Universidad Zaragoza

TRABAJO FIN DE GRADO

Análisis y optimización del protocolo Spice en escenarios de ancho de banda limitado

Autor:

David Lorite Solanas

Director:

Sergio Lopéz Pascual

Ponente:

José Ramón Gállego Martínez

Escuela de Ingeniería y Arquitectura

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Noviembre 2015

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

DECLARACIÓN DE AUTORÍA Y ORIGINALIDAD

(Este documento debe acompañar al Trabajo Fin de Grado (TFG)/Trabajo Fin de Máster (TFM) cuando sea depositado para su evaluación).

TRABAJOS DE FIN DE GRADO / FIN DE MÁSTER

D./Dª. ____,

con nº de DNI ______________________ en aplicación de lo dispuesto en el art. 14 (Derechos de autor) del Acuerdo de 11 de septiembre de 2014, del Consejo de Gobierno, por el que se aprueba el Reglamento de los TFG y TFM de la Universidad de Zaragoza,

Declaro que el presente Trabajo de Fin de (Grado/Máster) ____, (Título del Trabajo)
____
____
____
____
____
____,
es de mi autoría y es original, no habiéndose utilizado fuente sin ser citada debidamente.

Zaragoza, ____________________________________

Fdo: David Lorite Solanas

Análisis y optimización del protocolo Spice en escenarios de ancho de banda limitado

RESUMEN

Este TFG consiste en el análisis de ancho de banda del protocolo Spice en escenarios con ancho de banda limitado. Este análisis ha ido precedido por la creación de herramientas de análisis de ancho de banda, tanto ofrecidas por programas de terceros como nativas al propio protocolo. Con estas herramientas se han analizado los distintos protocolos de presentación de escritorio remoto más usados en la actualidad además de Spice. Con los datos obtenidos de los análisis anteriores se ha comparado el uso de ancho de banda con distintas restricciones y se ha estudiado como se adaptan a ellas. De este análisis además se han sacado conclusiones e ideas para mejorar el protocolo de Spice. Por último se han usado estas ideas para optimizar el protocolo Spice y se han comparado los resultados obtenidos, tanto objetivamente (uso del ancho de banda) como subjetivos (interactividad y calidad de imagen). Finalmente después de analizar los resultados, se ha obtenido una optimización que supera con creces las espectativas en ambos casos (subjetiva y objetiva con una media del 34.69% de mejora).

ÍNDICE GENERAL

1. Introducción 1

1.1. Propósito y motivación 1

1.2. Objetivo 2

1.3. Materiales y herramientas utilizadas 2

1.4. Organización de la memoria 2

2. Análisis previo 4

2.1. Público objetivo 4

2.2. Spice 4

2.3. Estado del arte 5

3. Creación de las herramientas métricas 6

3.1. Wireshark 6

3.2. Herramientas nativas 8

4. Configuración de los escenarios 12

4.1. Spice 12

4.2. PCOIP 13

4.3. RDP 14

4.4. ICA 15

4.5. Finalización e instalación de los programas usados en las pruebas 16

5. Análisis del ancho de banda 17

5.1. Puesta en marcha de las macros 17

5.2. Análisis objetivo 18

5.3. Análisis subjetivo 20

6. Optimización del protocolo y resultados 21

6.1. Método Ica 21

6.2. Método PCOIP 21

6.3. Método hibrido 22

6.4. Método hibrido con token bucket **23**
6.5. Resultados **25**
7. Conclusiones y líneas futuras **27**
Bibliografía **28**
Anexo A **29**
Anexo B **32**

Capítulo 1

Introducción

Spice (*the Simple Protocol for Independent Computing Environments*) [1] es un protocolo abierto de presentación de escritorio remoto. Fue creado originalmente por *Qumronet*, aunque está siendo desarrollado en estos momentos por *Red Hat*.

En este tipo de protocolos, un cliente se conecta a un servidor que aloja una máquina virtual llamada *guest*, y envía la pantalla u otra información de dicha máquina virtual por distintos canales TCP (o UDP) hasta el cliente, el cual presenta la información al usuario final. El uso de estos protocolos es útil sobre todo para empresas, por el ahorro en costes, ahorro en mantenimiento y seguridad sin costes adicionales. Además permite a los usuarios trabajar donde quieran y con la máquina que quieran. Este ahorro en mantenimiento es debido a que al haber solo servidores donde se encuentran las máquinas virtuales, modificarlos de cualquier forma es mucho más sencillo que hacerlo equipo por equipo, es decir, si antes se tenía que enviar a un técnico a actualizar ordenador por ordenador, ahora se puede automatizar esta misma actualización. El ahorro en costes se debe a que cuando se ha de instalar todo lo necesario, en vez de instalar ordenadores se instalan *thin clients* o clientes ligeros, que tienen todo lo necesario para correr el cliente con el protocolo de presentación de escritorio remoto.

1.1 Propósito y Motivación

El proyecto se ha llevado a cabo en flexVDI [2]. FlexVDI es una startup zaragozana que ofrece servicios de VDI (Virtual Desktop Infrastructure) con componentes en su mayor parte de código abierto. FlexVDI trabaja en sus programas con el protocolo Spice, es por esto que es interesante primeramente saber cómo funciona Spice frente a otros protocolos y si se puede optimizar de alguna forma. Esta comparación se llevara a cabo con Spice frente a los demás protocolos. Se usaran conexiones limitadas, es decir se limitará el ancho de banda para asemejarse a un escenario con un ancho de banda reducido.

Por lo tanto se ha hecho un análisis de Spice frente a los protocolos de RDP [3] (Remote Desktop Protocol), ICA [4] (Independent Computing Architecture) y PCOIP [5] (PC-over-IP). Estos protocolos son los más famosos y usados en la actualidad por las empresas. A diferencia de estos protocolos Spice es libre, los otros son privados y no podemos acceder a su código de ninguna forma. Gracias a esto podemos añadir a Spice funcionalidades libremente y sin ningún tipo de compromiso. Sabiendo que se puede modificar este protocolo, si se analiza como de buenos son estos protocolos con respecto a Spice, se pueden buscar mejoras para el mismo.

1.2 Objetivo

Hay dos objetivos. El primer objetivo es analizar la diferencia tanto subjetiva (la fluidez), como objetiva (consumos de ancho de banda) y con distintas conexiones.

El segundo objetivo es optimizar el protocolo Spice una vez tenemos los resultados para cada protocolo. Veremos si Spice lo hace mejor o peor en comparación con los demás. Si en efecto lo hace peor, es decir, consume más, entonces estudiaremos por qué lo hace peor. Adicionalmente también compararemos subjetivamente cómo de fluido se ve. Haremos el mismo estudio que el anteriormente citado, es decir un estudio subjetivo y compararemos si en este también sale peor. Finalmente, sabiendo estos datos, buscaremos una optimización tanto objetiva como subjetiva del protocolo Spice. En última instancia, expondremos los resultados obtenidos después de aplicar la optimización.

1.3 Materiales y herramientas utilizadas

Los programas y herramientas usados han sido el lenguaje de programación C [6], con el entorno de desarrollo *Eclipse* [7] para C. Para el análisis del ancho de banda se ha usado el analizador de protocolos *Wireshark* [8]. Para las restricciones de ancho de banda se ha usado el programa *traffic control* [9] para Linux. Se ha usado los programas cedidos por flexVDI, para la creación del escenario, estos han sido el servidor con el manager, el dashboard y el cliente. Para los otros escenarios se han usado las versiones de prueba de *VMWare* [10] (para el caso de PCoIP) y *Citrix* [11] (para el caso de ICA), además para este último se usó la versión de prueba de *Windows server 2013* [12] en dos máquinas virtuales creadas en el entorno de desarrollo de flexVDI. Por último los equipos de pruebas que se usaron fueron una maquina con *Windows 7* [13] y otra con *Fedora 22* [14].

1.4 Organización de la memoria

El contenido de la memoria se divide de la siguiente forma:

- **Capítulo 1:** Se realiza una introducción del TFG y el objetivo a alcanzar así como la motivación y propósito del mismo.
- **Capítulo 2:** Se sitúa el protocolo en el contexto actual y se ha realizado un análisis previo del funcionamiento del protocolo Spice. Se incluye también el estado del arte.
- **Capítulo 3:** Se explica la creación de las herramientas métricas de *Wireshark* así como las creadas en el código nativo de Spice.
- **Capítulo 4:** Se explica cómo ha sido el proceso de creación de los escenarios para cada protocolo.

- **Capítulo 5:** Se desarrollan las pruebas de ancho de banda así como el análisis objetivo y subjetivo de los mismos.
- **Capítulo 6:** Se optimiza el protocolo gracias a las conclusiones obtenidas del análisis previo y se explican los resultados obtenidos.
- **Capítulo 7:** Se realiza una pequeña conclusión final y algunos detalles a mejorar en un futuro.

Capítulo 2

Análisis Previo

2.1. Público objetivo:

El protocolo Spice está integrado en plataformas de VDI. En estas los usuarios son habitualmente gente con poco conocimiento de informática y redes. Estos usuarios quieren encender el ordenador sin impórtales cómo lo hace y que salga todo lo que ellos necesitan, principalmente una aplicación de ofimática y un navegador de internet. A estos usuarios además les interesa poder trabajar donde sea, en cualquier punto de España (o del mundo). En algunos lugares es posible que la cobertura a internet sea limitada.

Por lo tanto es interesante analizar el rendimiento de este protocolo en estos escenarios con recursos limitados y compararlo con otros protocolos que se pueden adaptar mejor a estas características para aprender de ellos.

2.2. Spice:

Como hemos visto Spice es un protocolo de presentación de escritorio remoto. Este protocolo conecta a un cliente con un servidor. El cliente tiene que saber de alguna forma (por medio de un manager o por otro mecanismo) la dirección ip del servidor y el puerto destino. Una vez creada la conexión TCP, se procede al intercambio de credenciales si el servidor ha optado por asignarle una contraseña. Cuando el cliente está validado, se procede a la creación de canales. Estos canales son conexiones dedicadas a un tipo de datos y van en diferentes puertos consecutivos. Hay numerosos canales pero los más importantes son:

- Main: Es el canal principal que controla los ajustes y opciones.
- Display: El encargado de enviar los datos relacionados con la pantalla.
- Inputs: El encargado de los periféricos, tales como el ratón o el teclado.
- Cursor: Es el canal que se encarga de indicar dónde se encuentra el cursos en cada momento.

Una vez creado el canal de Main, a los demás se les asignará en orden ascendente el puerto que corresponda. A partir de este momento se procede al intercambio de datos, cada uno por su canal correspondiente.

Estos canales suponen la parte más importante de la conexión y son los que usarán en su mayoría los recursos de ancho de banda. Haciendo una comparativa podemos ver cuál es el predominante de ellos y centrarnos en él.

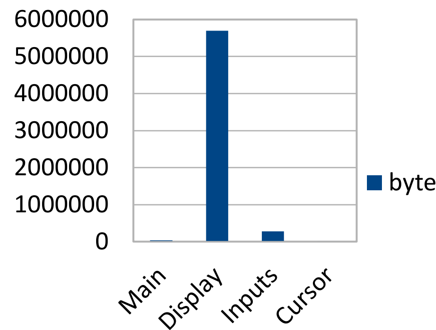


Figura 1: Consumo por canales

En la figura 1 se refleja cada canal con el envío de información que hace cada uno en una conexión convencional (conexión, navegación web y cierre). Como podemos observar el canal predominante es el de *display* y por eso nos vamos a centrar en él.

