Proyecto Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación.

Validación y corrección del protocolo de transporte MPTCP (Multipath TCP) para Ns3

Autor: Antonio Manuel Del Toro Domínguez

Tutor: Juan Manuel Vozmediano Torres

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Grado
Ingeniería de las Tecnologías de Telecomunicación

Validación y corrección del protocolo de transporte MPTCP (Multipath TCP) para Ns3

Autor:
Antonio Manuel Del Toro Domínguez

Tutor:
Juan Manuel Vozmediano Torres
Profesor titular

Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2018

Proyecto Fin de Grado: Validación y corrección del protocolo de transporte MPTCP (Multipath TCP) para Ns3

Autor: Antonio Manuel Del Toro Domínguez

Tutor: Juan Manuel Vozmediano Torres

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2018

El Secretario del Tribunal

A mi familia

A mi novia

AGRADECIMIENTOS

Quiero agradecer a todo el mundo que me ha estado apoyando todo este tiempo, en especial, a mis padres que sin ellos no habría tenido la oportunidad de estar donde estoy hoy en día. Por apoyarme y animarme siempre a conseguir nuevas metas.

Por otro lado a Carmen, mi novia, por todo lo que me ha tenido que aguantar, las conversaciones de tecnología que se ha tenido que tragar sin gustarle absolutamente nada. Sin embargo, siempre está al pie del cañón apoyándome y aconsejándome en todo. Mis hermanos, por aguantarme todos estos años, en especial Fernando, que gracias a él me empezó a gustar la tecnología y es con el que comparto más aficiones.

Agradecer de corazón a todas las personas que han hecho posible que haya podido alcanzar esta meta que un día me marqué.

Antonio Manuel Del Toro Domínguez

Sevilla, 2018

RESUMEN

Este proyecto aborda el estudio y la validación de la implementación en Ns3 del protocolo de transporte Multipath TCP, más adelante haremos referencia a él como *MpTCP*. Es un protocolo que aún no ha sido estandarizado por el IETF, aunque existen varias normas que aún no han sido validadas. La más actualizada es la **RFC 6824** y es una de las fuentes que he seguido para la realización de este proyecto. La principal ventaja de *MpTCP* frente a TCP es el aprovechamiento de todas las interfaces físicas del dispositivo para la transmisión de datos en una misma conexión.

La validación se ha llevado a cabo sobre la implementación en un simulador de código abierto, como es NS3, que está escrito en C++ y sus interfaces en Python.

El protocolo *MpTCP* está implementado en el simulador como una **extensión** del protocolo TCP que ya estaba implementado en este, no siendo necesario ningún cambio a nivel de aplicación. Posteriormente, se han hecho simulaciones para validar el modelo planteado, además de obtener conclusiones acerca del nuevo protocolo.

Durante el desarrollo del proyecto ha sido imprescindible el uso de herramientas como: **Wireshark** para comprobar visualmente el funcionamiento de la implementación y **eclipse** para depurar la implementación de una forma más visual que desde la línea de comandos.

Finalmente, se prueba que el protocolo MpTCP realmente es una mejora de TCP, para realizar esta afirmación ha sido necesario analizar los resultados obtenidos en las simulaciones planteadas.

ABSTRACT

This project tackles the study and the validation of the implementation of transport protocol Multipath TCP, afterwards we will reference this as MpTCP. MpTCP is a protocol that have not been standardized by IETF yet, although any standards exist have not been validated yet, the most update standard is RFC 6824 and one of the guides I have been following to do make this project. The main advantage of MpTCP is the use of multiple physical interfaces on the device for make a data transfer in the same device.

The validation has been done in an open source simulator, which is ns3. Ns3 is written in C++ and his interfaces in Python.

The MpTCP protocol is implemented on the simulator as an extension of TCP protocol, which it has been implemented in the simulator already, not needing any change at application level. Later I have made simulations to validate the model, moreover, I have obtained conclusions about the new protocol.

During the project developer, the use of tools, as Wireshark to probe visually the properly working on the implementation and eclipse to debug the implementation visually instead of command line it have been required.

Finally, it is tested that the MpTCP protocol really it is an improvement of TCP, for make this statement it has been necessary analyze the results obtain by simulations

ÍNDICE

1 ÍNDICE DE FIGURAS

2 INTRODUCCIÓN

La duda es el principio de la sabiduría.

- Aristóteles -

2.1 Motivaciones

La motivación en realizar este Trabajo Fin de Grado se debe a la evolución que se está produciendo en las redes de datos. Actualmente, los dispositivos móviles como por ejemplo los *Smartphone* disponen de múltiples interfaces de red o NIC (*network interface controller*), una radio y otra wifi. Los centros de procesamiento de datos en sus núcleos para aumentar la disponibilidad disponen de redundancia y los grandes servidores que reciben muchos accesos disponen de múltiples direcciones desde las cuales pueden ser accedidos.

Ante todos estos cambios que se están produciendo el protocolo *TCP* aún sigue restringiendo su conexión a una sola ruta, es decir, dirección **IP (ip:puerto) origen a dirección IP (ip:puerto) destino**. Debido a esto no soporta funcionalidades como tener varias ubicaciones (*multihoming*), alta disponibilidad (ya que si se cae la interfaz física asociada a una de las direcciones entre las que se establece la conexión *TCP*, la conexión se perdería) y agregación de enlaces.

Todos los recursos disponibles en la red tienen que ser aprovechados con el fin de obtener un mayor rendimiento y hacer que las conexiones sean más robustas. Por estos motivos cobra gran importancia el protocolo *MpTCP*, ya que su función estrella es aprovechar todas las interfaces de red del dispositivo para transmitir datos simultáneamente por todas las rutas disponibles. Todos los datos transmitidos se reorganizarían en el destino formando un único flujo de datos *MpTCP*.

2.2 Objetivos

Los objetivos propuestos para la realización del trabajo fin de grado son los siguientes:

- Conocer en profundidad el protocolo de transporte que sustituirá a *TCP* ya que extiende las funcionalidades del mismo, y además añade otras nuevas que lo mejora.
- Validar una implementación del protocolo *MpTCP* en el simulador ns3 para comprobar que realmente cumple con las especificaciones. Además, añadir mejoras para completar la implementación y que esta sea lo más completa posible para que pueda ser usada, tanto con fines académicos, como de investigación.
- Documentar todo el proceso realizado y todas las mejoras implementadas con el fin que el proyecto pueda ser continuado y mejorado. En el apartado 7.2. se pueden encontrar mejoras propuestas para una continuación del proyecto.
- Comprobar a través de las simulaciones realizadas que efectivamente el protocolo *MpTCP* es mejor a *TCP* en cuanto a robustidad y eficiencia.

2.3 Planificación

En esta sección se comenta la planificación que se ha seguido durante la realización del trabajo:

- La primera tarea realizada ha sido es estudio del protocolo, para ello se ha seguido la RFC 6824 que es donde está definido. Además, se han utilizado otras fuentes como https://multipath-tcp.org/pmwiki.php, la implementación de *MpTCP* en el *kernel* de Linux.
- Una vez se han adquirido los conocimientos sobre el funcionamiento del protocolo se estudiaron las dos implementaciones de MpTCP que existen para el simulador. Posteriormente, se elige una de las implementaciones, esta decisión se fundamenta en el apartado 4.2.
- El siguiente paso ha sido la integración de la implementación con el simulador. Debido a la poca información existente este paso ha llevado cierto tiempo hasta que la implementación ha estado totalmente funcional.
- Se han realizado pruebas para comprobar la validez de la implementación. Además, se añaden nuevas funcionalidades a la implementación y se adapta para adecuarla a nuestro objetivo.
- Codificación de simulaciones para obtener resultados. Es en este punto donde se han obtenido conclusiones sobre el protocolo implementado.
- Tras finalizar se ha elaborado un *script* para facilitar la integración de los ficheros fuentes con la versión 3.19 del simulador.

2.4 Simulador Ns

El papel del simulador en el desarrollo de este trabajo fin de grado es primordial, pero el funcionamiento del mismo no se considera objetivo de estudio en este trabajo. Se conoce a Ns (network simulator) como un simulador de eventos discretos usados principalmente en ambientes educativos y de investigación. Es software libre ofrecido bajo la versión 2 de *GNU General Public License*.

La versión del simulador que se ha usado es Ns-3, que es la versión que actualmente está en desarrollo. Las versiones anteriores Ns-2 y Ns-1 ya no tienen soporte y no son compatibles con la última versión. Esta última versión del simulador surgió durante el año 2008 y fue desarrollada por investigadores de las siguientes instituciones: Universidad de Washington, Instituto Tecnológico de Georgia y grupo INRIA.

El objetivo del simulador es configurar entornos de simulación con el propósito de realizar investigaciones y enseñanzas académicas.

La implementación del protocolo *MpTCP* está realizada en la actualización 3.19 del simulador, en el **ANEXO A: Instalación ns3** se ha detallado paso a paso la instalación del simulador. Para más información se puede consultar su documentación en https://www.nsnam.org/.

3 ANTECEDENTES

El placer más noble es el júbilo de comprender.

-Leonardo da Vinci-

En esta sección se hace una breve introducción a los dos protocolos fundamentales en los que se basa el desarrollo de este trabajo fin de grado, *TCP* y *MpTCP*.

3.1 Protocolo TCP

Si bien el protocolo *TCP* (*Transmission Control Protocol*) es uno de los pilares básicos para la realización de este trabajo fin de grado no es motivo de estudio dentro del mismo, se expondrán unas nociones básicas con el fin de ayudar a la compresión. Para más información sobre el protocolo se puede consultar la norma donde se encuentra definido. RFC 793.

Es el protocolo de transporte por excelencia junto con *UDP* (*User Datagram Protocol*), usado por la mayoría de los sistemas existentes. Son los dos protocolos de transporte más usados en el modelo *TCP/IP*.

A continuación, se detallan conceptos relevantes para la comprensión del protocolo:

- Puertos: Identifican a aplicaciones en las máquinas donde se está usando el protocolo para que, de esta forma, más de una aplicación puedan estar usando a la misma vez el protocolo.
- Número de secuencia: son necesarios para identificar los datos que se han transmitido. Al iniciarse la sesión *TCP* se establece un número de secuencia inicial aleatorio en ambos extremos que en cada transmisión de datos se va actualizando.
- Número de acuse de recibo: se utiliza para asentir al otro extremo de la conexión que ha llegado cierta información, además, indica el próximo número de secuencia que espera recibir. Es utilizado junto con otros indicadores para identificar congestión en la red.
- Suma de control: es un campo utilizado para garantizar que la información que sea recibida es íntegra, ya que esta suma de control se calcula antes de enviar el segmento y tras recibirlo, debiendo resultar la misma.
- Ventana de transmisión: es el número de datos (medidos en bytes) que puede transmitir para no llenar el buffer del destino.

Se denomina **segmento *TCP*** al conjunto de bits que forman las unidades de datos del protocolo. Todos los segmentos llevan una cabecera que no es más que información necesaria para el funcionamiento del protocolo. La cabecera siempre se sitúa sobre los datos del protocolo.

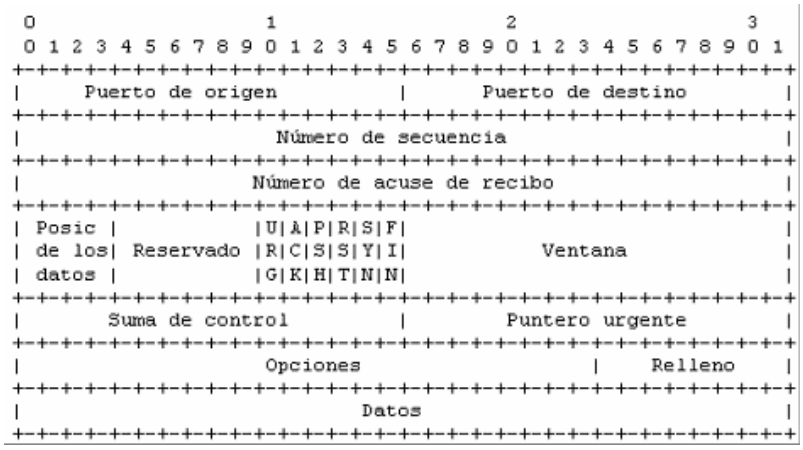

Ilustración 3.1: Cabecera *TCP*

3.1.1 Características

Las principales características de *TCP* son las siguientes:

- Orientado a conexión: Antes de transmitirse los datos, los dispositivos requieren el establecimiento de una conexión en el que se pondrán de acuerdo en la forma de transmitir la información.

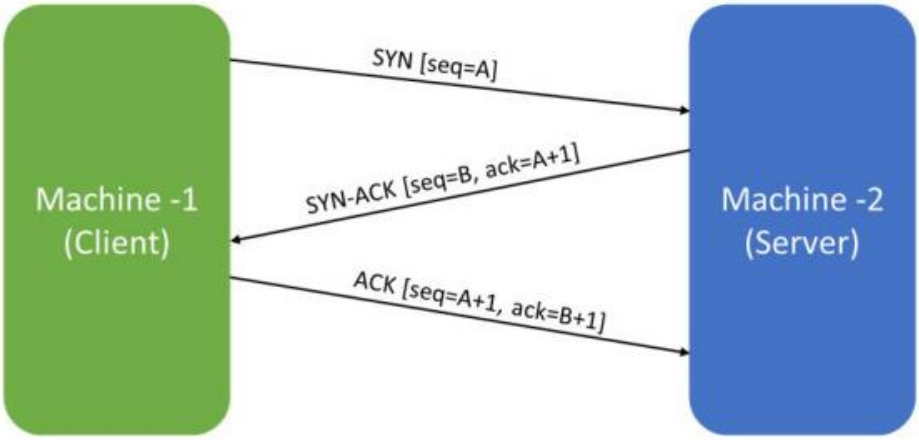

Ilustración 3.2 : Establecimiento sesión *TCP* (*three way handshake*)

- Bidireccional: Una vez que la sesión *TCP* ha sido establecida ambos dispositivos pueden enviar y recibir a la misma vez.
- Identificación de destino final: Cuando la sesión ha sido establecida, es identificada por un *socket*. De esta forma cada dispositivo puede tener múltiples sesiones activas sin que se produzca ningún conflicto.
- Control de congestión: El protocolo es capaz de determinar cuando la red está saturada, para ello hace uso de la realimentación de la red. Los principales indicadores de congestión en la red son:
    - Acuse de recibo (*ACK*): asiente al emisor de los datos que han sido recibidos y además indica el próximo número de secuencia que espera recibir.

- Tiempo Agotado (*Tout*): Según las características de la red se establece el tiempo que debería tardar en realizarse una transmisión.

- Servicio fiable: hace uso del acuse de recibo. De esta forma se asegura que toda la información que se ha enviado ha sido recibida. En caso contrario, se sabrá que parte de la información se ha perdido y será retransmitida.
- Cierre de comunicación acordado: Una vez las aplicaciones no tienen más datos para transmitir se ponen de acuerdo para terminar la comunicación

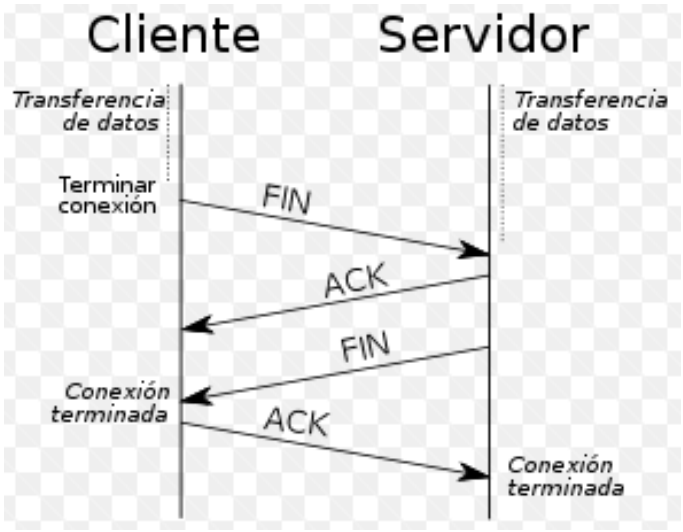

Ilustración 3.3 Cierre sesión *TCP*

3.2 Protocolo MpTCP

Multipath TCP es un protocolo de nivel 4 o transporte (en el modelo TCP/IP) que se implementa en el **campo de opciones** de su antecesor *TCP*. Dentro del campo de opciones el protocolo se identifica con el valor **30**. El objetivo principal de este protocolo es hacer que las conexiones *TCP* funcionen por distintas rutas simultáneamente, de esta forma sería como si una sesión *TCP* principal estuviera formada de **subsesiones** *TCP*.

Desde el punto de vista de la aplicación sería exactamente igual que *TCP* ya que se le ofrece la *API* del *socket* con la que la aplicación se comunicaría. Desde el nivel de aplicación solo sería visible un único flujo de datos al igual que en *TCP*.

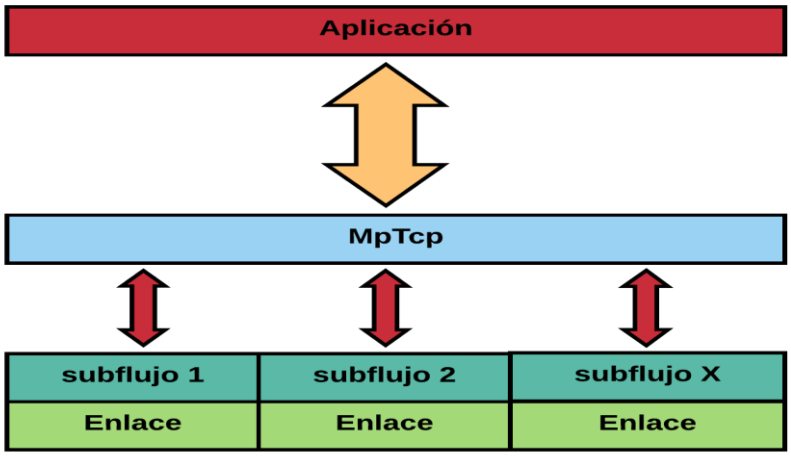

Ilustración 3.4: Comunicación *MpTCP* con aplicación

MpTCP fue definido por primera vez por el IETF en 2013 en la RFC 6824 aunque la versión más reciente de está RFC es del 4 de Marzo de 2018.

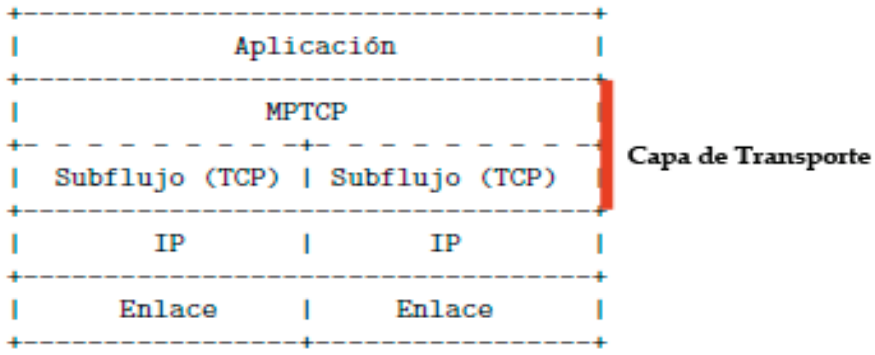

Ilustración 3.5: *MpTCP* en el modelo *TCP*/IP

Lo que propone *MpTCP* es aprovechar todas las interfaces de red disponibles en el dispositivo. De esta forma se establecería una única sesión MpTCP que enviara y recibiera segmentos de forma simultánea por todas las rutas disponibles hacia un mismo destino. De este modo, se aprovecharían todos los recursos del dispositivo a la hora de realizar una transmisión de datos.

3.2.1 Objetivos de diseño

Antes de avanzar en esta sección se desea hacer hincapié en la diferencia entre **subflujo** y **flujo** *MpTCP*.

Se entiende por **subflujo** como una conexión *TCP* normal que en el campo de opciones llevan las opciones necesarias para el funcionamiento de *MpTCP*. Los subflujos se establecen entre dirección IP/puerto origen y dirección IP/puerto destino.

Por otro lado, un **flujo** es un conjunto de uno o más subflujos. Los datos serán enviados por los distintos subflujos dependiendo del algoritmo empleado para gestionar el envio de información, de tal forma que todos los subflujos formarán un único flujo *MpTCP*. Las conexiones se establecen entre origen y destino, a ambos se les asigna un identificador al iniciar la comunicación.

De esta forma, aunque cada subflujo se establezca como una conexión *TCP*, todos serán establecidos entre los mismos identificadores de origen y destino.

En cuanto al diseño del protocolo se tuvieron en cuenta algunas premisas:

- El protocolo debía ser compatible con todos los mecanismos básicos de *TCP*.
- Cuando el protocolo estuviera en funcionamiento debía ser capaz de elegir la ruta más eficiente posible. En la implementación del protocolo se ha introducido un nuevo algoritmo que realiza esta selección, para ello se basa en el *RTT* de cada ruta disponible eligiendo la que tenga un valor menor.
- Una sesión *MpTCP* con un solo subflujo en ningún caso podría tener una eficiencia menor que una sesión *TCP*. Esto emula el peor de los casos que se podría dar.
- El protocolo ha de ser capaz de adaptarse rápidamente ante cambios en la topología. En el caso de que un enlace en el que se haya establecido un subflujo deje de estar disponible, el protocolo debe ser capaz de mantener la sesión y seguir enviando datos por los subflujos disponibles.
- Un subflujo no puede tomar más capacidad en un enlace que una conexión *TCP*. El objetivo de esto es no saturar los enlaces.
- El protocolo *MpTCP* debe tener compatibilidad hacia atrás, es decir, que si el host con el que se intenta conectar no soporta *MpTCP* se debe establecer una conexión *TCP* en su caso.
- La seguridad del protocolo deberá ser igual o mayor que la de TCP.

3.2.2 Funcionamiento

El objetivo de que *MpTCP* esté implementado en el campo de opciones de TCP es que no haya que realizar ninguna modificación a nivel de aplicación para soportar el protocolo Las opciones más utilizadas son las siguientes: [1]

- **OPT_MPC** {0x1E}: Indica si soporta *MpTCP*, se usa para establecer una sesión.
- **OPT_JOIN** {0x1F}: Se utiliza para añadir nuevo subflujo a una conexión previamente establecida.
- **OPT_ADD_ADDR** {0x20}: Anuncia nuevas direcciones a la sesión para indicar que también pueden ser alcanzados a través de ellas.
- **OPT_REMOVE_ADDR** {0x21}: Usada para eliminar direcciones. Su función es indicar que el host ya no está disponible en esa dirección, por lo que el subflujo se cerraría.
- **OPT_DSS** {0x22}: Se utiliza para la transmisión de datos, además indica cómo se deben mapear el flujo de datos contra el flujo de la sesión principal.
- **OPT_PRIO**: Opción para cambiar la prioridad entre los subflujos a la hora enviar información.
- **OPT_FAIL:** Notifica que se ha producido un fallo en el mapeo de los datos para obligar a la sesión a establecer un único subflujo

En la en la Ilustración 3.6 se puede apreciar la estructura de la cabecera *MpTCP*, se observa que es similar a la del protocolo *TCP*, Ilustración 3.1, con la salvedad del campo de opciones que es donde se implementa *MpTCP*. Tal como se menciona anteriormente, se puede comprobar que el valor presente en el campo de opciones es {**0x1e}** que en decimal se corresponde con el valor **30** que identifica al protocolo *MpTCP*. El campo ***subtype*** presente en la cabecera se usa para indicar el tipo de opción *MpTCP*.

[1]. El valor entre '{ }' hace referencia al valor hexadecimal de la opción.

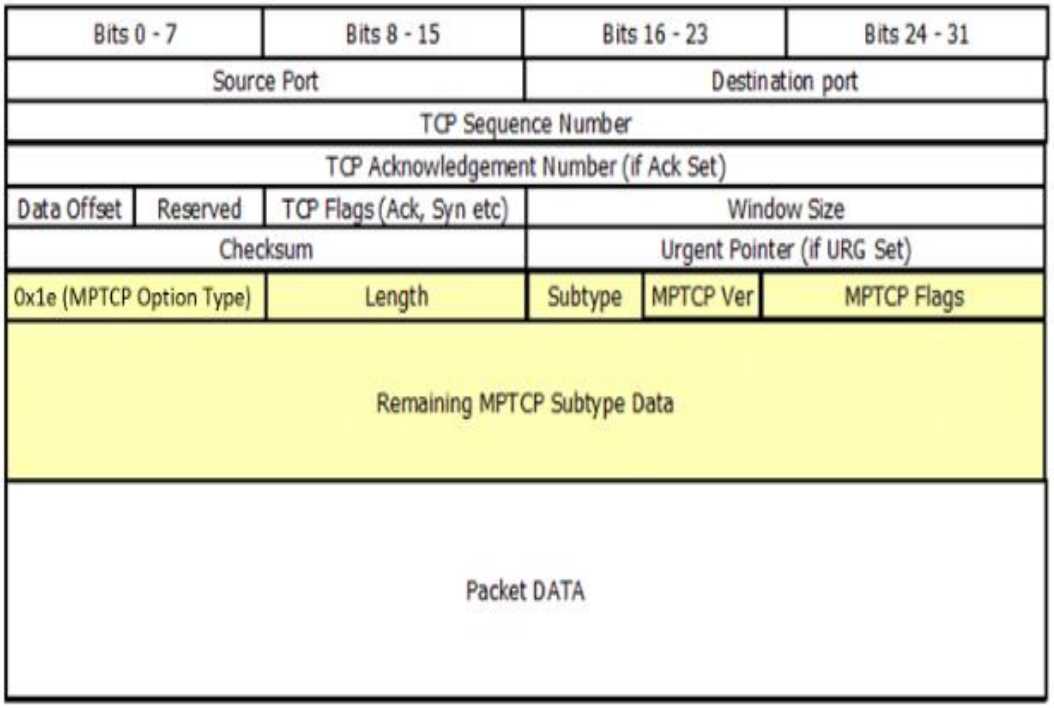

Ilustración 3.6: Cabecera *MpTCP*

3.2.2.1 Establecimiento de conexión

Se establece de la misma forma en la que se inicia una sesión *TCP* tal como se muestra en Ilustración 3.2, la única peculiaridad es que los paquetes **SYN**, **SYN/ACK**, **ACK** del *TWHS* llevan la opción **MP_CAPABLE** activa. Esta opción además de verificar si el host destino soporta *MpTCP* permite intercambiar información con el propósito de establecer subflujos adicionales.

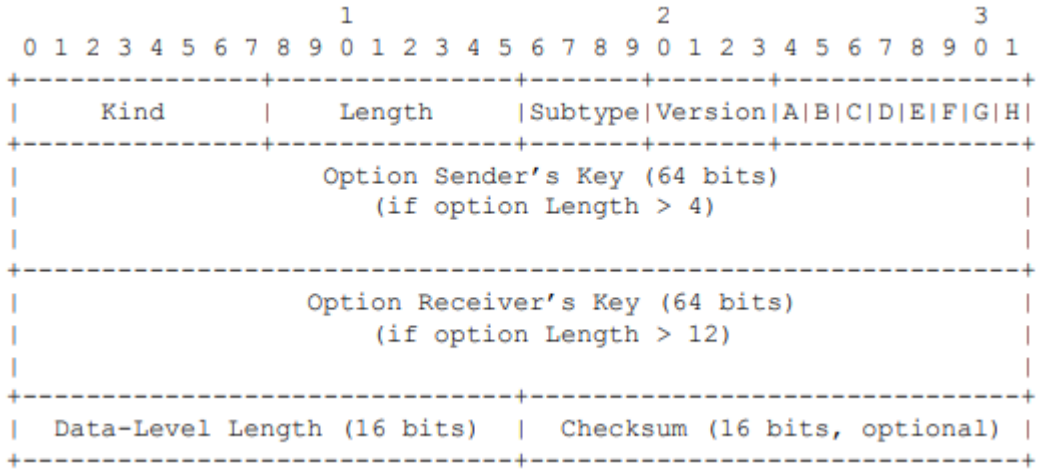

Ilustración 3.7: Opción MP_CAPABLE

En el caso de que el host destino no soporte *MpTCP* se deberá iniciar una sesión *TCP*, este comportamiento se conoce como compatibilidad hacia atrás o "*backward compatible"*. Se detecta que el destino no soporta *MpTCP*

cuando al recibir el **SYN/ACK** no se encuentra la opción **MP_CAPABLE** y por tanto es necesario establecer una sesión *TCP*. En su defecto puede que el destino soporte el protocolo pero no la red, debido a que no permita valores en el campo de opciones de *TCP*.

