Universidad de la República
Facultad de Ingeniería
Instituto de Computación

Técnicas para la apertura de puertos en las aplicaciones de Internet

Informe de Proyecto de Grado

28 de noviembre de 2012
Montevideo - Uruguay

Tutor
Rodríguez-Bocca, Pablo

Integrantes
Suero, Andrés
Kouyoumdjian, Federico

Usuario responsable
De Vera, Daniel

Resumen del proyecto

Este proyecto se enmarca en la investigación sobre la plataforma de distribución de video en Internet GoalBit (http://goalbit.sourceforge.net), basada en licenciamiento libre y código abierto. Esta plataforma utiliza tecnología P2P para la distribución del video al usuario final y para el transporte de video entre servidores.

El uso de P2P presenta ventajas y desafíos. Las principales ventajas son el ahorro de ancho de banda en la distribución al usuario final y la escalabilidad a miles de servidores para el caso del transporte entre servidores. El principal desafío para que la distribución tenga máxima eficiencia es disponer de una conexión abierta con el usuario final (algo difícil de lograr en entornos de conectividad limitada por Firewalls o NAT).

El objetivo del proyecto es entender la conectividad de los usuarios finales (firewalls y NATs), conocer las técnicas para abrir puertos, y realizar pruebas en laboratorio y prototipos. Para esto se realizaron las siguientes etapas:

- estudiar el estado del arte en tecnologías que permitieran establecer comunicación P2P entre clientes detrás de NATs,
- evaluar implementaciones de estas tecnologías que fueran fácilmente integrables al proyecto GoalBit,
- integrar en un prototipo funcional las implementaciones para quebrar NATs estudiadas,
- probar el desempeño del prototipo final,
- y obtener conclusiones a partir de estudios previos sobre la presencia de los distintos tipos de NATs en las redes domésticas.

Palabras clave

P2P, NAT Traversal, NAT, port, IP, Open Source, VLC, GoalBit, LibNice, Pjsip, ICE, STUN, STUNT, GPL

Tabla de Contenidos

1 Introducción

1.1 Motivación

Según [1] las IPs de IPv4 no son suficientes para cada usuario de Internet tenga su propia dirección IP. Esto, sumado a la búsqueda de mayor seguridad, llevó a la creación de subredes que utilizan la solución NAT (*Network Address Translation*, especificado en RFC 1631 [2]) que traduce IPs internas a la subred en IPs que son visibles desde afuera de la subred.

En Internet un *host* puede ser identificado por una IP o por una IP y número de puerto TCP/UDP. NAT es una técnica por la cual las IPs o IPs y puertos de los *hosts* son traducidos de IPs privadas a IPs públicas y viceversa. Tanto en software como en hardware ofrece enrutamiento transparente hacia los *hosts* según un mapeo entre IPs públicas y privadas, basado en el concepto de sesión de comunicación. La razón principal de su nacimiento es solucionar a corto plazo la escasez de direcciones IPv4.

Los NAT están pensados para que las conexiones sean generadas desde la red interna hacia la externa, en un enfoque cliente servidor [3]. En conexiones Cliente-Servidor, que exista NAT no es un problema, ya que el que inicia la conexión es el cliente, pero en redes P2P hay problemas ya que no hay clasificación entre clientes y servidores puesto que cada *peer* puede cumplir ambos roles, y actuar como ambos simultáneamente.

Las NATs/firewalls no permiten conexiones entrantes a IPs privadas a no ser que la IP privada haya iniciado la conexión antes o el NAT esté específicamente configurado. Esto trae cierta privacidad y seguridad ya que esconde la IP privada de los *hosts* pero dificulta la comunicación entre pares. Sobre todo si ambos están detrás de un NAT.

1.2 Objetivos

En este proyecto se cuenta con un protocolo P2P y el programa GoalBit que lo implementa. Este protocolo usa conexiones TCP para establecer la comunicación entre los pares y intercambiar paquetes. La dificultad que se presenta y que se pretende resolver en este proyecto es cuando ambos están detrás de NATs.

Para ello se deberá estudiar y entender sobre protocolos P2P y sobre técnicas e implementaciones que permitan establecer conexiones entre clientes que están detrás de NATs. Una vez entendido esto se hará un prototipo de aplicación que permita atravesar NATs, inspirándose en la implementación abierta GoalBit.

1.3 Resumen del contenido de este documento

El crecimiento constante del ancho de banda y la posibilidad de acceder a Internet cada vez más al alcance de la gente, han tenido una incidencia en el comportamiento, hábitos y costumbres de los usuarios en Internet. Esta abundancia de banda ancha ha producido una demanda de recursos que antes no eran posibles, entre ellos el intercambio de material multimedia como música y video.

El consumo de video por *streaming* ha sido un fenómeno que explotó con la aparición de servicios como YouTube [4], que por las características del video de cada vez más calidad, requirió cambios estructurales no solo a nivel de servidores sino a nivel de infraestructura y herramientas de software.

GoalBit es un proyecto de licenciamiento libre y código abierto, cuyo fin es proporcionar una plataforma que brinde una experiencia de alta calidad en el consumo de video en tiempo real. En esta plataforma, la distribución de la señal de video es basada en el protocolo BitTorrent de las redes entre pares (*Peer to Peer*, P2P), adaptado para lograr un mejor aprovechamiento del ancho de banda de los involucrados en el consumo de la señal de video. De esta manera se quita parte de la carga del proveedor original de la señal de video, haciendo que los requerimientos de capacidad de transmisión se vean reducidos, poniendo la distribución de video al alcance de más gente.

Nuestro proyecto se enmarca dentro del contexto de GoalBit, con el objetivo de incrementar la posibilidad de que dos clientes puedan establecer una conexión P2P independiente de la topología de la red entre ellos, cuantas más conexiones se puedan establecer entre los diferentes *peers* para compartir información ya adquirida de la fuente de video menor será la carga sobre ella y más *peers* podrán acceder al video. Las topologías que se encuentran más habitualmente en los hogares incluyen un enrutador (NAT) que brinda conectividad a Internet a la red interna del hogar; conocido como *home gateway*, y que típicamente implementa NAT. Una de las características de los NAT es que no permitirán que recibamos información de una fuente a la cual no se la pedimos (está basado en el modelo cliente servidor).

Existen protocolos como UPnP (*Universal Plug and Play* [5]) implementados en algunos NAT para poder acceder a la red interna del NAT y poder establecer una conexión P2P. GoalBit puede hacer uso de UPnP si se encuentra presente, pero en casos donde no se encuentra, existe la posibilidad de no poder establecer la conexión P2P, limitando el tamaño de la red de *peers*. Es aquí donde nuestra solución debe actuar para brindarle a GoalBit un espectro más amplio respecto a las topologías y NATs presentes en ellas con los cuales pueda lidiar y establecer una conexión P2P exitosa.

Una de las dificultades principales que se presenta para la solución de este problema es que los NATs son un mecanismo de traducción entre IPs y puertos locales e IPs y puertos públicos que están clasificados en cuatro tipos principales según su comportamiento: *Full Cone*, *Restricted Cone*, *Port Restricted Cone* y *Symmetric* los cuales poseen comportamientos diferentes. Varios proyectos han atacado la problemática generada por los NATs a las redes P2P; entre los cuales se encuentran *“Implementing NAT Traversal on BitTorrent, NUTSS”*, *“A New Method for Symmetric NAT Traversal in UDP and TCP”*, *NATBLASTER”*, *“TCP connections for P2P Apps”* y *“TCP Traversal a través de NATs y Firewalls”*.

Para nuestro prototipo analizamos las siguientes herramientas: PJNATH, JINGLE LIB, LIBNICE, MOJOP2P, GNUNET, XSTUNT, GNUTELLA, BITCOMET, SHAREAZA y LIBUTP. Se hicieron dos prototipos, uno para integrar PJNATH a LibGoalBit y otro para integrar LIBNICE a LibGoalBit. Se descartó el prototipo que usa PJNATH por sólo ser útil para protocolos UDP cuando LibGoalBit sólo funciona con protocolo TCP. Con el prototipo de LIBNICE obtuvimos resultados satisfactorios en 7 de 9 casos probados con lo que se logrará una mejora sensible en el proyecto GoalBit al integrarle la librería LIBNICE.

1.4 Resultados y conclusiones

Se logró:

- Entender con mayor profundidad sobre NAT, NAT-traversal, protocolos P2P, clientes, servidores y transporte de video en red.
- Integrar una librería que implementa el protocolo ICE para atravesar NATs al proyecto GoalBit.
- Modificar el software del servidor para comunicar los nuevos datos necesarios para establecer el NAT-traversal.

Las conclusiones a las que llegamos fueron:

- No existen muchas soluciones que cumplan todos los requisitos que debía tener la librería a integrar al proyecto GoalBit, pero las dos que seleccionamos tenían una comunidad muy activa y dispuesta a ayudar.
- Es de fundamental importancia la interacción con los creadores de las librerías o programas para establecer cuales son los comportamientos esperados de las librerías, y generar ambientes de pruebas que permitan esos comportamientos esperados.
- Hace un largo tiempo que el problema de NAT está vigente y aún en el caso de un alto despliegue de IPv6 seguirá existiendo el problema.
- Dada la alta cantidad y variedad de programas y *hardware* que intervienen en programas que utilizan redes de computadoras es vital poder emular distintas configuraciones y contar con estadísticas de los distintos programas y *hardware* que tienen los usuarios como para estimar los resultados que se van a obtener una vez llevados los sistemas a producción.

1.5 Estructura del documento

En el Capítulo 1 “Introducción” se introduce el tema, se plantean los objetivos y se establece la estructura del documento.

En el Capítulo 2 “Fundamento teórico” se presentan los distintos conceptos que son necesarios para entender los capítulos siguientes.

En el Capítulo 3 “Proyectos similares” se muestran distintas implementaciones de protocolos de NAT traversal.

En el Capítulo 4 “Soluciones existentes” se estudian distintas librerías existentes y se selecciona una para integrarla al proyecto GoalBit.

En el Capítulo 5 “Solución propuesta” se detallan los pasos que fueron necesarios para integrar la librería seleccionada al proyecto GoalBit.

En el Capítulo 6 “Evaluación” se plantea un estudio sobre la existencia de distintos tipos de NATs, cómo se generó el ambiente de prueba y cuáles fueron los resultados obtenidos.

En el Capítulo 7 “Conclusiones” se presentan las conclusiones del proyecto: qué se logró, qué

dificultades hubo y posibles extensiones o mejoras al trabajo realizado.

Finalmente hay un capítulo para el Glosario y otro para la Bibliografía.

2 Fundamento teórico

La Sección 2.1 de este capítulo describe conceptos generales de Internet que son necesarios para entender el resto del capítulo. La Sección 2.2 contiene una descripción de la plataforma GoalBit a la cual se le quiere integrar una librería que permita NAT Traversal. La Sección 2.3 describe el concepto de NAT y los distintos tipos que existen. La Sección 2.4 describe dos conceptos utilizados en comunicaciones P2P para la apertura de puertos. La Sección 2.5 describe técnicas que permiten NAT Traversal. La Sección 2.6 describe algunas propiedades que facilitan NAT Traversal. La Sección 2.7 describe algunos estándares que facilitan NAT Traversal.

2.1 Conceptos generales de Internet

2.1.1. Sockets

Socket es una interfaz de programación de aplicaciones (API) para la familia de protocolos de Internet TCP/IP, provista usualmente por el sistema operativo. Los *sockets* de Internet constituyen el mecanismo para la entrega de paquetes de datos provenientes de la tarjeta de red a los procesos o hilos apropiados. Un *socket* queda definido por un par de direcciones IP local y remota, un protocolo de transporte y un par de números de puerto local y remoto. Todo programa que usa la red utiliza *sockets*, en particular las aplicaciones P2P hacen un uso avanzado de los mismos con la intención de abrir puertos.

Primitivas de un *socket*

Todos los *sockets* proporcionan una serie de primitivas, las cuales están orientadas a ser utilizadas por sus usuarios, de forma que se pueda establecer una comunicación determinada, y marcar unas pautas dentro de la misma.
Cada *socket* puede ser usado de múltiples formas y por diferentes usuarios, lo que invita a que para su mayor comprensión, sea prácticamente indispensable estudiar el conjunto de primitivas que ofrece. A continuación mostramos las primitivas más importantes que ofrecen, estudiando más profundamente aquellas que tienen más relevancia dentro de este conjunto.

La primitiva Socket.

Esta primitiva permite la creación de un *socket*, es decir, la creación e inicialización de entradas en las diferentes tablas del sistema de gestión de archivos, que son: tabla de descriptores de procesos, tabla de archivos y estructuras de datos, conteniendo las características del *socket*. Entre estas características se encuentran:

- el tipo, el dominio y el protocolo;
- el estado del *socket* (conectado o no, enlazado, en estado de recibir o de emitir);
- la dirección del *socket* conectado (si hay alguno): al *socket* se le asocia un *buffer* de emisión y otro de recepción;
- punteros a los datos (en emisión y en recepción);
- un grupo de procesos para la gestión de mecanismos asíncronos.

La forma general de la primitiva que permite crear un *socket* y obtener un descriptor para utilizarlo es:

```
int socket (dominio, tipo, protocolo)
int dominio;      /* AF_INET, AF_UNIX, ... */
int tipo;         /* SOCK_DGRAM, SOCK_STREAM, ... */
int protocolo;    /* 0: protocolo por defecto */
```

En caso de fallo de la primitiva se devuelve el valor **-1** y en errno estará codificado el error producido. Si no hay fallo se devuelve un descriptor referenciando al *socket*, que se utilizará en llamadas posteriores a funciones de la interfaz.

El primer parámetro especifica la familia de *sockets* que se desea emplear. El segundo parámetro especifica el tipo de *socket*. El tercer parámetro especifica el protocolo que se va a usar en el *socket*. Normalmente, cada tipo de *socket* tiene asociado sólo un protocolo, pero si hubiera más de uno, se especificaría mediante este argumento. Generalmente su valor es **0,** en cuyo caso la elección del protocolo se deja en manos del sistema.

La primitiva Close.

Un *socket* es suprimido cuando ya no hay ningún proceso que posea un descriptor para accederlo. Esta supresión, la cual es el resultado de al menos una llamada a la primitiva *close*, implica la liberación de entradas en las diferentes tablas y *buffers* reservados por el sistema relacionados con el *socket*.

La primitiva Bind.

Cuando se crea un *socket* con la llamada `socket`, se le asigna una familia de direcciones, pero no una dirección particular. Después de esto, un *socket* no es accesible más que por los procesos que conocen su descriptor. Sin un mecanismo suplementario de designación, sólo los procesos que hayan heredado tal descriptor en su creación (por la primitiva `fork`) podrían utilizar un *socket*. La primitiva `bind` permite con una sola operación dar un nombre a un *socket*, es decir, asignar un nombre a un *socket* sin nombre.

```
int bind (sock, p_direccion, lg)
int sock;                          /* descriptor de socket */
struct sockaddr *p_direccion;      /* puntero de memoria a la dirección*/
int lg;                            /* longitud de la dirección */
```

`Bind` hace que el *socket*, cuyo descriptor es `sock`, se una a la dirección de *socket* específica en la estructura apuntada por `p_direccion`; le indica el tamaño de la dirección.
Para una utilización en un dominio particular, el puntero `p_direccion` apunta a una zona cuya estructura es la de una dirección en ese dominio (`sockaddr_un` para el dominio AF_UNIX y `sockaddr_in` para el dominio AF_INET).
El *socket* nombrado después de realizarse con éxito la primitiva (valor de retorno **0**), puede ser identificado por cualquier proceso por medio de la dirección que se le ha dado sin necesidad de poseer un descriptor. Un cierto número de primitivas específicas permiten explotar estas direcciones.

Las causas de error (valor devuelto **-1**) de petición de conexión son múltiples: descriptor incorrecto, dirección incorrecta, inaccesible o ya utilizada, o un *socket* ya conectado a una dirección.

* Dominio UNIX.

En el dominio UNIX, los *sockets* no se destinan más que a una comunicación local. Por tanto, les corresponden direcciones locales que son referencias UNIX idénticas a las de los archivos. Un *socket* del dominio UNIX, aparecerá después del nombrado en los resultados producidos por la orden `ls` con el tipo `s`. La supresión de una referencia de este tipo, es por medio de la orden `rm` o de la primitiva `unlink`.

* Dominio Internet.

a) Preparación de la dirección.
La conexión a una dirección Internet de un *socket* de este dominio necesita la preparación de un objeto que tenga la estructura `sockaddr_in`. Esto supone en particular:

- el conocimiento de la dirección de la máquina local (obtenida por medio de la primitiva `gethostname` y la primitiva `gethostbyname`) o la elección del valor INADDR_ANY;
- la elección de un número de puerto.

b) Recuperación de una dirección.
Un proceso puede disponer de un descriptor de un *socket* conectado o enlazado a una dirección pero sin saber cuál es (si la conexión ha sido realizada por otro proceso y el proceso ha heredado el descriptor o si la conexión ha sido realizada sin especificar el número de puerto). La primitiva:

```
int getsockname (sock, p_adr, p_lg)
int sock;                       /* descriptor del socket */
struct sockaddr *p_adr;         /* puntero a la zona de dirección */
int *p_lg;                      /* puntero a la longitud de la dirección */
```

permite recuperar la dirección relacionada con el *socket* del descriptor `sock`. Cuando se llama a esta primitiva, el tercer parámetro se utiliza como dato y como resultado:

- en la llamada, `*p_lg` tiene como valor el tamaño de la zona reservada a la dirección `p_adr` para recuperar la dirección del *socket*;
- en el retorno de la llamada, tiene como valor el tamaño efectivo de la dirección. El valor de retorno de la primitiva es **0** o **-1** según si la llamada ha tenido éxito o no.

La primitiva Connect.

Después de la creación de un *socket*, un proceso hace la llamada connect para establecer una conexión activa con otro proceso remoto, es decir, abre un circuito virtual entre dos *sockets*, el cual permite intercambios bidireccionales

```
int connect (sock, p_adr, lgadr)
int sock;                       /* descriptor del socket local */
struct sockaddr *p_adr;         /* dirección del socket remoto */
```

```
int lgadr;                    /* longitud dirección remota */
```

El socket correspondiente al descriptor `sock` será conectado o enlazado a una dirección local. `Connect` intenta contactar con el ordenador remoto con el objeto de realizar una conexión entre el *socket* remoto y el *socket* local especificado en `sock`.

La conexión puede establecerse (y la primitiva `connect` devuelve el valor **0**) si se cumplen las siguientes condiciones:

a) los parámetros son "localmente" correctos ;
b) la dirección `*p_adr` se asocia a un *socket* del tipo SOCK_STREAM en el mismo dominio que el *socket* local de descriptor `sock` y un proceso (servidor) tiene solicitado escuchar sobre este *socket* (por una llamada a `listen`);
c) la dirección `*p_adr` no está utilizada por otra conexión;
d) el archivo de conexiones pendientes del *socket* distante o remoto no está lleno.