2.3. Estado del arte:

Los protocolos de presentación de escritorio remoto están teniendo un gran crecimiento en los últimos años debido principalmente al aumento del ancho de banda global. Estos protocolos en su mayoría son privativos por lo que usar estas alternativas no era una opción al no poder modificar ninguna parte del código. Por lo tanto se decidió usar un protocolo libre. A continuación se detallan los protocolos más importantes con sus características:

- **RDP**: Protocolo propietario desarrollado por *Windows*. El protocolo usa por defecto el puerto TCP 3389 en el servidor para recibir las peticiones. Como principal ventaja es que viene por defecto instalado en todos los sistemas operativos *Windows*.
- **PCOIP:** Protocolo propietario desarrollado por *Teradici* del que hace uso *VMWare* y *Amazon*. Este protocolo usa por defecto el puerto UDP 3389. La ventaja es que viene integrado en los principales programas de *VMWare*.
- **ICA:** Protocolo propietario desarrollado por *Citrix Systems*. Este protocolo usa por defecto el puerto TCP 1494. La ventaja al igual que con *VMWare* es la integración en sus productos.

En conclusión, aunque hay alternativas muy competentes en sus campos, ninguna es de código libre. Por ello se decidió trabajar con Spice.

Capítulo 3

Creación de las herramientas métricas

Para poder analizar el uso del ancho de banda para los distintos protocolos, creamos y usamos las siguientes herramientas: *Wireshark* y herramientas nativas. Estas herramientas nos ayudarán a cuantificar la cantidad de ancho de banda usado en un momento específico.

3.1. Wireshark:

Wireshark es un analizador de protocolos. La funcionalidad que provee es similar a *tcpdump* pero añadiendo una interfaz gráfica, además de opciones de análisis y filtrado de la información. *Tcpdump* es una herramienta en línea de comandos para sistemas operativos basados en *Unix*, cuya utilidad es analizar el tráfico de la red. De forma resumida muestra en la línea de comandos los paquetes que circulan en tiempo real por la interfaz de red. Esto quiere decir que analizaremos los paquetes enteros, desde el nivel de aplicación hasta el nivel físico. De estos paquetes sacaremos posteriormente el uso total de una conexión, teniendo en cuenta todas las cabeceras. Todo esto es software libre y ejecutable en la mayoría de sistemas operativos. Además gracias a las herramientas disponibles seremos capaces de entender el protocolo de una forma general.

Wireshark usa disectores para hacer un análisis de los paquetes dependiendo del protocolo que se esté usando en esos momentos. Estos disectores son programas integrados en Wireshark y escritos en C, que decodifican el paquete y ordenan la información de forma ordenada y entendible. En el caso del disector de Spice, este está desactualizado y debido a esto se actualizo el mismo para analizar correctamente el uso que se hace del ancho de banda disponible.

Primeramente se actualizaron las cabeceras. Esto es un fichero llamado *packet-spice.h*, este fichero contiene una serie de valores enumerados que representan un estado de la conexión. El propio código del disector (archivo *packet-spice.c*) contiene una nota que nos explica cómo actualizar esta cabecera de forma automática, gracias al uso de una herramienta de uno de los repositorios de Spice. Esta herramienta aprovecha la definición del *Spice-protocol* incluido en uno de los repositorios, llamado

```
case SPICE_MSGC_INPUTS_KEY_SCANCODE:
        ti = proto_tree_add_text(tree, tvb, offset, 2, "Client KEY_SCANCODE
message");
        inputs_tree = proto_item_add_subtree(ti, ett_inputs_client);
        proto_tree_add_item(inputs_tree, hf_keyboard_code, tvb, offset, 2,
ENC_LITTLE_ENDIAN);
        offset += 2;
```

Figura 2: Código de packet-spice.c

Spice-common. Teniendo este repositorio descargado, lanzamos la instrucción correspondiente con python: "*python ./spice_codegen.py --generate-wireshark-dissector \spice.proto packet-spice.h*".

Con las cabeceras ya actualizadas, se procedió a probar el disector lanzando una conexión de Spice. Una vez testeado el disector, vimos que el principal problema era que cuando se presionaba la tecla de retroceso (*backspace*) el disector dejaba de funcionar correctamente. Cuando se presionaba está tecla el disector no la reconocía y después de eso todo ese canal resultaba ilegible y no se podían recuperar los datos. Al ver en *Wireshark* el paquete mal diseccionado, se podían apreciar dos cosas: primero que era un mensaje del canal de inputs, y segundo que el tipo de mensaje tenía un valor de 104. Este valor visto en la cabecera correspondía a un mensaje llamado *SPICE_MSGC_INPUTS_KEY_SCANCODE*. Una vez detectado de qué tipo de mensaje se trata, pasamos a buscar el mensaje en disector (*packet-spice.c*). En este archivo vamos a la función que disecciona los paquetes del canal input del cliente, *dissect_spice_inputs_client*. En esta función tenemos un *switch-case* que dependiendo del tipo de mensaje hace lo correspondiente. En nuestro

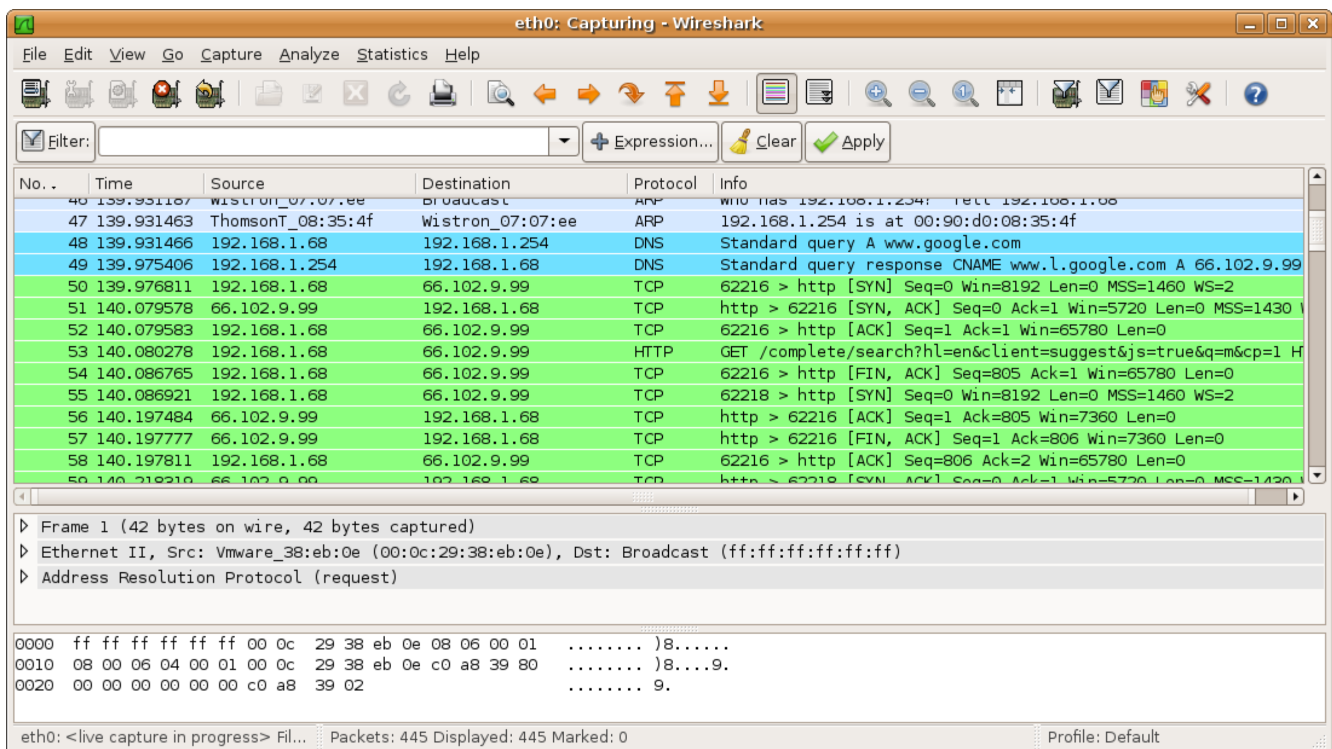


Figura 3: Wireshark capturando paquetes

caso debería haber un case con el nombre antes mencionado de S*PICE_MSGC_INPUTS_KEY_SCANCODE* (ver figura 2). Este caso no está presente y por lo tanto procedemos a añadirlo. Además tenemos que añadir al árbol un texto que contiene una descripción del mensaje, *Client KEY_SCANCODE message*. Añadimos el código recibido del teclado y finalmente añadimos un *offset* para que pueda seguir diseccionando los siguientes mensajes.

Una vez hecho esto, podemos pasar a analizar el uso de ancho de banda.

3.2. Herramientas métricas nativas:

También se quería hacer mediciones de la cantidad de datos enviada y recibida por el cliente de forma nativa. Así que para ello se añadió en el código del cliente de Spice funcionalidades nuevas. Se empezó buscando la función que se encarga del envío y recepción de datos. En este caso función de envió de datos se llama *spice_channel_flush_wire (*ver Anexo A*)*. Esta función se encarga de enviar los datos directamente sabiendo el tamaño del mensaje. Además se modificó la función para que cuando fuera a enviar el mensaje, antes de hacerlo llamará a una función creada anteriormente (*store_in_buffer*) para guardar el tamaño del paquete enviado una vez por segundo. Para contabilizar el tiempo en esta función se añadió un reloj (ver figura 4), para que en cada segundo se actualizara la información nueva. Este reloj viene dado por la función *g_get_monotonic_time*, la cual está en la librería de *Glib*, y devuelve un entero de 64 bits que son los microsegundos que han pasado desde el 1 de Enero de 1970 UTC. Entonces, si es la primera vez que entra a la función simplemente guardamos el tiempo del reloj y no hacemos nada más. Si ha pasado más de un segundo pero menos de dos, llamamos a la función de *store_in_buffer* y guardamos el tamaño enviado. Y por último si han pasado más de dos segundos entonces como en ese tiempo no se ha enviado nada, añadimos de forma iterativa ceros llamando a la función *store_in_buffer*.

```
time = g_get_monotonic_time ();
if (time2 == 0)
    time2 = time;
if (((time2 + 1000000)  < time) && ((time2 + 2000000) > time)){
    time2 += 1000000;
    store_in_buffer(TRUE, &my_buf,aclen, channel,time);
    aclen = datalen;
}else if((time2 + 2000000) < time){
    i = (time - time2)/ 1000000;
    for (guint j=1;j<i;j++){
        aclen = 0;
        store_in_buffer(TRUE,&my_buf,aclen, channel,time);
    }
    aclen = datalen;
    store_in_buffer(TRUE,&my_buf,aclen, channel,time);
    time2 += i * 1000000;
}else{
    aclen += datalen;
}
```

Figura 4: Código del spice_channel_flush_wire

Con esto en *store_in_buffer* iremos guardando en un vector circular, cada segundo con su tamaño correspondiente. Este vector es realmente una estructura que contiene un vector y dos enteros que hacen de marcador de cola y de cabeza. Esta función recibe como parámetros:

- Un booleano para marcar si es el cliente el que envía, o es el servidor.
- El puntero al buffer circular, donde guardaremos los datos.
- Un entero sin signo de 64 bits, que señala cuantos bytes se han enviado en un segundo.
- Un puntero a *SpiceChannel* (una estructura propia de Spice), para señalar el canal.
- Un entero sin signo de 64 bits, para marcar el tiempo.