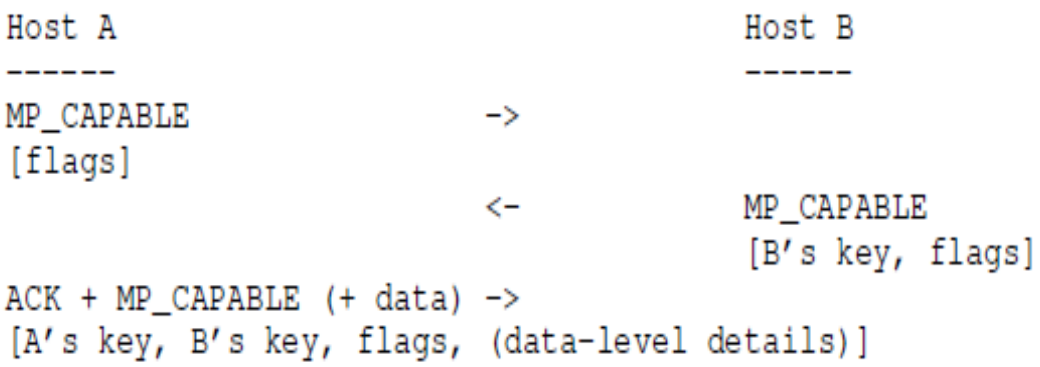

Ilustración 3.8: Establecimiento sesión *MpTCP*

En el establecimiento de la sesión se intercambian claves para identificar de forma única el origen y destino de la conexión. Será la única vez en la que las claves se intercambian sin cifrar, ya que, más adelante lo que se intercambiarán serán *tokens* generados a partir de estas claves.

Posteriormente, estas claves se usarán en el caso de querer añadir un nuevo subflujo, ya que será necesario la autentificación para realizar esta acción. Las claves deben ser únicas, puesto que el host debe almacenar la relación entre clave y *token* para identificar la sesión *MpTCP* sobre la que se está actuando en el caso de tener más de una activa al mismo tiempo. Este intercambio se puede observar en la Ilustración 3.8.

Se comprueba la situación de que no se genere el mismo *token* para distintas claves. En este caso no sabría sobre qué sesión *MpTCP* se está actuando.

3.2.2.2 Gestión de Subflujos

En este apartado se detallará el manejo de los subflujos una vez la sesión *MpTCP* se ha establecido correctamente.

3.2.2.2.1 Adición de subflujos

Una vez la sesión *MpTCP* ha sido establecida se pueden añadir nuevos subflujos, para ello se usa la opción **MP_JOIN**.

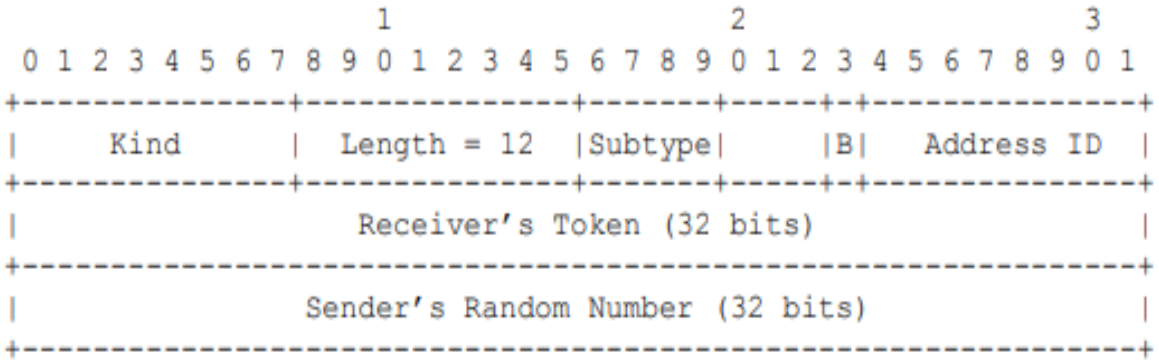

Ilustración 3.9 : Opción MP_JOIN para SYN inicial

El procedimiento para añadir un nuevo flujo es similar al del establecimiento. La única diferencia es que los paquetes intercambiados en el *TWHS* llevan la opción **MP_JOIN**. El *token* generado a partir de la clave intercambiada durante el establecimiento de la sesión es usado por el receptor para identificar a que conexión se está intentando añadir un nuevo subflujo.

El identificador de la dirección (*Address Id*) se usa para eliminar direcciones, ya que el paquete al pasar a través de dispositivos intermedios como Firewalls o NATs puede que su dirección original haya sido modificada. De esta forma se pueden eliminar las direcciones sin preocuparse cuál es la dirección origen en el destino. Cada host debe almacenar la relación entre *Address Id* y la dirección, así como la relación de: dirección local – dirección remota – subflujo asociado.

La opción **MP_JOIN** cuando va en el paquete que porta la bandera SYN, lleva activa la bandera B. Se usa para indicar si el subflujo que se quiere crear será un subflujo de resguardo (B=1), para transmitir cuando no haya ningún otro subflujo disponible, o será un subflujo normal que forme parte de la sesión actual (B=0).

El código HMAC es usado para la autentificación y se genera a partir de las claves intercambiadas en el establecimiento de la sesión y de números aleatorios (usados para evitar ataques de réplicas durante la autentificación). Hacer los intercambios de códigos HMAC en esta fase permite a los host tener un primer intercambio de datos aleatorios. De esta forma un atacante solo tiene una oportunidad de averiguar el código HMAC correctamente.

En la siguiente ilustración se muestran los mensajes intercambiados en el establecimiento de una sesión *MpTCP* y en la adicción de un subflujo adicional, donde el host A tiene dos interfaces distintas y el host B solo 1.

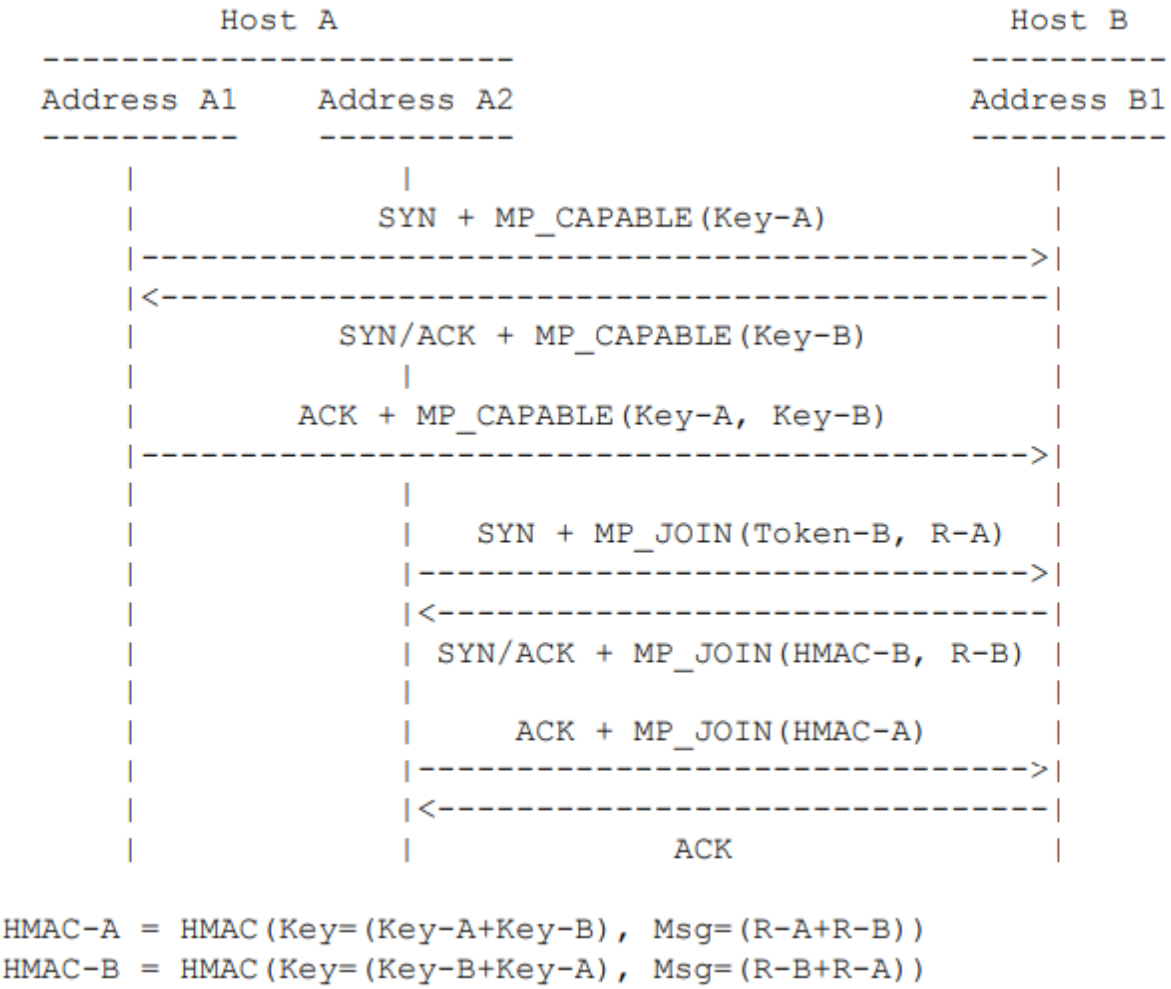

Ilustración 3.10: Establecimiento sesión y subflujo adicional *MpTCP*

Antes de establecer un nuevo flujo, el host debe conocer que direcciones a nivel de *IP* (*Internet Protocol*) que posee el host destino. *MpTCP* soporta la adición o eliminación de direcciones desde las que podemos alcanzar al host. El término "gestión de caminos" hace referencia al intercambio de información sobre los caminos o rutas alternativas. *MpTCP* hace uso de dos métodos para intercambiar esta información.

- Método 1: si el host dispone de alguna dirección adicional realiza un MP_JOIN poniendo como dirección origen la adicional y como destino el mismo sobre la que se ha establecido la sesión. Este es el caso que se ha representado en la Ilustración 3.10.

- Método 2: hace uso de la opción **ADD_ADDR**. Esta opción anuncia direcciones adicionales (opcionalmente puertos) en los que el host puede ser alcanzado.

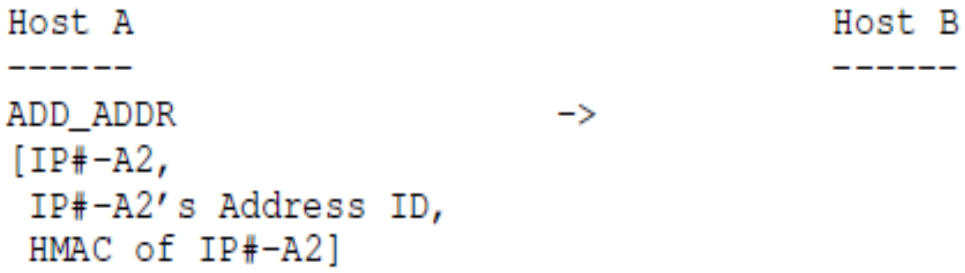

Ilustración 3.11: Anuncio de dirección

Actualmente hay 3 formas diferentes de anunciar rutas alternativas con la opción **ADD_ADDR**:

- Por defecto: No hace nada, ni anuncia direcciones disponibles (aunque existan), ni inicia nuevas conexiones. Acepta la creación pasiva de subflujos.
- FullMesh: Esta configuración establece el número máximo de subflujos disponibles.
- NdiffPorts: En vez de usar las distintas direcciones IP para iniciar nuevos subflujos, usa puertos distintos manteniendo la misma dirección IP. Esta configuración puede ser útil cuando solo se dispone de una interfaz y un sistema intermedio está limitando el ancho de banda de una conexión *TCP*.

Si una de las direcciones anunciadas deja de estar disponible *MpTCP* hace uso de la opción indicada anteriormente, **REMOVE_ADDR**, para indicar que cierta dirección ya no es posible usarla para alcanzar el destino.

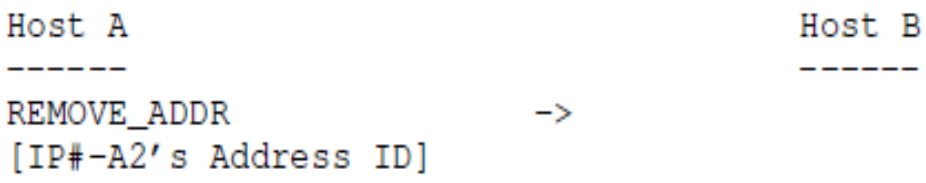

Ilustración 3.12: Eliminación de dirección

3.2.2.2.2 Transmisión de datos

Como se menciona anteriormente, se utiliza la opción **DSS** para realizar una transmisión de datos. Desde un nivel superior puede verse que a la hora de transmitir información hay una única entrada y salida de datos de la aplicación. El protocolo divide la información para transmitirla por uno o más subflujos, véase Ilustración 3.4. Para que *MpTCP* pueda entregar los datos de una manera fiable y ordenada, puesto que la transmisión puede realizarse utilizando diferentes rutas o caminos, se ha introducido un nuevo nivel de números de secuencia para numerar los segmentos.

Dentro de la opción DSS encontramos el campo **DSN** (*data sequence number*), que identifica a un segmento dentro del flujo *MpTCP*, y ACK Data para confirmar la recepción correcta dentro del flujo *MpTCP*.

De forma que, ahora existen dos jerarquías de números de secuencias, la más externa SN (*sequence number*) y ACK, es usada en los subflujos para garantizar la fiabilidad y orden de los segmentos que se transmiten por ellos. El otro nivel va en el campo de opciones de TCP y es utilizado para que los subflujos entreguen de una forma ordenada los datos en el flujo MpTCP, además para poder reenviar datos a través de distintos subflujos. Esto quiere decir, enviar datos por un subflujo y recibir el asentimiento por otro.

De esta forma la conexión se encuentra asegurada tanto a nivel de flujo (DSN y ACK Data) como a nivel de subflujo (SN y ACK de *TCP*). El campo DSN habilita al receptor a ordenar los paquetes recibidos en los diferentes subflujos sobre los que se establece la conexión. El DSN se va incrementando por cada transmisión, a este campo se le asocia un valor inicial en el *TWHS* del primer subflujo. Al terminar el establecimiento de la conexión se puede calcular el valor como los 64 bits menos significativos de código *hash* de la clave del emisor. La razón de esta forma de generarse es para dotar de resiliencia y no facilitar a los atacantes la tarea de predecir el DSN inicial.

En las ilustraciones que se encuentran a continuación se muestran una visión global de los niveles de jerarquía mencionados anteriormente y la opción *DSS* al completo.

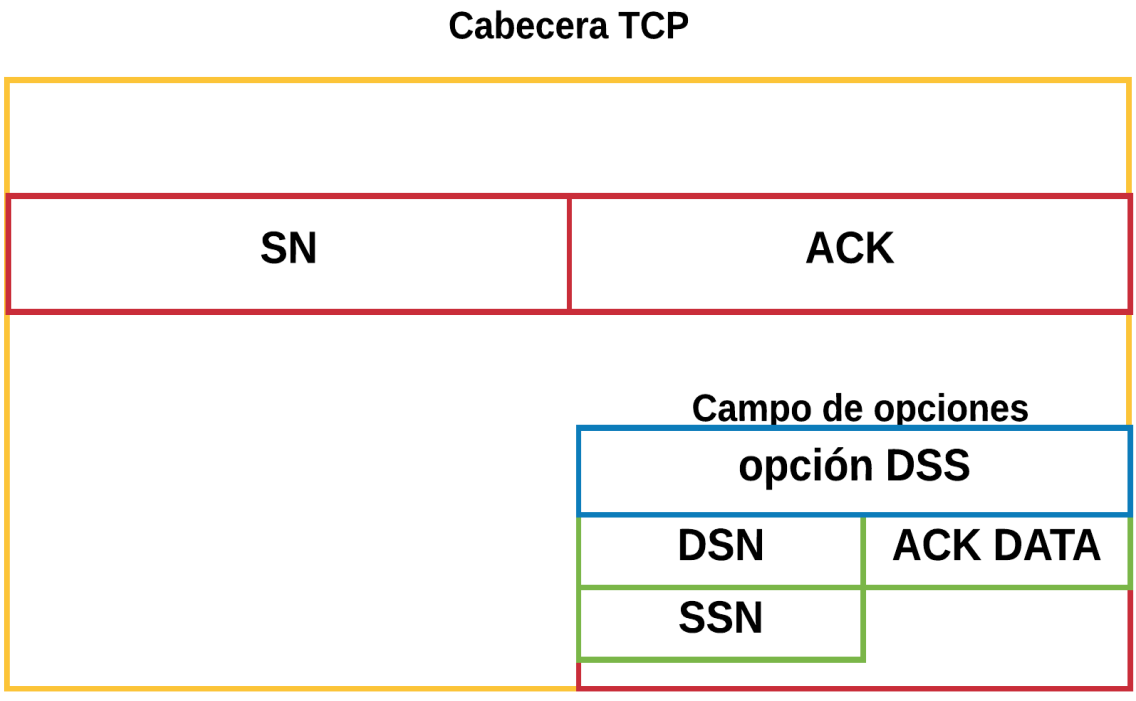

Ilustración 3.13: Niveles de jerarquía

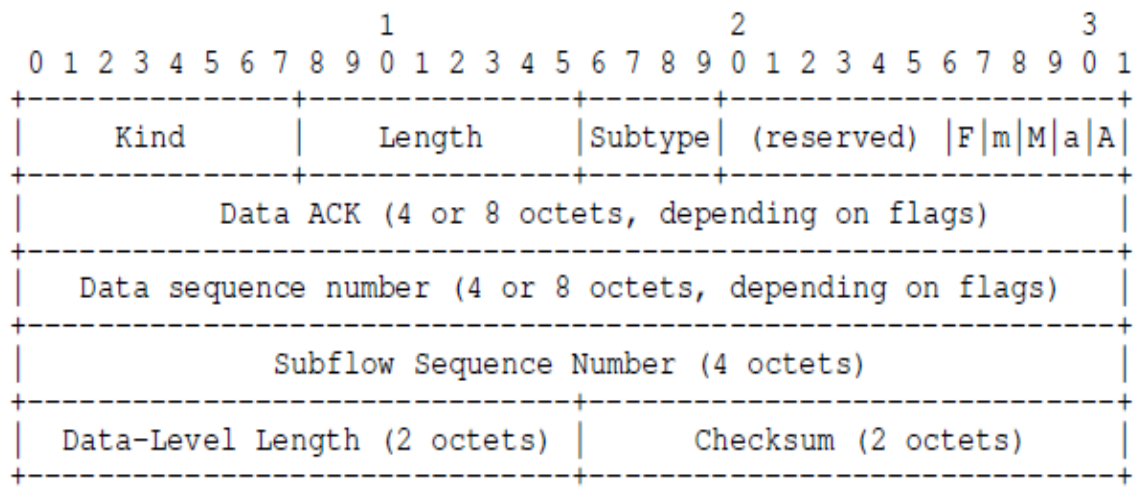

Ilustración 3.14: Opción DSS

3.2.2.2.3 Cierre de conexión

Para cerrar la conexión *MpTCP* es similar a la de *TCP*, la diferencia es que la señal **DATA_FIN** va dentro de la opción DSS, en concreto en la bandera '**F**' (véase Ilustración 3.3). Cuando se envía la señal **DATA_FIN** se supone que ya no hay más datos que enviar y la conexión se cierra en ambos extremos.

También existe la posibilidad de dejar de transmitir en un subflujo determinado, para ello se utiliza el mismo mecanismo que en *TCP* (bandera FIN).

3.2.3 Ventajas del uso

En este apartado se identificarán las principales ventajas que aporta *MpTCP* sobre *TCP*.

- Mayor rendimiento: Al tener la posibilidad de usar todas las interfaces disponibles en el dispositivo, es posible que el ancho de banda de la sesión sea mayor que si solo se usara una ruta. En el caso de que las rutas disponibles tengan un gran retardo se enviará toda la información por la mejor ruta y la conexión será similar a *TCP*.
- Robustez: Al disponer de caminos o rutas alternativas, el protocolo aumenta la fiabilidad de la conexión debido a que añade redundancia a la información. Antes caídas de enlaces el protocolo transmitiría por rutas alternativas sin tener que cerrar la conexión como pasaría en *TCP*. Además, dependiendo del algoritmo de congestión usado elegirá la ruta para enviar en función de la política de la organización.
- Balanceo de carga: Al poder elegir sobre que ruta transmitir puede repartir los datos entre las rutas disponibles con el fin de no congestionar ninguna ruta.

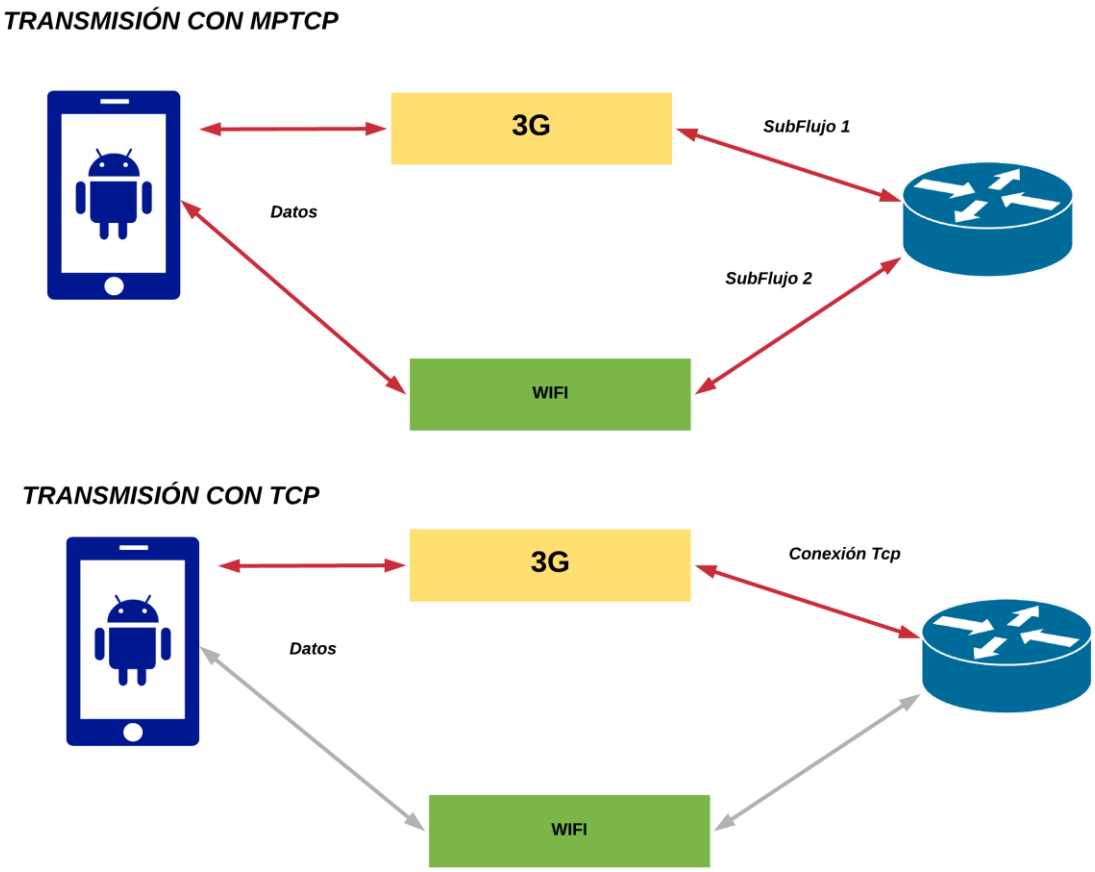

Ilustración 3.15: Transmisión *MpTCP* vs *TCP*

Como se muestra en la ilustración superior, si la transmisión se realizara con *MpTCP* se podría aprovechar ambas interfaces disponibles. De esta forma, sin que el usuario haya tenido que realizar ninguna configuración, su conexión estaría dotada de redundancia y de mayor caudal. En la ilustración anterior las rutas que se muestran en color rojo (las dos en el caso de *MpTCP* y solo la de arriba en el caso de que la transmisión se realice con *TCP*) indican que la comunicación se está llevando a cabo a través de ella.

3.2.4 Restricciones del protocolo

En este apartado se identifican las restricciones que se tienen que cumplir para que el protocolo funcione adecuadamente, dando por supuesto que los host origen y destino tengan implementado el protocolo *MpTCP*.

Los dispositivos intermedios como Firewalls, Routers, NATs, entre otros, deben permitir los mensajes que van en el campo de opciones de la cabecera *TCP*. Para ello deben permitir la **opción 30 de *TCP*** que identifica al protocolo *MpTCP*. Los dispositivos que lo implementen deben disponer al menos de dos interfaces físicas, ya que en el caso contrario no habrá ninguna diferencia significativa al uso de TCP.

3.2.5 Máquina de estados de MpTCP

En este apartado se muestra la máquina de estados al completo del protocolo *MpTCP* en la que se puede observar los posibles eventos que podrían pasar y los cambios de estados que estos desencadenan

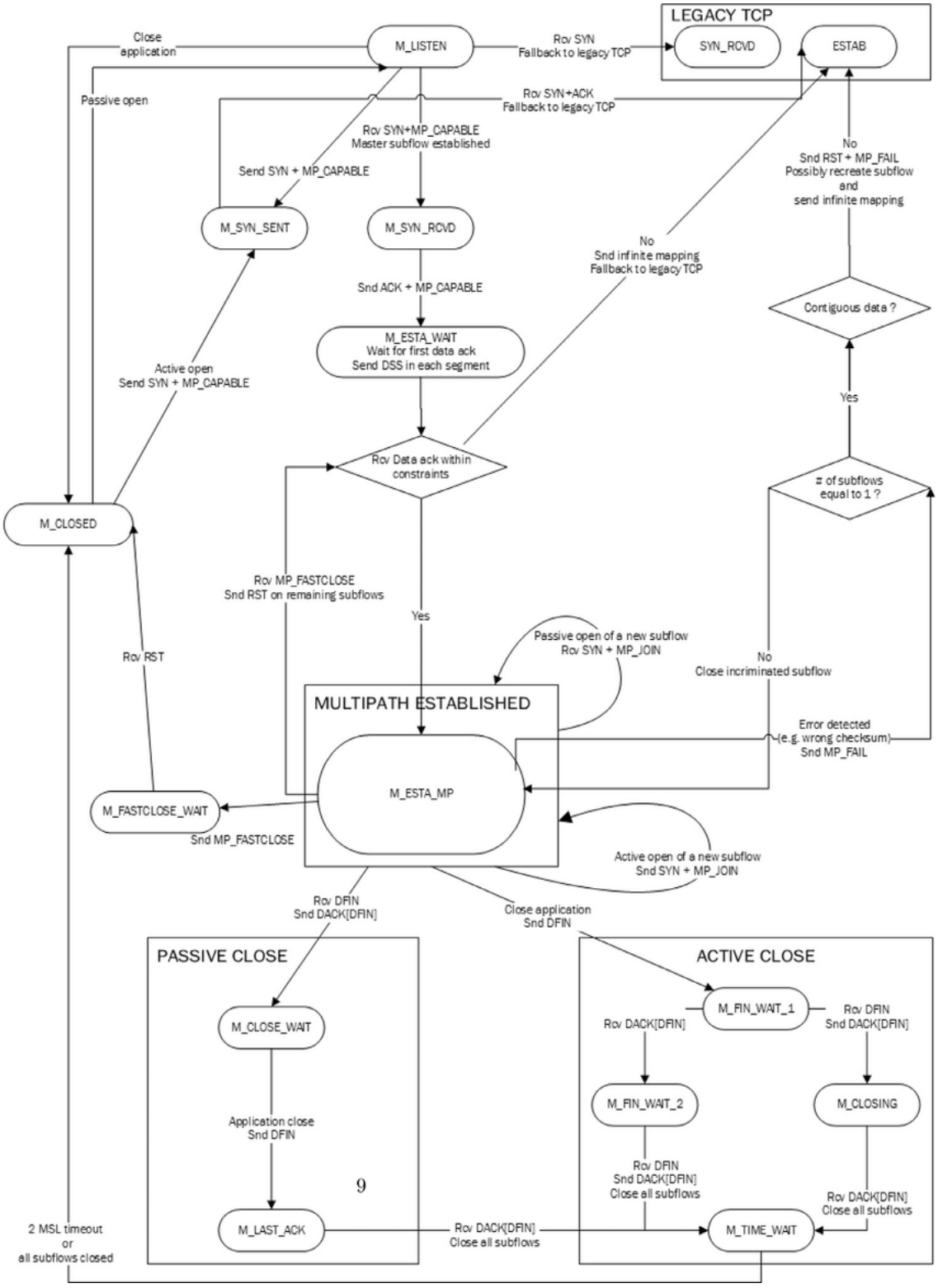

Ilustración 3.16: Máquina de estados *MpTCP*

4 MODELADO DE MPTCP

La fortuna juega a favor de la mente preparada.