En caso de éxito (valor de retorno **0** de la primitiva `connect`), el *socket* local `sock` está conectado con un nuevo *socket* y la conexión está pendiente hasta que el proceso llamado tenga conocimiento de ella a través de la primitiva `accept`. Sin embargo, el proceso que llama puede comenzar a escribir o a leer del *socket*.

En el caso de que no se cumpla alguna de las tres primeras condiciones, el valor devuelto por la primitiva es **-1**, indicando error.

El comportamiento de la primitiva es particular si no se cumple la condición d:

- si el *socket* es de modo bloqueante, el proceso se bloquea. La petición de conexión se repite durante un cierto tiempo; si al cabo de este lapso de tiempo la conexión no se ha podido establecer, el proceso es despertado.
- si el *socket* es de modo no bloqueante, la vuelta es inmediata. Sin embargo, la petición de conexión no se abandona en seguida (se repite durante el mismo lapso de tiempo). Finalmente, en el caso de que se realice una nueva solicitud de conexión del mismo *socket* (por una llamada a `connect`) antes de que se haya abandonado la petición anterior, la vuelta de la primitiva es igualmente inmediata.

La primitiva Write.

A través de esta primitiva, dos procesos pueden intercambiar información a través de una conexión TCP. En el caso de un cliente y un servidor, el cliente realizaría con esta primitiva peticiones, y el servidor les daría réplica.

Esta primitiva requiere tres argumentos: el descriptor de un *socket* al cual deben ser enviados los datos, la dirección de los datos a ser enviados y la longitud de los datos.

```
int write (sock, msg, lg)
int sock;                     /* descriptor del socket local */
char *msg;                    /* dirección de memoria del mensaje a enviar */
```

```
int lg;                 /* longitud del mensaje */
```

Normalmente, para facilitar la comunicación, la primitiva `write` no manda la información directamente sobre el *socket*, sino que lo hace sobre un *buffer* perteneciente al *kernel* del sistema operativo, lo que permite que la aplicación siga su ejecución mientras se transmiten los datos a través de la red. En el caso de que el *buffer* estuviese lleno, se bloquearía la aplicación hasta que los datos del mismo pudiesen ser mandados por la red.

La primitiva Read.

De la misma forma que con la primitiva `write` los procesos mandaban información a través de la red, con la primitiva `read`, los procesos esperan información procedente de otros procesos que desean comunicarse con ellos. El uso de una primitiva `write` por parte del proceso llamante, implica que para que el proceso receptor pueda recibir lo que ha escrito el primero, necesita ejecutar la primitiva `read`, de forma que el proceso receptor pueda leer los datos de una conexión TCP.

Al igual que la primitiva `write`, esta primitiva requiere también tres argumentos: el descriptor del *socket* a usar, la dirección del mensaje a enviar y la longitud de dicho mensaje.

```
int write (sock, msg, lg)
int sock;               /* descriptor del socket local */
char *msg;              /* dirección de memoria del mensaje a enviar */
int lg;                 /* longitud del mensaje */
```

Cuando llegan datos, `read` los copia del *socket* y los ubica en el *buffer* del área de usuario. En el caso de que no haya llegado ningún dato, `read` bloquea la aplicación hasta que llegue algún dato al *socket*. Si llega más información de la que cabe en el *buffer*, read solo extrae lo que tenga cabida en el *buffer*, y en el caso de que hayan llegado menos datos que espacio en el *buffer*, `read` los copia todos y devuelve el número de bytes leídos.

La primitiva Listen.

Cuando un *socket* es creado, no se encuentra en estado pasivo (por ejemplo, preparado para ser usado por un servidor) ni activo (por ejemplo, para ser usado por un cliente), hasta que la aplicación tome una acción determinada. Los servidores en modo conectado realizan la llamada `listen` para colocar un *socket* en modo pasivo (pasándolo como argumento), y prepararlo para futuras conexiones entrantes que tendrán lugar, es decir, no hará nada hasta que llegue una conexión. Para llamar con éxito a esta primitiva (con valor de retorno **0**), el proceso debe disponer de un descriptor de *socket* del tipo SOCK_STREAM.

La mayoría de los servidores consisten en un bucle infinito que acepta cada conexión entrante que le llega, la maneja, y después sigue en el bucle a la espera de otras conexiones. Si en el momento en el que el servidor atiende una petición (conexión) no llega ninguna otra conexión, no existe problema, pero este si existe en el momento en que el servidor está atendiendo una determinada conexión, y en ese instante llega otra. Para asegurar de que esta nueva conexión no se pierda, a la primitiva listen se le pasa un argumento que le indica al sistema operativo que encole las peticiones de conexión para un

socket. Así pues, los dos argumentos que se les pasa a la llamada `listen` son el *socket* que debe ser ubicado en modo pasivo, y el tamaño de la cola que va a ser usada por dicho *socket*.

```
int listen (sock, nb)
int sock;          /* descriptor de socket */
int nb;            /* número máximo de peticiones de conexión pendiente */
```

La primitiva Accept.

Esta primitiva permite extraer una conexión pendiente de la cola asociada a un *socket* para la cual se ha realizado una llamada a `listen`. De esta manera, el sistema toma conocimiento de un enlace realizado.

```
int accept (sock, p_adr, p_lgadr)
int sock;                          /* descriptor del socket */
struct sockaddr *p_adr;            /* dirección del socket conectado */
int *p_lgadr;                      /* puntero al tamaño de la zona reservada a p_adr */
```

Una vez extraída la petición de conexión, `accept` crea un nuevo *socket* con las mismas propiedades que `sock` y reserva (que se devuelve como resultado de la función) un nuevo descriptor de fichero para él. El *socket* de servicio creado se enlaza a un nuevo puerto (tomado del conjunto de puertos no reservados).

A la vuelta, también se recupera memoria en la zona apuntada por `p_adr`, la dirección del socket del cliente con el cual se ha restablecido la conexión. El valor de `*p_lgadr` se modifica: si en la llamada era el tamaño de la zona reservada para guardar la dirección, en el retorno da el tamaño efectivo de la dirección.

En el caso en el que no exista ninguna conexión pendiente, el proceso se bloquea hasta que exista una (o hasta que llegue una señal) a menos que el *socket* esté en modo no bloqueante.

2.1.2. Secuencia de conexión TCP (TCP Connection Sequence)

Según [6] y [7] en TCP las conexiones se establecen usando el acuerdo de tres vías. El servidor espera una conexión entrante ejecutando *listen* y *accept*, y especificando cierto origen o ninguno. El cliente ejecuta un *connect* especificando la IP y el puerto con el que se desea conectar. El *connect* envía un segmento TCP con el bit SYN encendido y el bit de ACK apagado, y espera una respuesta.

Al llegar el segmento al servidor, la entidad TCP revisa si hay un proceso que haya ejecutado un `listen` en el puerto indicado en el campo puerto de destino. Si no lo hay envía una respuesta con el bit RST encendido para rechazar la conexión.

Si algún proceso esta escuchando el puerto, ese proceso recibe el segmento TCP entrante y puede entonces aceptar o rechazar la conexión, si la acepta devuelve un segmento de confirmación de recepción SYN/ACK con los bits SYN y ACK encendidos. En la Figura 2.1(a) se ve la secuencia normal y en la Figura 2.1(b) se ve la secuencia en que dos *hosts* intentan simultáneamente establecer

una conexión.

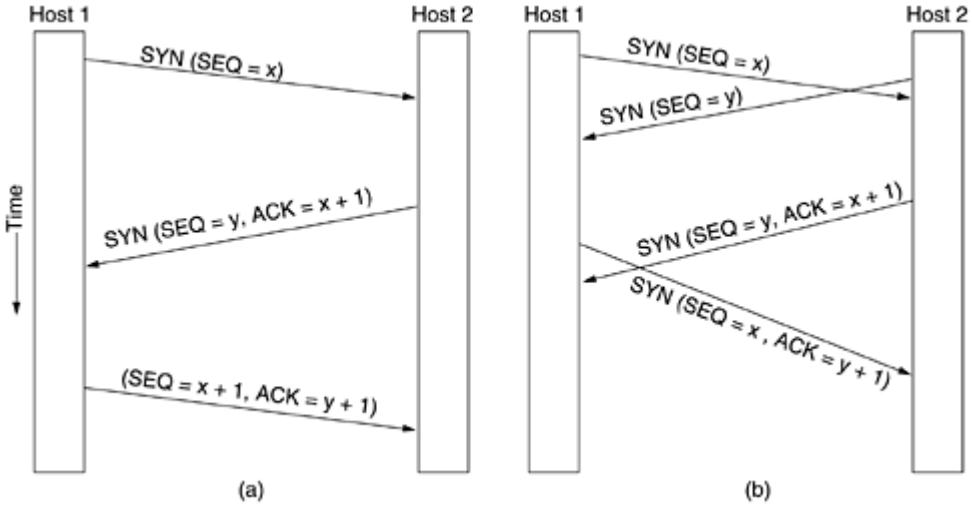

Figura 2.1: TCP Connection Sequence

El resultado de estos eventos es que solo se establece una conexión, no dos, pues las conexiones se identifican por sus puntos terminales. Si el primer establecimiento resulta en una conexión identificada por (x, y), al igual que en el segundo, solo se hace una entrada en la tabla, es decir, la de (x, y). Como veremos más adelante, la mayoría de los métodos de NAT-Traversal utilizan el proceso de conexión en formas peculiares con la intención de burlar la seguridad existente.

2.1.3. Sesión de comunicación

En [1] y [8] se define que una sesión de comunicación entre dos *hosts* es identificada por una 4upla (IP local, puerto local, IP destino, puerto destino). En [9] hay una discusión más amplia de dirección, nombre, puerto y ruta.

2.1.4. Enrutamiento entre dominios sin clases (*Classless Inter Domain Routing*)

Según [10] y [11] el grupo *Road Group* produjo el CIDR que permitió que los bloques de IPs que antes se asignaban en grupos grandes se asignaran con una granularidad más fina. Antes del CIDR los bloques de IPs venían en tres tamaños: 256, 65.000 y 16 millones (clases de IPs C, B y A respectivamente). De esta manera una organización con unos pocos miles de IPs recibían 65.000 IPs, desperdiciándose la mayoría de ellas. Con el CIDR los bloques de IPs pueden venir en cualquier potencia de 2. CIDR ha permitido un uso mucho más eficiente de las direcciones IPv4, a pesar de esto el NAT es la principal forma de conectarse a Internet por usuarios finales.

2.1.5. IP privada

Para contener la escasez de IP se definió IP privada e IP pública como se detalla en [12]. La Internet

Assigned Numbers Authority (IANA) reservó los siguientes tres bloques de direcciones IP para redes privadas:

- 10.0.0.0 - 10.255.255.255 (prefijo 10/8)
- 172.16.0.0 - 172.31.255.255 (prefijo 172.16/12)
- 192.168.0.0 - 192.168.255.255 (prefijo 192.168/16)

El primer bloque es el de 24 bits, el segundo de 20 bits y el tercero de 16 bits. Estos corresponden a una red A, 16 redes B y 256 redes C de la notación anterior a CIDR respectivamente.

2.1.6. SIP - Session Initiation Protocol

SIP, o *Session Initiation Protocol* es un protocolo de control y señalización usado mayoritariamente en los sistemas de Telefonía IP, que fue desarrollado por el IETF (RFC 3261 [13]). Dicho protocolo permite crear, modificar y finalizar sesiones multimedia con uno o más participantes y sus mayores ventajas recaen en su simplicidad y consistencia. En los últimos tiempos la mayoría del desarrollo de NAT-Traversal se ofreció en conjunto con los protocolos SIP y SDP de forma de disponer de una herramienta potente y única para las comunicaciones punto a punto.

Funciones SIP

El protocolo SIP actúa de forma transparente, permitiendo el mapeo de nombres y la redirección de servicios ofreciendo así la implementación de la IN (*Intelligent Network*) de la PSTN o RTC.

Para conseguir los servicios de la IN el protocolo SIP dispone de distintas funciones. A continuación se enumeran las más importantes:

- Localización de usuarios (SIP proporciona soporte para la movilidad),
- Capacidades de usuario (SIP permite la negociación de parámetros),
- Disponibilidad del usuario,
- Establecimiento y mantenimiento de una sesión.

En definitiva, el protocolo SIP permite la interacción entre dispositivos, cosa que se consigue con distintos tipos de mensajes propios del protocolo. Dichos mensajes proporcionan capacidades para registrar y/o invitar un usuario a una sesión, negociar los parámetros de una sesión, establecer una comunicación entre dos a más dispositivos y, por último, finalizar sesiones.

2.1.7. SDP - *Session Description Protocol*

Session Description Protocol (SDP), es un protocolo para describir los parámetros de inicialización de los flujos multimedia. Corresponde a un protocolo actualmente redactado en el RFC 4566 [14] por la IETF.

SDP está pensado para describir sesiones de comunicación multimedia cubriendo aspectos como anuncio de sesión, invitación a sesión y negociación de parámetros. SDP no se encarga de entregar los contenidos propiamente dichos sino de entablar una negociación entre las entidades que intervienen en la sesión como tipo de contenido, formato, y todos los demás parámetros asociados. Este conjunto de parámetros se conoce como perfil de sesión. SDP se puede ampliar para soportar nuevos tipos de medios y formatos.

Comenzó como componente del SAP (*Session Announcement Protocol*), pero encontró otros usos en conjunto con RTP (*Real-time Transport Protocol*), SIP y como formato independiente para describir sesiones *multicast*.

Descripción de la sesión

Una sesión se describe con una serie de atributos, cada uno en una línea. Los nombres de estos atributos son un carácter seguido por '=' y el valor respectivo. Existen parámetros opcionales, denotados con '=*'. Los valores pueden ser una cadena ASCII, o una secuencia específica de tipos separada por espacios. La sintaxis de SDP se puede ampliar y ocasionalmente se agregan nuevos atributos a la especificación. A continuación se muestra un formato para el uso de SDP:

v= (Versión del protocolo)
o= (Origen e identificador de sesión)
s= (Nombre de sesión)
i=* (Información de la sesión)
u=* (URI de descripción)
e=* (Correo electrónico)
p=* (Número telefónico)
c=* (Información de conexión)
b=* (Cero o más líneas con información de ancho de banda)

Una o más líneas de descripción de tiempo (Ver abajo "t=" y "r=")
z=* (Ajustes de zona horaria)
k=* (Clave de cifrado)
a=* (Cero o más líneas de atributos de sesión)
Cero o más descripciones de medios

Descripción de tiempo
t= (Tiempo durante el cual la sesión estará activa)
r=* (Cero o más veces de repetición)

Descripción de medios, si está presente
m= (Nombre de medio y dirección de transporte)
i=* (Título)
c=* (Información de conexión)
b=* (Cero o más líneas con información de ancho de banda)
k=* (Clave de cifrado)
a=* (Cero o más líneas de atributos de sesión)

2.2 GoalBit

GoalBit [15] [16] [17] es un sistema de distribución P2P capaz de difundir el contenido de una señal de video en vivo sin interrupciones, a altas tasas de transferencia y preservando su calidad. La difusión está basada en una adaptación del protocolo *BitTorrent* [18], y a diferencia de otros programas que utilizan este protocolo para la difusión de archivos de toda índole, GoalBit se enfoca en capturar flujos multimedia en vivo y reproducir el contenido en el momento. Está orientado principalmente a la distribución de televisión en vivo (*broadcast-tv*).

Fue desarrollado tomando como base el sistema VLC[19], el cual es un reproductor de video con la capacidad de reproducir video de varios tipos de fuente (*streaming*, *file* y *devices*). Esto aporta al sistema varias ventajas en cuanto a la reproducción de contenido multimedia en múltiples formatos, y

para la distribución del contenido por la red ya que dispone de módulos con muchas funcionalidades de gran utilidad para estos propósitos.

Para su desarrollo se incorporaron al reproductor multimedia VLC varios módulos con distintas funcionalidades. Entre ellos encontramos el módulo del cliente BitTorrent que está basado en la implementación del *Enhanced CTorrent* [20]. También se incorporó un módulo para el *tracker* que está basado en la implementación del Open *Tracker* [21]. Además de estos dos, se incluyeron módulos que se encargan del manejo de la ejecución y la interfaz gráfica entre otros.

Por la naturaleza del protocolo, el video nace en una o más fuentes, y se distribuye en piezas para poder ser compartido eficientemente entre los usuarios. Un usuario obtiene de manera directa o indirecta las piezas de video y las almacena para su inmediata o posterior reproducción además de compartirlas con otros usuarios de la red si así lo desea.

En la adaptación de este protocolo es que se produce la diferencia entre *GoalBit* y *BitTorrent* de cómo interpretar una red P2P. *GoalBit* tiene más categorías para los nodos de una red, que dependen tanto del papel que desempeñan como de sus capacidades de transferencia. Además, para la transmisión de video en vivo, se tienen restricciones de tiempo real que no se tienen en la transferencia de archivos de otro tipo.

Cuando la distribución de archivos (contenido) se realiza desde una servidor central, si la cantidad de clientes aumenta también aumenta la carga sobre el servidor central. Para solucionar este problema *BitTorrent* se basa en conexiones P2P entre clientes *seeds* y *peers*. Los *seeds* poseen una copia completa del contenido y el peer una parcial transformándose en *seed* cuando adquiere la totalidad de contenido. El descubrimiento de otros clientes se da a través de un tracker, el cual el *peer* conoce por la información incluida en un archivo *.torrent* [20] entre otros datos. De esta manera los *peers* pueden comunicarse con otros *peers* que posean piezas que ellos no poseen y los *seeds* pueden comunicarse con *peers* para brindarles las piezas del contenido que estos no poseen.

2.2.1. Arquitectura

A continuación introducimos brevemente los distintos tipos de nodos existentes en la plataforma *GoalBit* y la forma en que interactúan entre sí.

La transferencia de la secuencia de video es iniciada por un nodo llamado *broadcaster* que contiene al *broadcaster server* y al conjunto de *broadcaster-peers*. El *broadcaster server* es el encargado de obtener la secuencia de video del medio físico y transmitirle a los *broadcaster-peers* las piezas del video que están listas para ser enviados a través de la red. Los *broadcaster-peers* son los que inician la distribución de esas piezas en la red.

Luego tenemos a los *super-peers* y a los *normal-peers*, donde ambos son nodos que pueden conectarse y desconectarse en cualquier momento. Los *super-peers* son los únicos autorizados a comunicarse con los *broadcaster-peers* y también pueden comunicarse entre sí y con los *normal-peers*. Los *normal-peers* (también llamados simplemente *peers*) sólo pueden comunicarse entre sí y con los *super-peers*.

Los *super-peers* y los *peers* se diferencian en la cantidad de recursos que pueden brindar a la red, el objetivo de un *peer* es simplemente ver el video, mientras que los *super-peers* están para ayudar en la distribución a la mayor cantidad de *peers*.

