DPRINT(0,"Finished bouncing\n");
240   }
241   void P2pPlayer::deConfirmMsg(int frameNo){
242     map<int, P2pMessage *>::iterator result = inputMsgCache.find(frameNo);
243     if( result == inputMsgCache.end() ){
244       DPRINT(0,"player:dude! %u\n", frameNo);
245       return;
246     }
247     delete inputMsgCache[frameNo];
248     inputMsgCache.erase( result );
249   }
250
251   int P2pPlayer::inputReady( int frameNo ){
252     //first hand input
253     if( inputMsgCache.find( frameNo ) != inputMsgCache.end() ){
254       if( !inputMsgCache[frameNo]->isReady() )
255         return frameNo; //new input NOT ready
256       else
257         return -1; //new input ready
258     }
259     //bounce input
260     if( bounceInputMsgCache.find( frameNo ) != bounceInputMsgCache.end() ){
261       pair< multimap<int, P2pMessage *>::iterator,
262             multimap<int, P2pMessage *>::iterator> range;
263       range = bounceInputMsgCache.equal_range(frameNo);
264       int highestOriginalFrame=-1; //-1 == everything ok
265       for( multimap<int, P2pMessage *>::iterator i = range.first;
266             i != range.second;
267             i++){
268         if(   !i->second->isReady()
269               && i->second->getOriginalFrame() > highestOriginalFrame )
270           highestOriginalFrame = i->second->getOriginalFrame();
271
272       }
273
274       return highestOriginalFrame;
275
276     }
277
278     return -1; //no new input of any kind
279   }
```

```
280
281 void P2pPlayer::bounceInput( int oldInput, int newInput){
282   map<int, P2pMessage *>::iterator result = inputMsgCache.find(oldInput);
283   if(result != inputMsgCache.end()){
284     P2pMessage *bounced = inputMsgCache[oldInput];
285     inputMsgCache.erase(result);
286     bounceInputMsgCache.insert( make_pair(newInput,bounced) );
287   }else{
288     DPRINT(0,"pp: Couldn't bounce orig:%d, no such frame", oldInput);
289   }
290 }
```