-Luis Pasteur-

-

4.1 Implementaciones existentes

Existen numerosas implementaciones del protocolo, aunque la referente es la del *kernel* de Linux. La implementación que se ha elegido para este trabajo está realizada en el simulador de eventos ns3, pero está basada en la del *kernel* de Linux. Esta decisión se debe a la utilidad que tiene simulador para fines educativos y de investigación.

A continuación, se muestran las implementaciones en uso disponibles:

- *Kernel* de Linux (implementación de referencia)
- *Free BSD*
- *Apple IOS 7*
- *Apple Mac OS X 10.10*
- *Alcatel-Lucent*
- *Solaris*

4.2 Decisiones sobre el diseño

Antes de comentar el procedimiento seguido a la hora de validar la implementación, cabe mencionar que los lenguajes bases del simulador son C++ y Python. Con lo que antes de ponerse manos a la obra, se ha tenido que dar un repaso a ambos lenguajes, ya que la codificación y configuración del simulador lo requieren.

A la hora de elegir la implementación se han barajado varias opciones. Ambas implementaciones son de código abierto, y se encuentran alojadas en GitHub

1. El autor de esta implementación es **Morteza Kheirkhah** y la información disponible referente a la implementación como la implementación, las podemos encontrar en las siguientes direcciones:
    - Implementación: https://github.com/mkheirkhah/mptcp .
    - Documentación: documentación MpTCP Morteza

2. El autor de la segunda implementación es el laboratorio informático de París 6, **LIP6**. Tanto el código de la implementación como la documentación disponible la podemos encontrar en:
    - Implementación: https://github.com/lip6-mptcp/ns3mptcp.
    - Documentación: documentación MpTCP LIP6

Las dos están cerca de la **RFC 6824**. Están implementadas de formas distintas, entonces para tomar una decisión sobre cual realizar la validación y mejora ha sido necesario indagar en profundidad en las implementaciones.

Para ello se han implementado ambas, la primera en la versión 3.19 del simulador y la segunda en la versión 3.23. Se han analizado y revisado la escasa documentación que hay acerca de ambas. En la 2º opción solo he conseguido configurar una conexión *MpTCP* con un solo subflujo, mientras que en la 1º he configurado simulaciones con hasta 8 subflujos Este ha sido uno de los principales alicientes para decantarme por una implementación.

La opción elegida ha sido la primera, la de **Morteza**. Se ha considerado que era la más adecuada para conseguir los objetivos de este trabajo, validar una implementación de *MpTCP* y comprobar que mejora a *TCP*.

4.3 Estructura de la implementación

En esta sección se detallarán las características de la implementación.

4.3.1 Introducción a la implementación

Los ficheros fuentes del simulador se encuentran todos bajo el directorio ***src***, que se ubica en el directorio padre.

Dentro de la carpeta ***src***, se encuentran los archivos fuentes organizados en diferentes módulos en función de las temáticas de cada uno.

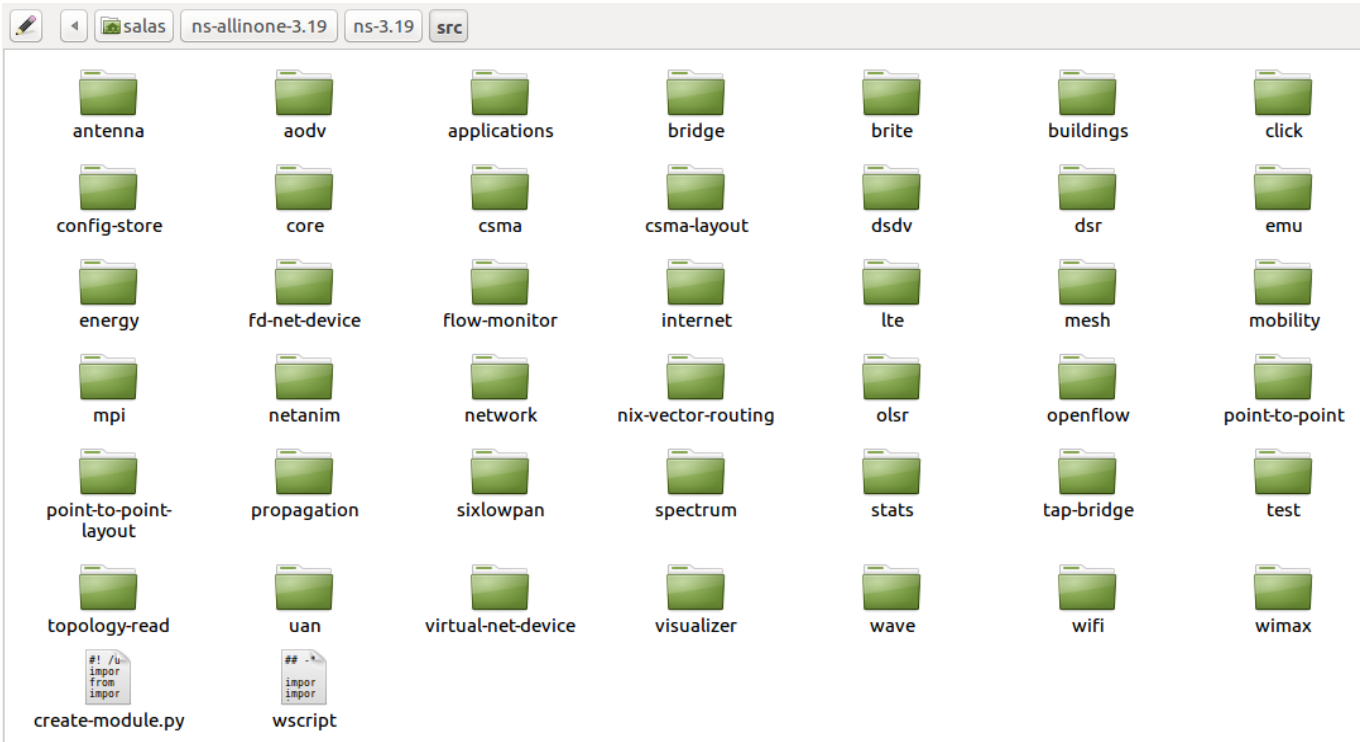

Ilustración 4.1: Directorio src

La implementación del protocolo ***MpTCP*** se ha llevado a cabo dentro del módulo internet, al igual que se encuentra la implementación de *TCP*. Todos los ficheros relacionados con la implementación del protocolo ***MpTCP*** se encuentran en este directorio. Los archivos fuentes de la implementación se encuentran en el subdirectorio *model*, mientras que los de test, se encuentran en el subdirectorio *test*.

4.3.2 Arquitectura de clases

Esta implementación consta de 2 clases principales para modelar el protocolo:

- ***MpTcpSocketBase***: Esta clase implementa el bloque de control del protocolo y exporta la *API* del socket a las aplicaciones existentes en ns3. Se encarga de tareas como el orden de los paquetes y el control de los distintos subflujos. Hereda de la clase ***TcpSocketBase*** que es la que implementa la lógica del protocolo *TCP*. En el lado del servidor se crea un objeto de la clase ***MpTcpSocketBase*** a la escucha, que instancia nuevos objetos de ***MpTcpSocketBase*** por cada conexión ***MpTCP***. En el lado del cliente, la instancia de ***MpTcpSocketBase*** representa una conexión ***MpTCP***.
- ***MpTcpSubflow***: Esta clase representa un subflujo. Un objeto de tipo ***MpTcpSocketBase*** puede tener varios objetos ***MpTcpSubflow***, uno por cada subflujo que tenga.

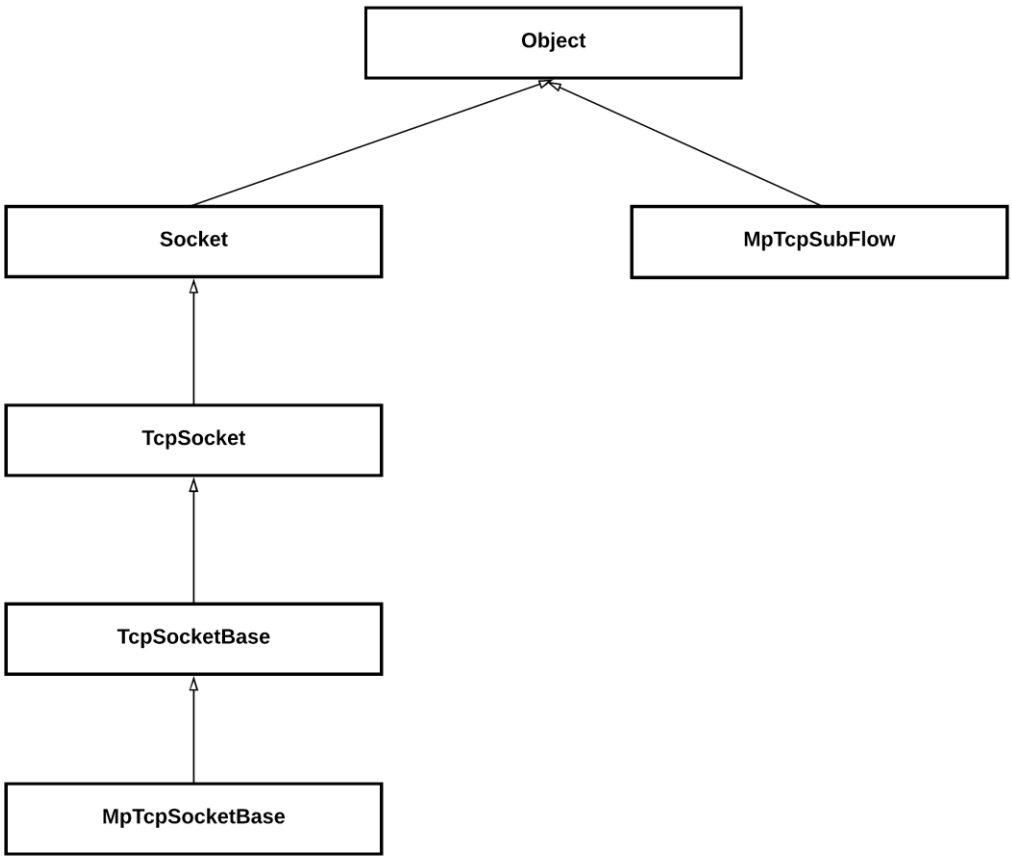

Ilustración 4.2: Arquitectura clases ns3

Además, se han hecho modificaciones en clases existentes con la finalidad de adaptar la implementación de *TCP* para soportar *MpTCP*. La clase TCPL4Protocol ha sido modificada para soportar la funcionalidad de *Tokens* en el establecimiento de nuevas conexiones usando la opción MP_JOIN.

4.3.3 Uso del protocolo

En este apartado se describe la forma en la que podemos hacer uso de la implementación y obtener resultados para posteriormente interpretarlos, que es el objetivo de las simulaciones.

4.3.3.1 Configuraciones previas

Antes de desarrollar una simulación se deberá tener en cuenta las siguientes consideraciones:

- Número máximo de subflujos: Para ello se hará uso del atributo ***MaxSubflows*** de la clase ***MpTcpSocketBase***.
- Algoritmo de selección de rutas: Este algoritmo elige el subflujo por el que se debe transmitir. El atributo que hay que configurar es ***SchedulingAlgorithm*** que pertenece a la clase ***MpTcpSocketBase.*** Actualmente solo hay dos algoritmos disponibles, ***Fastest_Rtt*** y ***Round_Robin***, ambos se explican en el apartado 4.4
- Gestión de rutas: Hace referencia a la forma que tiene el protocolo de anunciar las direcciones que tiene disponibles. Para ello es necesario configurar el atributo ***PathManagement*** de la clase ***MpTcpSocketBase***. Actualmente se puede configurar con tres valores distintos, ***FullMesh***, ***NdiffPort*** y ***Default***, cuyo funcionamiento se explica en el apartado 3.2.2.2.1.
- Algoritmo de control de congestión: Este algoritmo controla la congestión presente en la topología. Podemos configurarlo a través del atributo ***CongestionControl***.
- Tipo de socket: Tenemos que indicar el tipo de *socket* a crear. Se hace a través del atributo ***SocketType*** perteneciente a la clase ***TCPL4Protocol***.

En la siguiente ilustración podemos observar una forma fácil de configurar todos los atributos mencionados anteriormente:

```
/*MpTcp*/
//Socket MpTcp
Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue(MpTcpSocketBase::GetTypeId()));
// 3 Subflujos máximos
Config::SetDefault("ns3::MpTcpSocketBase::MaxSubflows", UintegerValue(3));
// Anuncia todas las direcciones
Config::SetDefault("ns3::MpTcpSocketBase::PathManagement", StringValue("FullMesh"));
// Para generar gráficas
Config::SetDefault("ns3::MpTcpSocketBase::LargePlotting", BooleanValue(true));
//Algoritmo menor RTT
Config::SetDefault("ns3::MpTcpSocketBase::SchedulingAlgorithm", StringValue("Fastest_Rtt"));

Config::SetDefault("ns3::MpTcpSocketBase::CongestionControl", StringValue("RTT_Compensator"));
```

Ilustración 4.3: Configuración Atributos

4.3.3.2 Aplicaciones

Para el uso del protocolo se han implementado dos aplicaciones: ***MpTcpPacketSink*** y ***MpTcpBulkSendApplication***, ambas aplicaciones se basan en las ya existentes, ***PacketSink*** y ***BulkSendApplication*** respectivamente. Estas aplicaciones se encuentran dentro del módulo de aplicaciones.

```
TypeId
MpTcpBulkSendApplication::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::MpTcpBulkSendApplication")
    .SetParent<Application> ()
    .AddConstructor<MpTcpBulkSendApplication> ()
    .AddAttribute ("SendSize", "The amount of data to send each time from application buffer to so
cket buffer.",
                   UintegerValue (140000), //512
                   MakeUintegerAccessor (&MpTcpBulkSendApplication::m_sendSize),
```

```
                   MakeUintegerChecker<uint32_t> (1))
    .AddAttribute ("Remote", "The address of the destination",
                   AddressValue (),
                   MakeAddressAccessor (&MpTcpBulkSendApplication::m_peer), MakeAddressChecker ())
    .AddAttribute ("MaxBytes",
                   "The total number of bytes to send. "
                   "Once these bytes are sent, "
                   "no data  is sent again. The value zero means "
                   "that there is no limit.",
                   UintegerValue (0), // 1 MB default of data to send
                   MakeUintegerAccessor (&MpTcpBulkSendApplication::m_maxBytes),
                   MakeUintegerChecker<uint32_t> ())
    .AddAttribute("FlowId",
                  "Unique Id for each flow installed per node",
                  UintegerValue(0),
                  MakeUintegerAccessor (&MpTcpBulkSendApplication::m_flowId),
                  MakeUintegerChecker<uint32_t> ())
    .AddAttribute("MaxSubflows",
                  "Number of MPTCP subflows",
                  UintegerValue(8),
                  MakeUintegerAccessor (&MpTcpBulkSendApplication::m_subflows),
                  MakeUintegerChecker<uint32_t> ())
    .AddAttribute("SimTime",
                  "Simulation Time for this application it should be smaller than stop time",
                   UintegerValue(20),
                   MakeUintegerAccessor(&MpTcpBulkSendApplication::m_simTime),
                   MakeUintegerChecker<uint32_t>())
    .AddAttribute("DupAck",
                 "Dupack threshold -- used only for MMPTCP and PacketScatter",
                  UintegerValue(0),
                  MakeUintegerAccessor(&MpTcpBulkSendApplication::m_dupack),
                  MakeUintegerChecker<uint32_t>())
    .AddAttribute("FlowType",
                  "The Type of the Flow: Large or Short ",
                  StringValue("Large"),
                  MakeStringAccessor(&MpTcpBulkSendApplication::m_flowType),
                  MakeStringChecker())
    .AddAttribute("OutputFileName",
                  "Output file name",
                  StringValue("NULL"),
                  MakeStringAccessor(&MpTcpBulkSendApplication::m_outputFileName),
                  MakeStringChecker())
    .AddAttribute ("Protocol", "The type of protocol to use.",
                   TypeIdValue (TcpSocketFactory::GetTypeId ()),
                   MakeTypeIdAccessor (&MpTcpBulkSendApplication::m_tid),
                   MakeTypeIdChecker ())
    .AddTraceSource ("Tx", "A new packet is created and is sent",
                     MakeTraceSourceAccessor (&MpTcpBulkSendApplication::m_txTrace))
  ;
  return tid;
}
```

Ilustración 4.4: Atributos MpTcpBulkSendApplication

Como podemos observar en la ilustración anterior, los atributos son similares a los de BulkSendApplication con la salvedad de los añadidos para dar soporte al protocolo ***MpTCP***. El atributo ***OutputFilename*** se ha utilizado para generar un fichero tras finalizar la simulación que recoge los datos más significativos.

La forma en la que se indica que el *socket* que usamos es del tipo *MpTcpSocketBase* es la siguiente:

```
void MpTcpBulkSendApplication::StartApplication (void) // Called at time specified by Start
{
  NS_LOG_FUNCTION (this);
  // Create the socket if not already
  if (!m_socket)
    {
      m_socket = DynamicCast<MpTcpSocketBase>(Socket::CreateSocket (GetNode (), m_tid));
      m_socket->Bind();
      m_socket->SetMaxSubFlowNumber(m_subflows);
      m_socket->SetFlowType(m_flowType);
      m_socket->SetOutputFileName(m_outputFileName);
      int result = m_socket->Connect(m_peer);
      if (result == 0)
```

```
        {
          m_socket->SetFlowId(m_flowId);
          m_socket->SetDupAckThresh(m_dupack);
          m_socket->SetConnectCallback(MakeCallback(
               &MpTcpBulkSendApplication::ConnectionSucceeded, this),
               MakeCallback(&MpTcpBulkSendApplication::ConnectionFailed, this));

          m_socket->SetDataSentCallback(MakeCallback(&MpTcpBulkSendApplication::DataSend, this));
          m_socket->SetCloseCallbacks (
            MakeCallback (&MpTcpBulkSendApplication::HandlePeerClose, this),
            MakeCallback (&MpTcpBulkSendApplication::HandlePeerError, this));

          m_socket->SetSendCallback(MakeCallback(&MpTcpBulkSendApplication::DataSend, this));
        }
      else
        {
          NS_LOG_UNCOND("Connection is failed");
        }
    }
  if (m_connected)
    {
      SendData ();
    }
}
```

Ilustración 4.5: Inicio aplicación MpTcpBulkSend

El método que se muestra en la figura anterior es el encargado de iniciar la aplicación y configurar el tipo de *socket* adecuado. Se realiza una conversión de *socket* de *TcpSocketBase* a *MpTcpSocketBase*. Esta conversión es posible debido a la arquitectura de clases que se muestra en 4.3.2.

De igual forma, el cambio más significativo de *MpTcpPacketSink* es la forma de inicializar la aplicación. De nuevo se configura el *socket* para que sea del tipo *MpTcpSocketBase*:

```
void
MpTcpPacketSink::StartApplication()    // Called at time specified by Start
{
  NS_LOG_FUNCTION (this);
  // Create the socket if not already
  if (!m_socket)
    {
      m_socket = DynamicCast<MpTcpSocketBase>(Socket::CreateSocket (GetNode (), m_tid));
      m_socket->Bind(m_local);
      m_socket->Listen();
      NS_LOG_LOGIC("StartApplication -> MptcpPacketSink got an listening socket " << m_socket
                   << " binded to addrs:port  " << InetSocketAddress::ConvertFrom(m_local).GetIpv4()
                   << ":" << InetSocketAddress::ConvertFrom(m_local).GetPort());
    }

  m_socket->SetRecvCallback(MakeCallback(&MpTcpPacketSink::HandleRead, this));
  m_socket->SetAcceptCallback(MakeNullCallback<bool, Ptr<Socket>, const Address &>(),
      MakeCallback(&MpTcpPacketSink::HandleAccept, this));
  m_socket->SetCloseCallbacks(MakeCallback(&MpTcpPacketSink::HandlePeerClose, this),
      MakeCallback(&MpTcpPacketSink::HandlePeerError, this));

}
```

Ilustración 4.6: Inicio aplicación MpTcpPacketSink

Si nos fijamos en ambas aplicaciones es necesario hacer la conversión al tipo de *socket* adecuado. Como podemos ver al principio de esta sección en la adición del atributo ***protocol***, que por defecto lo inicializa a ***TcpSocketFactory***, la factoría que se encarga de la creación de *sockets* de tipo *TcpSocketBase*.

4.3.3.3 Ficheros de salida

Tal como se ha comentado en el apartado anterior (4.3.3.2), el atributo ***OutputFilename*** determina el nombre del fichero de salida con los datos de las simulaciones. Si no se configura este parámetro, este fichero se generará siempre con el nombre por defecto. Por otro lado, la implementación también es capaz de generar gráficas para una representación más visual de los datos.

```
void
MpTcpSocketBase::DoGenerateOutPutFile()
{
  TypeId tid = this->GetTypeId();
  goodput = ((nextTxSequence * 8) / (Simulator::Now().GetSeconds() - fLowStartTime));
  Ptr<OutputStreamWrapper> stream = Create<OutputStreamWrapper>(outputFileName, std::ios::out |
std::ios::app);
  ostream* os = stream->GetStream();
  std::string tipoSocket = "";
  if(this->GetTypeId().GetName().compare("ns3::MpTcpSocketBase") ==  0){
          tipoSocket = "MpTcp";
  }else{ //en el caso contrario es TCP
          tipoSocket = "Tcp";
  }
  *os << "############################################# Datos Simulación Socket " << tipoSocket <<
" ############################################### \n\n";
  *os << "[NodeId: " << m_node->GetId();
  *os     << " ]\n[FlowId: ";
  *os   << flowId << " ]\n[FlowType: ";
  *os   << flowType << "]\n[Throughput: ";
  *os   << goodput / 1000000 << " Mbps]\n[FlowComplTime: ";
  *os   << (Simulator::Now().GetSeconds() - fLowStartTime);
  *os   << " seconds ]\n[timeOut:" << TimeOuts << " ]\n[fastReTx: ";
  *os   << FastReTxs << " ]\n[PartialAck: " << pAck << "]\n[FullAck: ";
  *os   << FullAcks << " ]\n[FastRecovery: " << FastRecoveries;
  *os   << " ]\n[Subflows Size: " << subflows.size() << " ]\n[EstSubflows: "<< GetEstSubflows();
  *os   << " ]\n[TypeId: " << tid.GetName() << " ]\n";
  *os   << "[Distribution algorithm: "<< PrintSchedulingAlgorithm(distribAlgo) << " ]\n\n";

  if(subflows.size() > 1){
          *os << "------------ Specific SubFlows ------------------\n\n";
          for(int a = 0; a < subflows.size(); a++){
              double meanRtt = 0;
              vector<pair<double, double> >::iterator it = subflows[a]->rttTracer.begin();
              while (it != subflows[a]->rttTracer.end())
                {
                  meanRtt = meanRtt + it->second;
                   it++;
                }
              *os << "[Src/port: " << subflows[a]->sAddr << "/" << subflows[a]->sPort
                << "  Dest/port: " << subflows[a]->dAddr << "/" << subflows[a]->dPort << " ]\n";

               meanRtt=(meanRtt/(subflows[a]->rttTracer.size()));
              *os << "[RTT mean Subflow " << a <<" : " << meanRtt
                  << " milliSeconds ]\n[Fast Retransmit threshold subflow "
                   << a << ": " << subflows[a]->m_retxThresh << " ]\n";
              *os << "[Numero de paquetes transmitidos por el SubFlow " << a
                   << " : " << subflows[a]->PktCount << " ]\n\n";
          }
          *os << endl;
  }else{
          double meanRtt = 0;
          for(int it = 0; it < subflows[0]->_RTT.size(); it++){
                  meanRtt = meanRtt + subflows[0]->_RTT[it].second;
          }
          meanRtt=(meanRtt/(subflows[0]->_RTT.size()));
          *os << "[RTT: "
          << meanRtt << " milliSeconds]\n[Fast Retransmit threshold: "
          << subflows[0]->m_retxThresh << " ]\n" << endl;
  }

  *os << "############################################# Fin Datos Simulación
      << ############################################################ \n\n";
  NS_LOG_DEGUG(Simulator::Now().GetSeconds() << " ["<< m_node->GetId()<< "] Goodput -> "
               << goodput / 1000000 << " Mbps");

                                                  }
```

Ilustración 4.7: Método que genera datos de salida

Este método anterior se ha implementado como mejora para que los resultados obtenidos en las simulaciones puedan ser comprobados visualmente, ya que por defecto no se genera ningún fichero. Los datos se generan una vez la sesión *MpTCP* ha finalizado, por eso es desde la misma función de cierre desde donde se ejecuta.