Además de los nodos mencionados, otra entidad presente en GoalBit es el *tracker*, que es el encargado de mantener la información de los *peers* que están viendo cada uno de los canales. Esa información es utilizada para entablar la comunicación entre los nodos.

Típicamente un *peer* que participa de GoalBit, cumple la siguiente secuencia de sucesos:

1. El *peer* elige ver un canal.
2. El *peer* se conecta con el *tracker* y obtiene la lista de otros *peers*.
3. El *peer* comienza a intercambiar piezas con esos *peers*.

La política de intercambio de piezas define la eficiencia de la transferencia de la señal. En relación con esta política, los *super-peers* son más solidarios porque disponen de mayor ancho de banda, mientras que los *normal-peers* son más estrictos y preservan una política de tipo “ojo por ojo, diente por diente”. La Figura 2.2 muestra cómo es la interacción entre las entidades que participan de GoalBit.

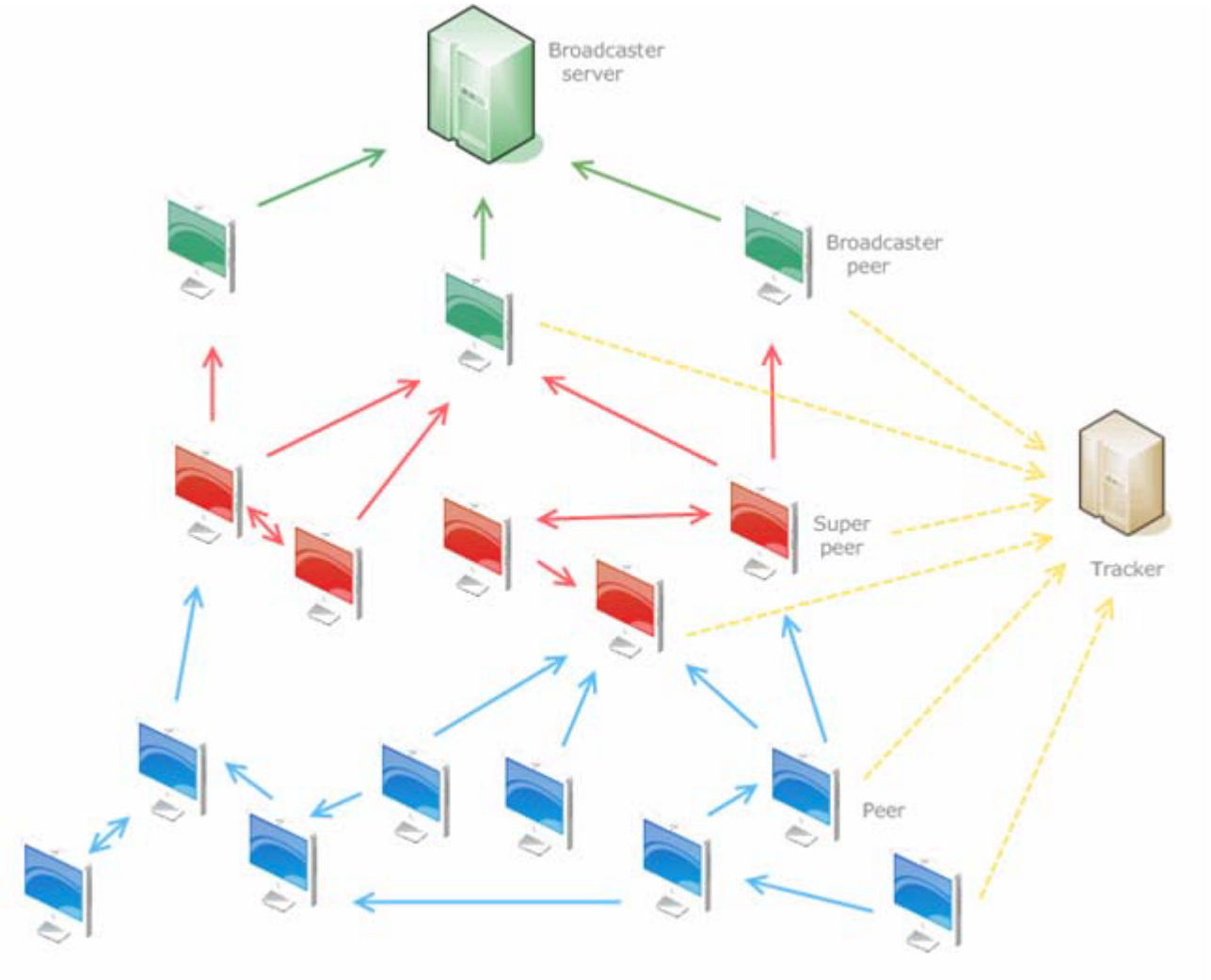

Figura 2.2: Interacción de las entidades de GoalBit Los normal-peers sólo se comunican con los super-peers y entre sí, los super-peers son los únicos autorizados a comunicarse con los broadcaster-peers. Estos últimos obtienen las piezas del broadcaster server. Todos los peers se comunican con el tracker.

Por tanto, el sistema contiene cuatro componentes diferentes: *broadcaster*, *super-peers*, *peers* (o *normal-peers*) y el *tracker* (ver Figura 2.3).

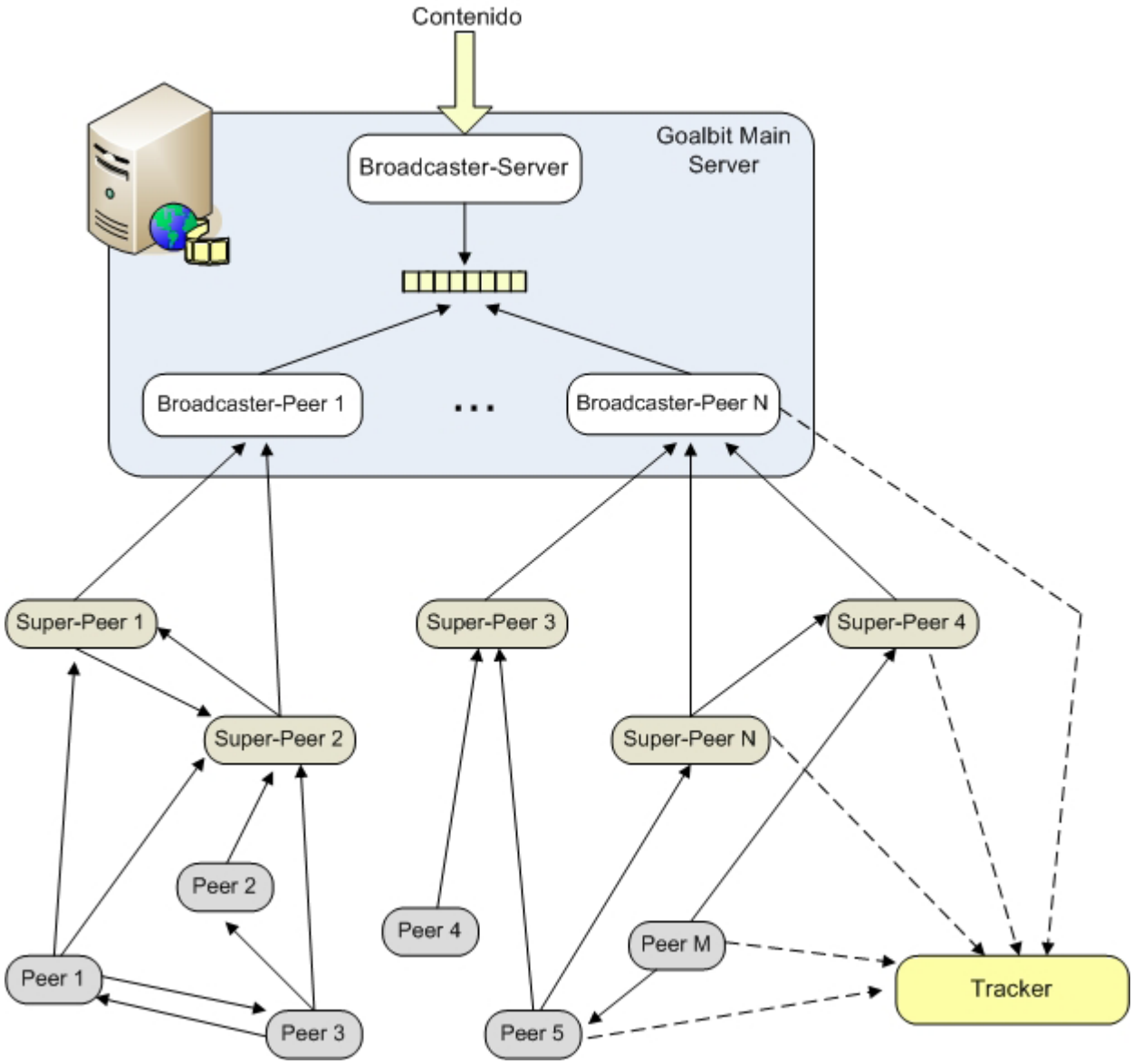

Figura 2.3: Componentes de la plataforma GoalBit.

El *broadcaster* se encarga del ingreso de video en la red y de su adaptación para ser distribuido en la plataforma GoalBit. Puede obtener el video de múltiples fuentes (las proporcionadas por VLC), por ejemplo un *streaming*, un DVD, una tarjeta de video, etc. El *broadcaster* tiene dos componentes: el *broadcaster server* y los *broadcaster-peers*.

Los *super-peers* son nodos que se caracterizan por tener buen ancho de banda y su finalidad es ayudar a los *broadcaster-peers* en la distribución masiva del contenido. Los *super-peers* consumen *chunks* de los *broadcaster-peers* y de otros *super-peers* y los distribuyen a los *peers*.

Los peers consumen el contenido desde los *super-peers* y de otros *peers*. La implementación actual exige que un peer elija ser *peer* o *super-peer*, pero en el futuro prevé que esto sea dinámico de acuerdo a sus recursos. Desde el punto de vista de la implementación los *peers* y *super-peers* se diferencian en el algoritmo con el que comparten las piezas.

El *tracker* cumple con todas las funciones de un tracker BitTorrent. Además, es quien decide el punto del video en donde debe comenzar a descargar y reproducir la señal cada *peer.*

GoalBit también puede ser utilizado en una modalidad más simple eliminando la jerarquía: cuando no se cuenta con *super-peers* en la red es posible utilizar nodos *broadcaster-super-peers* que ingresen el contenido en la red y lo distribuyan directamente a los *normal-peers* (ver Figura 2.4).

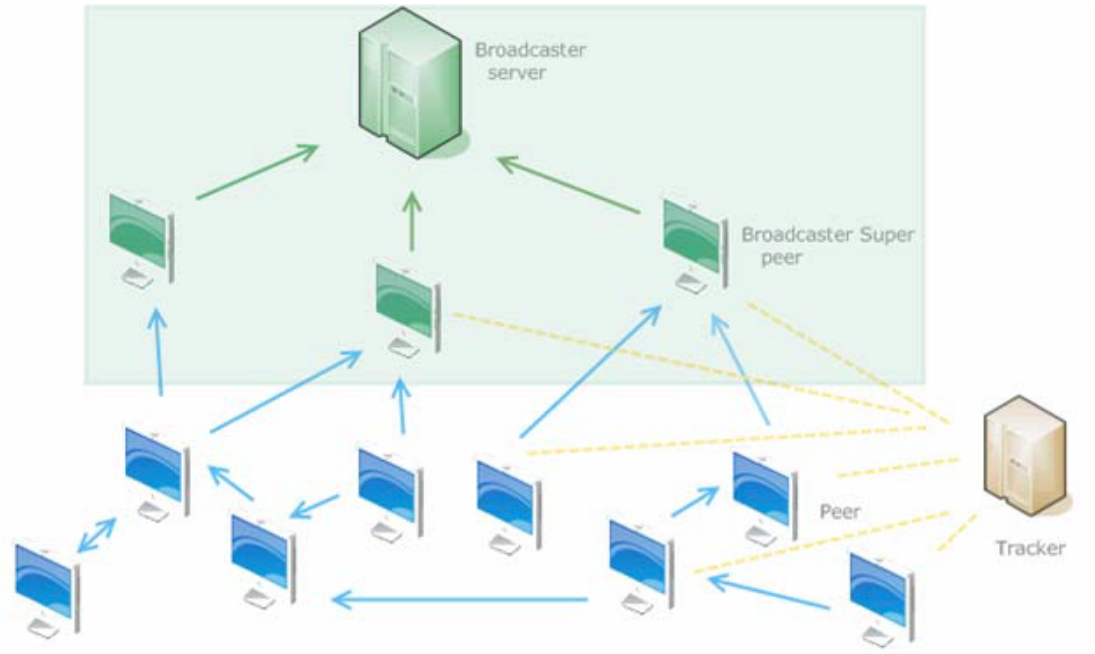

Figura 2.4: Componentes de la plataforma GoalBit simplificada.

2.2.2. GoalBit Transport Protocol (GBTP)

El *GoalBit Transport Protocol [17]* es una extensión al protocolo BitTorrent optimizada para la transmisión de contenido en vivo. Para permitir esto, el protocolo cuenta con varias adaptaciones ya que, a diferencia de BitTorrent, los *chunks* tienen que descargarse con cierto orden, de tal forma que se pueda reproducir el contenido de forma continua.

Una de las diferencias más importantes entre el GBTP y BitTorrent es que en este último los archivos a descargar tienen un largo determinado, mientras que para GBTP la señal no tiene un final, sino que es un flujo constante de contenido. Es por esto que GBTP tiene una ventana circular que indica el rango de las piezas con las que está trabajando cada peer en cada momento, y tanto el buffer donde se van guardando las piezas descargadas, como el *BitField* son relativos a esa ventana.

Otra diferencia es en el archivo .torrent (que en el caso de GoalBit es un archivo .goalbit), donde no se puede incluir el *CheckSum* de los *chunks*, ya que estos se van generando dinámicamente por el *broadcaster* a medida que se van obteniendo de la fuente y como fue indicado anteriormente, la cantidad de los mismos no es acotada. En [15] se puede consultar el formato de los archivos .goalbit y los mecanismos de seguridad (basados en firmas digitales) utilizados.

2.2.3. Active Buffer

Como parte de la extensión se introducen dos nuevos conceptos importantes en la especificación del protocolo: *Active Buffer* (buffer activo) y *Active Buffer Index* o ABI (índice del buffer activo), ver Figura 2.5. El *Active Buffer* es la secuencia consecutiva de *chunks* disponibles a partir de la línea de ejecución del reproductor. El *Active Buffer Index* es el máximo número de secuencia en el *Active Buffer.* En otras palabras, el *Active Buffer* es el video descargado hasta donde el *peer* puede reproducir ininterrumpidamente, medido en *chunks*.

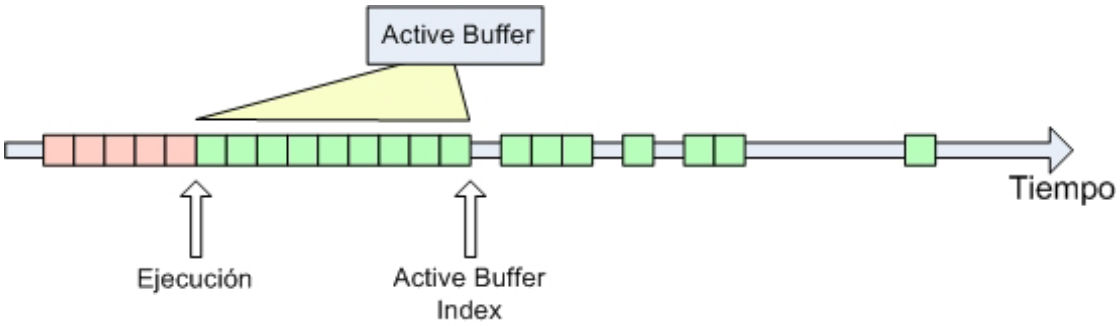

Figura 2.5: Active Buffer y Active Buffer Index

Los *chunks* se numeran del **0** a **MAX_PIECE_ID** (número máximo de *chunk* en el sistema), cuando se llega a este valor, se vuelve a comenzar desde **0**.

2.2.4. Comunicación Peer–Tracker

Al iniciarse, el *peer* se conecta con el *tracker* para unirse a un *swarm* de *peers* (conjunto de peers) y obtener el índice de pieza que es conveniente comenzar a reproducir. Esto determina inicialmente que el ABI del *peer* arranca en el índice de pieza que el *tracker* le comunicó. El número de pieza es el promedio de los ABIs del *swarm*.

Dependiendo del tipo de *peer*, el *tracker* devuelve distintos tipos de *swarms*: los *broadcaster-peers* no

utilizan *swarm*, los *super-peers* deben recibir la lista de los *broadcaster-peers* y de los otros *super-peers*, y los *peers* deben recibir una lista con algunos *super-peers* y muchos *peers*.
Para mejorar la eficiencia, el *swarm* informado por el *tracker* puede ser geográficamente próximo al *peer* que se está iniciando y además los peers del *swarm* deben estar descargando *chunks* cercanos (que tengan ABIs similares).

El tracker puede estimar el valor del ABI de cada *peer* a lo largo del tiempo a partir del último valor del ABI informado y el momento en que fue informado, basándose en que el consumo de *chunks* tiene velocidad constante. Sin embargo, algunas situaciones pueden alterar el valor del ABI del lado de los *peers* (por ejemplo, un *rebuffering*). Por este motivo, periódicamente los *peers* informan el valor del ABI actual al *tracker.*

Las comunicaciones periódicas entre el *peer* y el *tracker* (llamadas anuncios) incluyen dos campos enviados por el *peer* al *tracker*: *swarm* (tamaño máximo de *swarm* deseado) y ABI (ABI actual del peer). Cuando se está anunciando un nuevo *peer*, por defecto se usa *swarm* = **100** y ABI = **MAX_PIECE_ID** + **1**. Cuando se realiza comunicación periódica o información de *rebuffering*, *swarm* = **0** y ABI = ABI actual del *peer.* Cuando el *peer* comienza a quedarse sin *peers* en su *swarm*, puede solicitar más al *tracker, swarm* = **100** y ABI = ABI actual del *peer.*

2.2.5. Comunicación Peer–Peer

Una vez que el *peer* obtuvo el *swarm* y el ABI inicial del *tracker,* comienza a comunicarse con los *peers* del *swarm*.
La comunicación entre dos *peers* consta de los siguientes pasos:

1. *Handshake*: los *peers* aceptan comunicarse entre sí.
2. *Bitfield*: los *peers* intercambian sus *bitfields* (para informar qué *chunks* tiene cada uno).
3. *Interested*: un *peer* le comunica al otro que desea descargar un *chunk*.
4. *Unchoke*: un *peer* le comunica al otro que está dispuesto a recibir pedidos de él.
5. *Request*: un *peer* le solicita al otro una parte de un *chunk*.
6. *Piece*: un *peer* le envía al otro una parte de un *chunk*.

El flujo de la comunicación puede observarse en la Figura 2.6.