Con estas variables, el primer paso en nuestra función es crear e inicializar las nuevas variables. La primera será la variable de tipo *SpiceMsgOut**. Esta será la que luego enviaremos con la otra que contiene el mensaje, para ser enviada. Gracias al puntero a *SpiceChannel* sacamos la sesión, y creamos esa variable de tipo *SpiceSession*. Por último creamos la variable que será el mensaje *per se* de tipo *SpiceMsgcMainSendStatistics (*ver figura 5*)*. Este tipo de mensaje es creado por mí y es una estructura con los siguientes parámetros:

- Un entero sin signo, que señaliza el tamaño del vector que vamos a enviar.
- Un puntero a un entero sin signo, que señaliza el comienzo del vector.
- Un entero sin signo, que señaliza la marca de tiempo.

```
typedef struct SpiceMsgcMainSendStatistics{
    uint32_t stats_size;
    uint64_t timestamp;
    uint64_t *stats_data;
} SpiceMsgcMainSendStatistics;
```

Figura 5: Estructura del mensaje

Siguiendo a la función, lo siguiente es recoger el dato de tamaño y guardarlo en buffer circular, en la posición que corresponda. Actualizamos la posición siguiente y si es el final del buffer, entonces creamos el mensaje y lo enviamos. Para eso creamos una variable *SpiceChannel* y gracias a la función *spice_session_lookup_channel*, obtenemos el canal de *Main*. Ahora añadimos al mensaje los tres parámetros correspondientes. Añadimos el canal y el tipo de mensaje a la variable de tipo *SpiceMsgOut.* El tipo de mensaje ha sido añadido al enumerado de la cabecera, con el nombre de *SPICE_MSGC_MAIN_SEND_STATISTIC* por lo que simplemente añadimos este nombre que corresponderá a un número. Y por último enviamos el mensaje, pasándolo por el *marshaller*. El *marshalling* o serialización es un proceso de codificación de un objeto para que pueda ser enviado por una conexión de red

```
if(my_buf->head == my_buf->tail && client){
    SpiceChannel* channel2 = spice_session_lookup_channel(session, 0, SPICE_CHANNEL_MAIN);
    mss.stats_size = 8;
    mss.stats_data = my_buf->buffer;
    mss.timestamp = time;
    msg = spice_msg_out_new(SPICE_CHANNEL(channel2),SPICE_MSGC_MAIN_SEND_STATISTICS);
    msg->marshallers->msgc_main_send_statistics(msg->marshaller,&mss);
}
```

Figura 6: Código del store_in_buffer del cliente

El servidor recibirá el mensaje. En la función *main_channel_handle_parsed.* Gracias a que hemos añadido antes de enviar el mensaje un parámetro llamado tipo de mensaje, el case de esta función será capaz de diferencia este mensaje como uno del tipo *SpiceMsgcMainSendStatistics*. Este reconocerá el tipo de mensaje y pasará a crear el buffer con los datos que le han llegado. Creamos el mensaje con el formato antes mencionado de tipo *SpiceMsgcMainSendStatistics*. Desde este tipo de mensaje se saca el tamaño del mismo, y la marca de tiempo. Con el tamaño del mensaje podemos recorrer el mensaje y a su vez ir añadiendo a un vector todos los datos recibidos. Una vez tenemos el vector completo, llamamos a la función *store_in_buffer* que guardará esté en otro vector para tener una muestra más amplia. Este último vector es otro buffer circular pero protegido por un *mutex*. Esta protección es debida a que este buffer es leído por otra función que borra el mismo si se le pide y por lo tanto necesitamos acceder a ella por exclusión mutua. Finalmente guardamos el vector, la marca de tiempo y el tamaño. Por último desbloqueamos el hilo.

```
case SPICE_MSGC_MAIN_SEND_STATISTICS:{
    SpiceMsgcMainSendStatistics *statistics = (SpiceMsgcMainSendStatistics *)message;
    uint32_t size = statistics->stats_size;
    uint64_t time = statistics->timestamp;
    static uint64_t buffer[8];

    for (int i=0; i<size; i++ ){
        buffer[i] = statistics->stats_data[i];
    }
    store_in_buffer(statistics);
    break;
}
```

Figura 7: Código del main_channel_handle_parsed

Una vez que tenemos el vector guardado, el servidor solo devolverá esa información cuando se le pida explícitamente. Esta petición la realizará *Qemu*, a través de un comando de *QMP* (*Qemu Machine Protocol*). Qemu es un emulador de procesadores con capacidad de virtualización dentro de un sistema operativo. Este protocolo funciona con llamadas por terminal con formato *JSON* (*JavaScript Object Notation*). Creamos una petición con el nombre de *get_stats*, y con un parámetro llamado flag.

Cuando se haga la llamada desde *QMP* con el nombre de *get_stats* y el parámetro correspondiente, se entrará a la función de *Qemu qemu_spice_get_stats*. En esta función se crea el buffer y llamaremos a la función de Spice, llamada *spice_server_get_stats*. Esta función únicamente hace de puente entre *Qemu* y Spice, para el intercambio de variables. *Qemu* hace la petición de estadísticas con una bandera y Spice devuelve el buffer.

```
void store_in_buffer( SpiceMsgcMainSendStatistics *data)
{
    pthread_mutex_lock(&ring_buffer_lock);
    unsigned int next = (unsigned int)(my_buf.head + 1) % (450);

    my_buf.buffer[my_buf.head].stats_data = data->stats_data;
    my_buf.buffer[my_buf.head].stats_size = data->stats_size;
    my_buf.buffer[my_buf.head].timestamp = data->timestamp;
    my_buf.head = next;
    pthread_mutex_unlock(&ring_buffer_lock);
}
```

Figura 8: Codigo de store_in_buffer del Servidor

La función encargada de enviar el buffer a la anterior nombrada se llama *get_statistics*. De lo que se encarga es de calcular el espacio en memoria del buffer, reservar el espacio y copiar el buffer. Además borra el buffer si recibe en la variable de entrada un valor que sea distinto de 0. Esta función también está protegida con un *mutex*, por la misma razón que la otra función está protegida. Una vez se han hecho la operaciones de copiado, desbloqueamos el hilo y por último devolvemos el buffer que había sido copiado.

```
SpiceMsgcMainSendStatistics* get_statistics(int flag)
{
    pthread_mutex_lock(&ring_buffer_lock);
    size_t size = 450*sizeof(SpiceMsgcMainSendStatistics);
    SpiceMsgcMainSendStatistics* buff = g_malloc(size);
    memcpy(buff, my_buf.buffer, size);
    if (flag != 0)
        memset(my_buf.buffer, 0, size);
    pthread_mutex_unlock(&ring_buffer_lock);
    return buff;
}
```

Figura 9: Codigo de get_statistics

Con todo esto tenemos finalmente una herramienta nativa que nos permite ver el número de bytes que son recibidos o enviados por el cliente cada segundo.

Capítulo 4

Configuración de los escenarios:

En todos los casos, las pruebas serán hechas en máquinas virtuales recién instaladas y creadas de la misma forma y con la misma imagen del S.O, el cual será un *Windows 7*. El cliente desde donde se recogerán los datos será también un *Windows 7.*

4.1. Spice

En el caso del protocolo Spice, se utilizará flexVDI. FlexVDI es una solución de escritorio remoto, que ofrece todo lo necesario para funcionar. Usaremos tanto la parte del servidor (con el *manager* incluido), como la del cliente (*dashboard* y cliente). Dado que el proyecto está hecho conjuntamente con flexVDI, la infraestructura ya está montada (servidor con el *manager*). Para la parte del cliente hay que instalar el *dashboard* que ofrece flexVDI. El *dashboard* es una aplicación que permite crear y configurar el servidor a través del *manager*, además de para la creación de máquinas virtuales y la configuración de políticas para el posterior uso desde el cliente. Desde este mismo, creamos una máquina virtual *Windows 7*, que será la que hará de *guest*. Configuramos las políticas, para que el *manager* nos de esa máquina cuando nos conectemos desde el cliente. Una vez esto está hecho, instalamos el cliente ofrecido por flexVDI en la máquina de pruebas. El cliente es un programa, que nos ofrece la posibilidad de conectar a un *manager*. El *manager*, será el encargado de darnos el *guest* que hayamos configurado. Con todo esto hecho, conectamos y directamente arrancaremos la máquina virtual.

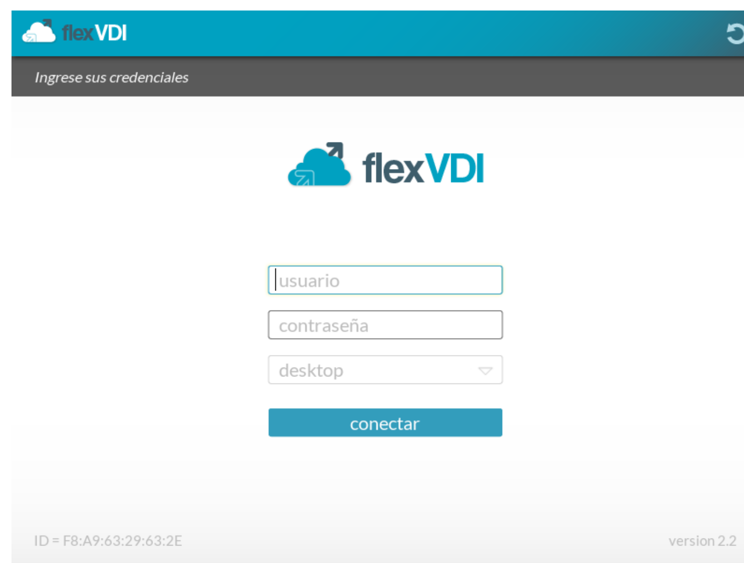


Figura 10: Inicio de sesión en flexVDI

4.2. PCoIP

En el caso del protocolo PCoIP, necesitamos las herramientas de *VMWare*. *VMWare* también ofrece todo lo necesario para funcionar. Para la parte del servidor y *manager* usaremos *ESXI Host* (sistema operativo independiente con la funcionalidad de hipervisor o monitor de máquina virtual, esto es una plataforma que permite el control de las maquinas virtuales). Esté se ha instalado en otro de los servidores cedidos por flexVDI. Una vez instalado en el servidor el *manager*, pasamos a la parte del cliente. En el cliente instalamos el equivalente a flexVDI del *dashboard* llamado en este caso *Vsphere client*. Desde aquí necesitamos crear una máquina virtual, la cual hay que configurar adecuadamente. Para configurarla, una vez tenemos instalado *Windows 7* debemos añadirle una dirección ip estática, para después instalar un plugin que ofrece *VMWare* para conectar el guest con el cliente directamente. Una vez hecho todo esto solo falta instalar el programa para el cliente, que en el caso de *VMWare* se llama *Horizon View*. Por último, para acceder a la máquina virtual lanzamos el programa *Horizon View*, ponemos la dirección ip que le habíamos puesto a esta, y accedemos a ella (la cual ya debería estar previamente arrancada).