Como podemos observar en la siguiente ilustración, no se genera el fichero de salida hasta que no hay más datos que transmitir.

```
int
MpTcpSocketBase::Close(uint8_t sFlowIdx)
{
        if (sendingBuffer.PendingData() > 0)
        { // App close with pending data must wait until all data transmitted from socket buffer
          NS_ASSERT(client);
          if (m_closeOnEmpty == false)
                {
                  m_closeOnEmpty = true;
                  if (flowType.compare("Large") == 0)
                        { // This is only true for background flows
                          flowCompletionTime = false;
                          DoGenerateOutPutFile();
                          GeneratePlots();
                        }
                  NS_LOG_INFO ("Socket " << this << " deferring close,
                                        Connection state " << TcpStateName[m state]
                                        << " PendingData: " << sendingBuffer.PendingData());
                }
          return 0;

        }
```

Ilustración 4.8: Función de cierre sesión *MpTCP*

Para que se generen las gráficas es necesario configurar el parámetro ***LargePlotting*** (con valor **true**) antes de iniciar la simulación, además del atributo de la aplicación *BulkSendApplication*, ***FlowType*** *(*con valor ***Large****)*. En la siguiente ilustración se muestra como se configuran ambos parámetros:

```
Config::SetDefault("ns3::MpTcpSocketBase::LargePlotting", BooleanValue(true));

//Fuente
MpTcpBulkSendHelper source("ns3::TcpSocketFactory",
                           InetSocketAddress(Ipv4Address(i1.GetAddress(1)), port));

source.SetAttribute("MaxBytes", UintegerValue(10000000));
source.SetAttribute("FlowType", StringValue("Large"));
source.SetAttribute("FlowId", UintegerValue(1));
source.SetAttribute("OutputFileName", StringValue("RESULT_MPTCP_VS_TCP.txt"));
ApplicationContainer sourceApps = source.Install(nodosEnlace1.Get(0));
sourceApps.Start(Seconds(0.0));
sourceApps.Stop(Seconds(1100.0));
```

Ilustración 4.9: Configuración para generar gráficas

Para la generación de las gráficas se han ido rellenando vectores de datos durante la simulación. Para rellenar estos vectores se ha suscrito a una traza para que cada vez que fueran cambiando estos valores se fueran almacenando. En la siguiente ilustración se muestran cuáles son los vectores que se han utilizado para generar las gráficas.

```
class MpTcpSubFlow : public Object
        {
        public:
                //ventana de congestión
                vector<pair<double, uint32_t> > cwndTracer;
                //Umbral
                vector<pair<double, uint32_t> > sstTracer;
                // RTO
                vector<pair<double, double> > rtoTracer;
                //RTT
                vector<pair<double, double> > rttTracer;
                //Numero de paquetes enviados
                uint64_t PktCount
```

Ilustración 4.10: Parámetros de estudio por subflujo

Una vez se han creado las gráficas, se genera un fichero de salida cuyo nombre hace referencia al algoritmo de control de congestión empleado, el tiempo que ha durado la simulación, el número máximo de subflujos, el tipo de *socket* empleado, el identificado del nodo y el del flujo *MpTCP*.

```
void
MpTcpSocketBase::GeneratePlotsOutput()
{
  stringstream oss;
         //control congestion                        //tiempo de simulacion
  oss << "PLOT_" << PrintCC(AlgoCC) << "_" << (uint32_t)Simulator::Now().GetSeconds();

                //n° max de subflujos        //tipo socket          //id del nodo
  oss <<"_"<< (int)maxSubflows << "_" << GetTypeIdName() <<"_" << m_node->GetId();

 // id del flujo mptcp
  oss << "_" << flowId;

  string tmp = oss.str();
  oss.clear();
  Ptr<OutputStreamWrapper> stream = Create<OutputStreamWrapper>(tmp.c_str(), std::ios::out);
  ostream* os = stream->GetStream();
  gnu.GenerateOutput(*os);

}
```

Ilustración 4.11: Método para generar fichero de gráficas

Un nombre de ejemplo para un fichero de salida puede ser: ***PLOT_Linked-Increase_521_1_MPTCP_0_1***. Para generar las gráficas en formato imagen es necesario situarse en el directorio base de ns3 (donde se genera el fichero anterior) y ejecutar el siguiente comando: **gnuplot4-x11 <nombreDelFicheroGenerado>**. Tras ejecutarlo se generan todas las gráficas en formato de imagen.

Actualmente, las gráficas que se generan por defecto son las siguientes: **pkt.eps, cwnd.eps, rtt.eps, rto.eps, ssh.eps.** En las que se muestra cómo evolucionan factores como el *RTT*, umbral de transmisión, *RTO*, paquetes enviados, con el tiempo. Todas las gráficas han sido adaptadas para representar los datos de la forma más adecuada, además de para mostrar los datos de todos los subflujos que se establecen durante la simulación.

```
void
MpTcpSocketBase::GeneratePlots()
{
  if ((m_largePlotting && (flowType.compare("Large") == 0)) ||
                                          (m_shortPlotting && (flowType.compare("Short") == 0)))
    {
      GenerateCwndTracer();
      GenerateRTT();
      GeneratePktCount();
      GeneratePlotsOutput();
      GenerateSendvsACK();
    }

}
```

Ilustración 4.12: Método para generar gráficas

4.4 Mejoras implementadas

El algoritmo de selección de rutas disponible en la implementación usada es ***Round_Robin***. Este algoritmo elige equitativamente la ruta para transmitir los datos (es el más simple). Para nada se trata de un algoritmo eficiente ya que pueden existir rutas que en un momento determinado no interese transmitir información por ellas, bien sea por que empiece a congestionarse o por que existan rutas más rápidas.

Para ello se ha definido un algoritmo más eficiente que se basa en el cálculo de *RTT* (***round trip time***), ***Fastest_Rtt***. El *RTT* se puede definir como el tiempo que tarda la información en llegar hasta el receptor, ser procesada, y volver hasta el emisor.

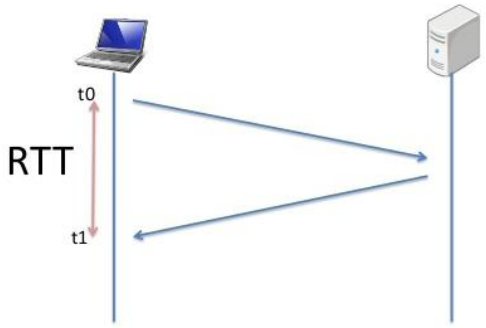

Ilustración 4.13: Round Trip Time

Este algoritmo, antes de transmitir, elige el subflujo que tenga un *RTT* más bajo, de esta forma se asegura que está transmitiendo por la ruta más rápida. Además de realizar la comprobación de *RTT*, es necesario comprobar que tenga espacio en la ventana de transmisión.

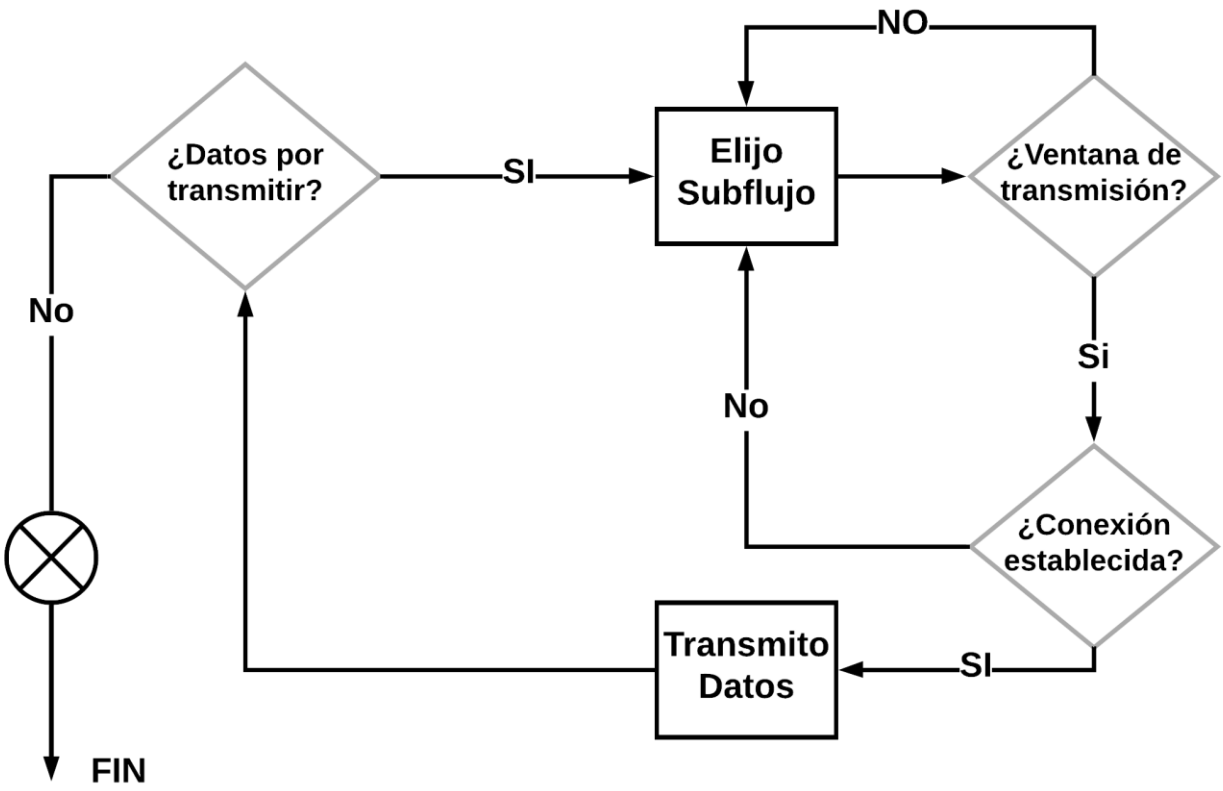

Ilustración 4.14: Transmisión a nivel de socket *MpTCP*

```
uint8_t
MpTcpSocketBase::getSubflowToUse()
{
  NS_LOG_FUNCTION(this);
  uint8_t nextSubFlow = 0;
  switch (distribAlgo){

              case Round_Robin:
              {
                     nextSubFlow = (lastUsedsFlowIdx + 1) % subflows.size();
                     break;
              }
              case Fastest_Rtt:
              {
                     if(!(subflows.size() == 1)){
                            Time lowestEstimate = Time::Max(); // El peor tiempo posible.
                            for(int i = 0; i < (int)subflows.size(); i++){
                                   Ptr<MpTcpSubFlow> sf = subflows[i];
                                   /**
                                   * En este punto, comprobamos si el subflujo tiene espacio en la
                                   * ventana de transmisión.
                                   * En el caso que no tenga, seguimos buscando en los demás
                                   * subflujos.
                                   */
                                   NS_LOG_DEBUG(" RTT Subflow "<< i << " "
                                   << sf->rtt->GetCurrentEstimate().GetMilliSeconds()
                                   << " mS Ventana Disponible: " << AvailableWindow(i));

                                   if(AvailableWindow(i) <= 0)
                                   {
                                          continue;
                                   }
                                   /**
                                    * El próximo subflujo sobre el que se transmitirá, será el que
                                    * tenga espacio en la ventana de transmisión y
                                    * además tenga el Rtt más bajo.
                                    */
                                   if(sf->rtt->GetCurrentEstimate() < lowestEstimate){
                                          lowestEstimate = sf->rtt->GetCurrentEstimate();
                                          nextSubFlow = i;
                                   }
                            }
                     }else{
                            /**
                             * En el caso que solo tengamos un subflujo,
                             * el siguiente para transmitir es el mismo subflujo
                             */
                            nextSubFlow = 0;
                     }
                     break;
              }
              default:
              break;
  }
  return nextSubFlow;

}
```

Ilustración 4.15: Algoritmos selección disponibles

Además de la definición del algoritmo en la clase encargada de controlar la simulación del protocolo, MpTcpSocketBase, se han realizado modificaciones con el fin de soportar el nuevo algoritmo implementado. Anteriormente, solo se comprobaba el subflujo a transmitir si la ventana de transmisión estaba vacía, luego no sería un algoritmo ***Round_Robin*** puesto que no transmite equitativamente por cada subflujo. Tras la modificación, antes de realizar una transmisión se comprueba el subflujo por el que enviar los datos.

Esta modificación se ha realizado debido a que antes no se estaba controlando la posibilidad de que existiera un nuevo algoritmo de gestión de rutas, sino que, se enviaba por un subflujo hasta que se quedara sin ventana de transmisión. Como ahora la comprobación se hace antes de la transmisión, en el caso de que la gestión sea a través de ***Round_Robin,*** se transmitirá por el próximo subflujo con ventana disponible. En el caso que se use ***Fastest_Rtt*** el próximo para transmitir será el subflujo con ventana disponible y menor *RTT*.

ÍNDICE DE PRUEBAS

5 PRUEBAS

Los científicos estudian el mundo tal y como es; los ingenieros crean el mundo que nunca han visto.

- Neal Stephenson -

En esta sección se hablará sobre el plan de pruebas seguido. Además, en él se mostrarán gráficas y ficheros generados durante la simulación para la validación del modelo.

5.1 Pruebas de rendimiento

En este apartado se detallan las pruebas que se han llevado a cabo para validar la implementación del protocolo. A continuación, se muestra la topología de red sobre la que se han realizado distintas pruebas:

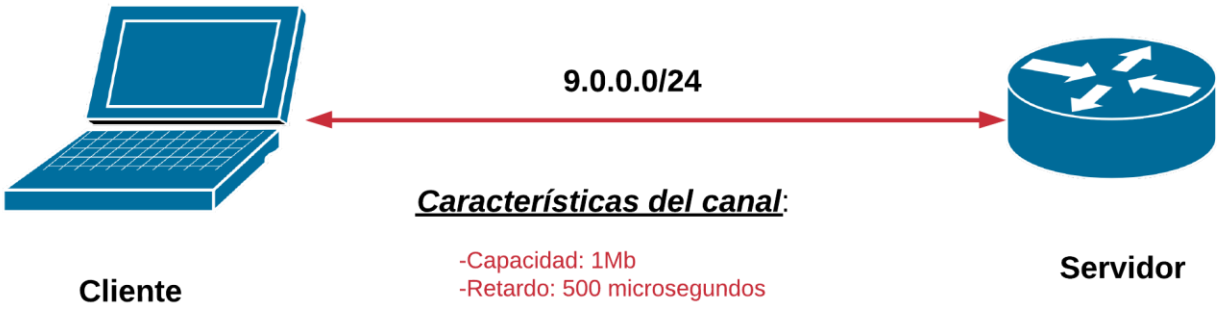

Ilustración 5.1: Topología de red

5.1.1 Validación de la implementación

Para una de las pruebas realizadas se han hecho dos simulaciones, una de ellas usando un socket del tipo *TCP* y la otra con un socket *MpTCP*. La conexión *MpTCP* se ha configurado de tal forma que solo soporte un subflujo con el fin de compararla con la conexión *TCP*.

```
/**
 * Author: Antonio Manuel Del Toro Domínguez
 * Código que se encarga de validar el modelo.
 * Para ello se se realizan dos simulaciones, una usando TCP y otra usando MPTCP con un único Flujo.
 *
 *                           enlace 1
 * Topología:  (nodo 0)<---------------------->(nodo 1)
 */
```

```
case MPTCP:{
        Config::SetDefault("ns3::TcpL4Protocol::SocketType",
                           TypeIdValue(MpTcpSocketBase::GetTypeId()));

        Config::SetDefault("ns3::MpTcpSocketBase::MaxSubflows", UintegerValue(1)); // Sink
        Config::SetDefault("ns3::MpTcpSocketBase::LargePlotting", BooleanValue(true));
        // TCP need one subflow only
        Config::SetDefault("ns3::MpTcpBulkSendApplication::MaxSubflows", UintegerValue(1));
        Config::SetDefault("ns3::MpTcpSocketBase::PathManagement", StringValue("Default"));

case TCP:{
        Config::SetDefault("ns3::TcpL4Protocol::SocketType",
                           TypeIdValue(TcpSocketBase::GetTypeId()));

        Config::SetDefault ("ns3::TcpL4Protocol::SocketType", StringValue ("ns3::TcpNewReno"));
/*
* Creamos el enlace
* */
        Ptr<RateErrorModel> errores = CreateObject<RateErrorModel> ();
        errores->SetUnit (RateErrorModel::ERROR_UNIT_PACKET);
        errores->SetRate (0);

        NodeContainer nodosEnlace1;
        nodosEnlace1.Create(2);

        PointToPointHelper enlace1;
        enlace1.SetDeviceAttribute("DataRate", StringValue("1Mbps"));
        enlace1.SetChannelAttribute("Delay", StringValue("500us"));
        enlace1.SetDeviceAttribute ("ReceiveErrorModel", PointerValue (errores));
        NetDeviceContainer devices1;
        devices1 = enlace1.Install(nodosEnlace1);

        InternetStackHelper stack;
        stack.Install(nodosEnlace1);

        Ipv4AddressHelper rango1;
        rango1.SetBase("9.0.0.0", "255.255.255.0");

       Ipv4InterfaceContainer i1 = rango1.Assign(devices1);

/**
* Configuramos las aplicaciones
*/

        uint16_t port = 9;
        PacketSinkHelper sink("ns3::TcpSocketFactory",
                           InetSocketAddress(Ipv4Address::GetAny(), port));

        ApplicationContainer sinkApps = sink.Install(nodosEnlace1.Get(1));
        sinkApps.Start(Seconds(0.0));
        sinkApps.Stop(Seconds(1100.0));

        BulkSendHelper source("ns3::TcpSocketFactory",
                           InetSocketAddress(Ipv4Address(i1.GetAddress(1)), port));

        source.SetAttribute("MaxBytes", UintegerValue(10000000)); //10 Mb de datos
        source.SetAttribute("FlowType", StringValue("Large"));
```

```
source.SetAttribute("FlowId", UintegerValue(1));
source.SetAttribute("OutputFileName", StringValue("RESULT_MPTCP_VS_TCP.txt"));
ApplicationContainer sourceApps = source.Install(nodosEnlace1.Get(0));
sourceApps.Start(Seconds(0.0));
sourceApps.Stop(Seconds(1100.0));
```

Ilustración 5.2: Prueba *MpTCP* vs *TCP*

Tras realizar las simulaciones se han obtenido los siguientes resultados, de esta forma lo que se quiere validar es que, en el peor de los casos, cuando *MpTCP* se establece con un solo subflujo se está comportando de la misma forma que TCP. Como se observa en los resultados, usando distintos tipos de *socket* para ambas simulaciones se han obtenido resultados similares, tanto duración de la simulación como el *RTT* calculado.

```
############################################## Datos Simulación Socket MpTcp ##################################################

[NodeId: 0 ]
[FlowId: 1 ]
[FlowType: Large]
[Throughput: 0.336332 Mbps]
[FlowComplTime: 234.768 seconds ]
[timeOut:882 ]
[fastReTx: 0 ]
[PartialAck: 0]
[FullAck: 0 ]
[FastRecovery: 0 ]
[Subflows Size: 1 ]
[EstSubflows: 1 ]
[TypeId: ns3::MpTcpSocketBase ]
[Distribution algorithm: Round_Robin ]

[RTT: 12.3557 milliSeconds]
[Fast Retransmit threshold: 3 ]

############################################## Fin Datos Simulación ###############################################################

############################################## Datos Simulación Socket Tcp ##################################################

[NodeId: 0]
[flowId: 1]
[FlowType: Large]
[Throughput: 0.336209]
[FlowComplTime: 237.947]
[timeOut: 0]
[FastReTxs: 0]
[PartialAck: 0]
[FullAcks: 0]
[FastRecovery: 0]
[TypeId: ns3::TcpSocketBase]
[RTT: 12.3535 milliseconds]


############################################## Fin Datos Simulación ###############################################################
```

En la siguiente ilustración se puede observar el comportamiento del *RTT* a lo largo de las simulaciones para cada uno de los tipos de *socket*. Se puede comprobar que los resultados presentados anteriormente se corresponden aproximadamente con lo que se muestra en la gráfica, ya que el valor que se presenta del *RTT* es la media de todas las medidas tomadas durante la simulación.

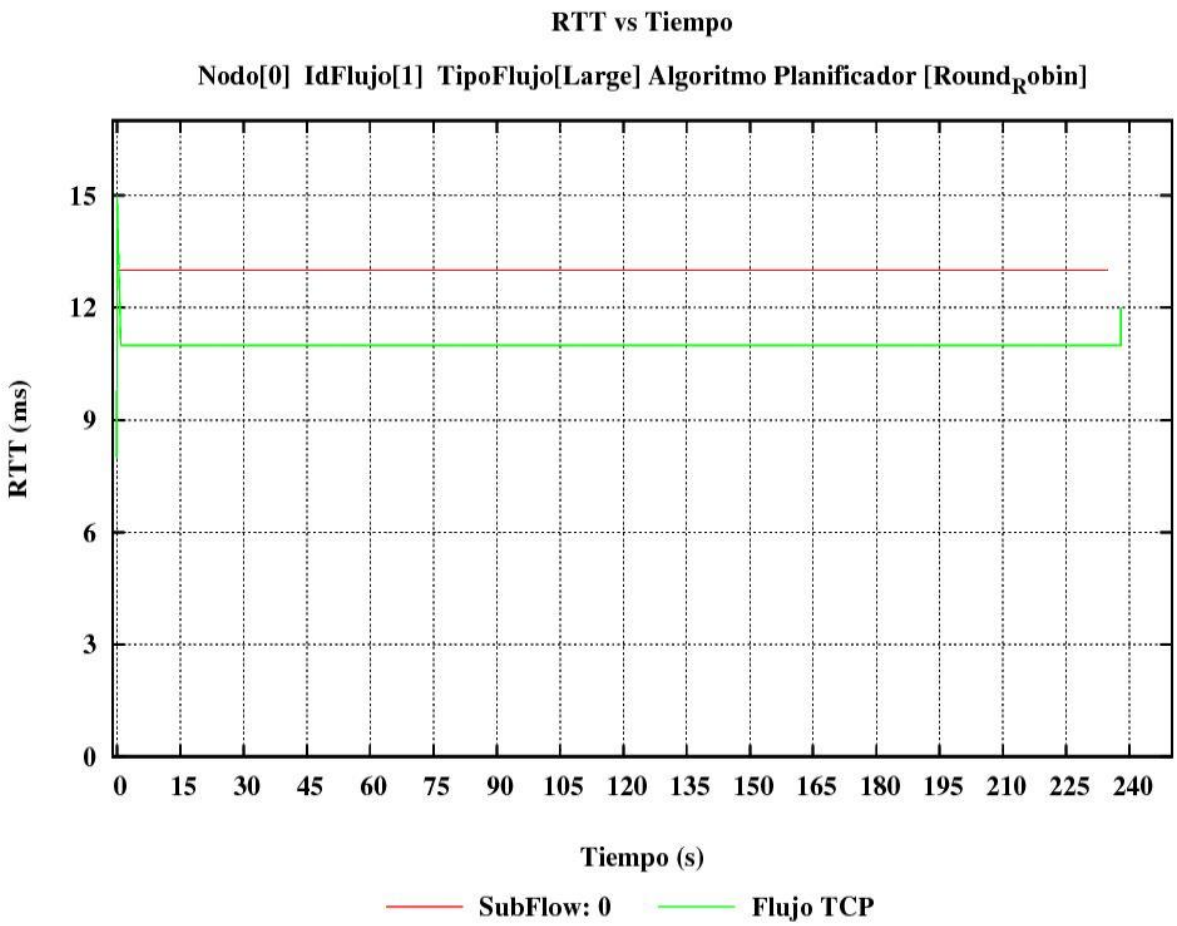

Ilustración 5.3: Comparación MpTCP y TCP 1 solo flujo

Otra prueba para validar la implementación ha sido comparar los cálculos teóricos de parámetros del modelo, con los que se han obtenido en la simulación. Entre los parámetros susceptibles de ser calculados, encontramos:

- *RTT* (*round trip time*): En el apartado 4.4 se detalla en que se basa este parámetro. Para ello se ha calculado el valor teórico de la siguiente forma:

  Bits = número de bits enviados.

  Capacidad = capacidad del canal.

  Retardo = tiempo que en que la información tarda en transmitirse por el canal.

  Tdatos = tiempo que tardan los datos en llegar al destino = (Bits/Capacidad) + Retardo

  Tack = tiempo que tarda en llegar el asentimiento al origen = (Bits/Capacidad) + Retardo

  $$RTT = Tdatos + Tack$$

  De esta forma, para los valores definidos en la simulación, tenemos que:

  Bits = 1462 bytes * 8 (a nivel de enlace)

  Capacidad = 1Mbps

  Retardo = 500 us

  Tdatos = ((11696 / 10^6) * 1000 + 0.5ms) = 12.19 ms

  Tack = ((42 / 10^6) * 1000 + 0.5ms) = 0.542 ms

  RTT = Tdatos + Tack = 12.732 ms

Una vez realizado el cálculo teórico podemos compararlo con el obtenido en la simulación. Para ello podemos comprobar los resultados anteriores en la sección de *MpTCP*, en la que hemos obtenido los resultados de la simulación y efectivamente se comprueba que el RTT es similar al calculado de forma teórica. Tras realizar esta validación y haber obtenido un resultado acorde a lo que se esperaba, se puede afirmar que una sesión *MpTcp* con un solo flujo se comporta de forma análoga que una sesión TCP.

5.1.2 Validación de mensajes con la herramienta Wireshark

Este apartado está dedicado a comprobar los mensajes *MpTCP* que se intercambian en la red. Para ello haremos uso del analizador de protocolos Wireshark. La herramienta puede descargarse desde su página web https://www.wireshark.org/download.html.

En el apartado 3.2.2 están definidas las opciones más usadas. El escenario en el que se analizarán los mensajes es el presentado en la Ilustración 6.1: Topología 1.

En primer lugar, observaremos la opción **OPT_MPC** que va en los segmentos intercambiados en el TWHS.

Justo tras el establecimiento de la conexión, debido a la configuración *FullMesh* (véase el apartado 3.2.2.2.1), se anuncian todas las direcciones desde las que los dispositivos están disponibles, con la opción **OPT_ADD_ADDR.**

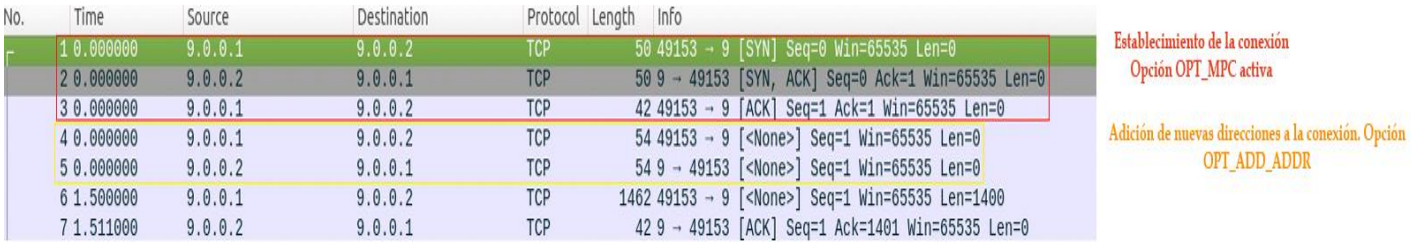

Ilustración 5.4: Validación de los mensajes

Este es el primer paquete intercambiado en la conexión, el que porta la opción **OPT_MPC**. Le siguen otros dos paquetes hasta completar el *TWHS*. Se puede comprobar cómo lleva activo el bit **SYN** además de llevar la opción **OPT_MPC** en el campo de opciones.