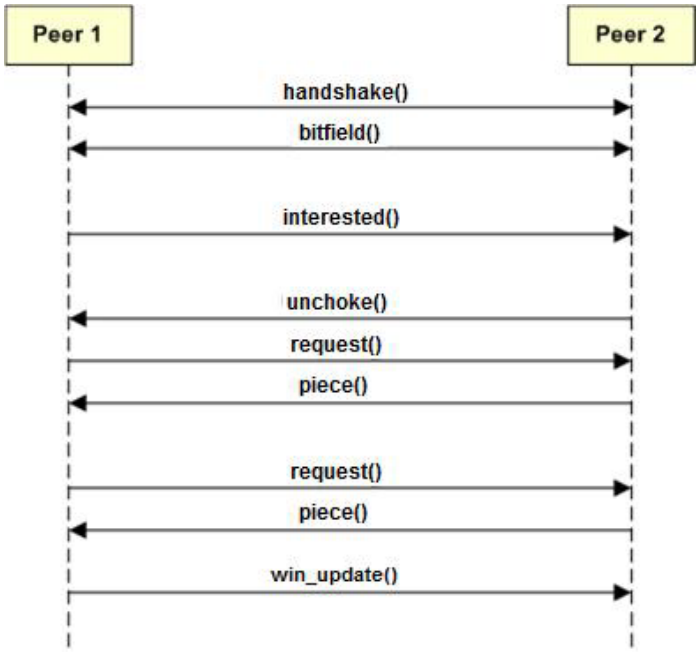

Figura 2.6: Comunicación entre peers.

2.2.6. Especificación del Handshake

En el proceso del *handshake* existen dos actores, denominados Iniciador (quien realiza el primer *handshake*) y Consultado (quien recibe un *handshake* sin haber realizado uno previamente).
El Iniciador informa en el mensaje de *handshake* la Identidad (*broadcaster-peer, super-peer, peer*) y el ABI.

El Consultado usa la siguiente estrategia para aceptar o no el *handshake*:

- Si el consultado es un *broadcaster-peer,* sólo aceptará un *handshake* si el iniciador es un *super-peer.*
- Si el consultado es un *super-peer* y el iniciador es un *peer,* se aceptará el *handshake*.
- Si el consultado y el iniciador son *super-peers*, y se cumple que ambos tienen piezas de interés mutuo, se aceptará el *handshake*.
- Si el consultado es *peer* (sea cual sea el iniciador), en caso de tener piezas de interés mutuo se aceptará el *handshake*.

En los dos últimos casos se utiliza el valor del ABI para determinar si los *peers* tienen piezas de interés mutuo, ya que al momento del *handshake* todavía no se ha realizado el intercambio de los *bitfields*. Con el valor del ABI cada uno de los *peers* puede saber hasta qué *chunk* a podido descargar el otro *peer* y a su vez, cuáles son los próximos *chunks* que intentarán descargar cada uno de ellos en el futuro cercano.
A continuación se incluye un pseudocódigo de la estrategia de *handshake* de GoalBit:

```
switch ( Consultado->identidad ) {
   case BROADCASTER_PEER :
      if ( Iniciador->identidad == SUPER_PEER )
         accept_handshake();
      else
         reject_handshake();
      break;
   case SUPER_PEER :
      if ( Iniciador->identidad == PEER )
         accept_handshake();
      else if ( Iniciador->identidad == SUPER_PEER )
         Consultado->analyze_handshake( Iniciador );
      else
      reject_handshake();
      break;
   case PEER :
      Consultado->analyze_handshake( Iniciador );
      break;
}

function accept_handshake() {
   //Esta función acepta el handshake.
}

function reject_handshake() {
   //Esta función rechaza el handshake.
}

function analyze_handshake( Iniciador ) {
   //Esta función acepta el handshake cuando ambos peers se pueden
   //"ayudar" mutuamente.
   bool can_i_help_him = this->can_i_help_him( Iniciador );
   bool can_he_help_me = this->can_he_help_me( Iniciador );
   if ( can_i_help_him && can_he_help_me )
      accept_handshake();
   else
      reject_handshake();
}

function can_i_help_him( Iniciador ) {
   //Esta función devuelve true si el peer Consultado tiene algún
   //chunk a partir del ABI del peer Iniciador.
}

function can_he_help_me( Iniciador ) {
   //Esta función devuelve true si el peer Consultado necesita algún
   //chunk en un entorno de radio X y centro el ABI del peer
   //Iniciador.
}
```

2.2.7. Especificación del envío del Bitfield

El *bitfield* se envía inmediatamente después del *handshake* con el objetivo de informar los *chunks* que el *peer* tiene. En ese proceso también intervienen dos actores: el *peer* local (local_peer) y el *peer* externo (external_peer). El local_peer es el *peer* que envía el *bitfield* y el external_peer es el *peer* que recibe el *bitfield*.
El *bitfield* se envía de acuerdo a la siguiente estrategia:

- En caso de que el local_peer sea un *broadcaster*, se envía un *bitfield* vacío (seeder_bitfield).

Esto es por convención e indica que el local_peer cuenta con todas las piezas.

- En caso de que el local_peer sea un super-peer y el external_peer sea un normal-peer, ocurrirá lo mismo que en el caso anterior, se enviará un seeder_bitfield. Si el external_peer es un super-peer, se enviará un *bitfield* que lleva la información de qué piezas contiene el local_peer en un rango convenido, a partir del ABI del external_peer.
- Si local_peer es un normal-peer se enviará un *bitfield* que lleva la información de qué piezas contiene el local_peer en un rango convenido, a partir del ABI del external_peer.

A continuación se incluye el pseudocódigo de la estrategia de envío de *bitfield* de GoalBit (observar que el *bitfield* que envía cada local_peer es adaptado a la ventana donde esta ejecutando el external_peer):

```
switch ( local_peer->identidad ) {
   case BROADCASTER_PEER :
      local_peer->send_seeder_bitfield();
      break;
   case SUPER_PEER :
      if ( external_peer->identidad == PEER )
         local_peer->send_seeder_bitfield();
      else if ( external_peer->identidad == SUPER_PEER )
         local_peer->send_leecher_bitfield( external_peer );
      break;
   case PEER :
      local_peer->send_leecher_bitfield( external_peer );
      break;
}

function send_seeder_bitfield() {
   //Envía un bitfield vacío, lo cual le indicará al external_peer
   //que el local_peer es un seeder, por lo que éste nunca le enviara
   //un mensaje de tipo HAVE.
   //Al recibir un bitfield vacío un peer puede considerar que quien
   //envía el mensaje siempre tendrá todos los chunks existentes.
}

function send_leecher_bitfield( external_peer ) {
   //Envía un bitfield a partir de external_peer->abi y de largo Y.
   //Cada vez que el local_peer reciba un nuevo chunk, si este es
   //mayor al external_peer->abi se enviara un mensaje de HAVE a
   //este.
}
```

2.2.8. Configuración de los Peers

Todos los *peers* (*broadcaster-peers*, *super-peers* y *normal peers*) cuentan con un conjunto de parámetros que permiten al usuario configurar distintos aspectos del cliente. Algunos de estos parámetros no afectan el comportamiento del protocolo, como la IP y el puerto donde el cliente atiende, mientras que otros parámetros sí. En el marco de nuestro proyecto, es este último conjunto de parámetros el que nos interesa evaluar de qué forma afectan al protocolo.

A continuación se describe cuáles son esos parámetros:

piece-length (bytes). Este parámetro permite configurar el tamaño en *bytes* del *chunk*. A mayor valor de éste parámetro, más segundos de reproducción se tienen por *chunk*, pero también se necesita más transferencia de datos entre los *peers* por *chunk*. A medida que la línea de tiempo de reproducción avanza, el reproductor de GoalBit consume *chunks*. Si se tiene disponible el siguiente *chunk* a reproducir, la reproducción continúa de forma normal, en caso contrario, el *chunk* se pierde, afectando a la calidad de la reproducción. Entonces, *chunks* más grandes brindan mayor tiempo de reproducción, bajando la tasa de pedidos de *chunks* del reproductor, pero debido a que se necesita transferir más *bytes* para un *chunk* completo, aumenta la probabilidad de no tener disponible los *chunks* a tiempo. En cambio, *chunks* más pequeños brindan menos tiempo de reproducción, aumentando la tasa de pedidos de *chunks* del reproductor, pero reducen la cantidad de bytes a transferir, lo que aumenta la probabilidad de tener los *chunks* disponibles a tiempo.

slice-length (bytes). Este parámetro permite configurar el tamaño máximo en *bytes* de los mensajes de tipo *piece* que son enviados a la red. Los *peers* no envían los *chunks* enteros a la red, sino que se envían de a partes, denominadas *slices*. Un valor más pequeño de este parámetro implica que para completar la transferencia de un *chunk* se deben enviar más mensajes, aumentando la probabilidad de pérdida de paquetes, pero disminuyendo la cantidad de información perdida en caso de pérdida de un paquete. Al contrario, un valor más grande de este parámetro implica menos mensajes para completar la trasferencia de un *chunk* (con el mínimo de un mensaje por *chunk*, cuando el *slice-length* es mayor o igual al *piece-length*), disminuyendo la probabilidad de pérdida de paquetes, pero aumentando la cantidad de información perdida en caso de una pérdida.

buffer-size (*chunks*). Este parámetro permite configurar la cantidad de *chunks* consecutivos que se deben tener para comenzar la reproducción. Valores mayores de este parámetro implican mayor transferencia de bytes para completar el *buffering* inicial, pero permite tener más tiempo de reproducción disponible inicialmente, lo que hace disponer de más tiempo para conseguir los *chunks* que le siguen a este *buffer*, y así ayuda a evitar pérdidas de *chunks* y futuros *rebufferings*. Valores menores de este parámetro implican menor cantidad de bytes para completar el *buffering* inicial, pero disminuye el tiempo de reproducción disponible inicialmente, y por ende menor tiempo para conseguir los *chunks* que le siguen, lo que puede causar pérdidas de *chunks* y *rebufferings* más frecuentes.

urgent-range-size (chunks). Este parámetro permite configurar la cantidad de *chunks* consecutivos que es deseable tener a partir de la línea de reproducción. Cuando faltan *chunks* en este rango el cliente prioriza los pedidos de éstos *chunks* para poder obtenerlos más rápido. Valores menores de este parámetro ayudan a evitar pérdidas de *chunks*, pero un valor demasiado pequeño puede restringir el rango de *chunks* solicitados. En cambio, valores grandes no previenen tanto la pérdida de *chunks*, pero permite una mayor diversidad de *chunks* solicitados.

exponential-beta. Este parámetro permite configurar la prioridad de petición de paquetes entre los más cercanos a la línea de reproducción y los más raros. Priorizar los *chunks* más cercanos a la línea de reproducción ayuda a prevenir la pérdida de *chunks*. Pero si se prioriza demasiado a estos *chunks*, todos los *peers* se van a concentrar en tener los mismos *chunks* (los cercanos a la línea de reproducción) y

ningún peer va a tener *chunks* raros (más lejanos a la línea de reproducción). Cuando la línea de reproducción avance, los peers no van a lograr conseguir los chunks ahora cercanos porque ningún peer se preocupó de conseguirlos antes.

2.3 NAT

Esta sección ha sido desarrollada a partir de [22], usando la terminología y taxonomía de NAT definida en [23], así como términos adicionales de [8]. En esta sección se explica como funciona el NAT y las principales configuraciones existentes.

Un *session endpoint* TCP o UDP es un par (IP, puerto), y una sesión es identificada por sus dos *session endpoints*. Para cada *host* involucrado, una sesión es identificada por una cuádrupla (IP local, Puerto local, IP remota, Puerto remoto). La IP de una sesión es normalmente la del flujo de paquetes que inician la sesión, el paquete SYN inicial para TCP o el primer datagrama para UDP.

De los varios tipos de NAT, el más común es el tradicional o *outbound* NAT, que provee un puente asimétrico entre una red privada y una pública. *Outbound* NAT por defecto permite solo a sesiones salientes pasar el NAT, paquetes entrantes son descartados a menos que el NAT los identifique como parte de una sesión iniciada desde la red privada. *Outbound* NAT tiene conflictos con los protocolos P2P porque cuando los dos *peers* que desean comunicarse se encuentran detrás de dos NATs diferentes, indiferente de quien trate de iniciar la sesión, el otro NAT la rechazará. Lo esencial del NAT *traversal* para crear sesiones P2P es que parezcan como sesiones salientes para los dos NATs.

Outbound NAT tiene dos variedades:

- *Basic NAT:* solo traduce la IP
- NAPT[24]: traduce los *endpoints* de sesiones completas. Se ha convertido en el más común porque permite a un *host* en una red privada compartir el uso de una sola IP pública.

NAT establece una traducción entre IP y puertos locales y públicos. Pero como se menciona en [25] las implementaciones de NAT difieren mucho.

Según [26] y [27] hay cuatro tipos principales de NAT: *Full Cone* (FC), *(Address) Restricted Cone* (AR), *Port (Address) Restricted Cone* (PAR) y *Symmetric* (SYM). A continuación se describen los mismos.

2.3.1. Uno a uno (*Full Cone*)

Es también conocido como *one-to-one* NAT. Una vez que una IP y puerto interno son mapeados a una IP y puerto externo respectivamente, todos los paquetes con la IP y puerto interno serán traducidos a la IP y puerto externo fijo. Aún más, cualquier *host* externo puede enviar paquetes al *host* interno enviando los paquetes a la IP mapeada externa. Se ilustra en la Figura 2.7.

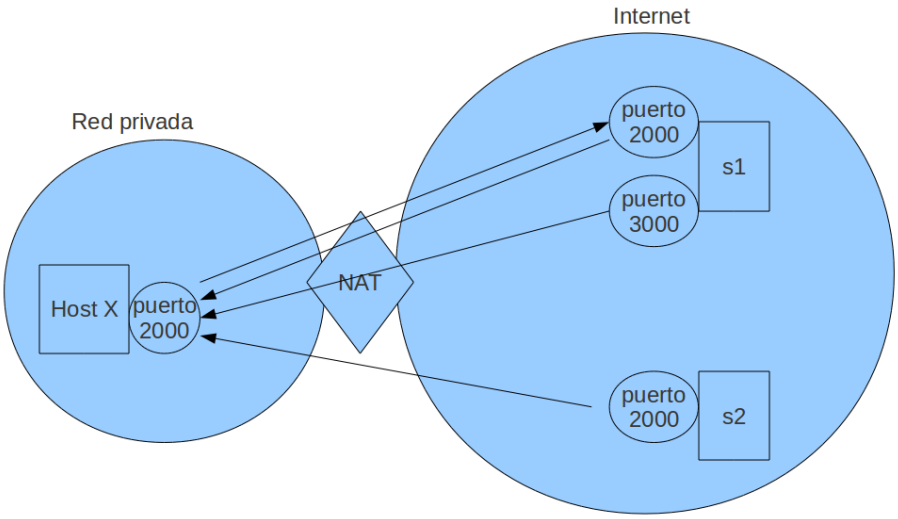

Figura 2.7: Full Cone NAT

2.3.2. Cono restringido por dirección ((Address) *Restricted Cone*)

Todos los pedidos desde una IP y puerto interna son mapeados a una IP y puerto externa fija. Es similar a FC excepto que a diferencia de FC, un *host* externo **s2** puede mandar paquetes a un *host* interno sólo si el *host* interno ha mandado paquetes previamente a la IP de **s2** a través del AR. Se ilustra en la Figura 2.8.

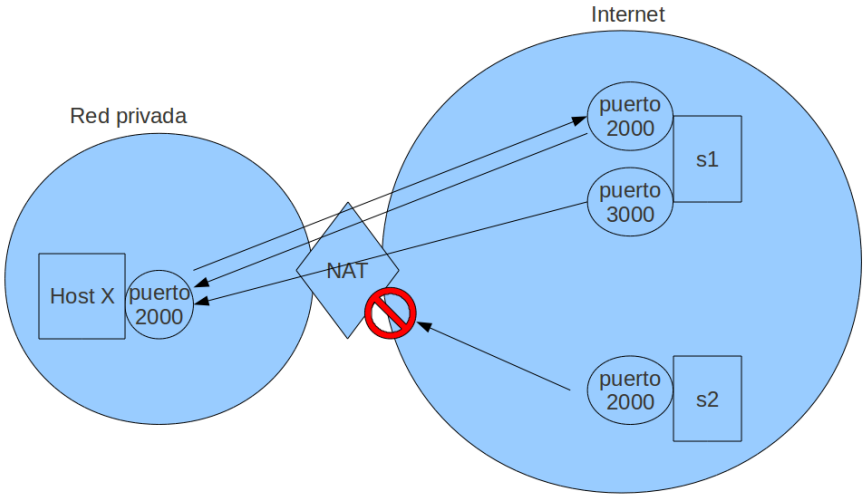

Figura 2.8: Restricted Cone NAT

2.3.3. Cono restringido a dirección y puerto (*Port (Address) Restricted Cone*)

Es similar al AR. Sin embargo, también toma los puertos en cuenta junto con la IP. Un *host* externo **s2** puede enviar un paquete con su IP y puerto origen, a un *host* interno solo si el *host* interno ha enviado previamente un paquete a dicho *host* en ese puerto. Se ilustra en la Figura 2.9.

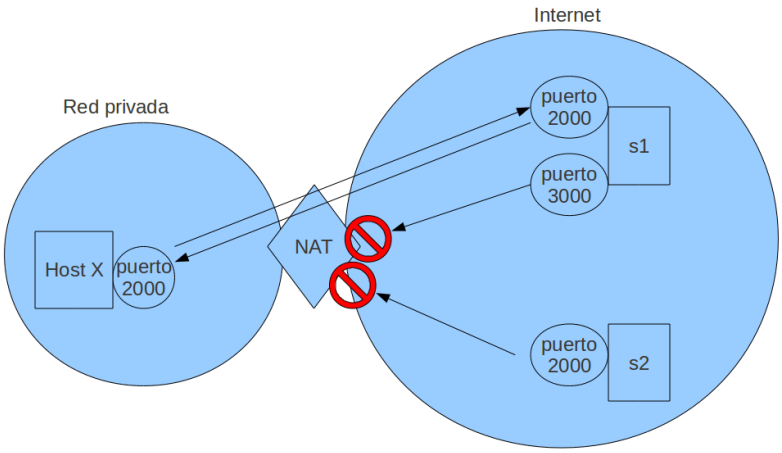

Figura 2.9: Port Restricted Cone NAT

2.3.4. Simétrico (*Symmetric*)

Cualquier pedido de una IP y puerto interna a una IP y puerto externa es mapeado a una única pareja de IP y puerto externo. Si el mismo *host* envía un paquete desde la misma IP y puerto pero a un destino distinto, un mapeo distinto es utilizado. Solo el *host* externo que recibe el paquete desde un *host* interno puede mandar un paquete de regreso al *host* interno. Se ilustra en la Figura 2.10.