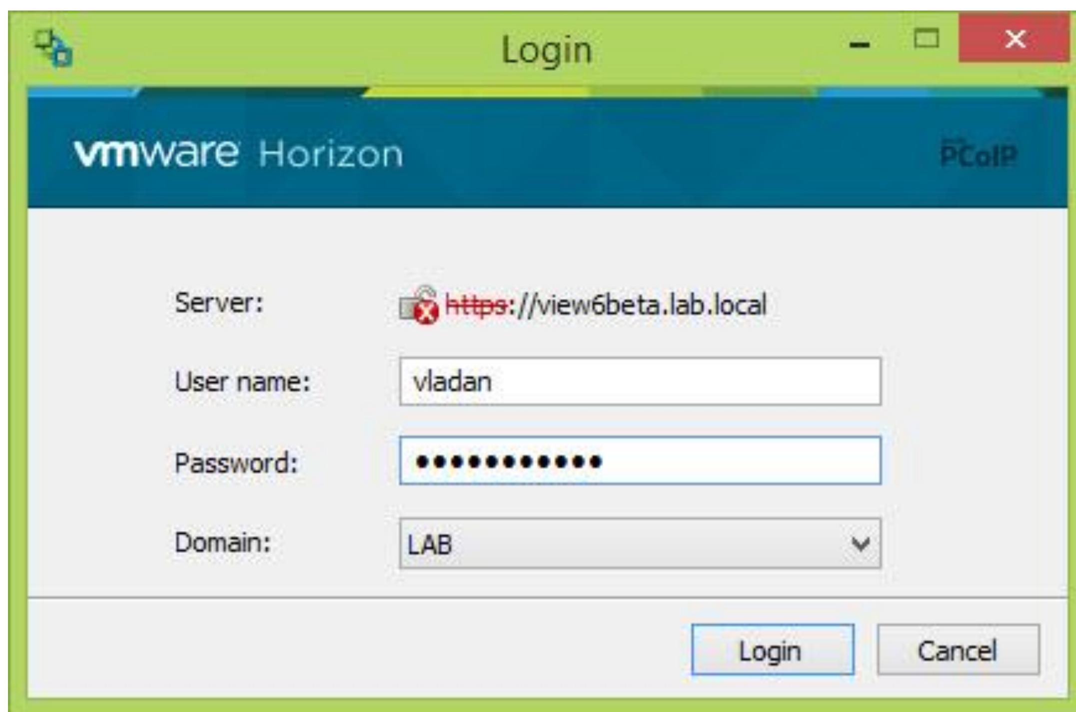


Figura 11: Inicio de sesión en VMWare Horizon View

4.3. RDP

En el caso del protocolo RDP, necesitamos las herramientas que ofrece *Windows*. En este caso la configuración es mucho más sencilla. Necesitamos haber creado una máquina virtual con cualquiera de las dos herramientas anteriormente explicadas. Una vez creada y con *Windows 7* instalado, también con esa herramienta accedemos a ella y configuramos el acceso remoto que ofrece *Windows*. Además necesitamos asignarle una dirección ip estática al igual que en el caso anterior. Con estas configuraciones hechas, gracias a que *Windows* lleva por defecto el programa para este protocolo, solamente necesitamos acceder a él desde nuestra máquina cliente y conectarnos a la ip que le hayamos puesto (la maquina tiene que estar previamente arrancada).

Figura 12: Inicio de sesión en RDP

4.4. ICA

En el caso del protocolo ICA, necesitamos las herramientas ofrecidas por *Citrix*. En este caso la configuración del escenario es la más costosa de las cuatro debido a que hay que instalar más máquinas con herramientas desconocidas por mí hasta el momento. Se necesita primeramente crear una máquina virtual de la misma forma que las hechas anteriormente. Pero antes de eso, necesitamos dos máquinas con Windows server. La primera que haga de *Active directory* y la segunda que haga de *Deliver controler*. Así pues creamos una máquina virtual con *Windows server* y la configuramos como *Active directory*, del cual esta máquina será administrador del dominio. Una vez tenemos creado el *Active directory*, creamos la segunda máquina virtual con *Windows server*. Esta segunda máquina será la que hará de *Deliver controler*, esto quiere decir que será la encargada de ofrecer las máquinas virtuales a los usuarios que se la pidan. Primeramente antes de instalar nada, necesitamos añadir está máquina al dominio del *Active directory*. Para más tarde configurar esta, necesitamos instalar *Citrix studio*, la herramienta ofrecida por *Citrix*. Desde esta herramienta, debemos crear un grupo de entrega, este grupo será asociado a uno o varios usuarios. Creamos paralelamente un catálogo de máquinas al que añadiremos la máquina virtual que hará de *guest*. Este catálogo es añadido al grupo de entrega. Posteriormente la máquina que hace de *guest*, la añadimos al dominio e instalamos también *Citrix studio*. Desde la máquina del cliente instalamos *Citrix receiver* y finalmente accedemos desde un navegador a la dirección del *Deliver controler*, instalamos el plugin y accedemos a la máquina con un usuario del dominio.

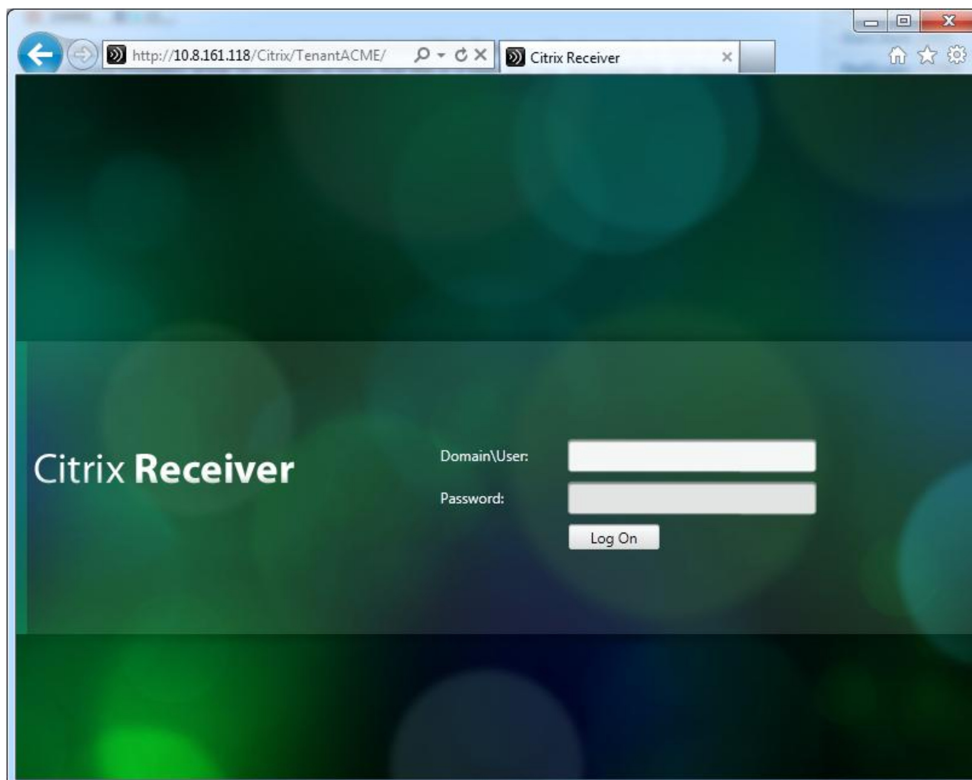


Figura 13: Inicio de sesión en Citrix

4.5. Finalización e instalación de los programas usados en las pruebas

Después de crear todas las máquinas correspondientes, instalamos el navegador que será usado para las pruebas, el cual será *Mozilla Firefox*. Se eligió este navegador debido a que es gratuito y usa menos recursos (menos memoria *Ram*).

Ahora que tenemos los cuatro escenarios creados y operativos, necesitamos crear una macro que haga siempre los mismos movimientos de ratón. Usamos un programa gratuito en *Windows 7* llamado *Pulover's macro creator*. Con este programa lo que hacemos es grabar tanto los movimientos como los clics o movimientos de la rueda del ratón. Estos movimientos se graban en el programa y se pueden modificar tanto el tiempo entre acciones, como la acción en si misma. Más tarde podemos reproducirlos en cada escenario simplemente dándole al botón de *play*.

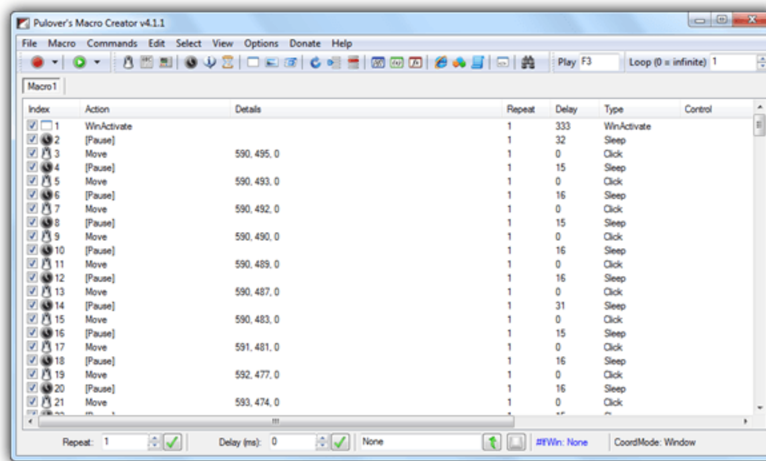


Figura 14: Pullover's Macro Creator

Para limitar el ancho de banda usamos una máquina con *Fedora*. Esta máquina tiene dos interfaces de red y gracias a esto la usaremos como *proxy*. Esto quiere decir que la conexión entrará por un interfaz y saldrá por el otro hasta llegar a la otra máquina. Usaremos la herramienta que tiene Linux, *Linux Traffic Control*. Como la herramienta de *Linux* es buena limitando el tráfico saliente pero no tan buena con el entrante, ya que del tráfico entrante no tenemos un control en el envío. Por lo tanto limitaremos ambas conexiones para hacerlo más real.

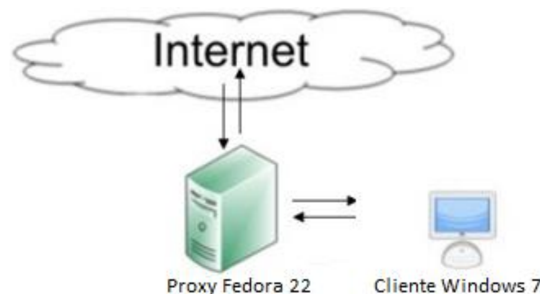


Figura 15: Escenario de pruebas

Capítulo 5

Análisis del uso de ancho de banda

Una vez tenemos las herramientas métricas operativas, es decir con los cambios efectuados en *Wireshark* y en el código nativo, finalmente se decide que para un análisis más homogéneo se usará *Wireshark* para todos los protocolos. Por lo tanto se procede a crear todos los escenarios.