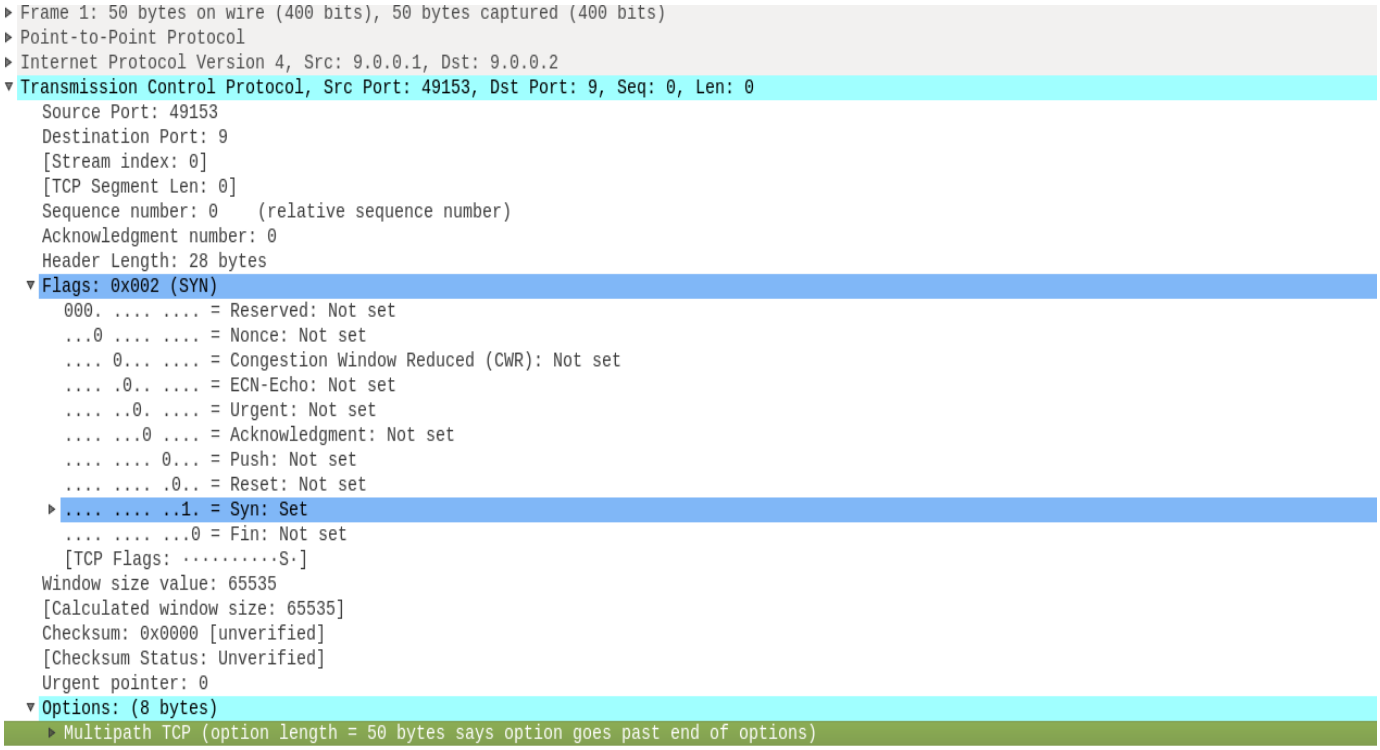

Ilustración 5.5: Opción MPC

En la siguiente figura observamos la opción **ADD_ADDR**, que como se indicó anteriormente, tiene un valor hexadecimal de 0x20. Se puede comprobar como en este caso el bit **SYN** no está activo Esta opción es transmitida en ambos sentidos de la comunicación para informar tanto al emisor como al receptor las direcciones disponibles. Una vez las direcciones han sido anunciadas, según la política de cada dispositivo (tal como se indica en el apartado 3.2.2.2.1) se establecerá o no nuevos subflujos en la sesión *MpTCP*.

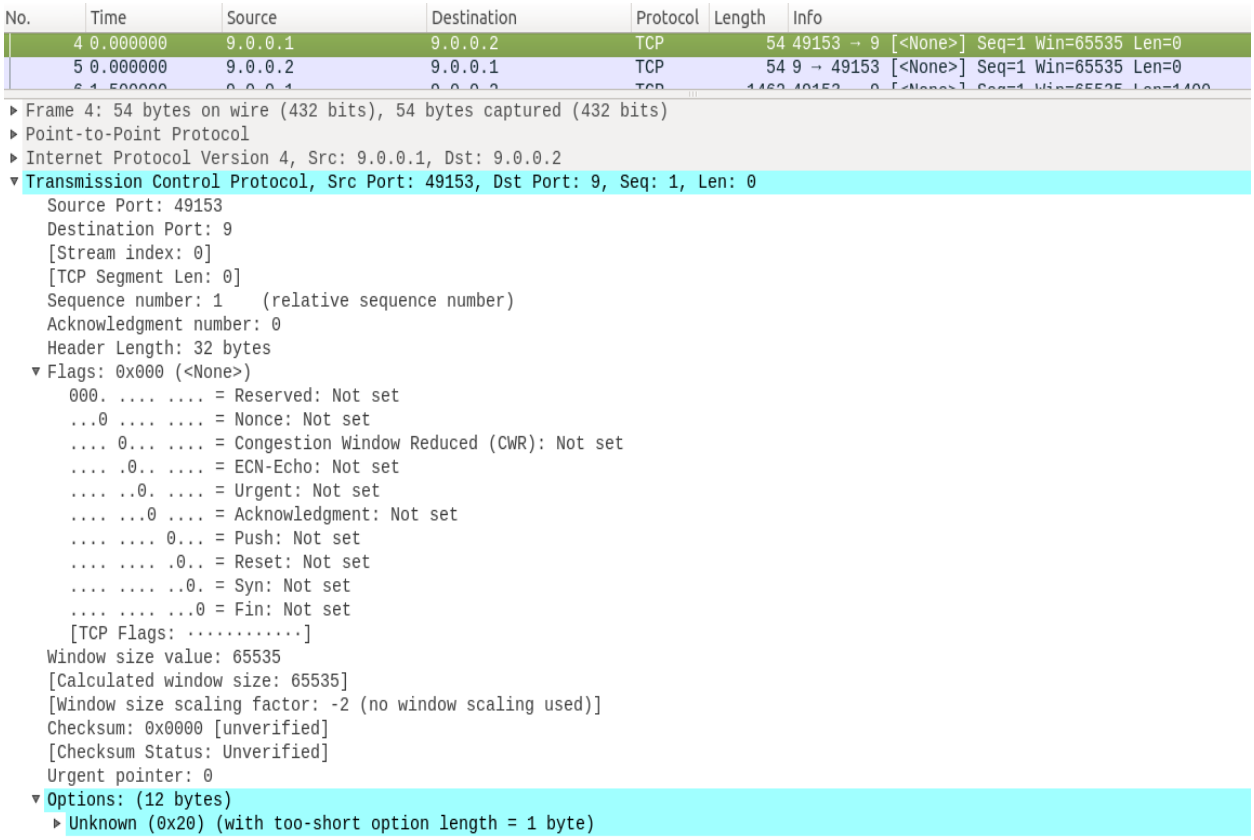

Ilustración 5.6: Opción ADD_ADDR

Esta opción, tal como se comentó en el apartado 3.2.2.2.1, es similar al *TWHS*, a excepción de la opción **JOIN** que lleva en el campo de opciones. Comprobamos que el bit **SYN** está activo, se debe a que se está iniciando un nuevo subflujo.

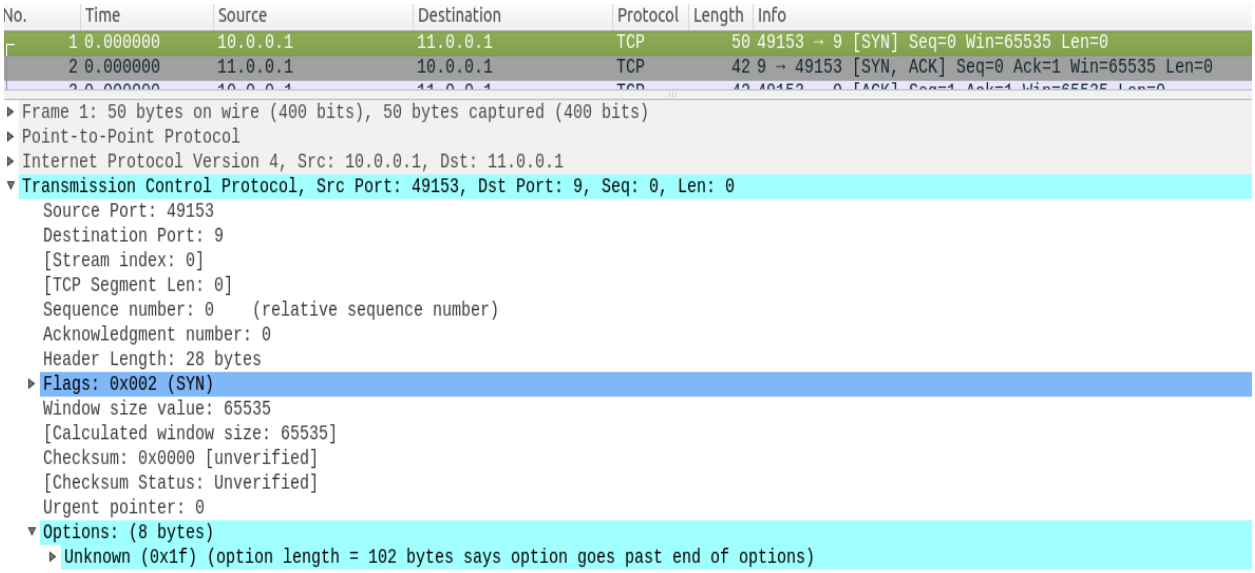

Ilustración 5.7: Opción JOIN

5.2 Pruebas unitarias automatizadas

En este apartado se describen las pruebas que se han implementado en el simulador para comprobar la validez de la implementación, tanto la actual como la de futuras mejoras implementadas.

Para ello se ha desarrollado el fichero ***mptcp-test.cc*** en el cual se implementa la suite de test de *MpTCP*, siguiendo la estructura de los test que se indica en el ANEXO D: Test en Ns3

5.2.1 Establecimiento de la sesión MpTCP

Puesto que la implementación que se está validando no dispone de ningún test para comprobar su validez, se ha creado una suite de test para validarla. Esta se ha situado dentro del directorio test del módulo de Internet, donde también se encuentran los test de *TCP*, valida que el establecimiento de una sesión *MpTCP* se establezca correctamente.

Para ello lo que se está comprobando es que el TWHS se esté produciendo correctamente cuando configuramos dos dispositivos en un enlace que usen *MpTCP*. En la siguiente ilustración se muestra la forma en la que se ha configurado la topología, puesto que cuando se codifican los test no deben usarse *helpers*.

```
void
MptcpTestCaseMpEnable::SetupDefaultSim (void)
{
        Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue(MpTcpSocket-
Base::GetTypeId()));
        Config::SetDefault("ns3::MpTcpSocketBase::PathManagement", StringValue("FullMesh"));

          const char* netmask = "255.255.255.0";
          const char* ipaddr0 = "192.168.1.1";
          const char* ipaddr1 = "192.168.1.2";
          Ptr<Node> node0 = CreateInternetNode ();
          Ptr<Node> node1 = CreateInternetNode ();
          Ptr<SimpleNetDevice> dev0 = AddSimpleNetDevice (node0, ipaddr0, netmask);
          Ptr<SimpleNetDevice> dev1 = AddSimpleNetDevice (node1, ipaddr1, netmask);

          Ptr<SimpleChannel> channel = CreateObject<SimpleChannel> ();
          dev0->SetChannel (channel);
          dev1->SetChannel (channel);

          Ptr<MpTcpSocketBase> server;
          Ptr<MpTcpSocketBase> client;
          server = DynamicCast<MpTcpSocketBase>(Socket::CreateSocket (node0,
                                                TcpSocketFactory::GetTypeId ()));

          client = DynamicCast<MpTcpSocketBase>(Socket::CreateSocket (node1,
                                                TcpSocketFactory::GetTypeId ()));
          uint16_t port = 50000;
          InetSocketAddress serverlocaladdr (Ipv4Address::GetAny (), port);
          InetSocketAddress serverremoteaddr (Ipv4Address (ipaddr0), port);

          server->Bind (serverlocaladdr);
          server->Listen ();
          server->SetAcceptCallback (MakeNullCallback<bool, Ptr< Socket >, const Address &> (),
                        MakeCallback (&MptcpTestCaseMpEnable::ServerHandleConnectionCreated,this));

          client->Connect (serverremoteaddr);

}
```

Ilustración 5.8: Test establecimiento conexión

Una vez hemos codificado el test para comprobar que la implementación lo cumple tenemos que ejecutar el test de validación. En el ANEXO D: Test en Ns3, se muestra cómo podemos depurar los test y obtener más información de si es apto o no.

```
Starting program: /home/salas/ns-allinone-3.19/ns-3.19/build/utils/ns3.19-test-runner-debug --suite=mptcp --verbose
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Connect -> SegmentSize: 536 tcpSegmentSize: 0 segmentSize: 536SendingBufferSize: 131072
0 [1] (0) NULL SendEmptyPacket -> backoffCount: 1 RTO: 3 cnTimeout: 3 cnCount: 5
[1] (0) SendEmptyPacket -> LOCALTOKEN is mapped to connection endpoint -> 1619568071 -> 0x66ad20 TokenMapsSize: 1
0x669490 [1](0) SendEmptyPacket -> [ SYN ] ReTxTimer set for SYN / SYN+ACK now 0 Expire at 3 RTO: 3 FlowType: NULL Header: 49153 > 50000 [ SYN ] Seq=264 Ack=0
 Win=65535 {OPT_MPC(1619568071)}
Listening socket receives SYN packet, it need to be CLONED... 49153 > 50000 [ SYN ] Seq=264 Ack=0 Win=65535 {OPT_MPC(1619568071)}
0 [0] (0) NULL SendEmptyPacket -> backoffCount: 1 RTO: 3 cnTimeout: 3 cnCount: 5
[0] (0) SendEmptyPacket -> LOCALTOKEN is mapped to connection endpoint -> 1289261353 -> 0x667620 TokenMapsSize: 1
0x66b360 [0](0) SendEmptyPacket -> [ SYN  ACK ] ReTxTimer set for SYN / SYN+ACK now 0 Expire at 3 RTO: 3 FlowType: NULL Header: 50000 > 49153 [ SYN  ACK ] Seq
=160 Ack=265 Win=65535 {OPT_MPC(1289261353)}
mpEnabled: 1
PASS mptcp 0.060 s
  PASS Establecimiento de conexión MpTcp 0.060 s
[Inferior 1 (process 11222) exited normally]
(gdb)
```

Ilustración 5.9: Depuración test *MpTCP*

6 CONFIGURACIONES TIPO

Somos lo que hacemos de forma repetida. La excelencia, entonces, no es un acto, sino un hábito.
–Aristóteles-

Una vez realizados el estudio y la implementación del protocolo, se mostrarán algunos de los resultados que se han obtenido que ayudarán a sacar las conclusiones.

6.1 Primera topología

A continuación, se muestra la topología que se ha configurado. El objetivo de esta topología es demostrar el funcionamiento del protocolo, así como las mejoras que se han implementado.

Para ello, se han realizado dos simulaciones que representan una transmisión de datos entre dos dispositivos, ambas usando el protocolo *MpTCP*. Una usa el algoritmo de selección de rutas ***Round_Robin*** y otra el ***Fastest_Rtt***. En ambas simulaciones la cantidad de datos a transmitir es la misma, **10 Mbyte**.

A continuación, se detallan las características de cada enlace:

- Enlace 1: Capacidad: **1 Mbps**, Retardo: **50us**
- Enlace 2: Capacidad: **10 Mbps**, Retardo: **500us**
- Enlace 3: Capacidad: **6 Mbps**, Retardo: **500us**

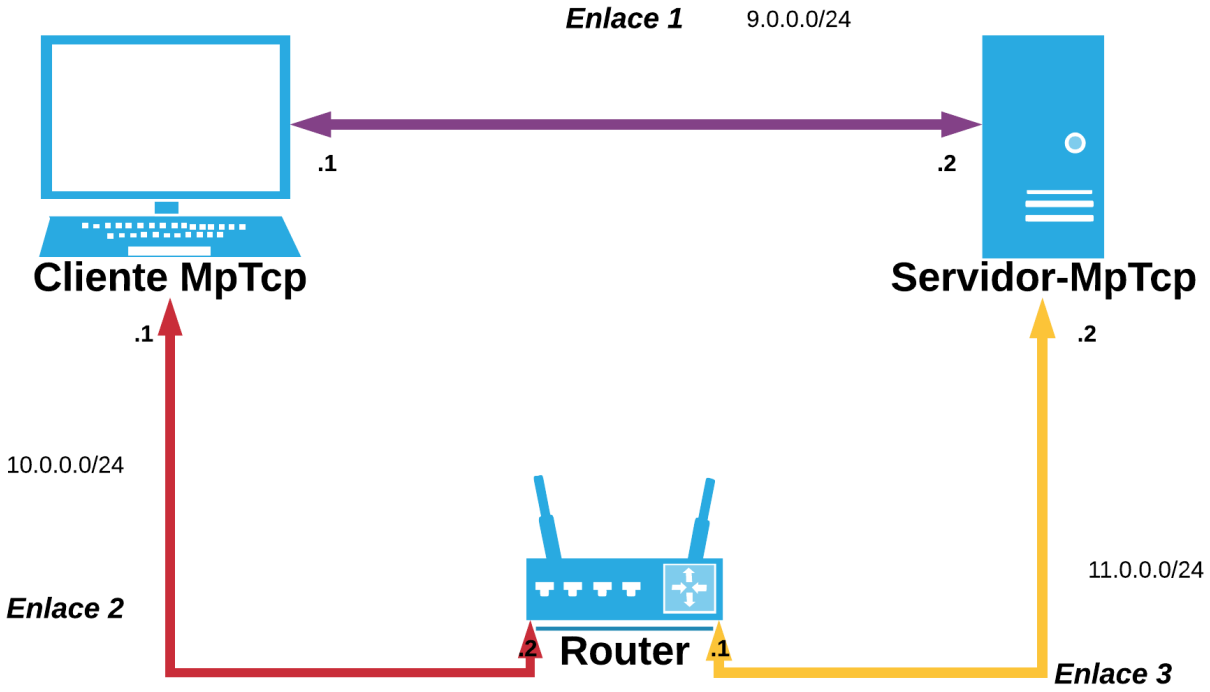

Ilustración 6.1: Topología 1

Los resultados de la simulación usando el algoritmo ***Fastest_Rtt*** han sido los siguientes:

```
############################################ Datos Simulación Socket MpTcp ##############################################

[NodeId: 0 ]
[FlowId: 1 ]
[FlowType: Large]
[Throughput: 2.03343 Mbps]
[FlowComplTime: 38.831 seconds ]
[timeOut:251 ]
[fastReTx: 163 ]
[PartialAck: 274]
[FullAck: 125 ]
[FastRecovery: 3444 ]
[Subflows Size: 2 ]
[EstSubflows: 2 ]
[TypeId: ns3::MpTcpSocketBase ]
[Distribution algorithm: Fastest_Rtt ]

------------ Specific SubFlows ------------------

[Src/port: 9.0.0.1/49153  Dest/port: 9.0.0.2/9 ]
[RTT mean Subflow 0 : 14.1294 milliSeconds ]
[Fast Retransmit threshold subflow 0: 3 ]
[Numero de paquetes transmitidos por el SubFlow 0 : 1405 ]

[Src/port: 10.0.0.1/49153  Dest/port: 11.0.0.1/9 ]
[RTT mean Subflow 1 : 2 milliSeconds ]
[Fast Retransmit threshold subflow 1: 3 ]
[Numero de paquetes transmitidos por el SubFlow 1 : 5645 ]


############################################ Fin Datos Simulación ########################################################
```

Si se observan los resultados se puede comprobar que la mayoría de los paquetes han sido enviados por la ruta que tenía un RTT menor, en este caso se corresponde la ruta compuesta por el enlace 2 y 3. El enviar los paquetes por la ruta más rápida dota de eficiencia en comparación de si el algoritmo usado fuera ***Round_Robin***, ya que no se estaría usando al máximo los recursos disponibles en la red. Se estaría desaprovechando el transmitir por una ruta más rápida. Como podemos observar en la siguiente ilustración, el número de paquetes transmitidos es superior por el subflujo 1. Era lo que esperábamos ya que el algoritmo ha elegido la mejor ruta para transmitir.

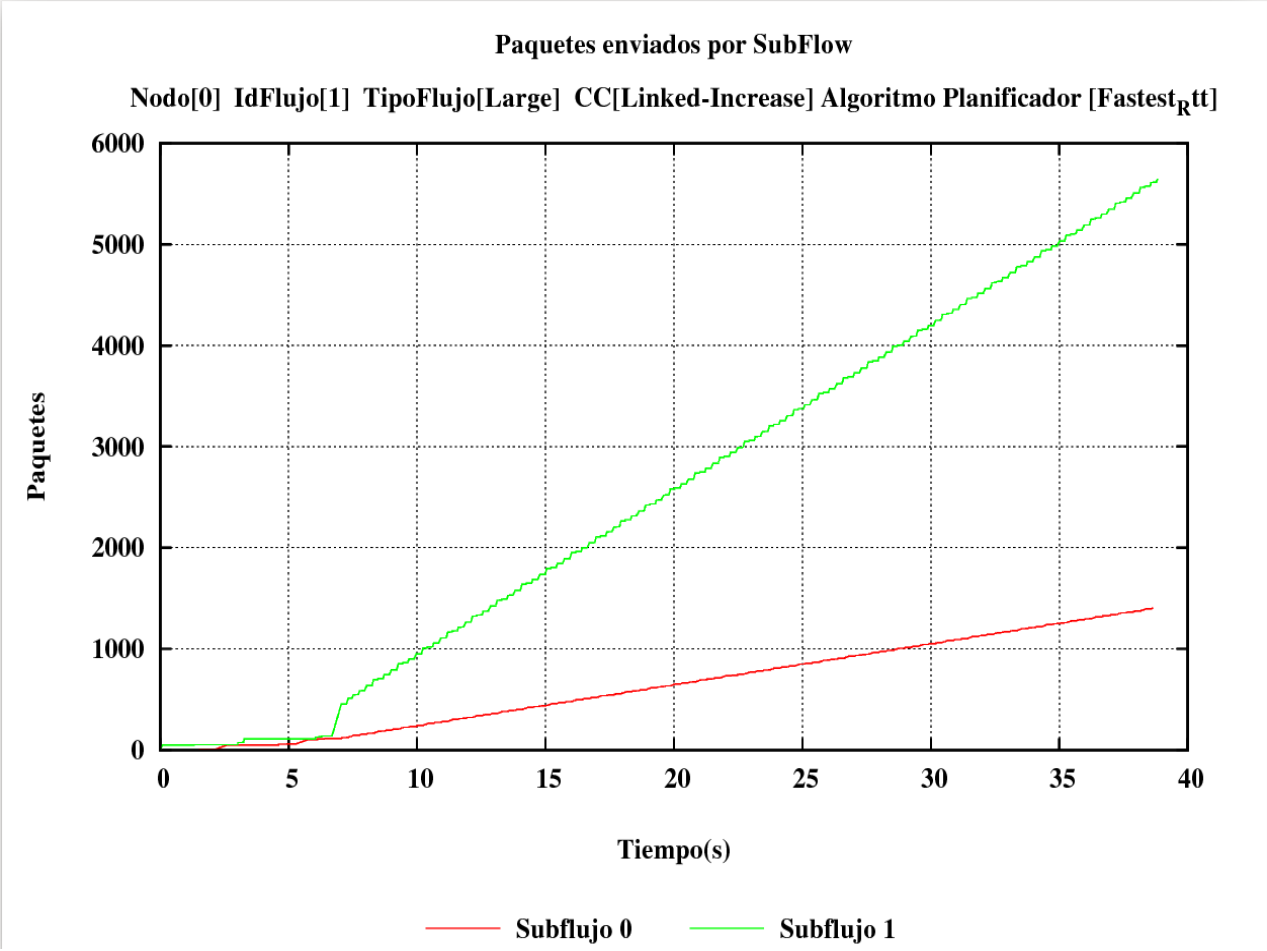

Ilustración 6.2: Fastest_Rtt paquetes vs tiempo

Ahora se presentan los resultados de la simulación realizada con el algoritmo ***Round_Robin***. Este algoritmo reparte equitativamente la capacidad de transmitir una información por un determinado subflujo de forma que, todos los subflujos disponibles disponen de la misma probabilidad de transmitir información.

```
########################################### Datos Simulación Socket MpTcp ##############################################

[NodeId: 0 ]
[FlowId: 1 ]
[FlowType: Large]
[Throughput: 0.686597 Mbps]
[FlowComplTime: 115.002 seconds ]
[timeOut:809 ]
[fastReTx: 1 ]
[PartialAck: 4]
[FullAck: 1 ]
[FastRecovery: 9 ]
[Subflows Size: 2 ]
[EstSubflows: 2 ]
[TypeId: ns3::MpTcpSocketBase ]
[Distribution algorithm: Round_Robin ]

------------ Specific SubFlows -------------------

[Src/port: 9.0.0.1/49153  Dest/port: 9.0.0.2/9 ]
[RTT mean Subflow 0 : 13.5082 milliSeconds ]
[Fast Retransmit threshold subflow 0: 3 ]
[Numero de paquetes transmitidos por el SubFlow 0 : 3246 ]

[Src/port: 10.0.0.1/49153  Dest/port: 11.0.0.1/9 ]
[RTT mean Subflow 1 : 2 milliSeconds ]
[Fast Retransmit threshold subflow 1: 3 ]
[Numero de paquetes transmitidos por el SubFlow 1 : 3804 ]


########################################### Fin Datos Simulación ########################################################
```

Se observa como el tiempo empleado para transmitir los mismos datos es mayor. En este caso se ha empleado 115 segundos frente a los 38.8 que ha tardado con el algoritmo ***Fastest_Rtt.*** El número de paquetes transmitido por cada subflujo es similar no teniendo en cuenta el RTT de cada subflujo, 13.5 ms para el subflujo 0 y 2 ms para el subflujo 1. En la Ilustración 6.2, se puede comprobar como el algoritmo está priorizando sobre la ruta con menor RTT. Por otro lado, si ahora comparamos el caudal obtenido en las distintas simulaciones observamos que la simulación usando el algoritmo basado en el *RTT* ha obtenido uno notablemente mayor que si usamos el algoritmo de reparto equitativo.

En la gráfica que se muestra a continuación podemos observar como usando el algoritmo ***Round_Robin*** la cantidad de paquetes transmitido por cada subflujo es similar. Esta situación podría ser de utilidad en el caso de que se quisiera hacer balanceo de carga sacrificando el rendimiento de la sesión.

Para concluir con los resultados obtenidos en esta topología se comenta que el algoritmo basado en el *RTT* que se ha implementado es más eficiente que el ***Round_Robin*** existente.