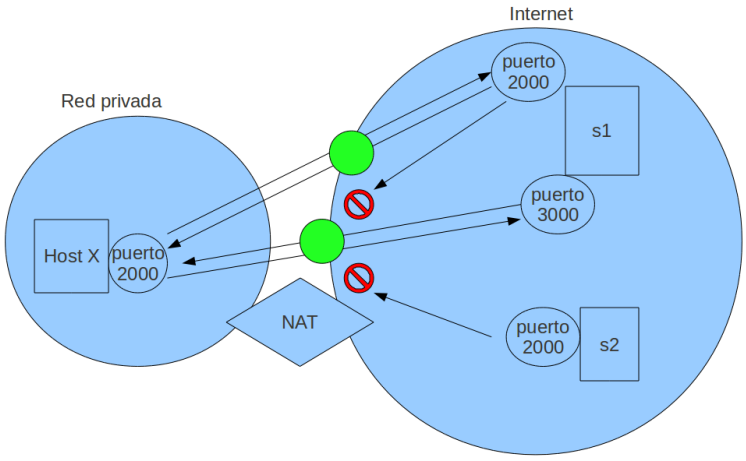

Figura 2.10: Symmetric NAT

2.4 Conceptos utilizados en comunicaciones P2P

2.4.1. Horquilla (Hairpin)

Según [28] es cuando un paquete da un "giro en U" dentro de un NAT y vuelve al mismo lado del que vino. El NAT detecta que el destino (IP, puerto) del paquete se encuentra en su tabla de NAT, traduce la

dirección y puerto de destino y redirige el paquete a la red privada de donde vino.

La capacidad de *Hairpinning* es importante para los programas P2P porque hay casos en que dos *hosts* en un mismo lado del NAT sólo se pueden comunicar usando sesiones que realizan *hairpin* a través del NAT. Los paquetes pueden ser tanto TCP como UDP.

2.4.2. Retransmisión (Relaying)

Según [22], el método más confiable, pero menos eficiente, para comunicación P2P a través de NAT es simplemente hacer que la comunicación vea a la red como una comunicación común cliente/servidor, a través de retransmisión. El protocolo TURN (*Traversal Using Relay* NAT) [29] define un método de implementar retransmisión en una forma relativamente segura, TURN se verá en la sección 2.7.5.

2.5 Técnicas que permiten el NAT Traversal

2.5.1. Perforación (Hole Punching, HP)

Es una técnica de redes de computación utilizada para establecer comunicación entre dos partes en organizaciones separadas que se encuentran ambas detrás de un NAT. Se envía un paquete con IP y puerto destino externo a la red, al salir este paquete de la red y pasar por el NAT se genera un “agujero” por el cual dependiendo del tipo de NAT se pueden aceptar conexiones entrantes de *hosts* externos a la red para el *host* origen en esa IP y puerto externo.

UDP

Según [22] UDP HP permite a dos *hosts* establecer una sesión P2P UDP con la ayuda de un servidor de rendezvous bien conocido, incluso para el caso donde los dos *peers* están detrás de NATs. Varios protocolos propietarios, como los de juegos *on-line* utilizan UDP HP.

Rendezvous Server

Hole punching asume que los dos clientes, **A** y **B**, ya poseen sesión activas UDP con un servidor de rendezvous **S**. Es a través de este que conocen la información necesaria para iniciar un *hole punching* con el objetivo de generar una conexión entre ellos. Cuando un cliente se registra con **S**, el servidor registra dos extremos para ese cliente: el par (dirección IP, puerto UDP) que el cliente cree estar utilizando para comunicarse con **S**, y el par (dirección IP, puerto UDP) que **S** observa como origen de los paquetes con los que se comunica con el. Nos referimos al primer par como el extremo privado y al segundo como el extremo público del cliente. El servidor puede obtener el extremo privado del cliente mismo en un campo dentro del mensaje de registro del cliente, y el extremo publico de la dirección IP y puerto UDP de origen en el encabezado UDP del mensaje de registro. Si el cliente no se encuentra detrás de un NAT, entonces su extremo público y privado serán iguales.

Algunos NATs poseen el comportamiento de registrar el cuerpo de los datagramas UDP buscando campos de 4-byte que se parecen a direcciones IP, y traducirlas como hacen con las que se encuentran en el encabezado. Para ser robusto frente a este comportamiento, las aplicaciones pueden desear ocultar

las direcciones IP en los cuerpos de los mensajes un poco, por ejemplo transmitiendo el complemento a unos de la dirección IP en vez de la dirección IP. Por supuesto que si la aplicación encripta sus mensajes, entonces este comportamiento del NAT muy probablemente no sea un problema.

TCP

Según [22], establecer una conexión P2P TCP entre dos *hosts* que se encuentran detrás de NATs es un poco más complejo que para UDP, pero es similar a nivel de protocolo. Cuando el NAT involucrado lo soporta es rápido y confiable como UDP HP. Incluso pueden ser más robustas que las comunicaciones UDP, debido que al contrario de UDP, la máquina de estados del protocolo TCP le da a los NATs en el camino una manera *standard* para determinar el tiempo de vida de manera precisa para una sesión TCP.

Sockets y Reuso de puertos TCP

Según [22], el desafío práctico más grande para las aplicaciones que desean implementar TCP HP no está en el protocolo sino en la interfaz (API). Debido a que la API de los *sockets Berkeley standard* fue diseñada para el modelo cliente/servidor, esta permite a un TCP *stream socket* ser el iniciador de una conexión saliente vía *connect()*, o escuchar por conexiones entrantes vía *listen()* y *accept()*, pero no ambas cosas a la vez. Los *sockets* TCP usualmente tienen una correspondencia uno a uno con números de puertos TCP en el localhost: después de que la aplicación ata un *socket* a un número de puerto local TCP particular, intentos de atar otro *socket* al mismo puerto TCP fallarán.

Para que el TCP HP funcione, necesitamos utilizar un solo puerto TCP local para escuchar por conexiones TCP entrantes y para iniciar múltiples conexiones TCP salientes de manera concurrente. Afortunadamente, la mayoría de los SOs soportan una opción especial para el *socket* TCP, comúnmente llamada `SO_REUSEADDR`, que permite a la aplicación atar varios *sockets* al mismo *endpoint* local siempre y cuando esta opción se utilice en todos los *sockets* involucrados. Sistemas BSD introdujeron una opción `SO_REUSEPORT`, que controla el re-uso de puertos de manera separada del re-uso de IPs: en dichos sistemas las dos opciones deben utilizarse.

Abriendo Streams TCP P2P

En [22], se supone que el *host* **A** desea iniciar una conexión TCP con el *host* **B**. Se asume que **A** y **B** tiene conexiones TCP activas con un servidor de Rendezvous **S**. El servidor guarda los *endpoints* públicos y privados de los clientes registrados, así como hace con UDP. A nivel del protocolo, TCP HP funciona muy similar al caso UDP:

1. El *host* **A** utiliza su sesión TCP activa con **S** para pedirle ayuda para conectarse con **B**.
2. **S** responde a **A** con los *endpoints* TCP público y privado de **B**, y al mismo tiempo envía a **B** los *endpoints* TCP público y privado de **A**.
3. Desde el mismo puerto TCP local que **A** y **B** utilizaron para registrarse con **S**, **A** y B asincronicamente hacen intentos de conexión saliente con los *endpoints* público y privado que les reporto **S**, mientras que simultáneamente escuchan por pedidos de conexión entrante en sus respectivos puertos TCP.

4. **A** y **B** esperan por un intento de conexión saliente que triunfe, y/o por un intento de conexión entrante que aparezca. Si uno de los intentos de conexión saliente falla debido a un error en la red como *"connection reset"* o *"host unreachable"*, el *host* simplemente re-intenta después de un tiempo pequeño, hasta un período máximo definido por la aplicación.

5. Cuando una conexión TCP se logra, los *hosts* se autentican cada uno para verificar que se conectaron con el *host* interesado. Si la autenticación falla, el *host* cierra la conexión y sigue esperando que otras tengan éxito. Los *host* utilizan la primera sesión TCP autenticada con éxito resultante de este proceso.

A diferencia de UDP, donde cada *host* solo necesita un *socket* para comunicarse con **S** y cualquier número de *peers* simultáneamente, con TCP cada *host* debe manejar varios *sockets* atados a puertos TCP locales en ese *host*, como muestra la Figura 2.11. Cada *host* necesita un *stream socket* representando su conexión con **S**, un *socket* para escuchar por pedidos de conexión de los *peers*, y al menos dos *stream sockets* adicionales con los cuales iniciar conexiones salientes a los otros *peers endpoints* públicos y privados.

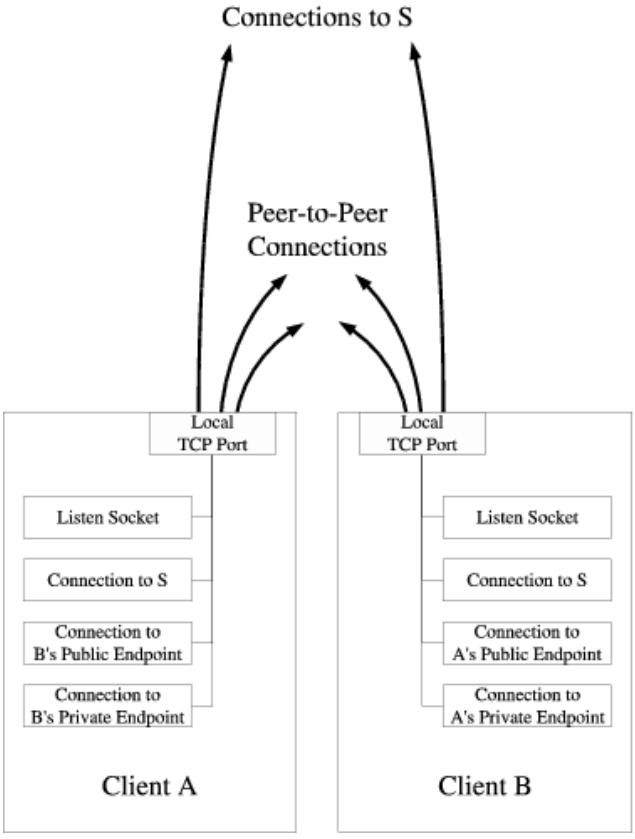

Figura 2.11: Sockets versus Puertos para TCP HP

Consideremos el escenario común donde los clientes **A** y **B** se encuentran detrás de diferentes NATs,

como muestra la Figura 2.11, y asumamos que los números de puerto mostrados en esa figura son para TCP en vez de UDP. Los intentos de conexión que **A** y **B** hacen al *endpoint* privado del otro fallan o se conectan con el *host* equivocado. Así como con UDP, debemos tener una manera de que la aplicación autentifique sus sesiones P2P, debido a la posibilidad de conectarse erróneamente con un *host* de la red privada que posee la misma IP privada que el *host* que en la red remota privada al que nos deseamos conectar.

Los intentos de los clientes por generar conexiones salientes al *endpoint* público del otro, generan en los respectivos NATs que se abran huecos permitiendo establecer una comunicación TCP directa entre **A** y **B**. Si los NATs se comportan bien, entonces un *stream* TCP P2P se forma automáticamente entre ellos. Si el primer paquete SYN de **A** hacia **B** llega al NAT de **B** antes de que el primer paquete SYN de **B** hacia **A** llegue al NAT de **B**, por ejemplo, entonces el NAT de **B** puede interpretar el SYN de **A** como un intento de conexión no solicitado y descartarlo. El primer paquete SYN de **B** hacia **A** debe subsecuentemente pasar debido a que el NAT de **A** ve este SYN como parte de la sesión saliente hacia **B** que el SYN de **A** inicio previamente.

Comportamiento observado por la aplicación

Según [22], lo que la aplicación cliente observa que sucede durante el TCP HP depende del *timing* y la implementación TCP involucrada. Suponiendo que el primer paquete SYN de **A** hacia el *endpoint* público de **B** es descartado por el **NAT B**, pero el subsecuente primer paquete SYN de **B** al *endpoint* público de **A** llega a **A** antes de que este retransmita su SYN. Dependiendo del SO involucrado, una de dos cosas puede suceder:

- La implementación TCP de **A** se da cuenta de que el *endpoint* de la sesión del SYN entrante corresponde con el de una sesión saliente de **A** que se estaba tratando de iniciar. El *stack* TCP de **A** por lo tanto asocia esta nueva sesión con el *socket* que la aplicación local en **A** estaba utilizando para *connect*`()` con el *endpoint* público de B. La llamada asincronica a *connect*`()` de la aplicación tiene éxito, y no pasa nada con los *sockets* que escuchan. Ya que el paquete SYN recibido no incluía un ACK para el previo SYN de **A** saliente, el TCP de **A** responde al *endpoint* público de **B** con un paquete SYN-ACK, la parte SYN simplemente como una respuesta al SYN saliente original de **A**, utilizando el mismo número de secuencia. Una vez que el TCP de **B** recibe el SYN-ACK de **A**, responde con su propio ACK para el SYN de **A**, y la sesión TCP entra en estado de conexión para los dos clientes.

- Alternativamente, la implementación de TCP de **A** puede darse cuenta que **A** tiene un *socket* activo escuchando en ese puerto esperando por una conexión. Ya que el SYN de **B** se parece a un intento de conexión entrante, el TCP de **A** crea un nuevo *stream socket* con el que asociar la nueva sesión TCP, y le entrega este nuevo *socket* a la aplicación a través de la siguiente llamada a *accept()*. El TCP de **A** responde a **B** con un SYN-ACK como arriba, y la conexión TCP continua de manera usual para el modelo cliente/servidor.

  Ya que el intento de *connect()* de **A** a **B** usó una combinación de *endpoints* destino y origen que está siendo ahora usada por otro *socket*, la que acaba de regresar a la aplicación a través de

accept(), el intento asíncrono de *connect()* de **A** debe fallar en algún punto, típicamente con un error "IP en uso". La aplicación tiene sin embargo el *stream socket* P2P que necesita para comunicarse con B funcionando, así que ignora el error.

El primer comportamiento es habitual para los SOs basados en BSD, mientras que el segundo comportamiento es más común en SOs Linux o Windows.

Apertura simultánea de TCP

En [22], se supone que el *timing* de varios intentos de conexión durante el proceso de HP se da de tal manera que los paquetes SYN de ambos clientes atraviesen sus respectivos NATs, abriendo nuevas sesiones TCP en cada NAT antes de llegar al otro NAT. En este caso de "suerte", los NATs no rechazan ninguno de los paquetes SYN iniciales, y los SYN se cruzan en el cable entre los dos NATs. En este caso los *hosts* observan un evento llamado *apertura simultánea de TCP:* cada peer TCP recibe un SYN mientras espera un SYN-ACK. El TCP de cada *host* responde con un SYN-ACK, cuya parte SYN repite el SYN enviado previamente por el *peer*, y su parte ACK reconoce el SYN recibido del otro *peer.*

Lo que la aplicación observa en este caso depende del comportamiento de la implementación TCP involucrada, como se describió previamente. Si ambos clientes implementan el segundo comportamiento, puede ser que todas las llamadas c*onnect()* asincronicas hechas por la aplicación últimamente fallen, pero la aplicación corriendo en cada *host* sin embargo recibe un *stream socket* TCP P2P funcionando a través del *accept()* como si el *stream* TCP se hubiera creado mágicamente por si mismo y fuese aceptado pasivamente por los *endpoints*. Siempre y cuando a la aplicación no le importe si recibe el *socket* TCP P2P de un *accept()* o de un *connect()*, el resultado del proceso es un *stream* funcional TCP en cualquier implementación que implemente la máquina de estados TCP *standard* especificada en RFC 793.

Agujereo Secuencial

Según [22], una forma para perforar TCP se da cuando los clientes intentan conectarse mutuamente de manera secuencial en vez de en paralelo. Por ejemplo:

1. **A** informa a **B** a través de **S** de sus deseos de comunicarse, sin simultáneamente escuchar en su puerto local;
2. **B** hace un intento de *connect()* con **A**, el cual abre un hueco en el **NAT B** pero falla debido a un *timeout* o RTS desde el NAT de **A** o desde el mismo **A**;
3. **B** cierra su conexión con **S** y hace un *listen()* en su puerto local;
4. **S** cierra en respuesta su conexión con **A**, indicándole a **A** que intente un *connect()* directamente con **B**.

El procedimiento secuencial podría ser útil con SOs Windows previos a XP SP2, los cuales no implementaban todavía apertura simultanea de TCP, o en APIs de *sockets* que no soportan *SO_REUSEADDR*. El procedimiento secuencial es más dependiente del *timing*, y puede ser más lento en el caso común y menos robusto en casos inusuales. En el paso (2), por ejemplo, **B** debe permitir que su intento de *connect() (*condenado a fallar) este activo lo suficiente como para que al menos uno de los paquetes SYN atraviesen todos los NAT en su lado de la red. Un tiempo de vida muy corto arriesga un

SYN perdido que descarrile el proceso, mientras que un tiempo de vida muy largo incrementa el tiempo total requerido para el procedimiento de HP. El procedimiento de HP secuencial también "consume" efectivamente ambas conexiones con el servidor **S**, requiriendo la apertura de conexiones nuevas con **S** para cada nueva conexión P2P que se quiera generar. El procedimiento de HP paralelo, en contraste, típicamente culmina tan pronto como los dos host hagan sus intentos de *connect()*, y permite a cada *host* retener y reusar una sola conexión con **S** por un tiempo indefinido.

2.5.2. Predicción de puerto (Port Prediction)

Es una técnica utilizada para predecir el puerto que se le asignará a la próxima conexión en un NAT basándose en el puerto asignado en la conexión previa. Se utiliza principalmente en los SYM, en los cuales la asignación de puerto puede ser secuencial (conexión **n** puerto **p**, conexión **n+1** puerto **p+x**) ó de modo aleatorio. En el caso aleatorio disminuye mucho su efectividad.

2.6 Propiedades de los NATs amigables

Según [22], no todos los NATs actuales cuentan con propiedades amigables, pero muchos lo hacen, y los NATs se están volviendo cada vez más *"P2P-Friendly"* ya que los fabricantes están reconociendo la demanda de protocolos P2P para voz sobre IP y juegos en red.

2.6.1. Traducción de *Endpoint* Consistente

Según [22], las técnicas que se describieron aquí de HP solo funcionan automáticamente si el NAT mapea de manera consistente un *endpoint* TCP o UDP en la red privada a un sólo *endpoint* público correspondiente controlado por el NAT. Un NAT que se comporta de esta manera es el NAT de cono.

Otro tipo de NAT que cuenta con traducción de *endpoint* consistente es el NAT simétrico cuando realiza una asignación de números de puerto. Ya que los NATs simétricos no proveen mayor seguridad que los NAT de cono con el filtrado de tráfico por sesión, los NATs simétricos se están volviendo menos comunes ya que los fabricantes están adaptando sus algoritmos para soportar protocolos P2P.