5.1. Puesta en marcha de las macros

Se crean los escenarios con los parámetros de conexión de 256Kbps, 512Kbps, 1Mbps y 10Mbps. Estos parámetros nos dan una muestra significativa de lo que ocurre en conexiones con distintos anchos de banda. Como estamos interesados en los escenarios con anchos de banda limitados, se ha hecho más hincapié en las conexiones menores de 1Mbps. Una vez creados estos escenarios se usa la macro anteriormente comentada con cada protocolo. Esta macro consiste en abrir el navegador *Mozilla Firefox* y hacer una serie de acciones. Estas acciones son: poner el nombre de flexVDI en la barra del buscador de *Google*, entrar en el primer resultado (que es la página web de flexVDI), esperar que cargue, entrar en la pestaña de blog, movernos por la página y salir. Medimos el uso del ancho de banda únicamente desde que se abre *Mozilla Firefox* hasta que se cierra. Se decidió hacer esta macro ya que era simple y nos da una imagen del uso de ancho de banda de un usuario habitual.

Estas pruebas se hicieron también con dos macros más. Una era viendo un video solamente. Y la otra consistía en acceder al *wordpad* de *Windows* y escribir una serie de caracteres para finalmente cerrar la aplicación. Pero viendo los resultados obtenidos con las tres macros, decidimos hacerlo con la web por dos razones. La primera es que el video en entornos en los que se usa VDI no se usa tanto como podría ser la navegación web y además la transmisión de video y su reproducción tiene unas características y una problemática muy peculiar (como se muestra en el proyecto de Martin Hagström [15]) que requerirían un análisis y unas soluciones diferentes a las propuestas en este trabajo. La segunda es que al comparar el uso de ancho de banda del *wordpad* con la navegación web, esta última tenía mucho más margen de mejora. En el caso del *wordpad* solamente había 100KB de diferencia entre Spice y el siguiente protocolo en el caso con más diferencia.

La prueba de navegación web fue la seleccionada y se hizo la prueba en cada escenario varias veces para asegurarnos de que devolvían datos deterministas. La prueba simplemente consistía en darle al botón de iniciar teniendo el *Wireshark* escuchando. Así que finalmente se obtuvieron los siguientes resultados.

5.2. Análisis objetivo

En la figura 16 tenemos representado en el eje de ordenadas el consumo total de datos en la sesión entera. En el eje de abscisas tenemos el ancho de banda disponible en esa prueba. En cada ancho de banda tenemos los diferentes datos para cada protocolo cada uno con un color. Viendo la gráfica podemos primeramente ver que el protocolo Spice es el que más consume de los tres con bastante diferencia. Además del consumo de ancho de datos, podemos ver como los protocolos hacen una adaptación a su ancho de banda cuando la conexión baja de 10 Mbps, excepto en el caso de Spice. Aunque en la figura 16 se pueda llegar a pensar que sí que hace algún tipo de adaptación es por cómo trabaja *Qemu*, que se envían menos datos. Esto ocurre porque la cola de datos de la imagen se llena y *Qemu* decide vaciarla con la reducción de envió de datos que esto conlleva.

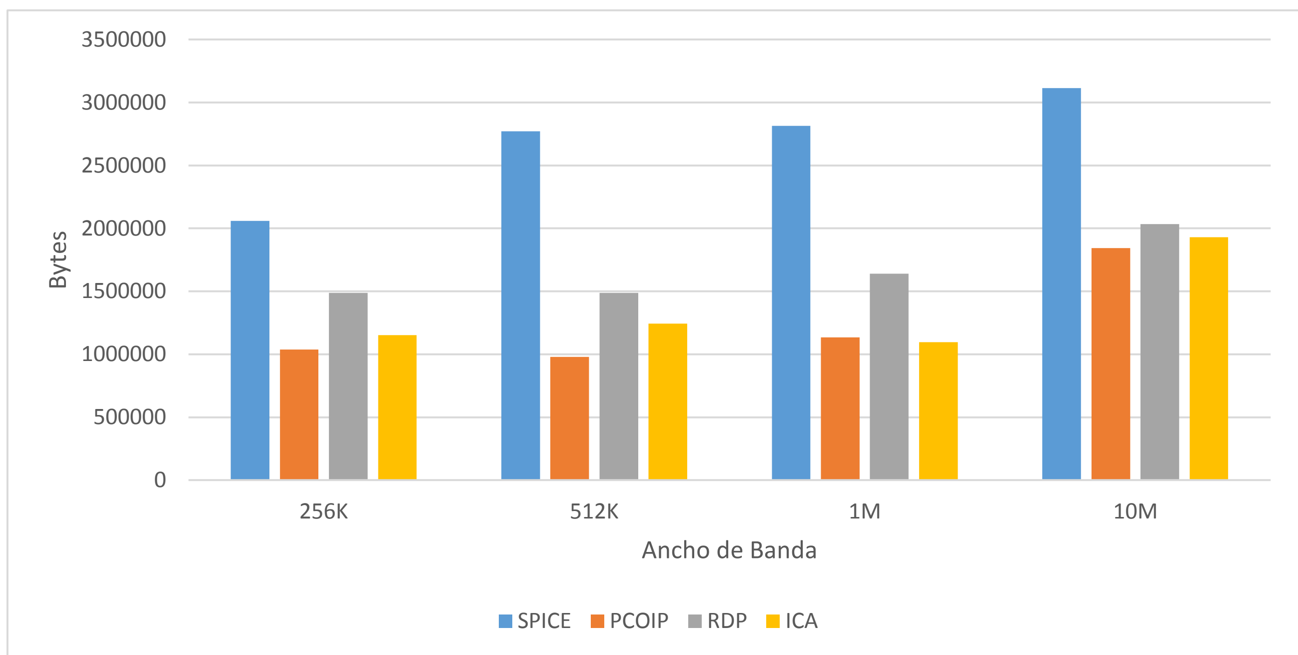


Figura 16: Consumo total de datos para los cuatro protocolos

Al ver más en profundidad con un ejemplo de ancho de banda con una limitación más agresiva, como es el de 256Kbps, y lo miramos segundo a segundo, obtenemos la figura 17. En esta grafica tenemos en el eje de ordenadas el ancho de banda y en el de abscisas tenemos los segundos. Si interpretamos los picos, podemos ver como el primer pico se corresponde a abrir el navegador *Mozilla Firefox*.

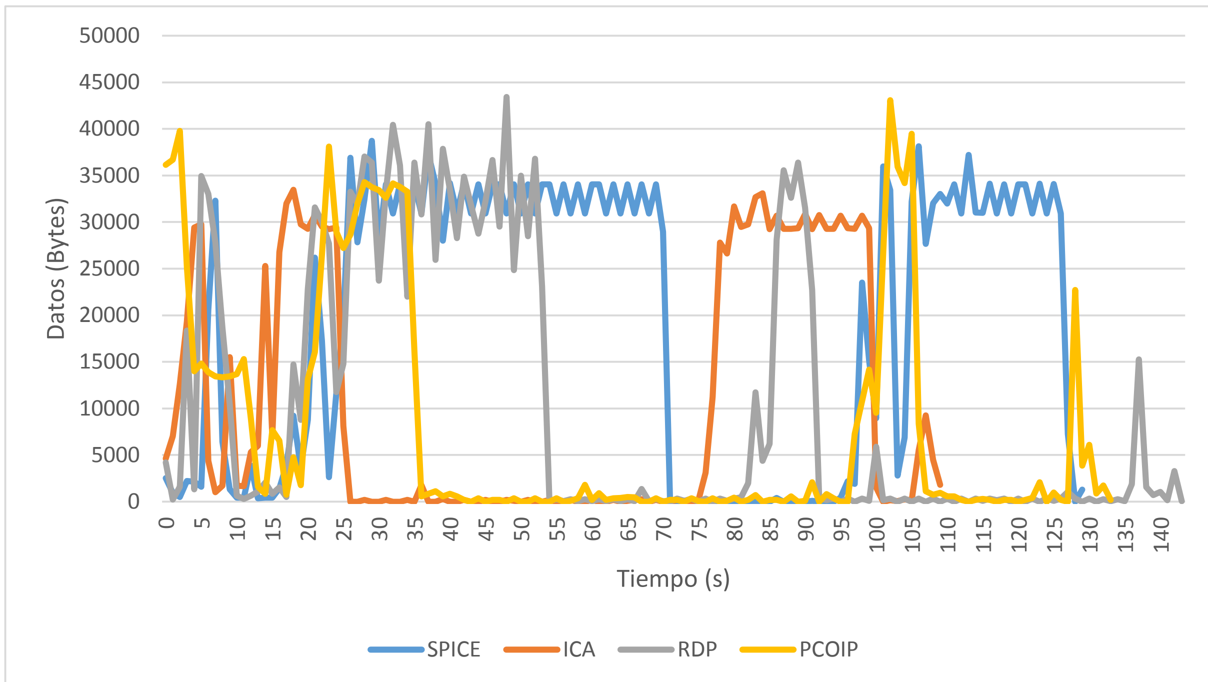


Figura 17: Uso del ancho de banda por segundo

El segundo pico no podemos apreciarlo del todo pero, podría corresponder a cuando damos al enter en el buscador *Google*. El tercer pico (en algunos protocolos podría llamarse meseta), se corresponde a la carga de la página de flexVDI, la cantidad de tiempo que le cuesta es debido a que la propia página tiene una animación. El cuarto pico es la carga del blog de flexVDI y el movimiento que sigue con el *scroll*.

Como se puede observar, el tiempo que está cada protocolo enviando a máxima velocidad es directamente proporcional al uso de datos final. En concreto Spice está el que más tiempo de todos los protocolos enviando a máxima velocidad. Esto es debido en mayor parte a que los demás protocolos se adaptan mejor a este escenario. Como los demás protocolos son privativos, no se puede ver que es lo que realmente están haciendo para adaptarse. Aun así en el caso de ICA se puede apreciar un aumento considerable de la compresión de imagen debido a la disminución de la calidad de imagen. Por otro lado en el caso de PCOIP se puede apreciar una disminución del envio de imágenes, debido al acortamiento en la animación de la página web. Se hicieron y probaron las dos soluciones en Spice, comprimiendo más como es el caso de ICA y enviando menos imágenes como es el caso de PCOIP, además de una solución que juntaba a las dos.

5.2. Análisis subjetivo

Una vez hecho el análisis objetivo mirando el uso de datos de cada protocolo, analizamos subjetivamente el comportamiento de los mismos. Se hizo un análisis de cada uno de los protocolos comparándolos unos con otros y entre ellos mismo sin restricción de ancho de banda.