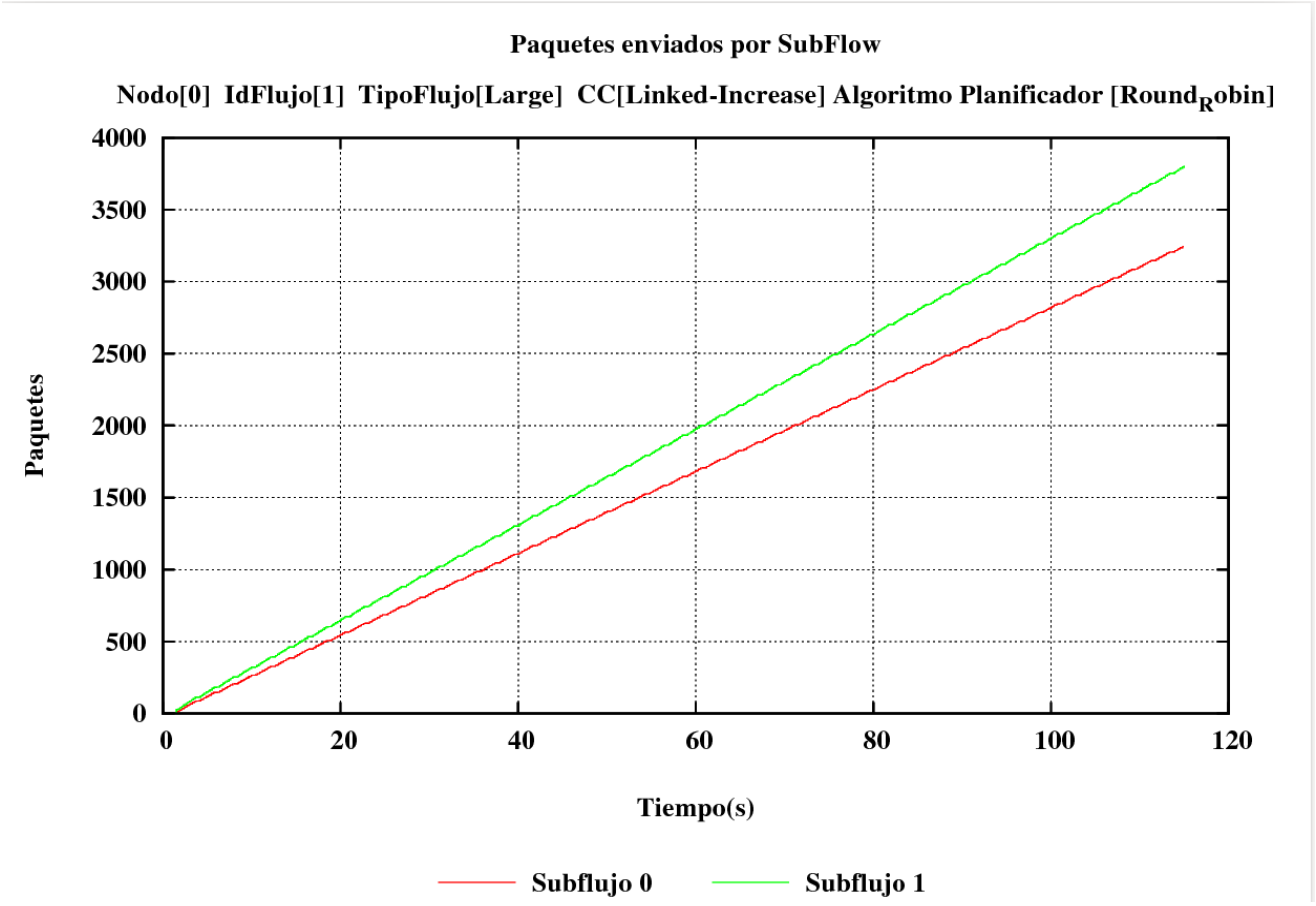

Ilustración 6.3: Round_Robin paquetes vs tiempo

6.2 Segunda topología

A continuación, se muestra un escenario donde existen conexiones redundantes entre distintas máquinas, lo que se puede asimilar a un centro de procesamiento de datos o ***CPD***. Es aquí donde *MpTCP* obtiene un mayor rendimiento que *TCP* debido a que las rutas disponibles para añadir redundancia serán aprovechadas para transmitir datos de una misma conexión. Se ha de tener en cuenta que el primer subflujo se establece por el enlace directo entre ambos dispositivos que en este caso es el enlace 1.

En este escenario lo que ha simulado es la diferencia entre usar *TCP* y *MpTCP* cuando existen más rutas disponibles, para comprobar que realmente se obtiene mayor rendimiento cuando se usa *MpTCP*. Para ello se va a realizar una simulación usando MpTCP y otra usando TCP. Todos los enlaces al iniciar la simulación tienen la misma capacidad y mismo retardo. La cantidad de datos a transmitir son **50 Mbyte**.

Destacar que durante estas simulaciones se han usado *sockets* de tipo *MpTCP*, debido a que en el apartado 5.1.1 se ha comprobado la equivalencia entre *TCP* y *MpTCP* con un solo subflujo.

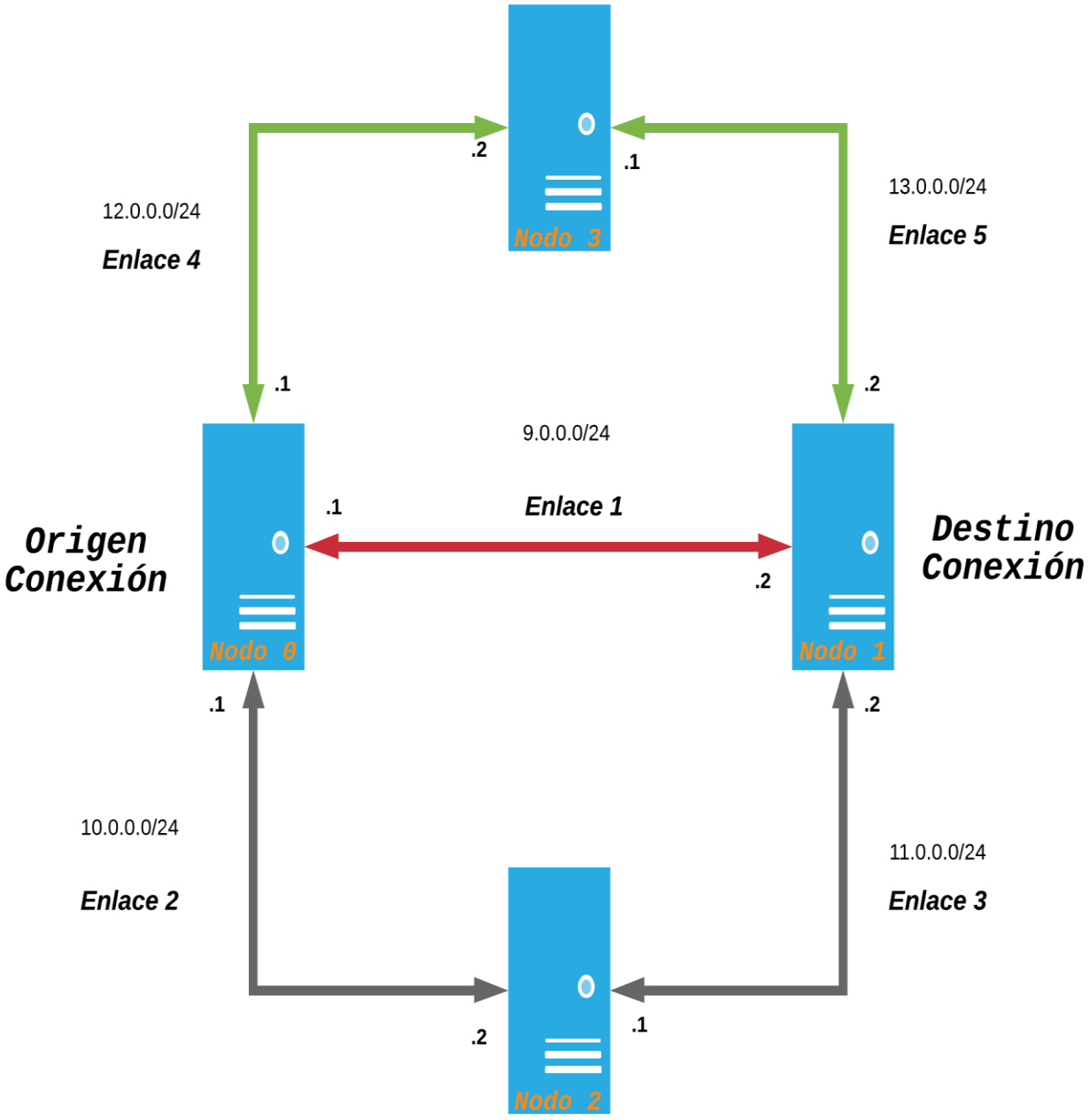

Ilustración 6.4: Topología 2

Al inicio de la simulación todos los enlaces tienen la siguiente configuración:

- Capacidad: **1 Mbps**, Retardo: **500us**

Los eventos planificados (Ilustración 6.6), modificarán los siguientes enlaces:

- Enlace 1: Capacidades: **100 Kbps**, **3 Mbps, 500 Kbps** Retardos: **5 ms, 500 us, 500 us**
- Enlace 5: Capacidad: **100 Kbps** Retardo: **3 ms**

Se establecen tantos subflujos como interfaces se disponga. De esta forma, los subflujos están establecidos entre los siguientes enlaces:

- Subflujo 0: enlace 1
- Subflujo 1: enlaces 2 y 3
- Subflujo 2: enlaces 4 y 5

Los enlaces que pertenecen al mismo subflujo se muestran del mismo color, en la Ilustración 6.4.

En esta simulación se han planificado eventos para emular congestión en los enlaces y así observar las decisiones del algoritmo planificador, que en este caso será ***Fastest_Rtt***. Para ello se ha elaborado la siguiente función que se encarga de ello.

```
/**
* Está función simula la congestión de un enlace punto a punto, para ello,
* recibe como parámetro los netDevice de cada dispositivo y el canal.
*/
void
cambiarCongestion (Ptr<NetDevice> nd, Ptr<NetDevice> nd1, std::string canal,
                     std::string rate,std::string delay){

        //Pasado un tiempo, el enlace (canal-1), disminuye o aumenta su capacidad y retardo.

      Config::Set("/ChannelList/" + canal + "/$ns3::PointToPointChannel/Delay",StringValue(delay));
      nd->SetAttribute("DataRate", StringValue(rate));
      nd1->SetAttribute("DataRate", StringValue(rate));

}
```

Ilustración 6.5: Función para emular la saturación del enlace

A continuación, se muestran los eventos planificados:

```
/**
* PROGAMAMOS EVENTOS DURANTE LA SIMULACIÓN:
*/
Simulator::Schedule(Seconds(10.0),cambiarCongestion,nodosEnlace1.Get(1)->GetDevice(0),
        nodosEnlace1.Get(0)->GetDevice(0),"1","100kbps","5ms");

Simulator::Schedule(Seconds(15.0),cambiarCongestion,nodosEnlace5.Get(1)->GetDevice(2),
        nodosEnlace5.Get(0)->GetDevice(1),"4","100kbps", "3ms");

Simulator::Schedule(Seconds(25.0),cambiarCongestion,nodosEnlace1.Get(1)->GetDevice(0),
        nodosEnlace1.Get(0)->GetDevice(0),"1","3Mbps", "500us");

Simulator::Schedule(Seconds(70.0),cambiarCongestion,nodosEnlace1.Get(1)->GetDevice(0),

        nodosEnlace1.Get(0)->GetDevice(0),"1","500Kbps", "500us");
```

Ilustración 6.6: Eventos planificados durante la simulación

Ahora se presentan los resultados que han sido obtenidos durante las simulaciones

```
########################################### Datos Simulación Socket MpTcp ###############################################

[NodeId: 0 ]
[FlowId: 1 ]
[FlowType: Large]
[Throughput: 1.94742 Mbps]
[FlowComplTime: 204.863 seconds ]
[timeOut:154 ]
[fastReTx: 818 ]
[PartialAck: 1100]
[FullAck: 719 ]
[FastRecovery: 7422 ]
[Subflows Size: 3 ]
[EstSubflows: 3 ]
[TypeId: ns3::MpTcpSocketBase ]
[Distribution algorithm: Fastest_Rtt ]

------------ Specific SubFlows -------------------

[Src/port: 9.0.0.1/49153  Dest/port: 9.0.0.2/9 ]
[RTT mean Subflow 0 : 38.4139 milliSeconds ]
[Fast Retransmit threshold subflow 0: 3 ]
[Numero de paquetes transmitidos por el SubFlow 0 : 15972 ]

[Src/port: 10.0.0.1/49153  Dest/port: 11.0.0.2/9 ]
[RTT mean Subflow 1 : 43.9566 milliSeconds ]
[Fast Retransmit threshold subflow 1: 3 ]
[Numero de paquetes transmitidos por el SubFlow 1 : 16657 ]

[Src/port: 12.0.0.1/49153  Dest/port: 13.0.0.2/9 ]
[RTT mean Subflow 2 : 220.724 milliSeconds ]
[Fast Retransmit threshold subflow 2: 3 ]
[Numero de paquetes transmitidos por el SubFlow 2 : 2992 ]


########################################### Fin Datos Simulación ########################################################

 ########################################### Datos Simulación Socket Tcp ################################################

 [NodeId: 0 ]
 [FlowId: 1 ]
 [FlowType: Large]
 [Throughput: 0.765739 Mbps]
 [FlowComplTime: 521.007 seconds ]
 [timeOut:1 ]
 [fastReTx: 1979 ]
 [PartialAck: 4]
 [FullAck: 1978 ]
 [FastRecovery: 1981 ]
 [Subflows Size: 1 ]
 [EstSubflows: 1 ]
 [TypeId: ns3::MpTcpSocketBase ]
 [Distribution algorithm: Fastest_Rtt ]

 [RTT: 34.0878 milliSeconds]
 [Fast Retransmit threshold: 3 ]

 ########################################### Fin Datos Simulación ########################################################
```

Observando los resultados de la simulación usando *MpTCP* se puede comprobar cómo, al tener el algoritmo ***Fastest_Rtt*** activado, la mayoría de los paquetes han sido transmitidos por la ruta con un RTT menor. Se puede comprobar que el tiempo de la transmisión es notablemente menor al de la simulación TCP, puesto que estamos usando tres rutas distintas en la misma transmisión. En las siguientes gráficas se muestran algunos parámetros controlados en la simulación, como son el número de paquetes enviado y el RTT de cada uno de los flujos de datos.

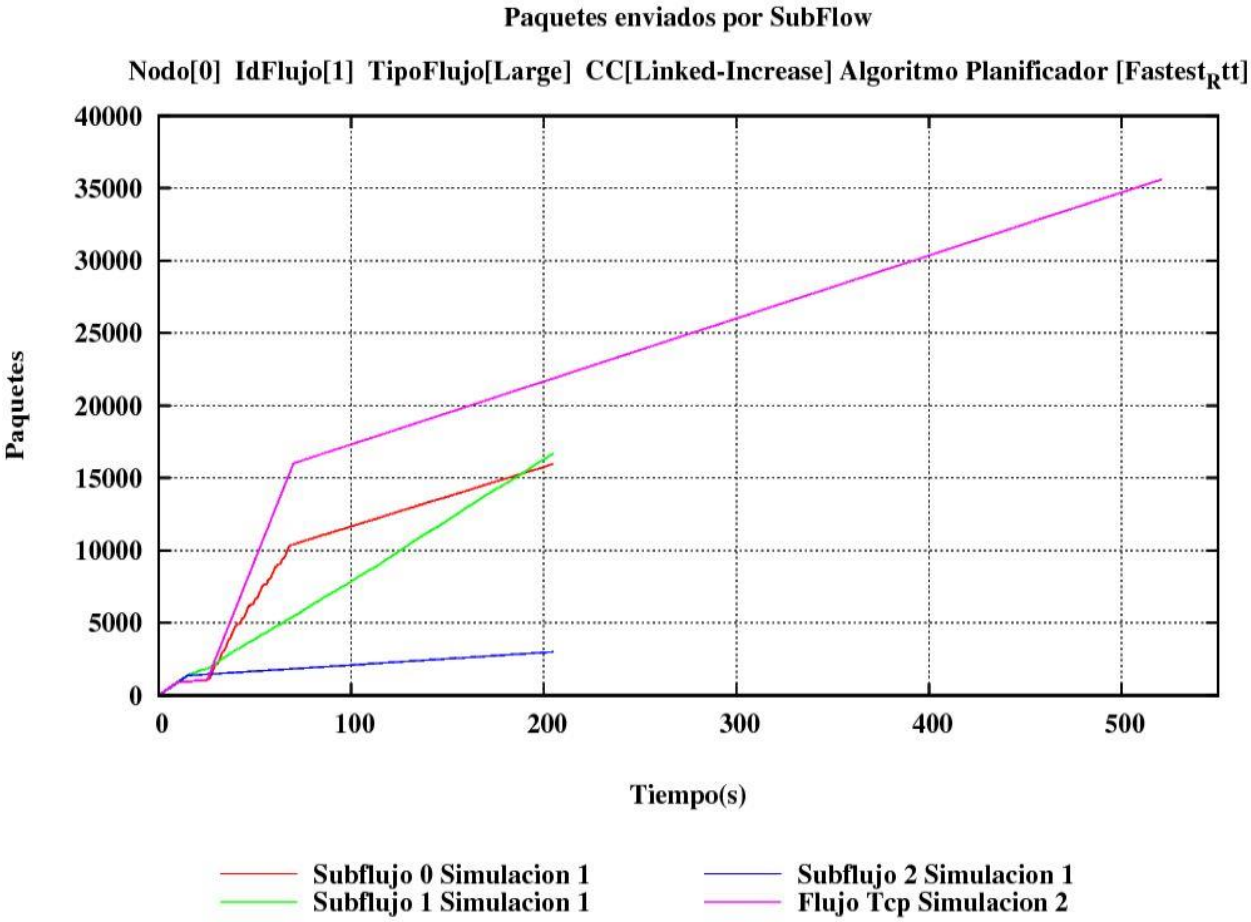

Ilustración 6.7: Paquetes por Subflujos MpTCP y Flujo TCP

En la Ilustración 6.8, se muestra el comportamiento del *RTT* ante los eventos que han sido programados. Observando la Ilustración 6.6, donde se han planificado los eventos, los tiempos en los que estos han sido programados coinciden con los cambios bruscos que se presentan en ambas gráficas Si nos fijamos estos cambios se producen a la par en ambas gráficas (Paquetes enviados y *RTT*), debido a que los cambios en el *RTT* de cada subflujo influyen directamente sobre que subflujo se elige para transmitir, si estamos usando el algoritmo ***Fastest_Rtt*** en el caso de *MpTCP*.

En el caso de TCP también se puede observar que tras producirse los eventos ambas gráficas cambian, esto se debe a que estamos directamente modificando las características del enlace, y esto se refleja en una pendiente mayor para la gráfica de paquetes en el caso de que se mejore las propiedades del enlace o menor en caso contrario.

Se puede observar como el protocolo *MpTCP* ante un aumento inesperado del *RTT* en un subflujo empieza a transmitir los datos por el subflujo con menor *RTT*, que es justamente lo que debe de hacer el algoritmo. Esto se puede observar en la Ilustración 6.7 a partir del instante 25 que es cuando se ha programado el evento para aumentar la capacidad y disminuir el retardo del enlace 1. Por lo que el subflujo 0 será el que tenga un menor retardo y la mayoría de los paquetes serán enviado por ese subflujo.

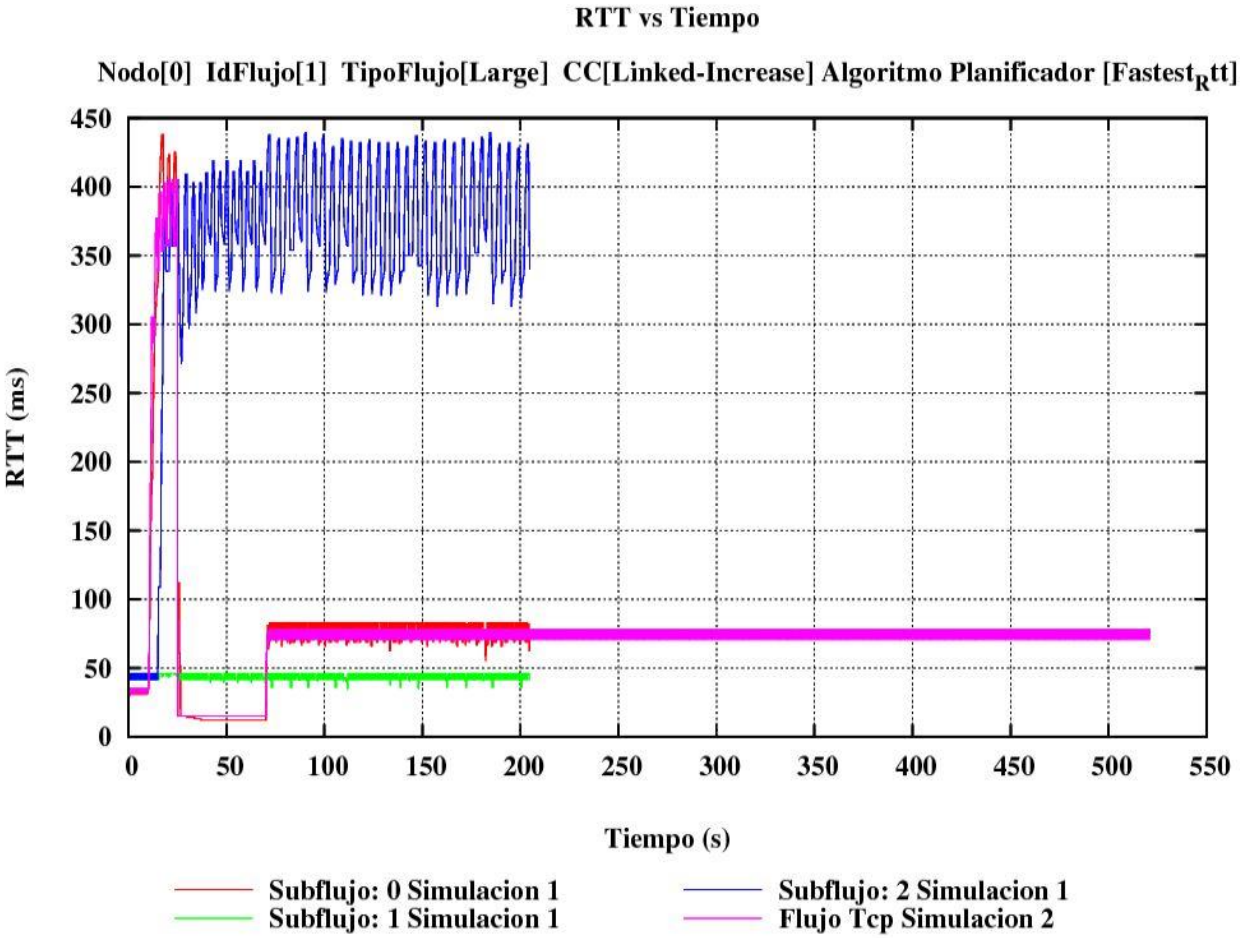

Ilustración 6.8: RTT por Subflujo MpTCP y Flujo TCP

Observamos que el tiempo que ha tomado la simulación es notablemente mayor que la simulación donde hemos usado *MpTCP*, 521 segundos frente a los 204 anteriores. Era lo que se esperaba, debido a que en la simulación *MpTCP* para la misma cantidad de datos a transmitir se han utilizado tres flujos distintos. Por lo tanto, el caudal de la sesión cuando usamos *MpTCP* es bastante mayor que si usamos *TCP*, lo que lleva a la conclusión de, si tenemos enlaces redundantes y el protocolo *MpTcp* activo, la conexión mejoraría notablemente.

A continuación, se muestra un diagrama de los mensajes intercambiados durante la conexión *MpTCP* establecida en la topología (Ilustración 6.4). Se detallarán solo los mensajes implicados en: Establecimiento de la conexión, anuncio de direcciones, establecimiento de nuevos subflujos, cierre de subflujos y envío de datos. En el cierre de los subflujos solo se ha representado el del principal, pero cada subflujo establecido se cierra como una conexión *TCP* (véase Ilustración 3.3).En el diagrama se ha representado mediante 3 puntos consecutivos que se producen otros intercambios de mensajes no reflejados.

En primer lugar y como se ha comentado en apartados anteriores de esta memoria, los mensajes que se intercambian son los del establecimiento de la conexión. Una vez que la conexión ha sido establecida, tanto origen como destino de la conexión *MpTCP* anuncian las direcciones de sus interfaces sobre las que no está establecida la conexión. El primer subflujo puede empezar a enviar información antes de anunciar las direcciones disponibles. Una vez las direcciones han sido anunciadas es en este momento donde se negocia el establecimiento de nuevos subflujos, para ello se hace uso de la opción JOIN en la que se debe identificar la conexión *MpTCP* sobre la que se quiere añadir el nuevo subflujo.

Cuando los subflujos adicionales hayan sido establecidos, estos pueden empezar a enviar información. Cuando no se desee enviar más información por un subflujo, de forma análoga a *TCP*, se envía un paquete con el bit **FIN** activo.

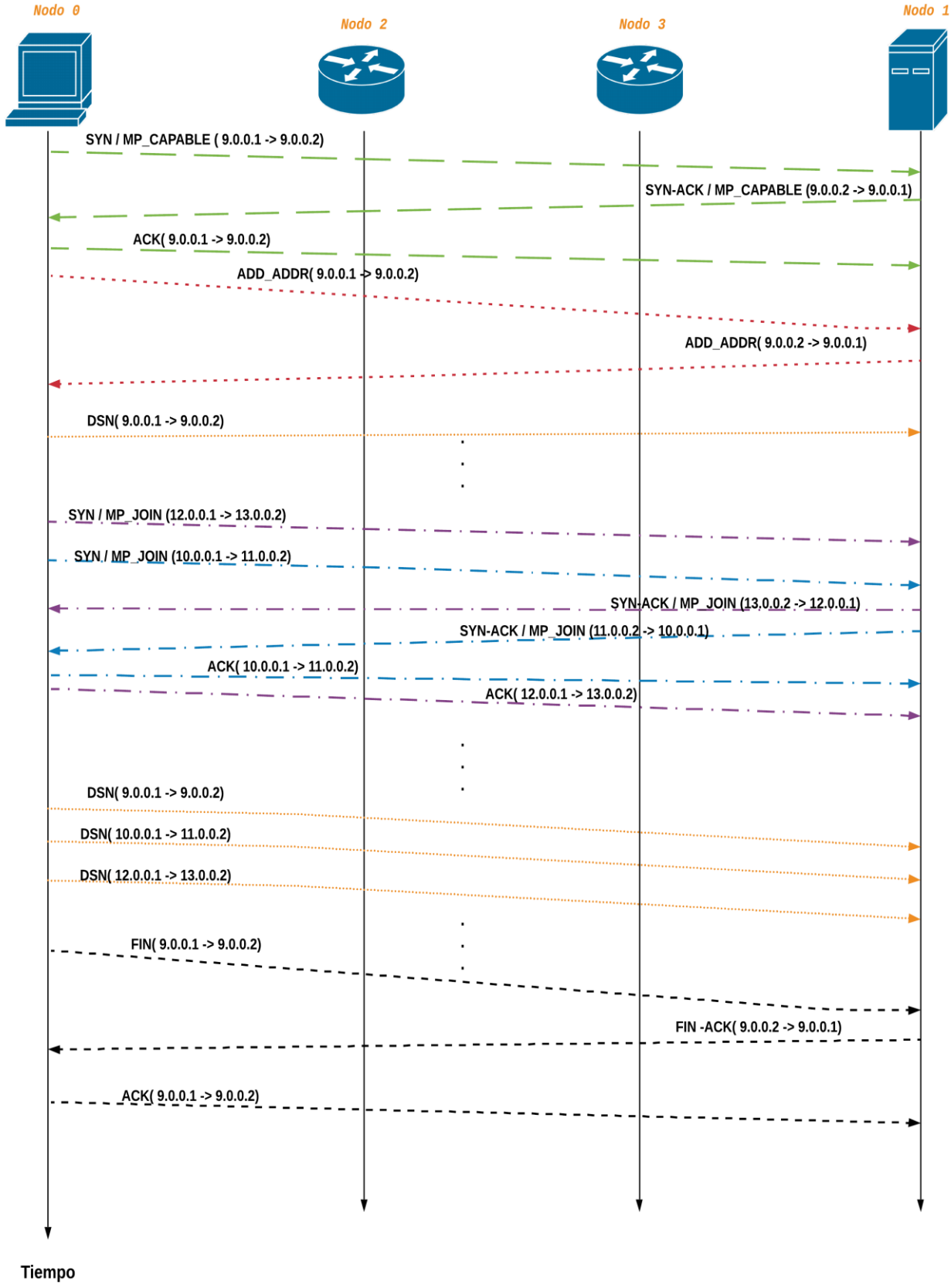

Ilustración 6.9: Mensajes intercambiados en la topología 2

7 CONCLUSIÓN Y LÍNEA DE CONTINUACIÓN

No basta con tener un buen ingenio, lo principal es aplicarlo bien

- Descartes-

7.1 Conclusión

Tras la realización de este proyecto fin de grado se ha adquirido una base técnica y funcional de los protocolos *MpTCP* y *TCP* que ha sido necesaria para validar la implementación. Tras el análisis de todas las simulaciones que se han planteado a lo largo del documento podemos afirmar que efectivamente *MpTCP* es una mejora de *TCP*. Si se ha diseñado bien, en el peor de los casos para el protocolo *MpTCP* se comportará como *TCP*.