2.6.2. Manejando solicitudes de conexión TCP no solicitadas

Según [22], cuando un NAT recibe un paquete SYN en su lado público para lo que parece ser un intento no solicitado de conexión entrante, es importante que el NAT descarte el paquete SYN de manera silenciosa. Algunos NATs rechazan activamente dichas conexiones entrantes devolviendo un paquete TCP RST o un reporte de error ICMP [30], que interfiere con el proceso de TCP HP. Este comportamiento no es necesariamente fatal, siempre y cuando los programas reintenten los intentos de conexión saliente como está especificado en el paso 4 del proceso descrito previamente (sección 2.5.1), pero puede hacer que el proceso de HP tarde más tiempo.

2.6.3. Dejar la carga útil intacta

Según [22], [31] y [32] algunos NATs son conocidos por escanear ciegamente la carga útil del paquete buscando valores de 4-bytes que se parezcan a IPs, y traducirlas como lo harían con IPs que se

encuentran en el cabezal del paquete, sin saber nada del protocolo de aplicación en uso. Este mal comportamiento afortunadamente parece ser poco común y los programas pueden protegerse a si mismos fácilmente ofuscando las IPs que mandan en los mensajes, por ej. mandando el complemento binario de la IP deseada.

2.6.4. Traducción de *Hairpin*

Según [22], algunas situaciones de NAT multinivel requieren traducción de *Hairpin* para que pueda funcionar el HP TCP o UDP como se describió previamente en el escenario multinivel de la Figura 3.2, por ejemplo, depende de que NAT C soporte traducción de *Hairpin*. Desafortunadamente los NATs que soportan la traducción de *Hairpin* son raros, pero afortunadamente los escenarios de red en los que se requiere dicha funcionalidad también son raros. Los escenarios en los cuales se presentan múltiples niveles de NAT se están volviendo más comunes debido al agotamiento del espacio de IPs v4, por lo tanto es importante el soporte de traducción de *Hairpin* en las próximas implementaciones de NAT.

2.7 Estándares que facilitan el NAT-traversal

2.7.1. Establecimiento de conexión interactivo (*Interactive Connectivity Establishment*, ICE)

Según [33] , ICE es una extensión de dos protocolos, STUN y TURN. STUN le permite a un *host* saber su IP global y el puerto UDP asignado por su NAT más lejano. Esta IP puede ser transmitida luego por SIP (bajo los procedimientos de ICE) para permitir conectividad UDP entre dos *hosts.*

ICE se utiliza tanto para descubrir y transmitir una lista de posibles medios de comunicación (IPv6[34], IPv4 pública, IPv4 privadas, STUN, TURN), y para probar estos modos. SIP puede servir para esta función de establecimiento de conectividad más amplia, en parte, porque SIP utiliza URI (por ejemplo, usuario@dominio) para identificar y descubrir *hosts,* lo que permite que mensajes de señalización SIP pasen a través de NAT y recojan la información necesaria (tipo de NAT, IPs públicas y privadas, puertos, etc.).

RFC 3264 [35] define un intercambio de mensajes *Session Description Protocol* (SDP) (ver RFC 4566 [14]) en dos fases con el propósito de establecer sesiones multimedia. Este mecanismo de ofrecer/responder es utilizado por protocolos como *Session Initiation Protocol* (SIP) [13]. Protocolos que utilizan ofrecer/responder son difíciles de operar a través de NATs. Debido a que su propósito es estableces un flujo de paquetes, ellos tienden a portar la dirección de IP y el puerto del origen del medio dentro de sus mensajes, lo cual es una situación conocida por causar problemas con NAT [31].

Los protocolos también buscan generar un flujo de datos directo entre los participantes de manera que no haya una capa de aplicación intermediaria entre ellos. Esto es hecho para reducir la latencia, perdida de paquetes, y reducir los costos operacionales de realizar el *deploy* de la aplicación. Sin embargo, esto es difícil de lograr a través de NAT como ya hemos establecido desde el comienzo del documento.

Numerosas soluciones fueron definidas para permitir a estos protocolos operar a través de NAT. Estas incluyen *Application Layer Gateways* (ALGs), el *Middlebox Control Protocol* [36], la especificación original del *Simple Traversal of UDP Through NAT* (STUN) [8], y *Realm Specific IP* [37] [38] junto con las extensiones de descriptores de sesiones necesarias para hacerlas funcionar, como ser el atributo *Session Description Protocol* (SDP) [14] para el *Real Time Control Protocol* (RTCP) [39]. Lamentablemente estas técnicas tienen pros y contras que hacen de ellas optimas para ciertas topologías pero una mala elección en otras. Lo que se necesita es una sola solución que sea suficientemente flexible para trabajar bien en todas las situaciones.

La técnica de *Interactive Connectivity Establishment* (ICE) es una técnica para NAT traversal para *streams* basados en UDP (siendo cierto también que ICE puede extenderse para soportar otros protocolos de trasporte como ser TCP [ICE-TCP]) que se establecen a través del modelo de oferta/respuesta, y funciona incluyendo múltiples direcciones IP y puertos en ofertas y respuestas SDP, las cuales son probadas con pruebas de conectividad peer-to-peer. Las direcciones IP y los puertos incluidos en el SDP y las pruebas de conectividad son realizados con el ya visto STUN, ICE también hace uso de *Traversal Using Relays around NAT* (TURN) [29], una extensión de STUN. Como ICE intercambia múltiples direcciones de IP y puertos para cada *stream* de datos, también permite la selección de la dirección para cada *multihomed* y *dual-stack hosts*, y por esta razón no cumple con RFC 4091 [40] y [41].

Visión general de ICE

En un *deployment* típico de ICE tenemos dos *endpoints* (conocidos como AGENTS en la terminología de RFC 3264) que desean comunicarse. Ellos pueden comunicarse indirectamente mediante algún protocolo de señalamiento (como SIP), a través del cual ellos pueden efectuar un intercambio de oferta/respuesta de mensajes SDP [35]. No es responsabilidad de ICE el intercambio del SDP, este se asume que es provisto por otro mecanismo. Al principio del proceso de ICE los agentes son ignorantes de sus propias topologías. En particular ellos pueden estar detrás de un NAT (o múltiples capas de NATs). ICE permite a los agentes descubrir suficiente información acerca de sus topologías como para encontrar potencialmente uno o más caminos a través de los cuales puedan comunicarse.

La Figura 2.12 muestra un entorno de despliegue típico de ICE. Los dos endpoints son L y R. Ambos están detrás de sus respectivos NATs pero pueden no estar al tanto de ello. Las propiedades y tipo de NATs son también desconocidas. Los Agentes L y R son capaces de realizar un intercambio oferta/respuesta en el cual pueden intercambiar mensajes SDP, cuyo propósito es iniciar una sesión de comunicación entre L y R. Típicamente este intercambio ocurrirá a través de un servidor SIP.

Ademas de los agentes, un servidor SIP y NATs, ICE se utiliza con servidores STUN o TURN en la red. Cada agente tiene su propio servidor STUN o TURN, o pueden ser el mismo.

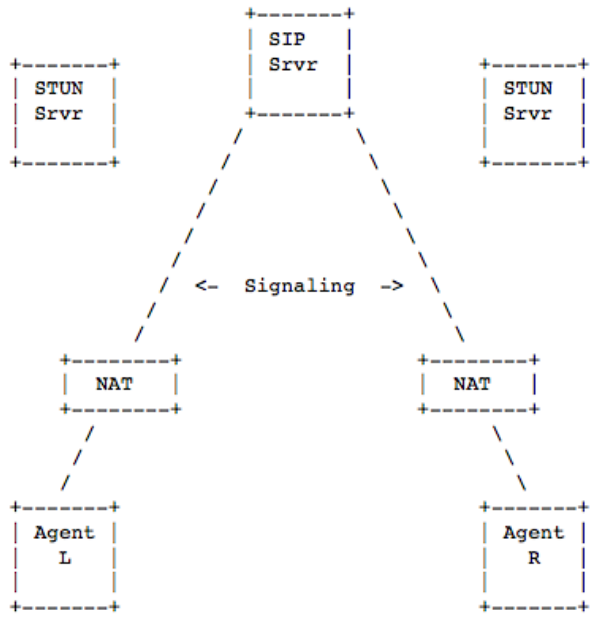

Figura 2.12: Escenario de Deployment de ICE

La idea básica detrás de ICE es la siguiente: cada agente tiene una variedad de candidatos TRANSPORT ADDRESSES (combinaciones de direcciones IP y puerto para un protocolo de transporte particular, el cual es siempre UDP para esta especificación) que puede utilizar para comunicarse con el otro agente. Esta puede incluir:

- Un dirección de transporte en una interface de red directamente conectada.
- Una dirección de transporte traducida en el lado publico de un NAT (dirección *"server reflexive"*)
- Una dirección de transporte asignada desde un servidor TURN (una "dirección retransmitida").

Potencialmente cualquiera de las direcciones de transporte candidatas de L pueden ser utilizada para comunicarse con cualquiera de las direcciones de transporte candidatas de R. En la práctica sin embargo, muchas combinaciones no funcionarán. Por ejemplo, si L y R se encuentran detrás de NATs, será improbable que puedan comunicarse directamente.

El propósito de ICE es descubrir cual par de direcciones funcionará. La manera en que ICE hace esto es probando sistemáticamente cada par probable (en un orden cuidadoso) hasta que encuentra uno o más que funcionen.

Recopilación de direcciones candidatas

Para ejecutar ICE, un agente tiene que identificar todas sus direcciones candidatas. Un CANDIDATO es una dirección de transporte (IP y puerto para un protocolo de transporte, siendo unicamente UDP especificado aquí).

Se definen tres tipos de candidatos, algunos derivados de interfaces de red lógicas o físicas, otras detectables vía STUN o TURN. Naturalmente, un candidato disponible es el obtenido directamente de la interfaz local. Dichos candidatos son llamados HOST CANDIDATE. La interfaz local puede ser *ethernet* o *WiFi*, o puede ser obtenida a través de un mecanismo de *tunneling* como una Virtual *PrivateNetwork* (VPN) o una *Mobile IP* (MIP). En todos los casos, la interfaz de red que aparece para el agente como una interfaz local y sus puertos (y por lo tanto candidatos) se pueden asignar.

Si un agente posee varias IP, se obtiene un candidato por cada dirección de IP. Dependiendo de la ubicación del PEER (el otro agente en la sesión) en la red IP en relación con el agente, el agente puede ser alcanzable por el peer a través de una o más de esas direcciones IP.

Luego, el agente utiliza STUN o TURN para obtener candidatos adicionales. Estos vienen en dos sabores: direcciones traducidas en el lado público de un NAT (SERVER REFLEXIVE CANDIDATES) y direcciones en servidores TURN (RELAYED CANDIDATES). Cuando los servidores TURN son utilizados, ambos tipos de candidatos son obtenidos del servidor TURN. Si sólo servidores STUN son utilizados, sólo candidatos *server reflexive* son obtenidos de él. La relación de estos candidatos con el candidato *host* se muestra en la Figura 2.13. En esta figura, ambos tipos de candidatos son descubiertos utilizando TURN. En la figura la notación **X:x** significa dirección de IP **X** y puerto UDP **x**.

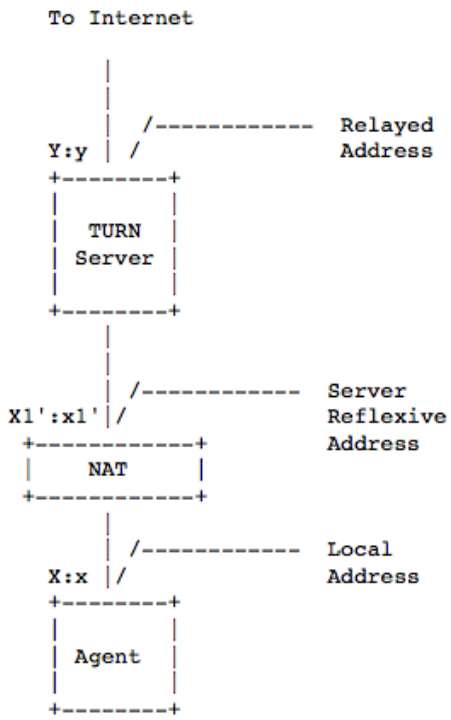

Figura 2.13: Relación de Candidatos

Cuando un agente envía el pedido TURN *Allocate* desde la dirección de IP y puerto **X:x**, el NAT (asumiendo que hay uno) creara un *binding* **X1:x1**, mapeando este *server reflexive candidate* al *host*

X:x. Los paquetes salientes enviados desde el *host* candidato serán traducidos por el NAT al candidato *reflexive* del *server.* Paquetes que arriban enviados al *server reflexive candidate* serán traducidos por el NAT al candidato del *host* y reenviados al agente. Llamamos al candidato del *host* asociado con un candidato *reflexive* del servidor la BASE. “Base” se refiere a la dirección a la que un agente envía para un candidato en particular. Por lo tanto, como un caso degenerado, los candidatos *host* también tienen base, pero es igual al candidato *host.*

Cuando hay muchos NATs entre cada agente y el servidor TURN, el pedido al TURN creara un *binding* con cada NAT, pero solo el más “lejano” candidato *reflexive* del *server* (el más cercano al *server* TURN) será descubierto por el agente. Si el agente no se encuentra detrás de un NAT, entonces el candidato base será el mismo que el candidato *reflexive* del *server* y el candidato *reflexive* del *server* se vuelve redundante y será eliminado.

El pedido *Allocate* llega al servidor TURN. El servidor TURN asigna un puerto desde su dirección IP local **Y**, y genera una respuesta *Allocate*, informado al agente de este candidato *relayed.* El servidor TURN también informa al agente del candidato *reflexive,* **X1':x1'** copiando la dirección origen del transporte del pedido *Allocate* en la respuesta *Allocate.* El servidor TURN actúa como un *relay* de paquetes, *forwarding* el tráfico entre **L** y **R**. En orden de enviar tráfico a **L**, **R** envía tráfico al *server* TURN a **Y:y** y el servidor TURN hace el *forward* a **X1:x1** que pasa por el NAT donde es traducido a **X:x** y entregado a **L**.

Cuando sólo servidores STUN son utilizados, el agente envía un pedido de STUN *Binding* [42] a su STUN *server.* El servidor STUN informará al agente del candidato *reflexive* del *server* **X1:x1** copiando la dirección origen del transporte en la respuesta del *Binding.*

Chequeo de Conectividad

Una vez que **L** ha obtenido todos los candidatos, los ordena de prioridad más alta a más baja y los envía a **R** sobre el canal de *signaling.*
Los candidatos son enviados en atributos en la oferta del SDP. Cuando **R** recibe la oferta, realiza el mismo proceso para obtener sus candidatos y responde con su propia lista. Al final de este proceso, cada agente tiene una lista completa de sus candidatos y los del *peer.* Arma pares con ellos, y cada agente planifica una serie de pruebas. Cada prueba es una transacción STUN pedido/respuesta que el cliente efectuará en un par en particular enviando un pedido STUN desde el candidato local al candidato remoto.

El principio básico del *checking* de conectividad es simple:

1. Ordenar los pares de candidatos en orden de prioridad.
2. Enviar *checks* a cada par de candidatos en orden de prioridad.
3. Reconocer *checks* recibidos desde el otro agente

Con ambos agentes realizando un *check* en un par de candidatos, el resultado es un *handsheke* de 4 vías:

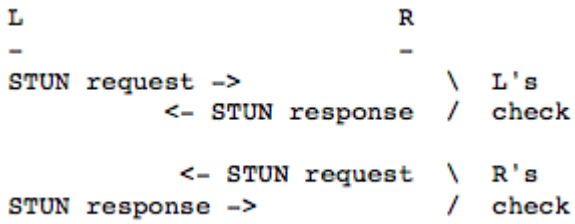

Figura 2.14: Prueba básica de conectividad

Es importante notar que los pedidos al STUN son enviados desde y hasta exactamente la misma dirección IP y puerto que serán utilizados por contenido (ej.: RTP y RTCP). Por consiguiente, los agentes demultiplexan STUN y RTP/RTCP utilizando el contenido de los paquetes, en lugar de el puerto a través del que arribaron. Por fortuna, este demultiplexado es fácil de hacer, especialmente para RTP y RTCP.

Debido a que la respuesta al pedido de STUN *Binding* es utilizada para *check* de conectividad, la respuesta del STUN *Binding* contendrá la dirección de transporte traducida del agente en el lado publico de cualquier NAT entre el agente y su *peer.* Si esta dirección de transporte es diferente de la de otros candidatos que el agente ya aprendió, representa un nuevo candidato, llamado PEER REFLEXIVE CANDIDATE, el cual es probado por ICE de la misma manera que cualquier otro candidato.

Como una optimización, en tanto **R** recibe el mensaje de *check* de **L**, **R** comienza un mensaje de *check* de conectividad a ser enviado a **L** en el mismo par de candidatos. Esto acelera el proceso de encontrar un candidato valido, y es llamado TRIGGERED CHECK.

Al final del *handshake*, **L** y **R** saben que pueden enviar (y recibir) mensajes punto a punto en ambas direcciones.

2.7.2. Enchufar y usar universal (*Universal Plug and Play*, UPnP)

Según [1], UPnP es una arquitectura y estándar abierto para conexión inteligente tanto de dispositivos cableados como inalámbricos. El descubrimiento automático de servicios, direccionamiento y facilidad de configuración lo hacen una buena opción para pequeñas redes domésticas o de oficina (*Small Office/Home Office*, SOHO). UPnP es promovido por Microsoft. Se le dice universal porque es independiente del SO y lenguaje de programación. La especificación está basada en TCP/IP. Cuando un nuevo *host* necesita una conexión, el dispositivo UPnP puede configurar automáticamente la IP de red, anunciar su presencia en una subred, y permitir el intercambio de descripción de dispositivos y servicios.

2.7.3. Utilidades para atravesar sesiones en NAT (*Session Traversal Utilities for NAT*, STUN)

STUN [42] es un protocolo que sirve como herramienta para otros protocolos en atacar el problema de atravesar NATs. Puede ser usado por un *host* para determinar que IP y puerto le asignó el NAT. También puede ser usado para probar la conectividad entre dos *hosts*, y como un protocolo de *keep-alive* para mantener los mapeos del NAT. STUN funciona con muchos NATs existentes, y no requiere ningún comportamiento especial de los mismos.

STUN no es una solución de NAT traversal por si misma. Pero si es una herramienta para ser usada en el contexto de una solución de NAT traversal. Este es un cambio importante con respecto a la especificación del RFC 3489, que presentaba a STUN como una solución completa.

El protocolo definido en la especificación, provee una herramienta para manejarse con los NATs. Provee una forma de que el *host* determine que dirección y puerto públicos asignó el NAT a sus correspondientes IP y puerto privados. También provee una forma de que un *host* mantenga vivo un mapeo del NAT. Con algunas extensiones se puede usar el protocolo para hacer pruebas de conectividad (MMUSIC-ICE), o para hacer *relay* de los paquetes entre dos *hosts* (BEHAVE-TURN). ICE es un uso de STUN. SIP *Outbound* es otro uso de STUN.