- **Spice:** En el caso de Spice, se aprecia una notable disminución de la interactividad. El caso más grave ocurre cuando se carga la página web. En este caso pasa mucho tiempo desde que empieza a cargar hasta que está totalmente cargada, notándose incluso como la animación se mueve poco a poco. Desde el punto de vista de calidad en la imagen, se nota muy poca diferencia. Se ve bien, pero esto es más algo malo visto el uso de ancho de banda, ya que necesita más tiempo para cargar.
- **RDP:** En el caso del software de *Windows*, se nota también una disminución de la interactividad pero no tan grave como la de Spice. La animación en este caso está totalmente desaparecida, directamente sale la imagen final. En el mismo caso de la carga de la página web, el tiempo de carga es grande, pero no lo es tanto como en el caso de Spice. La calidad en la imagen es similar a Spice.
- **PCOIP:** En el caso de PCOIP, la interactividad se ve disminuida pero es el que mejor sale parado de los cuatro. Si no se hubiera probado sin restricción podríamos pensar que el programa funciona correctamente. Responde rápido y no tarda mucho en cargar la página web. La animación se ve acortada pero todavía se puede apreciar el movimiento original de la misma. En términos de calidad no ve muy alterada y está al nivel de los dos anteriores protocolos.
- **ICA:** En el último caso, el protocolo ICA tiene también una interactividad menor pero esta al mismo nivel que PCOIP. La animación cuando cargamos la página web se ve prácticamente completa, sin ningún acortamiento apreciable. En el caso de la calidad en la imagen, se puede apreciar una disminución bastante notable. En este caso es el que peor sale parado, viéndose muy claramente que la compresión ha sido muy agresiva.

Una vez analizados los cuatros protocolos, se puede apreciar que la gráfica objetiva se corresponde con las sensaciones subjetivas, esto quiere decir que al usarse menos ancho de banda funcionan mejor, por ejemplo las animaciones son más fluidas. En orden descendente empezando con el que mejor se comporta sería: PCOIP, ICA y RDP estaría a la par que Spice. En conclusión a estos analizados, se probaron las soluciones que se suponen que usan PCOIP e ICA y en última instancia una solución hibrida de las dos.

Capítulo 6

Optimización del protocolo Spice y resultados

6.1. Método ICA

Primero se probó a aumentar la compresión. Viendo cómo actúa ICA, teóricamente si aumentamos la compresión estaremos usando menos ancho de banda, pero el algoritmo que está soportado y que más comprime, es *jpeg*. El valor por defecto del coeficiente de compresión que está implementando en este algoritmo cuando detecta que la conexión es de 10Mbps o menor, es de 85. Este valor se mantiene desde que te conectas hasta el final. Se aumentó el factor de compresión hasta 25 (100 es sin nada de compresión). Pero este algoritmo no es el mejor en términos absolutos y la diferencia que se apreció no era notable por dos razones.

La primera razón de que no era notable es debido al sistema de envió de imágenes que usa Spice. Cuando Spice crea una conexión y crea los canales, por el canal de *display* envía la primera y última imagen completas. Desde este momento, de todos y cada uno de los paquetes que se envían por este canal, ninguno es una imagen completa. Se envían trozos de la imagen que en muchas ocasiones no se pueden comprimir correctamente debido a que tienen trasparencias. Este caso era fácilmente apreciable en la barra del explorador de *Windows*, la cual nunca llegaba a comprimirse.

Se ha visto que de este modo y debido a la restricción de Spice que solo es capaz de comprimir en *jpeg*, no alcanzamos resultados aceptables. Se prueba entonces la otra solución propuesta.

6.2. Método PCOIP

Para llevar a cabo la solución de reducir el número de imágenes por segundo, hay que tener en cuenta que Spice no envía ningún tipo de imagen completa. Por esto lo que se hizo fue adaptar el procedimiento que usa PCOIP a cómo funciona Spice. Como no se pueden descartar imágenes y reducir de forma sencilla el número de imágenes por segundo, se decidió en cambio que dejará de enviar los trozos y enviará imágenes completas. Esto se hace así porque no se puede tampoco dejar de enviar los trozos, ya que sino la imagen resultante podría no tener sentido.

Como Spice sí que puede enviar una imagen entera (envía la primera), se cambió el envio de partes por envio de imágenes. Al crearse el canal de *display*, Spice inmediatamente envía la primera imagen, esto se hace en la función *on_new_display_channel_client*. En esta función se llama a la función que envía la imagen entera, la cual se llama *red_worker_push_surface_image*. Después, como se ha comentado anteriormente va enviando partes de la imagen conforme va cambiando. Estas partes se

llaman ítems, que vienen dados a través de *Qemu* y se guardan en una cola para enviarlos en cuanto hay ocasión. Además para que se vacíe la cola de ítems por enviar, hace falta llamar a la función *red_current_flush.* Estas dos funciones tienen que ser añadidas en la función encargada de manejar los ítems. Debido al tipo de variable que puede ser un ítem, además de al tipo de canal a que pertenece, se decidió hacerlo en la función donde convergen todos los ítems del canal de *display*. Esta función se llama *display_channel_send_item (*ver Anexo A*)*. De esta forma se consigue enviar una imagen cada vez que cambia algo de la pantalla.

Al hacer las pruebas con esta solución se vio que la mejora no era solo inexistente sino que todavía hacía uso de más datos. Esto es debido a que ahora además de enviar cuando cambiaba algo, enviábamos una imagen completa.

Se decidió entonces viendo el tamaño de la imagen y del ancho de banda disponible, que enviara una imagen cada segundo. Para ello añadimos un reloj en la misma función y se comprobó cuanto consumía en este caso. Además se comprobó de forma subjetiva como se veía. De forma objetiva y conforme al uso de datos, como era de esperar había bajado considerablemente. En cambio de forma subjetiva se apreciaba una bajada de interactividad aunque gracias a que la conexión no se saturaba no era demasiado grave.

6.3. Método hibrido

Para la segunda prueba, se decidió que ya que Spice usaba un sistema bastante inteligente de envío de datos discriminatorio, se haría una prueba con un formato hibrido entre el que había antes y el de una imagen completa. Esto quiere decir que, gracias a que se siguen recibiendo los ítems que

```
DrawablePipeItem *dpi = SPICE_CONTAINEROF(pipe_item, DrawablePipeItem, dpi_pipe_item);
Drawable *item = dpi->drawable;
RedDrawable *drawable = item->red_drawable;
SpiceRect rect;
rect = drawable->bbox;
if (checksend==TRUE){
    if (send_bottom < rect.bottom)
        send_bottom = rect.bottom;
    if (send_right < rect.right)
        send_right = rect.right;
    if (send_top > rect.top)
        send_top = rect.top;
    if (send_left > rect.left)
        send_left = rect.left;
}else{
    send_bottom = rect.bottom;
    send_right = rect.right;
    send_top = rect.top;
    send_left = rect.left;
}
send_size = ((send_bottom - send_top)*(send_right - send_left) * 32)/100;
red_current_flush(worker, 0);
checksend = TRUE;
break;
```

Figura 18: Código de la función display_channel_send_item

cambian, ahora se decide ir guardando la zona de intersección de los cambios y enviarla cada segundo. De esta forma estamos aprovechando los ítems que anteriormente enviaba, ya que llevaban las coordenadas de la región que cambiaba. Para hacer esto debemos leer ese ítem e ir guardando la región que va cambiando, para finalmente hacer una intersección de todas ellas. Cuando llegase el tiempo de enviar, en ese momento enviaríamos solo esa intersección ahorrándonos ancho de banda y pudiendo enviar más imágenes por segundo. Para hacer esto, primero obtenemos el *DrawablePipeItem* del ítem. Luego el *Drawable* del anterior y finalmente el *RedDrawable.* De este último sacamos las cuatro coordenadas del *bbox* y de ese el *rect.bottom*, el *rect.right*, el *rect.top* y el *rect.left*. Con las cuatros coordenadas comprobamos si las coordenadas están dentro de la anterior intersección, si no lo están, actualizamos con las nuevas.

Gracias a esta solución se está ahorrando mucho más, ya que si tenemos en cuenta que si antes al pasar el ratón por alguna pestaña podía cambiar, teníamos que enviar la pantalla entera. Con la nueva solución solo enviamos una parte que puede llegar al orden de más de 100 veces más pequeña.

Con esta solución se ha solucionado el problema de la interactividad y el problema del uso del ancho de banda, pero todavía el sistema no es adaptativo. El sistema no usará de forma eficiente el ancho de banda disponible, ya que enviará cada segundo, independientemente de que pudiera enviar antes.

6.4. Método hibrido con token bucket

Finalmente probamos la última solución, más compleja, pensando en cómo funciona un sistema de *token bucket*. Además se envían menos imágenes por segundo y se añade la compresión más agresiva con coeficiente de 25. Este sistema consiste en lo siguiente. Tenemos un cubo con una capacidad, la cual en nuestro caso es la capacidad de la conexión en un segundo divida para 40. Esto se hace para conseguir un ratio de 25 imágenes por segundo. Este ratio es el que comúnmente se usa en los videos comerciales (cine o televisión). Cada 40ms (que es el tiempo resultante de la anterior operación) se comprueba cuánto ocupa aproximadamente la sección de la imagen que vamos a enviar. Calculamos cuantos pixels tiene la imagen, sabiendo cuantos bits son necesarios para señalizar un pixel, y añadiéndole la compresión. La fórmula usada para calcular este tamaño es, el tamaño del cuadrado por 32 y divido por 100 que corresponde a la compresión aproximada que se produce cuando tenemos un valor de 25. Entonces hay cuatro casos posibles:

- Si está lleno el cubo y la imagen a enviar ocupa menos que el cubo se envía directamente.

- Si el cubo está lleno y la imagen a enviar ocupas más que el cubo, entonces se envía pero al cubo se le resta el tamaño de la imagen. Como será mayor que la capacidad del cubo este se pondrá en negativo.

-Si el cubo no está lleno, pero no está en negativo, enviamos solo si la imagen que hay que enviar es menor que el cubo.