MpTCP es un paso para el futuro de Internet y de las comunicaciones ya que cada día son más los dispositivos que se conectan a la red y que además disponen de más de una interfaz, por lo que seguir usando en estos dispositivos TCP es un atraso. Se estaría desaprovechando todos los recursos que los dispositivos ofrecen. Por otro lado, y hablando de los centros de procesamiento de datos (*CPD*), donde no se para de procesar información, es necesario usar MpTCP por la redundancia que aporta además de por el balanceo de carga que ofrece a los dispositivos.

En conclusión a todas las pruebas y simulaciones que se han realizado a lo largo de esta memoria se puede decir que efectivamente se ha comprobado que el algoritmo *Fastest_Rtt* que se ha implementado es más eficiente que el de reparto equitativo *Round_Robin*. Por otro lado, en la mayoría de las situaciones que nos imaginemos será más eficiente *MpTCP* ya que solo está aprovechando los recursos disponibles para transmitir datos, mientras que *TCP* sigue restringido a solo una interfaz para realizar la transmisión de datos. La versatilidad que aporta el protocolo estudiado es primordial, ya que en estos días, es raro que un dispositivo no disponga de más de una interfaz física.

7.2 Línea de continuación

El estado actual de la implementación de *MpTCP* realizada en el simulador está cerca de la RFF **6824**. Cumple con los requisitos necesarios para el objetivo de este proyecto; mostrar la mejora de *MpTCP* sobre *TCP*. La implementación se encuentra totalmente funcional, pero por otro lado se pueden añadir mejoras que por motivos de tiempo no han sido factibles.

A continuación, se detallan algunas características que se podrían implementar para mejorar su funcionamiento y completar su estado de desarrollo.

7.2.1 Eliminar direcciones que previamente hayan sido añadidas

En el estado actual, si en la simulación se ha configurado el parámetro de gestión de rutas como *FullMesh* (véase apartado 3.2.2.2), todas las direcciones disponibles en el host son anunciadas. Actualmente, esas direcciones no son eliminadas.

Por ejemplo, si en un determinado momento una interfaz del dispositivo deja de estar disponible, se debe enviar por uno de los subflujos disponibles un segmento con la opción **OPT_REM_ADD**. Al recibirla en el destino debe actualizar la estructura en la que se almacenen las direcciones remotas, eliminado la dirección.

7.2.2 Implementar opción MP_PRIO

Esta opción no está implementada actualmente. Se usa para cambiar la prioridad a la hora de transmitir por un subflujo sin la necesidad de que la aplicación tenga que reiniciar la conexión al completo. El primer paso para implementar la opción sería tener una lista con todos los subflujos disponibles. Posteriormente, tendrían que asignarse una prioridad a cada subflujo.

Otra funcionalidad asociada a esta opción es tener un subflujo de *backup* o "respaldo", de manera que siempre se transmita por el mismo subflujo. En el momento que se desee, bien sea por que el enlace deje de estar disponible o por que el enlace empiece a saturarse, se pueda dar prioridad al enlace de *backup* para empezar a transmitir por él y dejar de hacerlo por el actual.

Con respecto a esta opción se puede obtener más información accediendo al siguiente enlace: https://tools.ietf.org/html/draft-samar-mptcp-socketapi-00#page-4

7.2.3 Añadir soporte para IPv6

Actualmente, la implementación del protocolo solo soporta IPv4. Sería una mejora añadir soporte para poder usarlo además con IPv6, ya que cada día tiene más cota de uso que su antecesor debido al agotamiento de las direcciones. Entre otras cosas esto es debido a que cada vez más dispositivos se conectan a Internet, por tanto debe de asignársele una dirección.

ANEXOS

ÍNDICE DE ANEXOS

ANEXO A: INSTALACIÓN NS3

El proyecto está desarrollado en la versión 3.19 del simulador, la cual se puede descargar desde le siguiente enlace (https://www.nsnam.org/release/ns-allinone-3.19.tar.bz2). El proyecto ha sido realizado sobre el sistema operativo Ubuntu 16. Las pautas indicadas aquí son para realizar una instalación del simulador para soportar la implementación de *MpTCP*, para una instalación con configuraciones personalizadas véase https://www.nsnam.org/wiki/Installation.

Los pasos a seguir para la instalación del simulador son los siguientes:

- Acceder al directorio principal del simulador, para ello se ha ejecutado el siguiente comando desde la consola suponiendo que tras descargarlo se ha almacenado en la carpeta de descargas: ***cd Descargas/ns3/***
- Configurar el paquete, para ello se ha ejecutado el siguiente comando. Es necesario configurar algunas variables para el compilador de C++: ***CXXFLAGS='-g -w -std=c++11 -lssl -lcrypto' ./waf configure***. Siendo importantes las opciones '**-g**' para luego poder depurar el programa desde el entorno Eclipse y '***-std=c++11***' para indicarle la versión del lenguaje que debe usar. Una vez ejecutemos el comando se obtiene la salida de que el paquete ha sido correctamente configurado.
-

```
---- Summary of optional NS-3 features:
Python Bindings                : not enabled (Python library or headers missing)
BRITE Integration              : not enabled (BRITE not enabled (see option --with-brite))
NS-3 Click Integration         : not enabled (nsclick not enabled (see option --with-nsclick))
GtkConfigStore                 : not enabled (library 'gtk+-2.0 >= 2.12' not found)
XmlIo                          : not enabled (library 'libxml-2.0 >= 2.7' not found)
Threading Primitives           : enabled
Real Time Simulator            : enabled
Emulated Net Device            : enabled
File descriptor NetDevice      : enabled
Tap FdNetDevice                : enabled
Emulation FdNetDevice          : enabled
PlanetLab FdNetDevice          : not enabled (PlanetLab operating system not detected (see option --force-planetlab))
Network Simulation Cradle      : not enabled (NSC not found (see option --with-nsc))
MPI Support                    : not enabled (option --enable-mpi not selected)
NS-3 OpenFlow Integration      : not enabled (Required boost libraries not found)
SQlite stats data output       : not enabled (library 'sqlite3' not found)
Tap Bridge                     : enabled
PyViz visualizer               : not enabled (Python Bindings are needed but not enabled)
Use sudo to set suid bit       : not enabled (option --enable-sudo not selected)
Build tests                    : not enabled (defaults to disabled)
Build examples                 : not enabled (defaults to disabled)
GNU Scientific Library (GSL)   : not enabled (GSL not found)
'configure' finished successfully (1.145s)
```

Ilustración 0.1: Configuración del paquete

- Una vez el paquete ha sido configurado con las opciones necesarias se tiene que construir, para ello se ha ejecutado el siguiente comando: ***./waf***. Una vez ejecutado se empiezan a compilar obteniendo el siguiente resultado.

```
salas@localhost:~/Escritorio/mptcp-1.0/ns-allinone-3.19/ns-3.19$ ./waf
Waf: Entering directory `/home/salas/Escritorio/mptcp-1.0/ns-allinone-3.19/ns-3.19/build'
Waf: Leaving directory `/home/salas/Escritorio/mptcp-1.0/ns-allinone-3.19/ns-3.19/build'
'build' finished successfully (0.186s)

Modules built:
applications              bridge                    config-store
core                      internet                  mpi
network                   point-to-point            stats

Modules not built (see ns-3 tutorial for explanation):
brite                     click                     openflow
visualizer
```

Ilustración 0.2: Construcción del paquete

- Tras haber construido el paquete podemos ejecutar simulaciones. Estas deben situarse dentro del directorio ***scratch*** y para lanzarlas basta con ejecutar el siguiente comando: ***./waf –run 'nombreDeLaSimulacion'.***

```
salas@localhost:~/Escritorio/mptcp-1.0/ns-allinone-3.19/ns-3.19$ ./waf --run scratch-simulator
Waf: Entering directory `/home/salas/Escritorio/mptcp-1.0/ns-allinone-3.19/ns-3.19/build'
Waf: Leaving directory `/home/salas/Escritorio/mptcp-1.0/ns-allinone-3.19/ns-3.19/build'
'build' finished successfully (0.188s)
Scratch Simulator
salas@localhost:~/Escritorio/mptcp-1.0/ns-allinone-3.19/ns-3.19$
```

Ilustración 0.3: Ejecución de una simulación

ANEXO B: ECLIPSE Y NS3

Para la comodidad a la hora de programar las simulaciones se hace muy recomendable el uso de entornos de desarrollos como Eclipse. En este caso se ha usado en la versión ***Neon 3*** (descarga). Ahora se detallarán los pasos para la configuración de ns3 en eclipse. Se hace hincapié en la opción '***-g***' a la hora de configurar el paquete, ya que una de las principales ventajas de usar el entorno es que podemos depurar código fácilmente y sin la opción no es posible.

Una vez se ha realizado la instalación de eclipse se deben seguir los siguientes pasos:

- Creación de un nuevo proyecto: Podremos hacerlo desde **File→New→C/C++ Project**. A la hora de indicar la dirección del proyecto tenemos que cambiar la del *workspace* de eclipse por el directorio principal de ns3.

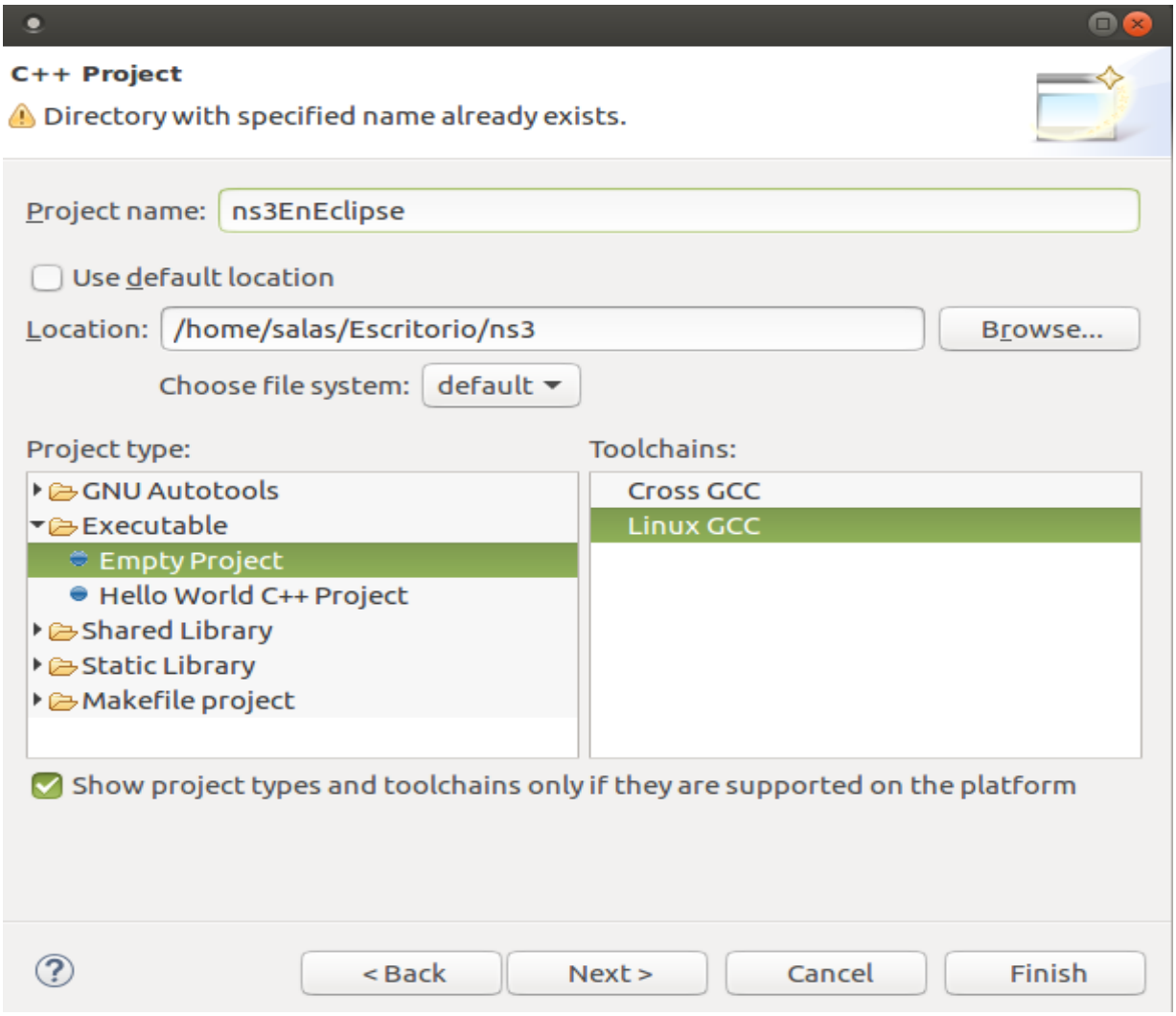

Ilustración 0.1: Creación nuevo proyecto

Tras haber creado el proyecto tenemos que configurar el constructor de este. Esta configuración se realiza desde ***Project→Properties***. Aunque configuremos el constructor del paquete desde eclipse, es necesario realizar la configuración de este. Esta configuración no se puede realizar desde eclipse, por tanto ha de realizarse tal y como se indica en el ANEXO A: Instalación ns3

Para no indicar la ruta completa podemos usar la variable ***workspace_loc***, que nos indica la ruta donde se encuentra el proyecto. Además, tenemos que definir el comportamiento del constructor, que esto se hace desde la pestaña '*behaviour*':

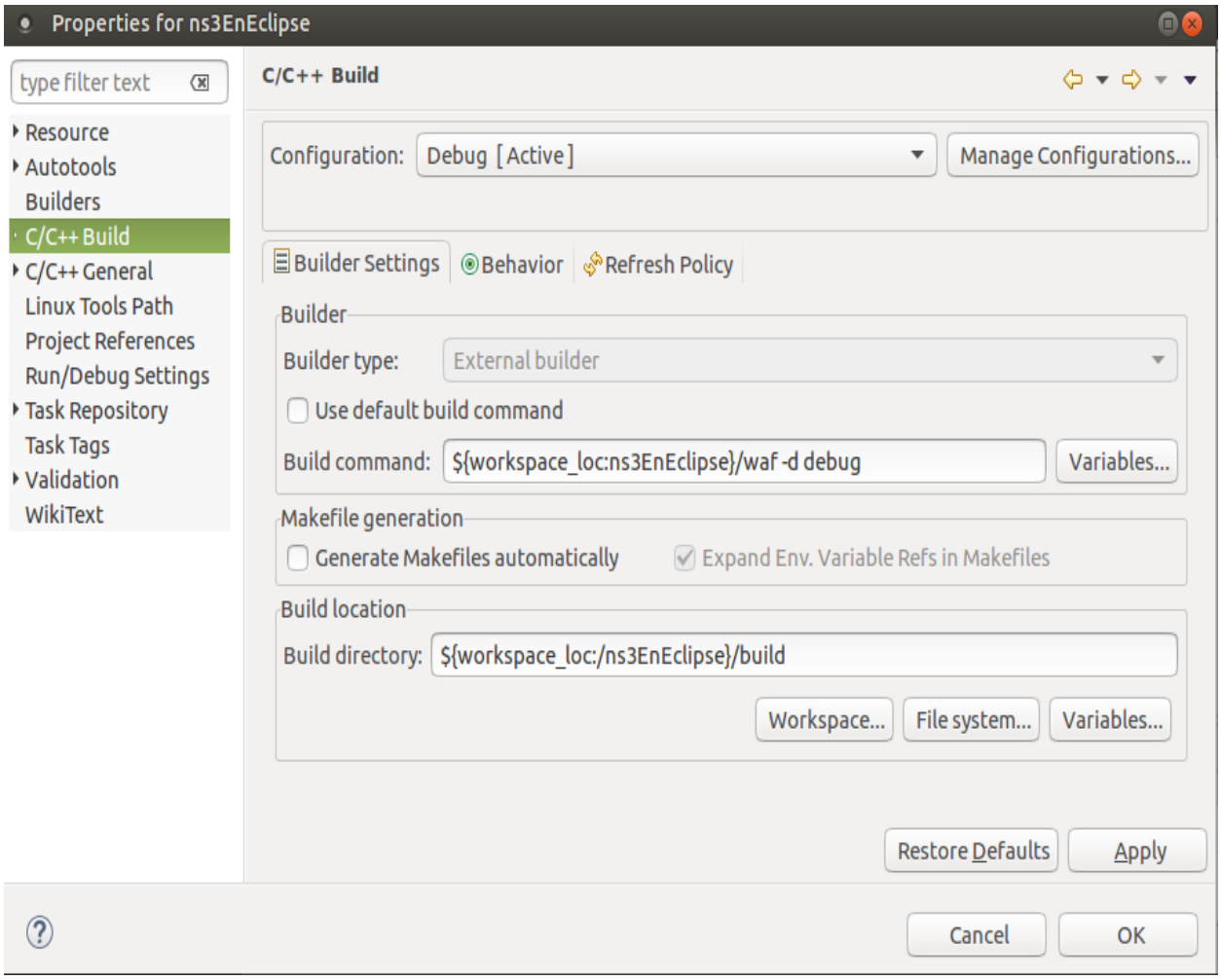

Ilustración 0.2: Creación nuevo proyecto 2

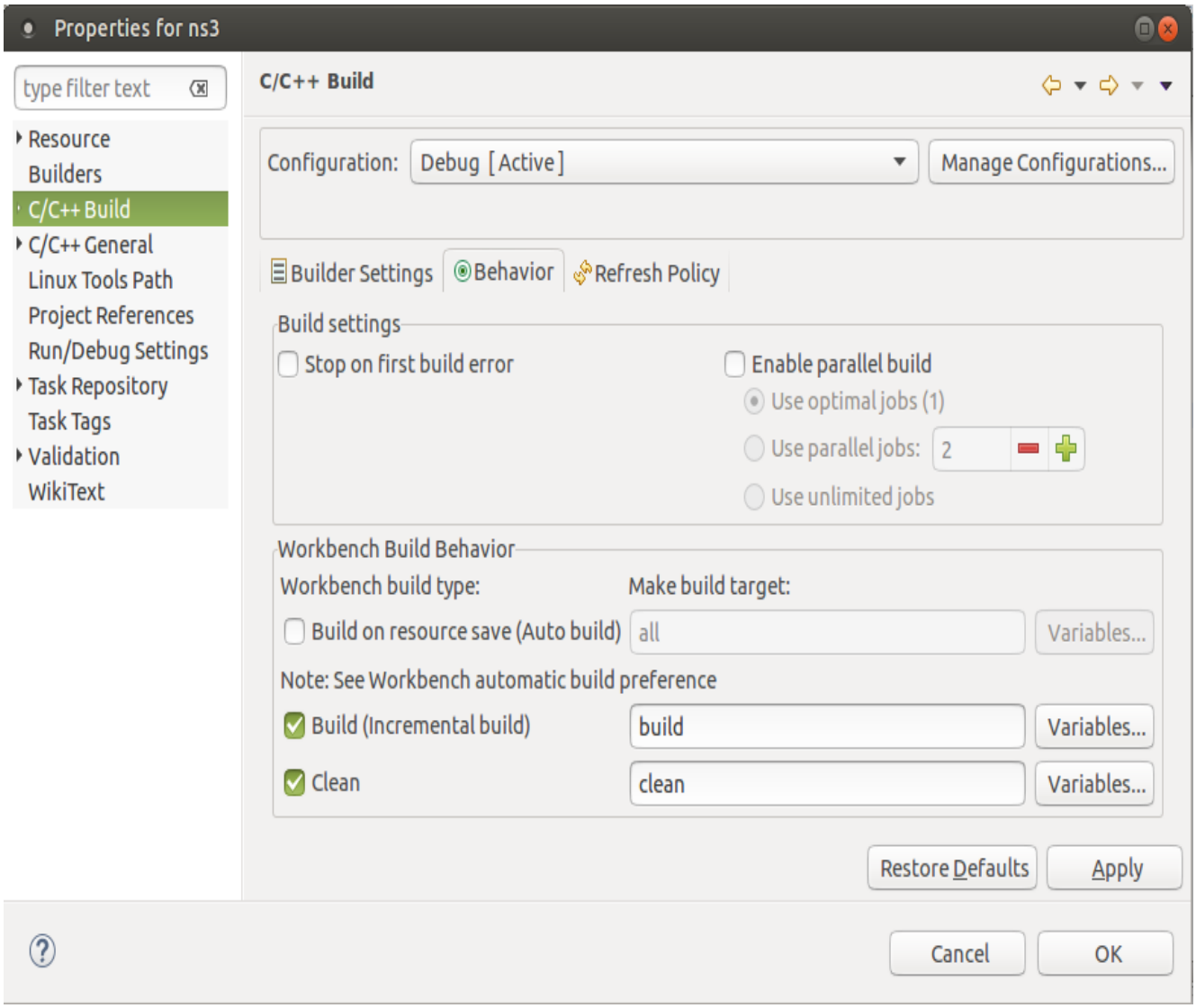

Ilustración 0.3: Creación nuevo proyecto 3

- El siguiente paso es crear una aplicación para ejecutar las simulaciones. Se puede crear ***desde Run→Debug Configurations***. Aquí podemos elegir la simulación que queramos depurar, además en este apartado es importante configurar las variables de entorno. La más importante es ***LD_LIBRARY_PATH*** que debe tener el valor de la ruta donde se almacenan los archivos binarios tras la compilación. Otra variable importante es ***NS_LOG*** que la usa el simulador para saber de qué componentes mostrar la información durante la simulación.

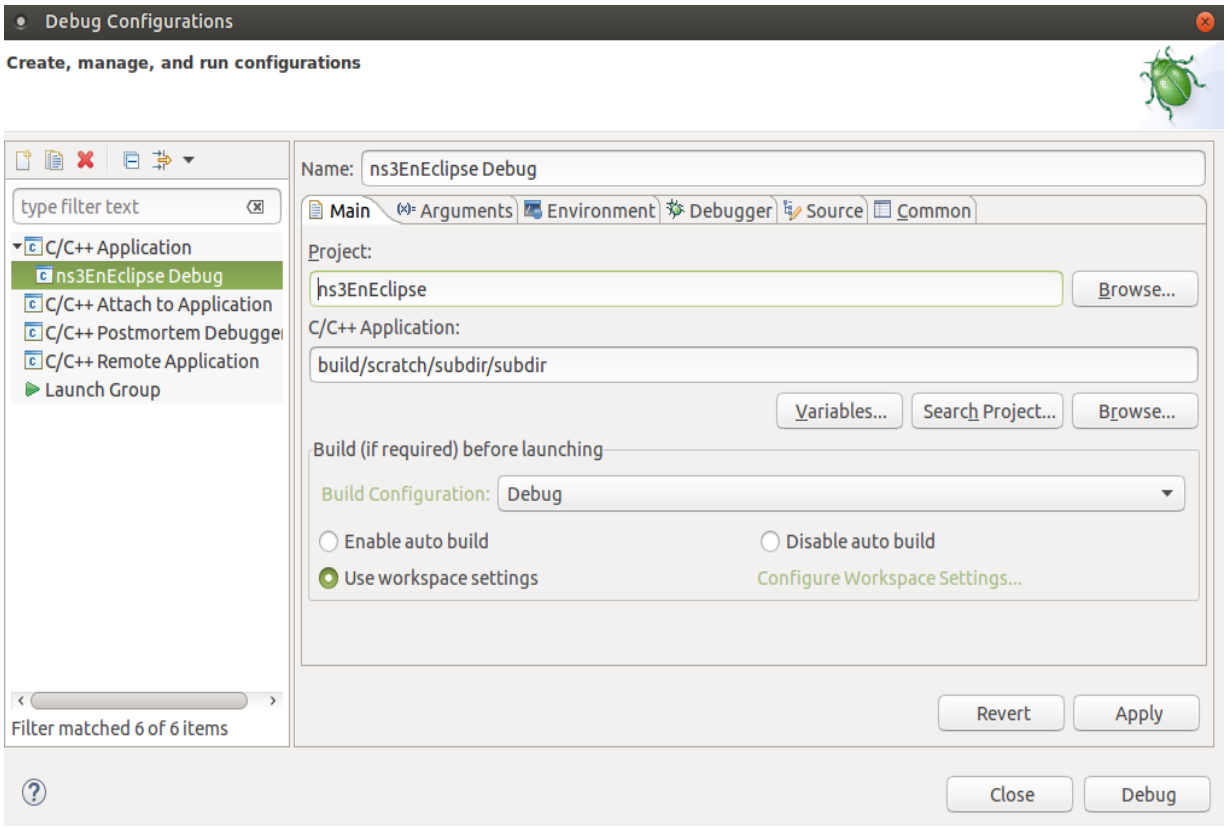

Ilustración 0.4: Configuración depurador

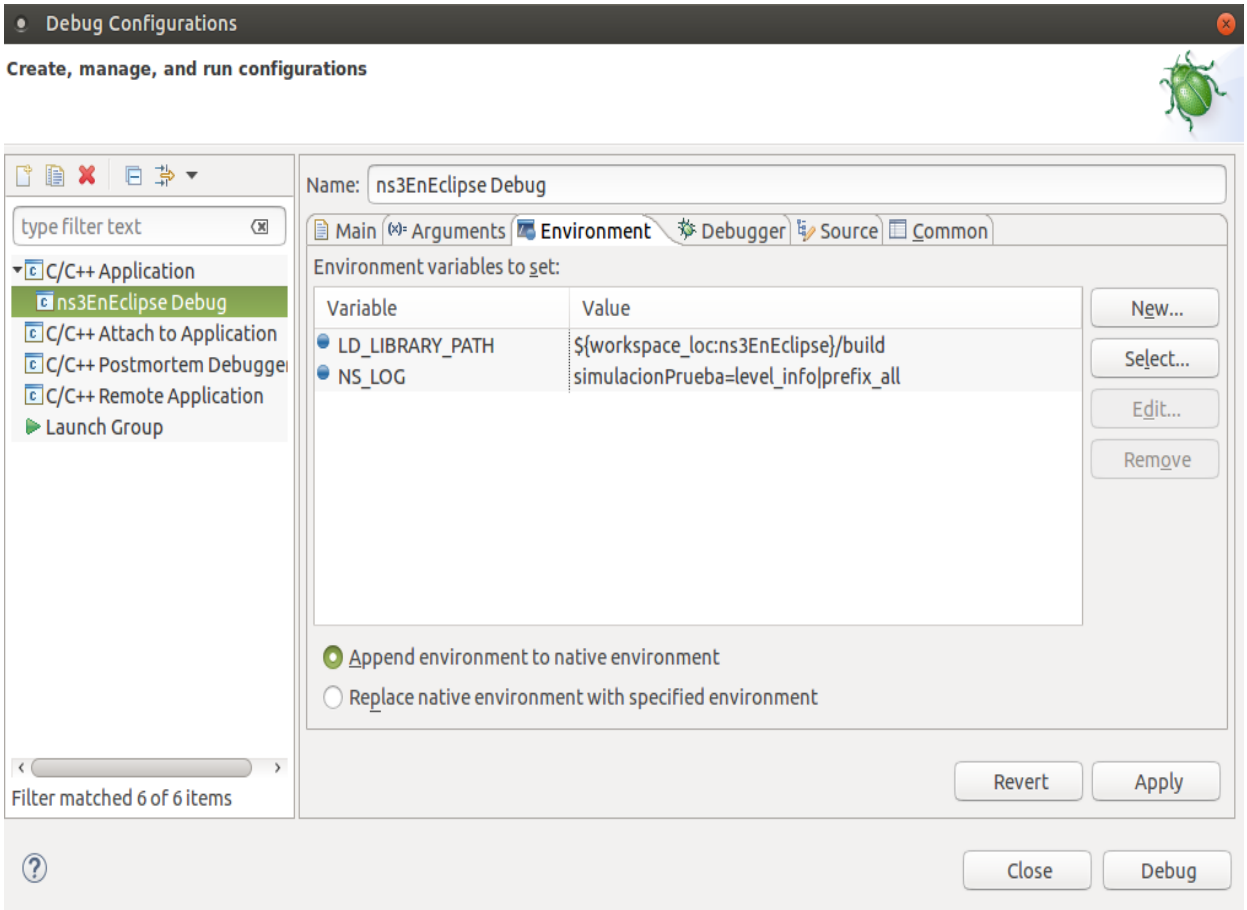

Ilustración 0.5: Configuración variables de entorno

- El último paso es ejecutar una simulación para la cual hemos configurado el depurador.