Cuando se definió por primera vez STUN (STUN clásico), en el RFC 3489, STUN era una solución completa al NAT traversal. En esa solución, un cliente descubría si estaba detrás de un NAT, determinaba su tipo de NAT, descubría su IP y puerto público en el NAT más externo, y utilizaba esa IP y puerto en los cuerpos de los protocolos, como el SIP. Sin embargo, la experiencia desde la publicación del RFC 3489 ha mostrado que esto no funciona lo suficientemente bien en la práctica. La IP y puerto descubiertos mediante STUN a veces se pueden usar para comunicarse con un peer y a veces no pueden. El STUN clásico no tenía manera de descubrir si iba a funcionar o no, y no proveía ningún remedio en los casos que no funcionaba. Aún más, el algoritmo del STUN clásico para clasificar a los tipos de NATs tenía fallas ya que muchos NATs no eran claramente clasificables dentro de los tipos definidos en el RFC 3489.

Para descubrir la presencia de NATs se mandan STUN *requests* y se recibe STUN *responses*. STUN es un protocolo dentro del modelo cliente/servidor. Se mandan *binding requests* a través de UDP o *shared secret requests* sobre TLS en TCP. Al principio de la sesión de comunicación, mediante un secreto compartido desde el *host* se pide un usuario y *password* temporal al servidor para el chequeo y autenticación de las siguientes conexiones. El servidor generalmente tiene IP pública. El pedido UDP puede atravesar varios dispositivos NAT hasta llegar al servidor STUN.

El servidor puede descubrir la última IP y puerto que fue modificado por un NAT. El servidor le devuelve al cliente esa IP y puerto. Comparando su IP y puerto local con lo que le responde el servidor, el cliente puede saber si está detrás de un NAT o no. Mandando dos paquetes desde la misma IP y puerto a dos servidores STUN distintos se puede descubrir si se está detrás de un NAT simétrico o no.

STUN no funciona con NATs simétricos ya que la conexión depende de la IP y puerto, que va variando según el servidor con el que se conecte el cliente. Para solucionar esto el IETF propone el uso de TURN. La ventaja de STUN es que no requiere cambios en los dispositivos NAT. Pero al no funcionar para NATs simétricos se pierde gran parte del mercado corporativo. STUN es lento y poco popular ya que requiere cambios a nivel de aplicación y un servidor con IP pública [1].

2.7.4. Utilidades para atravesar sesiones en NAT TCP (*Session Traversal Utilities for NAT TCP*, STUNT)

STUNT [43] es un protocolo que extiende STUN para incluir TCP. STUN es el protocolo que un *host* usa para determinar qué dirección global y puerto UDP le ha asignado el NAT más externo, y para determinar qué tipo de NAT es.

STUNT no se ha especificado en detalle todavía, pero su uso TCP es similar al que tiene STUN para UDP. La idea básica es que el *host* establezca múltiples conexiones TCP (o UDP) con un servidor público (el STUNT *server*). El STUNT *server* registra la dirección global y el puerto global asignados por el NAT, y le pasa esa información al *host*. Usando SIP, el *host* le pasa al *host* remoto la dirección y puerto global esperados (y toda otra dirección y puerto con el que se pueda comunicar), y a su vez recibe las direcciones y puertos del *host* remoto. Luego ambos *hosts* envían un paquete saliente con destino la dirección y puerto remoto. Al atravesar el NAT estos paquetes generan el mapeo en el NAT, permitiendo paquetes entrantes desde el *host* remoto. A esto se le llama agujerear el NAT. A continuación pueden fluir paquetes en ambos sentidos.

2.7.5. Atravesamiento usando un retransmisor (*Traversal Using Relay NAT*, TURN)

Según [43], TURN permite a un *host* seleccionar un *TCP-relay* que sea globalmente direccionable, que puede utilizarse subsecuentemente de puente para una conexión TCP entre dos *hosts* atrás de NATs. A diferencia de STUN, TURN no permite conexiones directas entre dos *hosts* detrás de NATs.

2.7.6. Compuerta de capa de aplicación (*Application Level Gateway*, ALG)

Según [1], NAT generalmente sólo manipula IPs y puertos en los datagramas. No se toma en cuenta las IPs que están embebidas en la carga del datagrama. Esto genera que los paquetes tengan dificultad en atravesar ciertos dispositivos NAT. ALG se ubica en los dispositivos NAT/firewall para modificar la carga de manera transparente y de este modo trabaja en conjunto con el NAT para ofrecer enrutamiento transparente de los paquetes. ALG generalmente requiere que se remplace o modifique el dispositivo de NAT/firewall y su configuración, lo que hace difícil su implantación.

2.7.7. Teredo

Según [27], Teredo es un método para NAT Traversal propuesto por Microsoft. Está basado en la tecnología de *tunneling* de IPv6. Un cliente Teredo obtiene una IPv6 Teredo del servidor Teredo. Utiliza una IPv6 con *tunneling* IPv6 en UDP/IPv4. Un cliente Teredo se comunica con otros clientes Teredo y otros nodos IPv6 a través de un *relay* Teredo. Un servidor Teredo debe tener una IPv4 pública y una IPv6 pública. Un Teredo *relay* también. El Teredo *relay* provee enrutamiento entre los clientes

Teredo y nodos en la Internet IPv6. Teredo no funciona bien con NATs simétricos.

2.7.8. Alianza de red viva digital (*Digital Living Network Alliance*, DLNA)

Según [44], DLNA es una organización responsable de definir guías de interoperabilidad para permitir compartir archivos digitales entre distintos aparatos de consumo como computadoras, impresoras, cámaras, teléfonos móviles y otros aparatos multimedia. Estas guías se basan en estándares públicos pero las guías son privadas (disponibles con costo).

2.7.9. Optimización del tráfico en capa de aplicación (*Application-Layer Traffic Optimization*, ALTO)

Según [45], ALTO es un servicio diseñado y especificado por un grupo de trabajo del IETF para dar a los programas información que permita seleccionar a los *peers* de un modo mejor que el aleatorio. ALTO puede tomar distintos enfoques en el balanceo de factores como máximo ancho de banda, mínimo tráfico a través de dominios, menor costo al usuario, etc. El grupo va a considerar las necesidades de BitTorrent, redes P2P sin tracker, y otros programas, como redes de distribución de contenido (CDN) y selección de *mirrors*.

3 Proyectos similares

En este capítulo se recopilan varios proyectos que han hecho pruebas de concepto sobre el NAT traversal. Los proyectos analizados utilizan técnicas basadas en los protocolos del capítulo anterior pero se enfocan en obtener resultados prácticos. A partir de este capítulo se tiene una visión general de las dificultades y ventajas que tienen los distintos métodos para atravesar NATs.

Según [1], para el diseño e implementación de programas y protocolos P2P se debe:

- Evitar usar IPs y puertos TU en la carga de los paquetes. Esto es porque NAT no los modifica y de este modo se filtran IP privadas a Internet que no serán alcanzables por otros *hosts*. En caso de ser necesario se podría usar ALG pero esto agrega complejidad.
- No usar ALG ya que hace que los dispositivos aumenten de costo y que decrezca su popularidad.
- Tener una sola conexión. Configurar los temporizadores de NAPT NAT TU para permitir el máximo uso de las sesiones de conexión y para reciclar los puertos TU.
- Usar un método híbrido de P2P para los *hosts* que tengan IP pública y cliente-servidor para los que no tengan.

Los proyectos similares estudiados fueron los siguientes:

- Implementando atravesamiento de NAT en BitTorrent (Implementing NAT Traversal on BitTorrent)
- NUTSS: Un método basado en SIP para la conectividad de redes TCP y UDP (NUTSS: A SIP-based Aproach to UDP and TCP Network Connectivity)
- Un método nuevo para atravesar NATs simétricos en UDP y TCP (A New Method for Symmetric NAT Traversal in UDP and TCP)
- NATBLASTER
- Conexiones TCP para aplicaciones P2P (TCP Connections for P2P Apps)
- Atravesamiento TCP en NATs y Firewalls (TCP Traversal a través de NATs y Firewalls)

3.1 Implementando atravesamiento de NAT en BitTorrent (Implementing NAT Traversal on BitTorrent)

Si un sólo cliente está detrás de NAT la conexión es posible. Si los dos están detrás de NAT es necesario configurar los NATs para que la conexión sea posible. *Relaying* es el más confiable pero menos eficiente (requiere más procesamiento, más ancho de banda y tiene más latencia). *Reversal* involucra al servidor sólo al inicio pero requiere un cliente con IP pública. HP es simple pero depende del comportamiento de NATs, los cuales varían mucho; y además está resuelto para UDP, pero hay problemas para TCP [46].

3.1.1. Predicción de puerto (*Port Prediction*)

1) Cuando el cliente se conecta al servidor, el servidor debe determinar qué tipo de NAT usa el cliente.
2) De acuerdo al tipo de NAT del cliente, el servidor debe decirles a los otros clientes qué puerto

va a usar el cliente cuando intenten establecer una conexión con él.

La parte 2) es complicada cuando hay muchos clientes en la red y todavía no se ha encontrado una solución robusta.

3.1.2. Implementación

En el trabajo original [46], modifica BitTorrent para que se use un puerto fijo para hacer pedidos y escuchar. De este modo si el NAT del cliente mapea las IP privada a una sola IP pública para cualquier destino del paquete; y al llegar un paquete exterior sólo filtra si la IP y puerto destino del paquete exterior no coincide con una IP y puerto que tenga mapeado el NAT, se puede lograr la conexión.

La implementación sigue los siguientes pasos:

1) **A** se conecta con **S**
2) **B** se conecta con **S**
3) **S** le dice a **A** que se conecte con **B**
4) **A** intenta conectarse con **B** y falla
5) **S** le dice a **B** que se conecte con **A**
6) **B** intenta conectarse con **A** y lo logra

3.2 NUTSS: Un método basado en SIP para la conectividad de redes TCP y UDP (NUTSS: A SIP-based Aproach to UDP and TCP Network Connectivity)

El nombre NUTSS [43] esta dado por sus componentes arquitectónicos: NAT, URI, *Tunnel*, SIP, y STUNT. NAT extiende el espacio de IPs, la URI es el esquema de nombrado punto a punto, el componente *Tunnel* se refiere a la necesidad de encapsular algunos protocolos de bajo nivel en UDP, específicamente *Mobile* IP, IPsec, e incluso IPv6. SIP es el protocolo utilizado para negociar comunicaciones a nivel de red. NUTSS utiliza SIP incluyendo ICE, pero también incluye mecanismos para establecer la conexión TCP/UDP. STUNT es un protocolo que extiende STUN para incluir TCP.

3.2.1. La solución

La solución al problema de generar conexiones cuando los dos *hosts* se encuentran detrás de NATs vista desde el punto de vista de los *hosts*. Se genera una conexión TCP que parece ser una conexión normal de 3 vías con los dos *hosts* creyendo ser los iniciadores de la misma. Este enfoque minimiza las probabilidades de que el NAT impida la conexión por no parecer estándar.

Por simplicidad, en la Figura 3.1 solo se muestran los NATs **N** y **M** que son los más lejanos a los *hosts* en su conexión a la red pública de Internet.

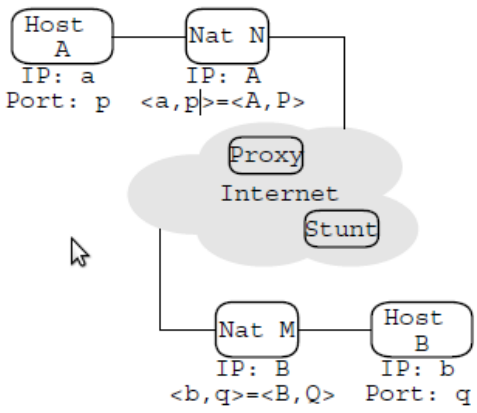

Figura 3.1: Arquitectura de NUTSS

Tenemos STUNT *server*(s) y un *proxy* que puede ser una computadora que utilizan **A** y **B** para establecer conexión, o podría representar una infraestructura compuesta por SIP *proxys* a través de la cual **A** y **B** puedan intercambiar mensajes.

La manera en que iniciamos la conexión TCP que describimos es simétrica para **A** y **B**. La describimos desde el punto de vista de **A** como se muestra en la Figura 3.2. **A** y **B** utilizan el estándar TCP *Stack* y API que viene en el SO. Notemos que las lineas continuas indican mensajes TCP enviados por el SO, las lineas punteadas representan mensajes enviados por los *hosts* y sus respectivos STUNT *Servers*, o entre ellos mismos a través de uno o más *proxys*.

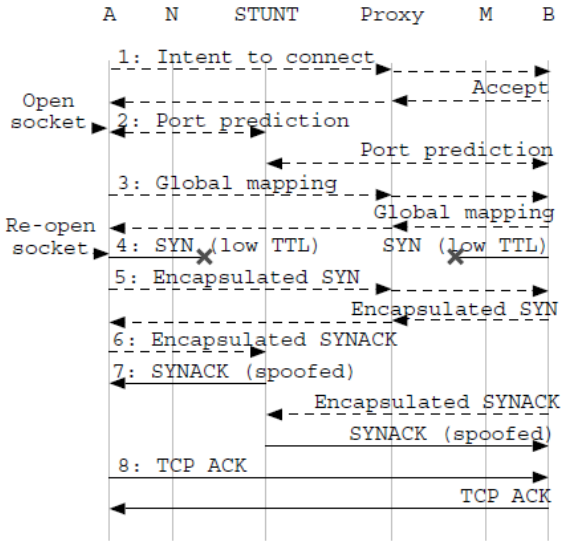

Figure 3.2 Inicio de conexión TCP

#	Source	Dest.	Contents
1	over Proxy		Intent to establish connection
2	STUNT port prediction		
3	over Proxy		A's global mapping A:P
4	a:p (A:P)[1]	B:Q	SYN, low TTL, Seq# SA
5	over Proxy		Encapsulated SYN, default TTL, source A:P
6	from A to STUNT		Encapsulated packet 7 (below)
7	B:Q (spoofed)	A:P (a:p)[1]	SYNACK, Seq#SB, ACK#SA+1, default TTL
8	a:p (A:P)[1]	B:Q (b:q)[2]	ACK, Seq#SA+1, Ack#SB+1, default TTL

Figura 3.3: Contenido de los paquetes

Al comienzo, **A** y **B** establecen el interés de conectarse con cada uno a través del *proxy*. Luego cada uno predice su GA y GP (*Global Address* y *Global Port*). Si alguno de los NAT es del tipo *Cone* entonces el *host* detrás de él puede predecir sus GA y GP antes del paso 1, para luego utilizarlo desde múltiples conexiones TCP. Cada *host* debe abrir un *socket* RAW para poder ver la primer copia de SYN que el SO envíe. El *host* **A** envía su GA y GP a B a través de uno de los *proxys* y asimismo recibe los datos de **B**. Luego de recibir los GA y GP de **B**, **A** inicia un saludo TCP con él. El SYN enviado como parte del acuerdo de 3 vías tiene que haber tenido un TTL lo suficientemente pequeño como para llegar entre **N** y **M**. La razón para esto es que el NAT M puede cerrar el hueco que creó **B** si ve un SYN inesperado que proviene de afuera.
El *host* **A** envía el contenido del TCP SYN (paquete 3) que escuchó vía el RAW *socket* en un mensaje a **B** (vía el *proxy*, paquete 5). De la misma forma, **A** recibe el contenido del paquete TCP SYN de **B** a través de un mensaje. El *host* **A** construye un mensaje conteniendo su SYN y el recibido de **B** y envía el mensaje a un STUNT *server*. De estos dos SYN el STUNT *server* puede crear el SYN ACK que **A** y el NAT **N** esperan ver (ej. conteniendo el número de secuencia que el *host* **B** generó , y el número de ACK que **B** hubiera producido si hubiera recibido el SYN de **A**). El SYN ACK es transmitido desde el STUNT *server* al NAT **N**, suplantando la IP de origen para que parezca que proviene del *host* **B** vía NAT M (esto puede hacerse con el RAW *socket*).
El NAT **N** encuentra el SYN ACK que espera ver y lo traduce al destino **A**. El paquete es recibido por el SO de **A** que reconoce el SYN ACK con el ACK y por lo tanto completa el acuerdo de 3 vías. El ACK final tiene el TTL por defecto y llega a **B**, terminando el acuerdo de 3 vías.

3.2.2. Conclusión

Este concepto presenta un *hack* que expande el espectro de comunicación a través de NAT traversal, la arquitectura no se encuentra finalizada, pero el objetivo real de la presentación de este concepto según los autores, es motivar a la experimentación para rever si ICE/STUN/TURN deben ser expandidos más allá del foco de transporte de medios para incluir todo tipo de comunicación de datos P2P incluyendo la

preparación para la transición a IPv6.

3.3 Un método nuevo para atravesar NATs simétricos en UDP y TCP (A New Method for Symmetric NAT Traversal in UDP and TCP)

En [27] se propone un nuevo método para UDP *multi-HP* con las siguientes características:

- establece una conexión UDP entre dos *hosts* detrás de NATs,
- está basado en *port prediction* y valores limitados de TTL,
- controla los números de puerto para permitir el NAT Traversal de NATs simétricos,
- funciona bien para otros tipos de NATs,
- puede ser extendido para NAT Traversal en TCP

3.3.1. Fase 1

Al cliente se le llamará *echo client* y al servidor *echo server* porque hay una serie de intercambios de paquetes entre ellos. Un *echo client* se comunica con dos servidores **S1** y **S2**. **S1** y **S2** registran la IP y puerto del *echo client*, y luego son transformadas por el NAT **A**. Los pasos de esta fase son:

1. el *echo client* se comunica con **S1**. Luego, **S1** analiza el número de puerto mapeado por el NAT **A**,
2. s1 le manda el número de puerto al *echo client*,
3. el *echo client* manda un paquete a **S2**. Incluye información obtenida sobre el número de puerto en el NAT **A** cuando el *echo client* se comunicó con **S1**. Luego, **S2** analiza el número de puerto del NAT **A** y lo registra. **S2** también registra el número de puerto en el NAT **A** cuando el *echo client* se comunicó con **S1** en el paso 1).

3.3.2. Fase 2

En esta fase el *echo server* se comunica con **S1** y **S2** de una manera similar a la fase 1.

1. El *echo server* se comunica con **S1**. Luego, **S1** analiza el número de puerto mapeado por NAT **B**
2. **S1** le manda el número de puerto al *echo server*
3. el *echo server* manda un paquete a **S2**. El paquete incluye el número de puerto del NAT **B** obtenido de la comunicación del *echo server* con **S1** en el paso 1. de 3.3.2. Luego **S2** analiza el número de puerto del NAT **B** y lo registra. Luego **S2** registra la información del número de puerto del NAT **B** obtenida cuando el *echo server* se comunicó con **S1** en el paso 1 de 3.3.2.