-Si el cubo no está lleno, y además está en negativo, no se envía nada.

```
gboolean check(RedWorker* worker)
{
    if(checksend == TRUE){
        if (cubo == 10000){
            cubo -= send_size;
            red_worker_push_surface_image(worker, 0);
            checksend = FALSE;
        }
        else if (cubo > 0 && cubo < 10000 && send_size < cubo){
            cubo -= send_size;
            red_worker_push_surface_image(worker, 0);
            checksend = FALSE;
        }
        send_size = 0;
    }
    cubo += 10000;
    if (cubo > 10000)
        cubo =10000;

    return TRUE;
}
```

Figura 19: Código de la función check

Al final de estas comprobaciones al cubo se le rellena la capacidad correspondiente. Para hacer todo esto, primero necesitamos tener en un hilo un reloj para hacer las comprobaciones cada 40ms. Este reloj lo añadimos a la función principal llamada *red_worker_main*. En esta función que hay un for infinito, añadimos nuestro reloj con la llamada a la función *check*. La función check tiene por parámetro una variable de tipo *RedWorker.* Esta variable es necesaria para enviar la imagen, ya que contiene información crucial para ser enviada. Ya en la función de check tenemos una comprobación de la variable checksend, que es con la que hacemos la decisión de si tenemos que enviar o no. Esta comprobación se refiere a lo antes expuesto, solo queremos enviar si hay algo que enviar. La variable que usamos de comprobación llamda *checksend* solo se actualizara a True cuando entra a la función *display_channel_send_item.* Una vez que esta variable está a True, hacemos las comprobaciones del cubo. Si enviamos ponemos la variable *checksend* a False. Además en la función *display_channel_send_item,* seguimos teniendo la actualización de la región de convergencia.

```
long time;
static long time2;
struct timespec monotime;
clock_gettime(CLOCK_MONOTONIC, &monotime);
time = monotime.tv_nsec;
if(time2 == 0)
        time2 = time;
if((time2 + 40000000) < time || (time < time2)){
        check(worker);
        time2 = time;
}
red_push(worker);
```

Figura 20: Codigo de la funcion red_worker_main

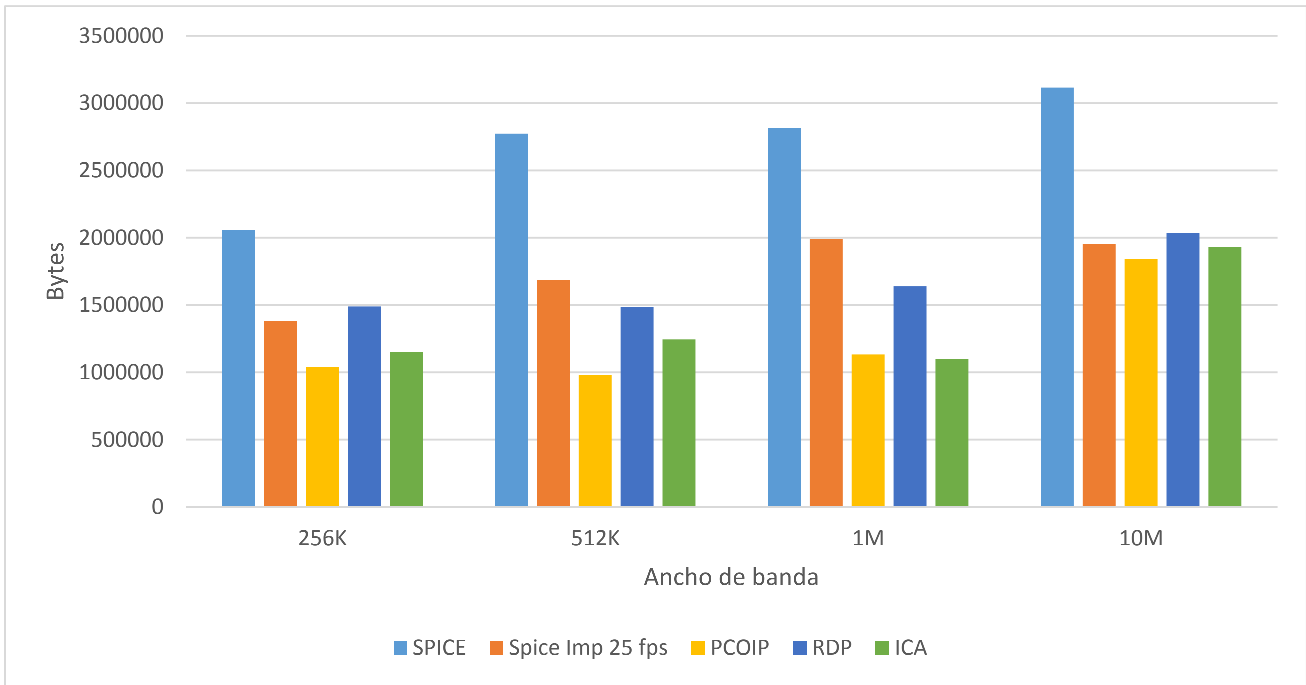


Figura 20: Consumo total de datos con la mejora

Con esto conseguimos que si hay que enviar muchas cosas pequeñas, se envíen sin perder en interactividad, ya que no colapsarán la conexión, pero si en algún momento se empiezan a enviar muchas imágenes grandes, entonces inevitablemente la cantidad de imágenes por segundo caerá, ya que la conexión se saturará, pero aun saturándolo nos adaptaremos a ella y enviaremos conforme a la capacidad de la conexión.

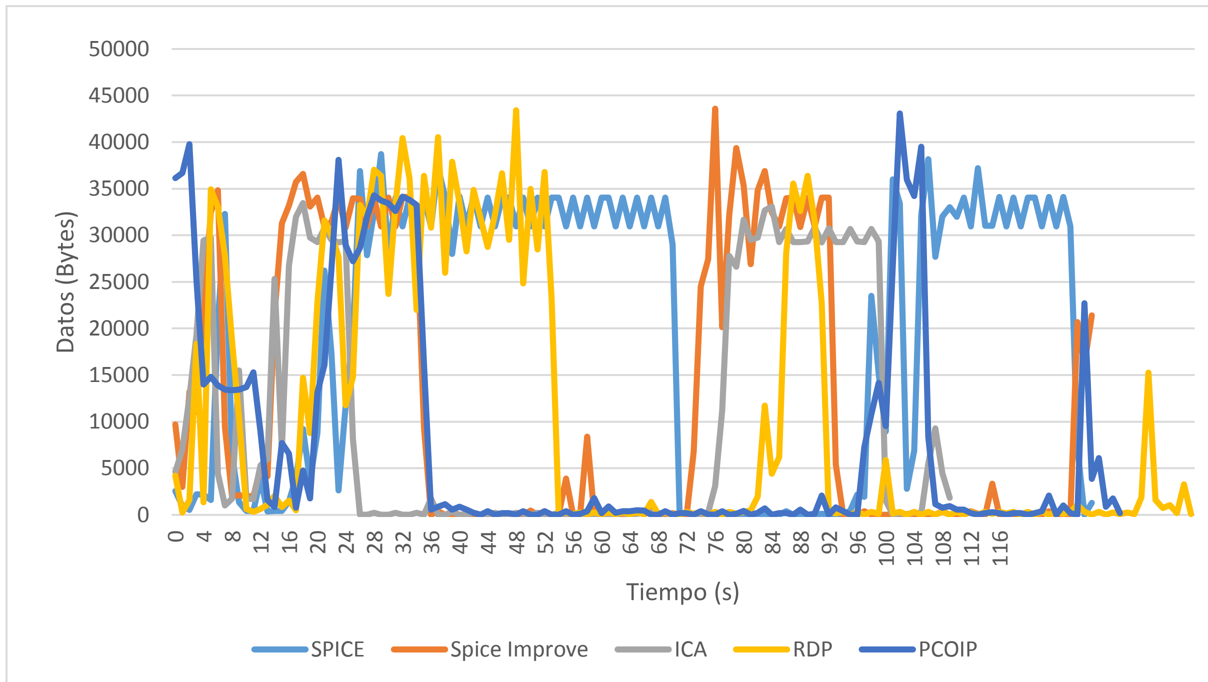


Figura 21: Uso de ancho de banda por segundo

6.5. Resultados

Con estas mejoras, se hicieron las pruebas anteriores y los resultados objetivos son los que podemos ver en la figura 20. Como podemos apreciar en dicha figura los resultados finales a 25fps, tienen una mejoría muy sustancial. Todavía no llegan al nivel de los otros protocolos excepto en el caso de 10Mbps donde están a la par. De todas formas se puede apreciar como conforme se baja el ancho de banda el uso de este se ve disminuido también.

Viendo la figura 21 con más detalle y segundo a segundo, podemos ver cómo en efecto el tiempo que se mantiene a máxima velocidad está reducido considerablemente.

De forma subjetiva también hay un cambio significativo. La transición de la página web donde antes podía tardar minutos ahora se produce en pocos segundos. Y en forma de interactividad no se nota apenas diferencia con los demás protocolos. Además se hizo una encuesta entre cuatro tipos de protocolos entre los compañeros de la empresa.

El material fue analizado con estos cuatro protocolos: Spice, RDP, PCOIP y el Spice con las mejoras. ICA finalmente no pudo ser analizado debido a problemas de la licencia. Esta encuesta fue realizada a un total de trece personas y en la que se valoró la interactividad y calidad de cada uno de los protocolos. La valoración de estos protocolos comprendía cinco valores:

- 5: Excelente
- 4: Muy Buena
- 3: Buena
- 2: Mala
- 1: Muy mala

En la figura 22 se puede ver como subjetivamente se mejoró la interactividad perdiendo algo de calidad.

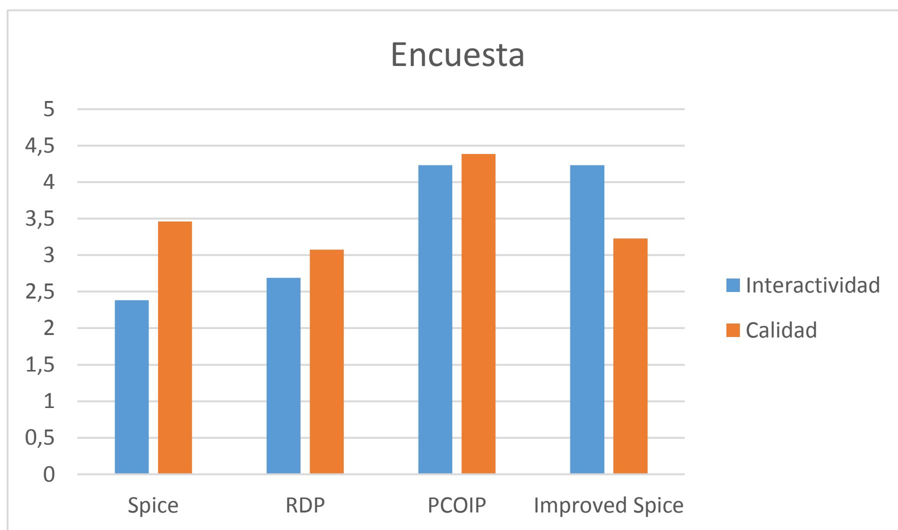


Figura 22: Encuesta subjetiva de calidad

Capítulo 7

Conclusión final y líneas futuras

Como conclusión final, se ha conseguido crear herramientas métricas capaces de medir los datos de forma correcta en cada uno de los protocolos y hacer una optimización del protocolo Spice con una mejoría que ha cumplido los objetivos con creces. En la franja de los 256Kbps la mejora subjetiva fue muy satisfactoria y en la parte objetiva salió una mejora del 32.94%. En la franja de los 512Kbps la mejora fue del 39.21%. En la de 1Mbps fue de 29.37%. En la de 10Mbps fue de 37.26%.

Aun con todas las mejoras todavía otras alternativas se comportan mejor y por ello hay todavía bastante margen de mejora. Como mejoras que se podrían hacer posteriormente, se encuentra la adaptación del ancho de banda dinámico. Es decir que si la conexión una vez creada fuera variando, cambiáramos el nivel de compresión y el límite de nuestro *token bucket*. Finalmente falta también enviar estas optimizaciones al grupo encargado del desarrollo de Spice para su implementación a nivel global.

BIBLIOGRAFÍA

[1] Página oficial de Spice. Disponible en http://www.spice-space.org/.

[2] Página oficial de flexVDI. Disponible en http://flexvdi.com/es/.

[3] RDP. Documentación disponible en https://msdn.microsoft.com/en-us/library/windows/desktop/bb892075(v=vs.85).aspx

[4] ICA. Documentación disponible en http://support.citrix.com/article/CTX116890

[5] PCoIP. Documentación disponible en https://techsupport.teradici.com/link/portal/15134/15164/Article/1456/Documentation-Center

[6] Lenguaje C.