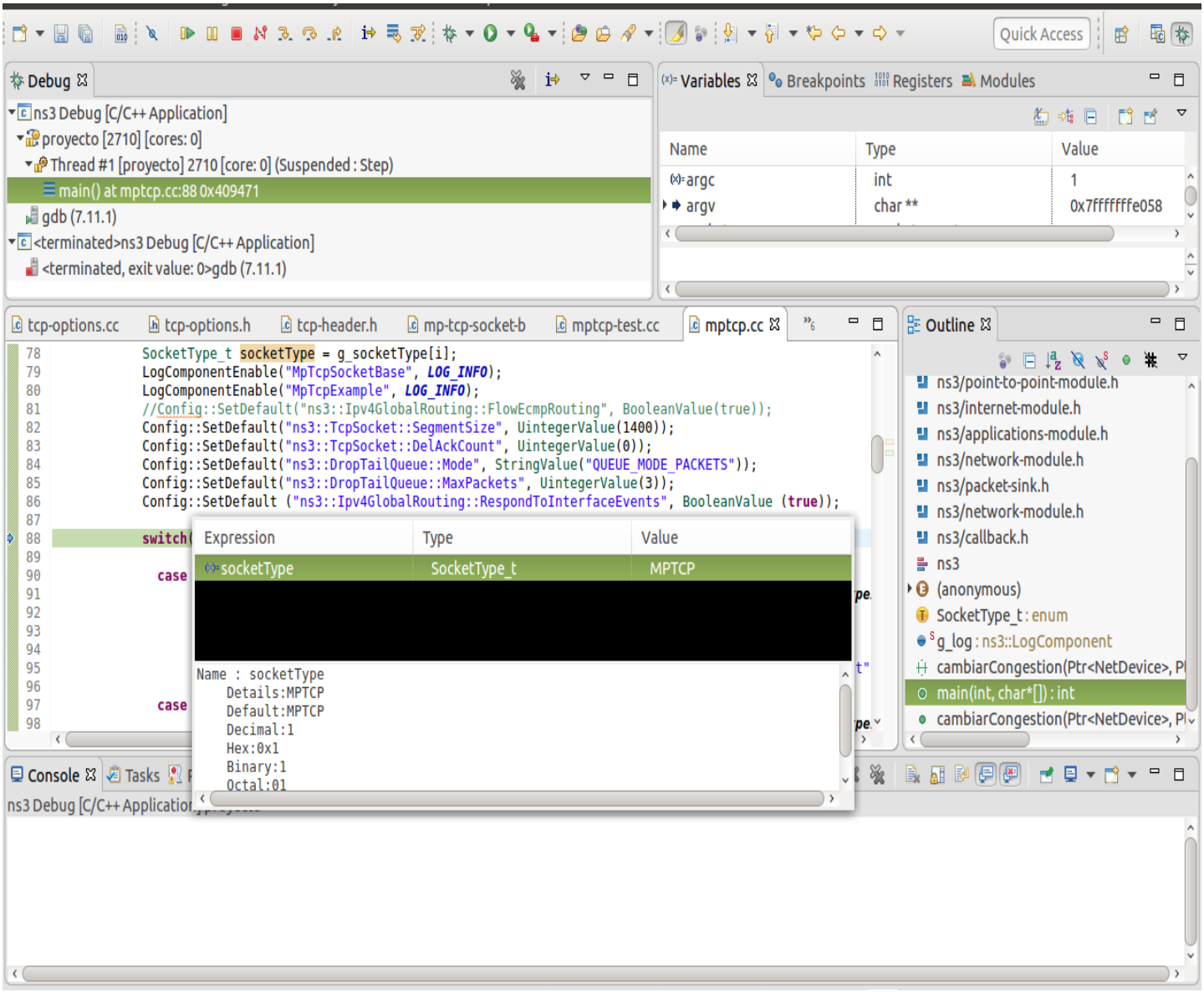

Ilustración 0.6: Ejecución en modo depuración

ANEXO C: INSTALACIÓN DE MPTCP EN NS3

Como se ha comentado anteriormente, la versión que se ha usado del simulador es la 3.19. Es un requisito para que la implementación de *MpTCP* se instale correctamente. Existe la posibilidad de realizar una instalación del simulador únicamente con los componentes necesarios para el funcionamiento de *MpTCP*. Para ello, la configuración del paquete se realizaría con la siguiente instrucción: "***CXXFLAGS='-g -w -std=c++11 -lssl -lcrypto' ./waf configure --enable-test --enable-modules=internet,applications,point-to-point***", donde estamos indicando los módulos que queremos añadir.

El código de la implementación se proporciona en un fichero comprimido con el siguiente contenido:

- **scriptConfigMptcp.sh**: Script que se encarga de configurar el simulador. En primer lugar, se descarga la versión 3.19 del simulador y coloca los ficheros fuentes que han sido modificados en los directorios correspondientes. En segundo lugar, configura el proyecto con las opciones necesarias y lo compila. Finalmente, se muestra mensaje para ejecutar los test de validación.

```
#Autor : Antonio Manuel Del Toro Domínguez
clear;
        echo "#########################################################"
        echo ""
        echo "Descargando el simulador ns3 versión: 3.19..."
        echo ""
        echo "#########################################################"

        if [ ! -f ns-allinone-3.19.tar.bz2 ]; then
                wget https://www.nsnam.org/release/ns-allinone-3.19.tar.bz2
                error=$?
                if [ $error -eq 1 ]; then
                echo "No se ha podido descargar los fuentes, comprueba la conexión a internet."
                echo "Puede descargarlo opcionalmente en la siguiente URL:
                        'https://www.nsnam.org/release/ns-allinone-3.19.tar.bz2'"
                fi
        else
                error=0
        fi


        if [ $error -eq 0 ]; then

                echo ""
                echo "Descomprimiendo el simulador ..."
                tar -xvf ns-allinone-3.19.tar.bz2
                cp -r MpTcp\ Sources ns-allinone-3.19/ns-3.19
                cd  ns-allinone-3.19/ns-3.19/MpTcp\ Sources

                echo "#########################################################"
                echo ""
                echo "Copiando documentos al directorio correspondiente..."
                echo ""
                echo "-------------------------------------------------------"
                echo "Copiando documentos a 'src/internet/'"
                cp ./Internet/model/*.cc ./../src/internet/model
                cp ./Internet/model/*.h ./../src/internet/model
                cp ./Internet/test/* ./../src/internet/test
                cp ./Internet/wscript ./../src/internet
                echo "-------------------------------------------------------"
                echo "Copiando documentos a 'src/stats/model'"
                cp ./Stats/model/* ./../src/stats/model

                echo "-------------------------------------------------------"
```

```
                echo "Copiando documentos a 'src/aodv/test'"
                cp ./aodv/test/* ./../src/aodv/test

                echo "-------------------------------------------------------"
                echo "Copiando documentos a 'src/test/ns3tcp/"
                cp ./test/* ./../src/test/ns3tcp/

                echo "-------------------------------------------------------"
                echo "Copiando documentos a 'src/applications/helper'"
                cp ./Aplicaciones/helper/*.cc ./../src/applications/helper
                cp ./Aplicaciones/helper/*.h ./../src/applications/helper
                echo "-------------------------------------------------------"
                echo "Copiando documentos a 'src/applications/model'"
                cp ./Aplicaciones/model/*.cc ./../src/applications/model
                cp ./Aplicaciones/model/*.h ./../src/applications/model
                cp ./Aplicaciones/wscript ./../src/applications

                echo "-------------------------------------------------------"
                echo "Copiando las simulaciones al directorio 'scratch'"
                cp -R ./Simulaciones/* ./../scratch
                echo "-------------------------------------------------------"
                echo ""
                echo "#########################################################"
                echo ""
                echo "-------------------------------------------------------"
                echo "Copiando script para borrar los ficheros de las simulaciones ...'"
                cp ./borrar.sh ./../
                echo "-------------------------------------------------------"
                echo ""
                echo "#########################################################"


                echo "############  Configurando el Simulador  ###############"
                echo ""
                cd ..
                # Se está configurando para construir el paquete solo con los módulos necesarios
                # para el funcionamiento de la implementación.
                # Si se desea construir el paquete con todos los módulos, basta con
                # eliminar la opción '--enable-modules'

                CXXFLAGS='-g -w -std=c++11 -lssl -lcrypto' ./waf configure --enable-test --enable-
                modules=internet,applications,point-to-point
                ./waf
                error=$?

                if [ $error -eq 0 ]; then

                        zenity  --title="Ejecutar test" --question --text="¿Desea ejecutar los test de
                        validación?" --ok-label="Si" --cancel-label="No"
                        ejecutar=$?
                        if [ $ejecutar -eq 0 ] ; then ##   Ejecutamos los test de validación  ###
                                ./test.py
                        else
                                exit
                        fi

                fi
                ./test.py
        fi


exit
```

Ilustración 0.1: Script de configuración de ns3

- **scriptSimulaciones.sh**: Script que se encarga de abstraer la complejidad del simulador para ejecutar las simulaciones. Tras ejecutarlo se muestra al usuario un mensaje visual donde puede elegir fácilmente la simulación que desee ejecutar.
- **MpTCP Sources**: dentro de este directorio encontramos todos los ficheros fuentes que han sido modificados y que deben ser copiados antes de construir el paquete a su directorio correspondiente. Además, dentro de cada subdirectorio encontramos el fichero wscript, que existe para cada módulo de ns3. En este fichero deben añadirse los nombres de los nuevos ficheros fuentes que se añaden a cada módulo.

La estructura dentro de este directorio es la siguiente:

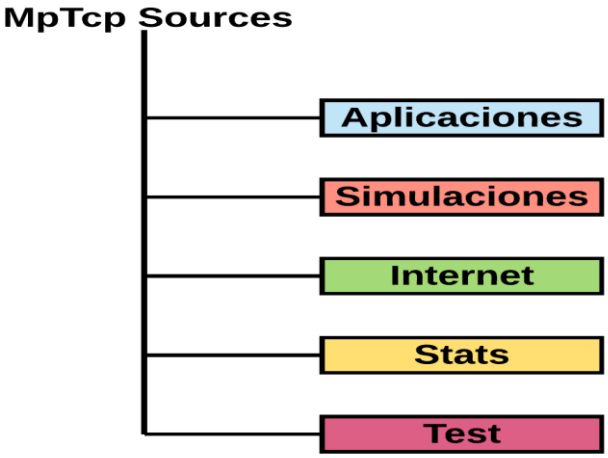

Ilustración 0.1: Directorio *MpTCP* Sources

A continuación, se detallan las instrucciones para ejecutar estos ficheros scripts:

- Se debe descomprimir el fichero en el directorio donde se quiera instalar el simulador.
- En el caso de que no hayamos descargado el simulador, el script lo hará. Si tenemos el simulador descargado, debemos asegurarnos de descomprimir el fichero en el mismo directorio y además debe colocarse el simulador en la siguiente ruta "ns-allinone-3.19/ns-3.19".
- En los casos que sea necesario los scripts necesitarán permiso de ejecución. Para ello desde la consola de se le podrán conceder ejecutando el siguiente comando **" *chmod +x 'nombre del script'* ".**
- Los scripts se ejecutarán de la siguiente forma: **"./nombre-del-script"**

Se muestran ilustraciones del proceso de instalación:

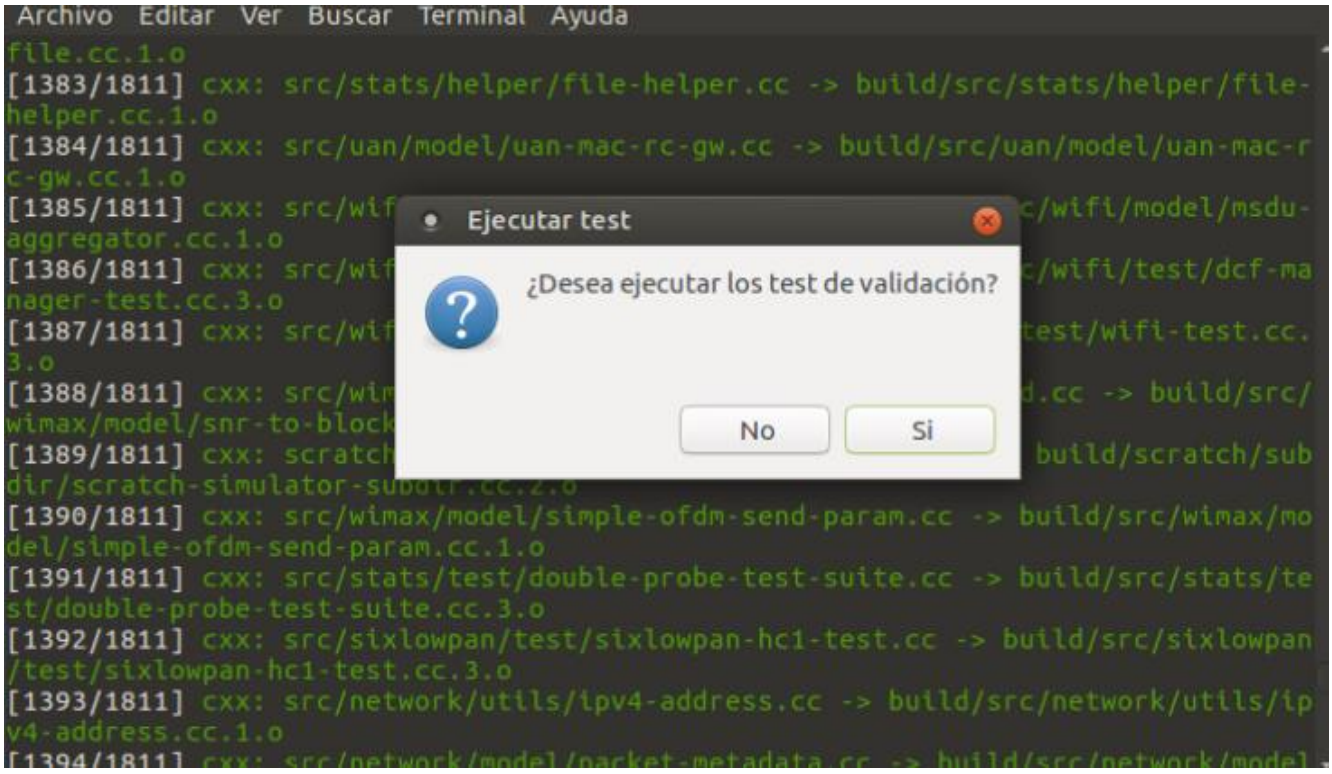

Ilustración 0.2: Diálogo ejecución de test

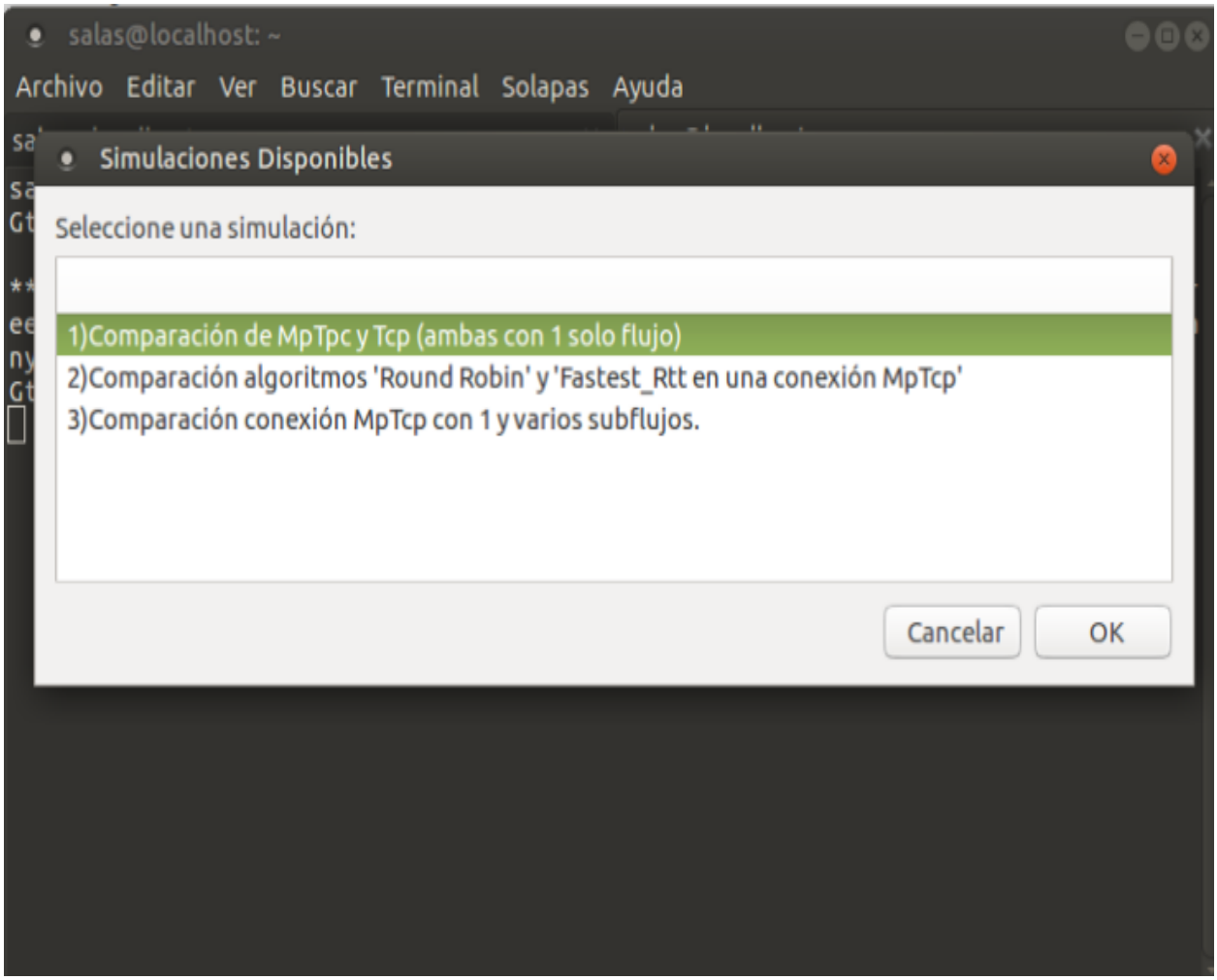

Ilustración 0.3: Menú para la selección de simulaciones

Los ficheros **wscript**, se proporcionan, pero si se quisieran modificar para añadir otros ficheros fuentes, se debería hacer de la siguiente forma:

- Los ficheros fuentes (los .cc) deberían incluirse en la siguiente variable:

  obj.source = [‘nombreDelFichero’.cc,…,]

- Los ficheros de cabecera (los .h)

  headers.source = [‘nombreDelFichero’.h,…,]

ANEXO D: TEST EN NS3

Los test en ns3 se utilizan para validar que la instalación del simulador se ha realizado correctamente. Además, cuando se desarrollan nuevas implementaciones en el simulador es necesario volver a ejecutar los test debido a que algo haya podido dejar de tener el funcionamiento deseado.

Por otro lado, el desarrollo de una nueva implementación para el simulador conlleva la codificación de los test de esa implementación, ya que si en un futuro se quiere trabajar sobre esa versión será necesario comprobar que todo sigue funcionando de forma adecuada.

La ejecución de los test se realiza de forma automática a través del script ***scriptConfigMptcp.sh***, pero si esta se quisiera realizar de forma manual podría realizarse con la instrucción ***./test.py*** que debe ser ejecutada desde el directorio base de ns3.

```
PASS: TestSuite sequence-number
PASS: TestSuite devices-point-to-point
PASS: TestSuite global-route-manager-impl
PASS: TestSuite ipv4-address-generator
PASS: TestSuite ipv4-address-helper
PASS: TestSuite ipv4-list-routing
PASS: TestSuite ipv4-packet-info-tag
PASS: TestSuite ipv4-raw
PASS: TestSuite ipv4-header
PASS: TestSuite ipv4-fragmentation
PASS: TestSuite ipv4-forwarding
PASS: TestSuite ipv4-protocol
PASS: TestSuite ipv6-extension-header
PASS: TestSuite ipv6-list-routing
PASS: TestSuite ipv6-packet-info-tag
PASS: TestSuite ipv6-protocol
PASS: TestSuite ipv6-raw
PASS: TestSuite tcp
PASS: TestSuite udp
PASS: TestSuite ipv6-address-generator
PASS: TestSuite ipv6-dual-stack
PASS: TestSuite ipv6-fragmentation
PASS: TestSuite ipv6-forwarding
PASS: TestSuite ipv6-address-helper
PASS: TestSuite rtt
PASS: TestSuite mptcp
PASS: TestSuite udp-client-server
60 of 60 tests passed (60 passed, 0 skipped, 0 failed, 0 crashed, 0 valgrind errors)

***  Note: ns-3 examples are currently disabled. Enable them by adding
***  "--enable-examples" to ./waf configure or modifying your .ns3rc file.
```

Ilustración 0.1: Salida de ejecución de Test

Para obtener más información de los test, de cara al desarrollo, pueden depurarse desde la línea de comando ejecutando la siguiente instrucción:

1. *.’/waf --command-template="gdb %s" --run test-runner’*
2. *‘r --suite="nombreDelTest" –verbose’*

A modo de ejemplo, en la siguiente ilustración se muestra la depuración del test de ***TCP***:

```
Source recv data="yzabcdefghijklmnopqr"
Source recv data="stuvwxyzabcdefghijkl"
Source recv data="mnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefghij"
Source recv data="klmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdefgh"
Source recv data="ijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcdef"
Source recv data="ghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd"
PASS tcp 1.610 s
  PASS Send string data from client to server and back total=13 sourceWrite=1 sourceRead=1 serverRead=1 serverWrite=1 useIpv6=0 0.060 s
  PASS Send string data from client to server and back total=100000 sourceWrite=100 sourceRead=50 serverRead=20 serverWrite=100 useIpv6=0 1.550 s
[Inferior 1 (process 11159) exited normally]
(gdb)
```

Ilustración 0.2: Depuración suite *TCP*

Para la codificación de los test en ns3 debe tenerse en cuenta la estructura de los mismos. Es necesario crear una suite para el protocolo que hayamos implementado (que debe heredar de ***TestSuite***), y dentro de la suite añadiremos casos para probar distintas características de la implementación. Cada caso que creemos dentro de la suite de los test debe implementar la clase ***TestCase*** y sobrescribir al menos el método **doRun()**, que es el que se encarga de iniciar el test. A modo de ejemplo, en la siguiente ilustración se muestra como sería la suite de *MpTCP*.

Si desea obtener más información acerca de los test en ns3 puede consultar la documentación en https://www.nsnam.org/docs/manual/html/test-framework.html. Para ver el código fuente de la implementación de los test de ns3 para *MpTCP* puede revisar el apartado 5.2 Pruebas unitarias automatizadas.

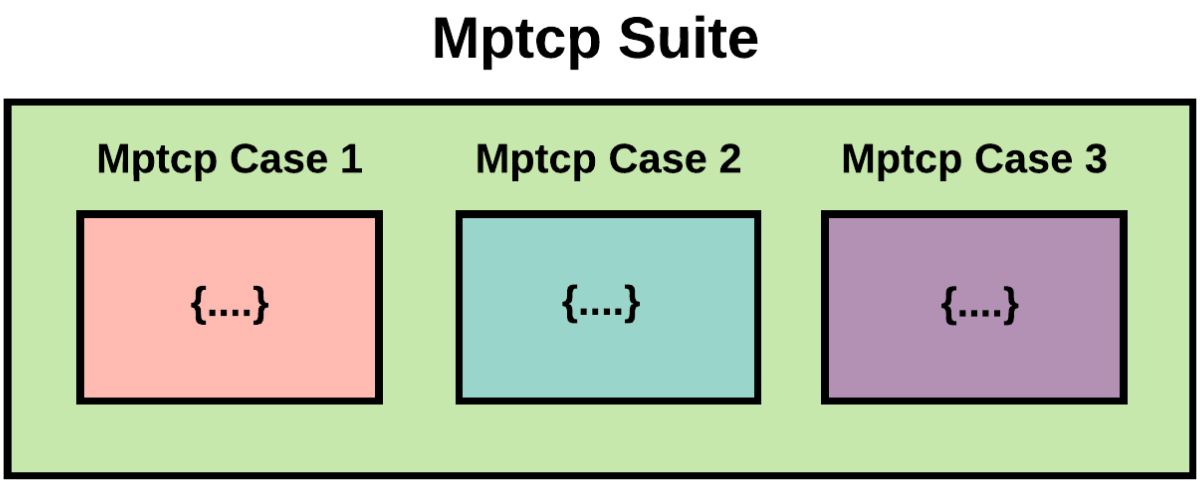

Ilustración 0.3: Test de *MpTCP*

BIBLIOGRAFÍA

http://blog.multipath-tcp.org/blog/html/index.html. (s.f.).

http://www.tcpipguide.com/free/t_TCPCharacteristicsHowTCPDoesWhatItDoes.htm. (s.f.).

http://www0.cs.ucl.ac.uk/staff/M.Handley/papers/9346-login1210_bonaventure.pdf. (s.f.).

https://datatracker.ietf.org/meeting/83/materials/slides-83-mptcp-3. (s.f.).

https://en.wikipedia.org/wiki/Multipath_TCP. (s.f.).

https://github.com/lip6-mptcp/ns3mptcp. (s.f.).

https://github.com/mkheirkhah/mptcp. (s.f.).

https://github.com/multipath-tcp/mptcp. (s.f.).

https://hal.sorbonne-universite.fr/hal-01382907/document. (s.f.).

https://tools.ietf.org/pdf/draft-ietf-mptcp-rfc6824bis-11.pdf. (s.f.).

https://www.blackhat.com/docs/us-14/materials/us-14-Pearce-Multipath-TCP-Breaking-Todays-Networks-With-Tomorrows-Protocols.pdf. (s.f.).

https://www.cisco.com/c/en/us/support/docs/ip/transmission-control-protocol-tcp/116519-technote-mptcp-00.html. (s.f.).

https://www.researchgate.net/publication/283335405_Multipath-TCP_in_ns-3. (s.f.).

GLOSARIO

API
Interfaz de programación de aplicaciones usadas por otro software como abstracción, 21, 34

bandera
En informática se le denomina a un bit que almacena un valor con un significado concreto., 26, 29
En informático se le denomina a un bit que almacena un valor con un significado concreto., 26

buffer
Espacio reservado para el almacenamiento temporal de información., 19, 35, 39

Firewalls
Parte del sistema o de la red que tiene cuya función es controlar los accesos y privilegios., 26, 30

host
En informática hace referencia a dispositivos o computadoras que están conectadas a la red., 23, 24, 25, 26, 27, 30, 65

IP
Protocolo de comunicación de datos situado en la capa de red según el modelo OSI., 17, 19, 21, 22, 27

kernel
Se conoce como el núcleo del sistema operativo, 18, 32

Multihoming
Puede ser accedido desde distintas direcciones IP, 17

NAT
Mecanismo utilizado en la redes IP para intercambiar datos entre redes con direcciones no compatibles., 26, 30

Script
Archivo de órdenes, 18, 76, 77, 79

Socket
Interfaz de aplicación para la familia de protocolos de Internet TCP/IP, 20

token
Cadena de caracteres que tienen un significado específico durante la realización de una acción., 25, 26

TWHS
Hace referencia al establecimiento de una conexión *TCP*., 24, 26, 28, 48, 50, 51