[7] Entorno de desarrollo Eclipse. Disponible en http://www.eclipse.org/callisto/c-dev.php.

[8] Analizador de protocolos Wireshark. Disponible en https://www.wireshark.org/.

[9] Linux Traffic Control. http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html

[10] Página oficial de VMWare. Disponible en http://www.vmware.com/es.

[11] Página oficial de Citrix. Disponible en https://www.citrix.es/.

[12] Windows Server 2013.

[13] Windows 7.

[14] Fedora 22. Disponible en https://getfedora.org/es/workstation/.

[15] HAGSTRÖM, Martin. Remote desktop protocols: A comparison of Spice, NX and VNC. 2012. Disponible en http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A530960&dswid=-2988

[16] Libre Office. Disponible en https://es.libreoffice.org/.

ANEXO A. Funciones adicionales

Parte modificada de la función Spice_channel_flush_wire

```
static void spice_channel_flush_wire(SpiceChannel *channel,
                                     const void *data,
                                     size_t datalen)
{
    SpiceChannelPrivate *c = channel->priv;
    const char *ptr = data;
    size_t offset = 0;
    GIOCondition cond;
    //***************************************************************
    gint64 time;
    static guint64 aclen = 0;
    static gint64 time2 = 0;
    static ring_buffer my_buf = { {0}, 0, 0 };
    gint i;
    time = g_get_monotonic_time ();
    if (time2 == 0)
        time2 = time;
    if (((time2 + 1000000)  < time) && ((time2 + 2000000) > time)){
        time2 += 1000000;
        store_in_buffer(TRUE, &my_buf,aclen, channel,time);
        aclen = datalen;
    }else if((time2 + 2000000) < time){
        i = (time - time2)/ 1000000;
        for (guint j=1;j<i;j++){
            aclen = 0;
            store_in_buffer(TRUE,&my_buf,aclen, channel,time);
        }
        aclen = datalen;
        store_in_buffer(TRUE,&my_buf,aclen, channel,time);
        time2 += i * 1000000;
    }else{
        aclen += datalen;
    }
    //***************************************************************
```

Función store_in_buffer en la parte del cliente

```
void store_in_buffer(gboolean client, ring_buffer *my_buf, guint64 data,SpiceChannel* channel, guint64 time )
{
    SpiceMsgOut *msg;
    SpiceSession *session = channel->priv->session;
    SpiceMsgcMainSendStatistics mss;

    unsigned int next = (unsigned int)(my_buf->head + 1) % (8);
    my_buf->buffer[my_buf->head] = data;
    my_buf->head = next;
    if(my_buf->head == my_buf->tail && client){
        SpiceChannel* channel2 = spice_session_lookup_channel(session, 0, SPICE_CHANNEL_MAIN);
        mss.stats_size = 8;
        mss.stats_data = my_buf->buffer;
        mss.timestamp = time;
        msg = spice_msg_out_new(SPICE_CHANNEL(channel2),SPICE_MSGC_MAIN_SEND_STATISTICS);
        msg->marshallers->msgc_main_send_statistics(msg->marshaller,&mss);
    }

}
```

Case de la función main_channel_handle_parsed en la parte del servidor

```
case SPICE_MSGC_MAIN_SEND_STATISTICS:{
    SpiceMsgcMainSendStatistics *statistics = (SpiceMsgcMainSendStatistics *)message;
    uint32_t size = statistics->stats_size;
    uint64_t time = statistics->timestamp;
    static uint64_t buffer[8];

    for (int i=0; i<size; i++ ){
        buffer[i] = statistics->stats_data[i];
    }
    store_in_buffer(statistics);
    break;
}
```

Función store_in_buffer en la parte del servidor

```
static ring_buffer my_buf = { {{0}}, 0, 0 };
static pthread_mutex_t ring_buffer_lock;

void store_in_buffer( SpiceMsgcMainSendStatistics *data)
{
    pthread_mutex_lock(&ring_buffer_lock);
    unsigned int next = (unsigned int)(my_buf.head + 1) % (450);

    my_buf.buffer[my_buf.head].stats_data = data->stats_data;
    my_buf.buffer[my_buf.head].stats_size = data->stats_size;
    my_buf.buffer[my_buf.head].timestamp = data->timestamp;
    my_buf.head = next;
    pthread_mutex_unlock(&ring_buffer_lock);
}
```

Función get_statistics en la parte del servidor

```
SpiceMsgcMainSendStatistics* get_statistics(int flag)
{
    pthread_mutex_lock(&ring_buffer_lock);
    size_t size = 450*sizeof(SpiceMsgcMainSendStatistics);
    SpiceMsgcMainSendStatistics* buff = g_malloc(size);
    memcpy(buff, my_buf.buffer, size);
    if (flag != 0)
        memset(my_buf.buffer, 0, size);
    pthread_mutex_unlock(&ring_buffer_lock);
    return buff;
}
```

Función check en la parte del servidor

```
static void red_worker_push_surface_image(RedWorker *worker, int surface_id);
long int  contador = 0;
long int send_size = 0;
long int cubo = 10000;
gboolean check(RedWorker* worker)
{
    if(checksend == TRUE){
        if (cubo == 10000){
            cubo -= send_size;
            red_worker_push_surface_image(worker, 0);
            checksend = FALSE;
        }
        else if (cubo > 0 && cubo < 10000 && send_size < cubo){
            cubo -= send_size;
            red_worker_push_surface_image(worker, 0);
            checksend = FALSE;
        }
        send_size = 0;
    }
    cubo += 10000;
    if (cubo > 10000)
        cubo =10000;

    return TRUE;
}
```

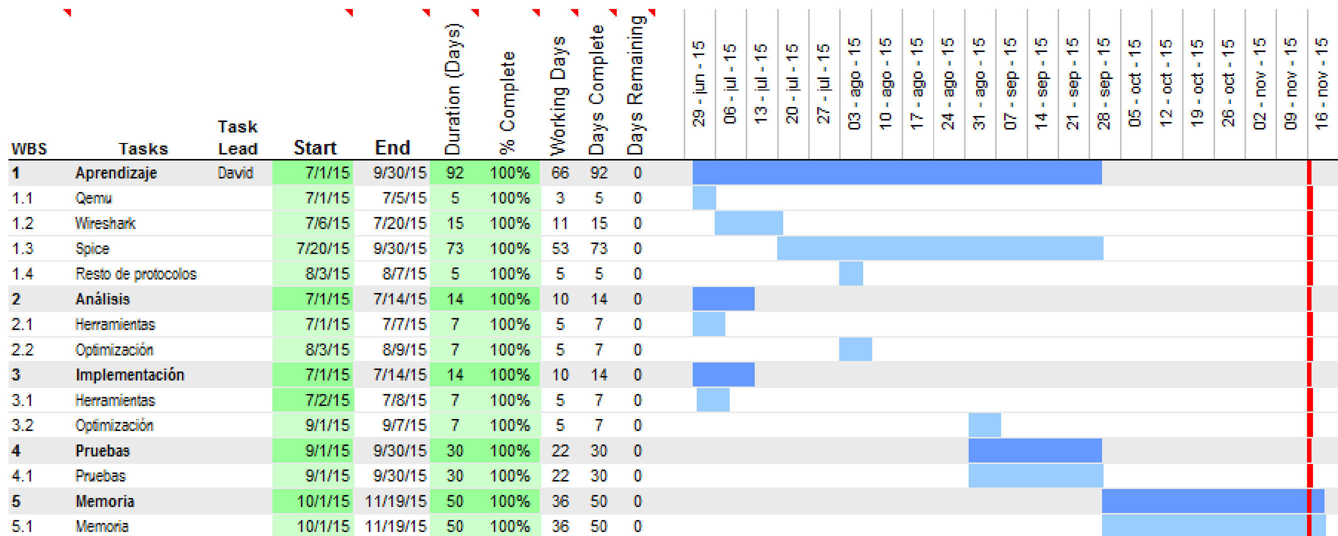
Parte modificada de la función display_channel_send_item

```
static void display_channel_send_item(RedChannelClient *rcc, PipeItem *pipe_item)
{
    SpiceMarshaller *m = red_channel_client_get_marshaller(rcc);
    DisplayChannelClient *dcc = RCC_TO_DCC(rcc);
    DisplayChannel *display_channel = DCC_TO_DC(dcc);
    RedWorker *worker = display_channel->common.worker;

    contador++;
    red_display_reset_send_data(dcc);
    switch (pipe_item->type) {
    case PIPE_ITEM_TYPE_DRAW: {
        DrawablePipeItem *dpi = SPICE_CONTAINEROF(pipe_item, DrawablePipeItem, dpi_pipe_item);
        Drawable *item = dpi->drawable;
        RedDrawable *drawable = item->red_drawable;
        SpiceRect rect;
        rect = drawable->bbox;
        if (checksend==TRUE){
            if (send_bottom < rect.bottom)
                send_bottom = rect.bottom;
            if (send_right < rect.right)
                send_right = rect.right;
            if (send_top > rect.top)
                send_top = rect.top;
            if (send_left > rect.left)
                send_left = rect.left;
        }else{
            send_bottom = rect.bottom;
            send_right = rect.right;
            send_top = rect.top;
            send_left = rect.left;
        }
        send_size = ((send_bottom - send_top)*(send_right - send_left) * 32)/100;
        red_current_flush(worker, 0);
        checksend = TRUE;
        break;
    }
```

ANEXO B

Diagrama de Gantt



WBS	Tasks	Task Lead	Start	End	Duration (Days)	% Complete	Working Days	Days Complete	Days Remaining
1	**Aprendizaje**	David	7/1/15	9/30/15	92	100%	66	92	0
1.1	Qemu		7/1/15	7/5/15	5	100%	3	5	0
1.2	Wireshark		7/6/15	7/20/15	15	100%	11	15	0
1.3	Spice		7/20/15	9/30/15	73	100%	53	73	0
1.4	Resto de protocolos		8/3/15	8/7/15	5	100%	5	5	0
2	**Análisis**		7/1/15	7/14/15	14	100%	10	14	0
2.1	Herramientas		7/1/15	7/7/15	7	100%	5	7	0
2.2	Optimización		8/3/15	8/9/15	7	100%	5	7	0
3	**Implementación**		7/1/15	7/14/15	14	100%	10	14	0
3.1	Herramientas		7/2/15	7/8/15	7	100%	5	7	0
3.2	Optimización		9/1/15	9/7/15	7	100%	5	7	0
4	**Pruebas**		9/1/15	9/30/15	30	100%	22	30	0
4.1	Pruebas		9/1/15	9/30/15	30	100%	22	30	0
5	**Memoria**		10/1/15	11/19/15	50	100%	36	50	0
5.1	Memoria		10/1/15	11/19/15	50	100%	36	50	0

Figura 23: Diagrama de Gantt

En la figura 23 podemos observar el diagrama de Gantt, se aprecia el tiempo empleado en el aprendizaje sobre todo del protocolo Spice debido a los continuos cambios que se iban realizando y de los que se necesitaba aprender nuevamente.