3.3.3. Fase 3

En esta fase el método realiza una predicción de puertos. Como fue descrito en 3.3.1, el NAT **A** mapea el número de puerto dos veces, una vez en el paso 1 y otra en el paso 3. Por ejemplo, si el NAT **A** usa el puerto **5361** en el paso 1 y el **5362** en el paso 3, podemos predecir que el *punching mode* del NAT **A** es incremental y que el puerto predecido es **5363**. El nuevo método puede predecir el *punching mode* como incremental, decremental, o modo *skip*. Luego comunica la IP global objetiva, y el *punching mode* al *echo client* y al *echo server.* El *echo client* y el *echo server* reciben la información e inician multi-HP para establecer comunicación entre ellos.

1. Basado en los dos tipos de información comunicadas en 3.3.1 y 3.3.2, podemos predecir un puerto adecuado para el HP. También podemos determinar el *punching mode*. **S2** manda información conteniendo el puerto predecido y el *punching mode* al *echo server.*
2. Basado en esa información el *echo server* manda un gran número de paquetes. Estos paquetes tienen un puerto de destino fijo y un TTL bajo. El *echo server* hace un *bind* del puerto. Los paquetes luego son enviados al *echo client*.
3. Usando los dos tipos de informaciones de 3.3.1 y 3.3.2 se puede predecir un puerto adecuado para el HP. **S2** envía la información del puerto predecido y del *punching mode* al *echo client* de manera análoga al paso 1.
4. Con la información del paso 3, el *echo client* manda muchos paquetes al *echo server.* Estos paquetes tienen un puerto fijo. El echo client realiza un *bind* del puerto. Después de haber mandado todos los paquetes el *echo client* pasa al modo receptor.

En el paso 4, el NAT **B** recibe muchos paquetes UDP del *echo client*. Si uno de los puertos origen del *echo client* coincide con el puerto destino mapeado por el NAT **B**, el NAT **B** traduce el paquete y lo manda al *echo server* correctamente. El *echo server* cierra todos los puertos que tenía abiertos excepto el que recibió el paquete correctamente.

1. El *echo server* responde al *echo client*. Se establece la conexión P2P entre el *echo client* y el *echo server.*

Usando este nuevo método, los paquetes enviados parecen pertenecer a una comunicación normal UDP por lo que tienen menos posibilidades de ser descartados por alguna configuración de seguridad en el NAT, se puede predecir exactamente el tipo d*e port translation* que usan los NAT, y al usarse mil puertos hay más chances de acertar si se está usando un *port translation* desconocido.

3.4 NATBLASTER

Según [47], NATBLASTER se utiliza para establecer conexiones TCP entre *hosts* detrás de NATs. Se realizan dos hipótesis validas acerca de la red para los NATs probados. Primero se asume que los *hosts* no ven los paquetes *ICMP TTL Exceeded* de la red externa. Estos paquetes si son recibidos por un peer o *buddy* (se usa la palabra peer y *buddy* solamente para distinguir entre los dos miembros de una conexión P2P, sería como decir peer A = peer y peer B = *buddy*) terminarían el intento de conexión TCP. Muchas de las soluciones dependen de la habilidad de iniciar una conexión TCP mandando un paquete SYN inicial con TTL demasiado bajo. Una vez que el paquete SYN es descartado en la ruta, paquetes ICMP TTL Exceeded son retornados al NAT. Los NATs usados para la implementación no reenviaban los *ICMP TTL Exceeded* a la red privada en respuesta a paquetes TCP. Aún cuando un NAT reenvía paquetes *ICMP TTL Exceeded* a la red privada, los *firewalls* pueden ser desplegados en el *host* para bloquear dichos paquetes. La segunda hipótesis es que el NAT no va a anular el mapeo creado si recibe un *ICMP TTL Exceeded.* Alternativamente, se podría dejar el valor de TTL con el valor por defecto, y confiar en que el NAT destino no generará paquetes TCP RST. En la práctica esta es una opción viable ya que muchos NATs no generan paquetes RST para ayudar a defenderse del *port scanning*.

3.4.1. Técnicas

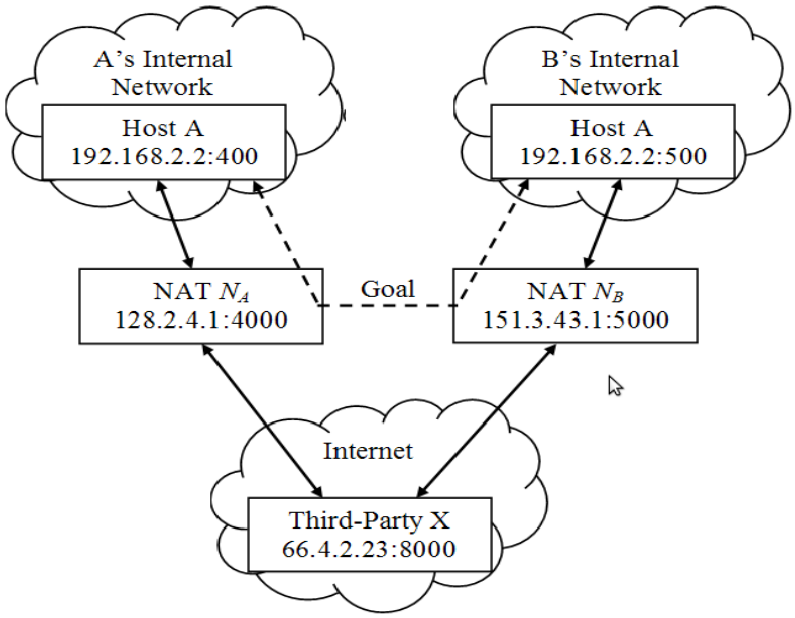

Figura 3.4: Situación a resolver por NATBLASTER

En la Figura 3.4 se muestra la situación planteada: el objetivo es permitir una conexión TCP entre **A** y **B**, que están detrás de las NATs **N_A** y **N_B**. Se desarrollaron varias técnicas para permitir conexiones TCP dependiendo de las propiedades de las NATs y las propiedades de la red. Si se considera esta información como el trío <asignación de puertos de **N_A**, asignación de puertos de **N_B**, *source routing* disponible>, se tienen los siguientes casos:

1. <predecible, predecible, LSR>
2. <predecible, predecible, no LSR>
3. <aleatorio, predecible, LSR>
4. <aleatorio, predecible, no LSR>
5. <aleatorio, aleatorio, LSR>
6. <aleatorio, aleatorio, no LSR>

El caso <aleatorio, predecible, X> es equivalente al caso <predecible, aleatorio, X>

3.4.2. Diagnóstico antes de la conexión

Loose Source Route (LSR) es una opción de IP que permite al creador de un paquete IP especificar una lista de IPs obligatorias a ser usadas en el *routing* del paquete. El resultado de esta opción es que cada IP indicada en la lista va a recibir el paquete en el orden indicado en la lista. Esta opción introduce un

riesgo de seguridad, ya que un atacante puede escuchar en una sesión si está en la ruta. Debido a este riesgo, muchos enrutadores descartan los paquetes que tengan esta opción activada.

Para determinar si LSR está disponible desde **A** a **B** pasando por **X**, **A** simplemente intenta conectarse con **B** haciendo LSR del paquete a través de **X**. Si **X** recibe el paquete, el LSR está disponible de **A** a **B** para el primer trayecto de **A** a **X**. Si **X** no recibe un paquete después de un determinado tiempo se puede asumir que LSR no está disponible. Como **X** sólo puede saber si LSR está disponible de **A** a **X**, debe obtener paquetes desde **B** mediante LSR. Si los obtiene entonces se puede hacer LSR entre **A** y **B** a través de **X**.

3.4.3. Implementación

En este paper se presentan métodos para los 6 casos pero los únicos que se implementan son los casos 2 y 4. Para ello se utiliza el lenguaje C en entorno Linux utilizando las librerías libnet y libpcap. El caso 2 seguramente abre conexiones mientras que el caso 4 las abre exitosamente con alta probabilidad (esta probabilidad está determinada por el número de SYNs y SYN+ACKs enviados). El caso 6 no se implementó porque es muy difícil adivinar el mapeo de puerto origen y puerto destino en el NAT de un *buddy.* Las posibilidades en el caso 4 eran 64.511 mientras que en el caso 6 son 64.511 * 64.511 = 4.161.669.121. Los casos 1, 3 y 5 no se implementaron porque generalmente LSR no está disponible en Internet y tendría pocas chances de tener éxito.

Se utilizó la implementación *standard* Berkeley, aumentándola con llamadas de sistema adicionales cuando fue necesario. Por ejemplo, cuando se mandaba un paquete SYN pero se necesitaba saber su número de secuencia, el paquete se manda con *connect*(), después de haber levantado un hilo que monitoree la interfaz por la que se va a mandar dicho paquete. Tanto en el caso 2 como el 4 los *peers* deben tener permisos de super usuario ya que lo requiere el uso de libpcap y libnet. El *helper* puede correr como usuario sin privilegios ya que no debe realizar ni *spoofing* ni *sniffing.*

3.5 Conexiones TCP para aplicaciones P2P (TCP Connections for P2P Apps)

El método NatTrav [48] se propone resolver el problema de los NATs mediante software. Los pares *"recipient"* esperan conexiones de los pares *"initiator"*. Para ello se registran con un *broker* intermediario, proveyendo una IP de red y una URI que identifica al par *recipient.*

Los *connection broker* facilitan la conexión a los *recipient* de dos maneras:

1. Proveyendo la IP de red actual del receptor
2. Facilitando el NAT Traversal si el receptor está detrás de un NAT

Los *connection broker* se replican para que la solución tenga mayor disponibilidad y escalabilidad.

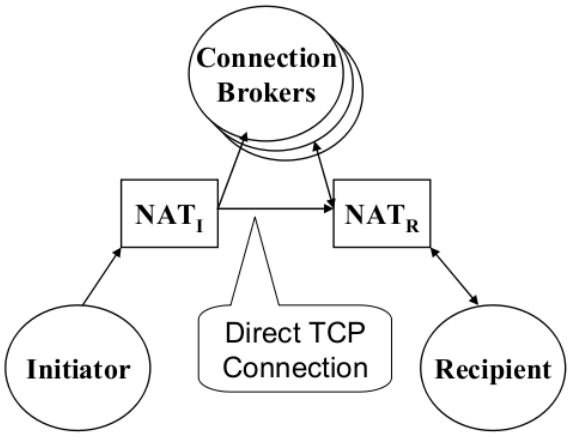

Figura 3.5: Arquitectura de NatTrav

En la Figura 3.6 se muestra el protocolo de NAT traversal para mapeos *cone-type*.

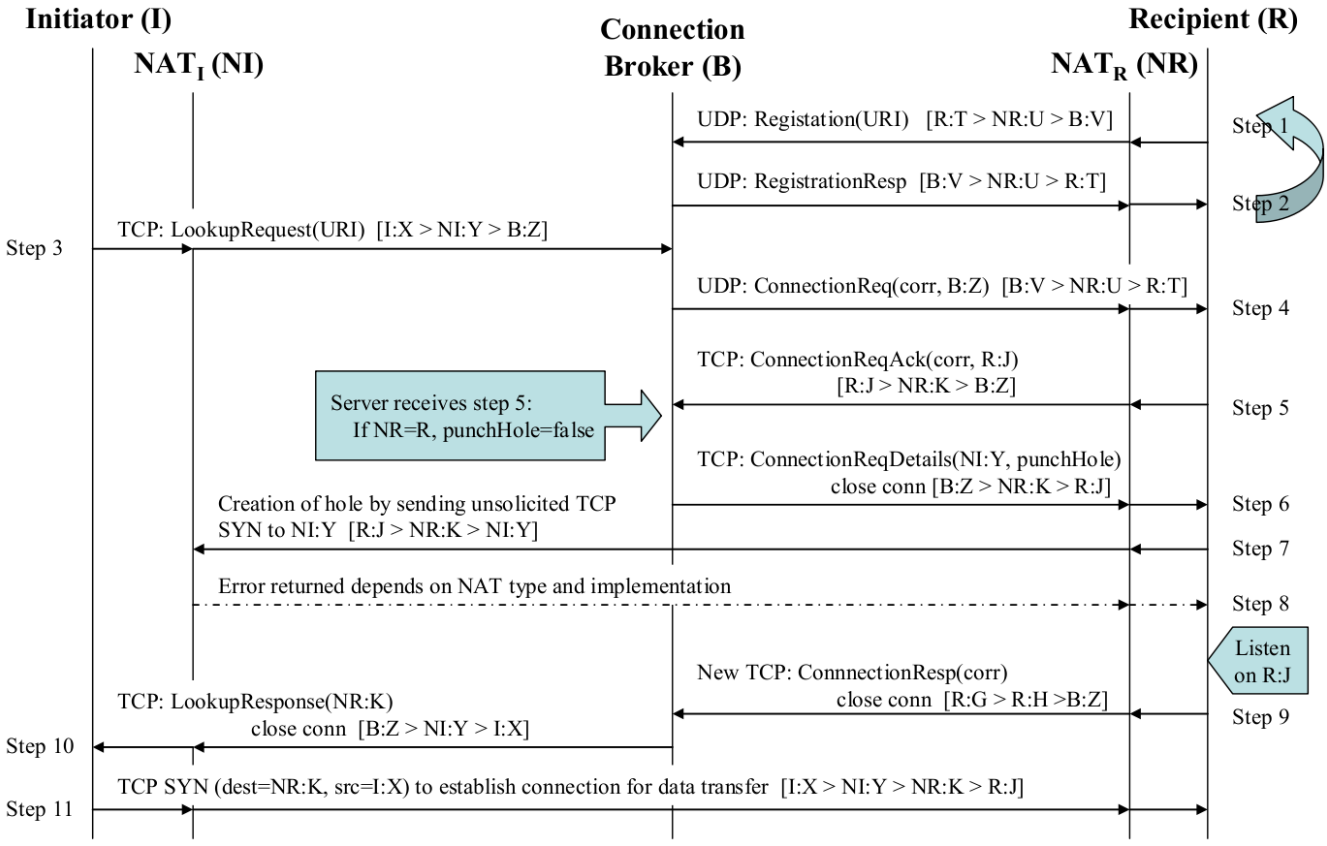

Figura 3.6: Protocolo NAT Traversal (Las IPs y puertos IP usadas para las comunicaciones se muestran en paréntesis rectos. Por ejemplo: [I:X > NI:Y > B:Z] significa comunicación desde el iniciador con IP I y puerto X con el broker de IP B y puerto Z a través del NAT I con IP pública NI y puerto público Y)

Se puede usar *port prediction* para extender la solución a mapeos simétricos, pero estudios recientes muestran que los NATs domésticos raramente usan mapeos simétricos. En estos estudios se muestra que de 42 versiones de NAT, 4 no siguen la especificación del IETF para NATs y 1 siempre usa mapeo simétrico. De los restantes 37, la mayoría usa mapeo *cone-type* a no ser que el puerto ya esté siendo usado por otra máquina atrás de el NAT, en ese caso se usa mapeo simétrico. En este último caso NatTrav falla pero cuando se reintenta, NatTrav usa un nuevo puerto y probablemente tiene éxito. Los teléfonos de Internet tampoco funcionan bien con NATs simétricos por lo que se cree que la industria se va a alejar de este tipo de NATs.

3.5.1. Registro del receptor

En los pasos 1 y 2, un receptor se registra con un *connection broker* proveyendo su URI. Se puede usar UDP o TCP para registrarse. Usando UDP (como se ve en la Figura 3.6) permite que un número grande de receptores se registren con cada *connection broker*, sin embargo los receptores deben mandar regularmente mensajes de registro para que el *connection broker* pueda mandar *connection requests* a los receptores a través de su NAT (paso 4). Usar TCP para el registro requeriría que los *connection brokers* mantengan un gran número de sesiones TCP abiertas.

3.5.2. Búsqueda

En el paso 3, cuando el iniciador quiere establecer una conexión TCP con el receptor, manda un *Lookup Request* al *connection broker.* El *connection broker* contesta (paso 10) con una IP y puerto público que el iniciador puede usar (paso 11) para establecer una conexión TCP directa desde *initiator* al receptor (a través de los NATs) como se muestra en la Figura 3.6.

El *connection broker* chequea si hay una conexión activa para el *recipient*. Para saber esto el *connection broker* puede que tenga que contactar a otros *connection brokers.* Si esto sucede, manda un *Connection Request* al receptor en la IP UDP y puerto que el receptor (o su *proxy)* usó para registrarse (paso 4). El *Connection Request* contiene un *correlator* (corr) para saber en qué *Lookup Request* van a estar trabajando los siguientes pasos y la IP y puerto del *connection broker* que el *recipient* debe usar en los siguientes pasos.

3.5.3. Haciendo el agujero (*Punching the Hole*)

En los pasos 5 y 6 el receptor usa TCP para intercambiar la información de IP de red especificada en el *Connection Request* con el *connection broker.* El *connection broker* compara la IP privada del *recipient* (**R**) con la IP pública del *recipient* (**NR**). Si son iguales, el *recipient* no está detrás de un NAT. Si son distintas, el receptor está detrás de un NAT y se le comunica que haga un agujero en el NAT para los mensajes que vengan desde la IP y puerto público del *initiator* (**NI:Y**).

El truco es usar la misma IP y puerto IP para las comunicaciones TCP con el *connection broker* y para la conexión TCP directa entre el *initiator* y el *recipient.* Como se está agujereando NATs *cone-type* sabemos que los NATs van a usar el mismo puerto *proxy* para la comunicación con el *broker* y con los *peers.* En el ejemplo de la Figura 3.6, el *initiator* usa el puerto **X** cuando se comunica con el *broker*

(paso 3), pero el *broker* ve que viene desde el puerto **Y** de NAT **I**. Análogamente, el receptor usa el puerto **J** (paso 5) y el *broker* ve el puerto **K** del NAT **R**. Para agujerear, el receptor cierra la conexión TCP con el *broker* usada en los pasos 5 y 6, y en el paso 7 usa el puerto **J** para intentar abrir una conexión TCP con **NI:Y**. Dependiendo del tipo e implementación del NAT **I**, el paso 8 puede dar *timeout*, TCP *reset response*, o (si NAT **I** es *full-cone*) error de puerto en uso (porque NAT **I** reenvió el TCP SYN a **I:X**). En cualquier caso, la conexión TCP no será exitosa pero el intento permite al NAT **R** reenviar comunicaciones desde **NI:Y**, mandadas a **NR:K**, a **R:J**.

Entre los pasos 8 y 9, se levanta un *Server-Socket* para escuchar en **R:J**. Luego se manda un *Connection Response* al *broker* (paso 9). En el paso 10, el *broker* le comunica al *initiator* que puede establecer una conexión TCP directa con el receptor (a través de los NATs) usando **NR:K**. En el paso 11, la conexión TCP ha sido establecida.

3.6 Atravesamiento TCP en NATs y Firewalls (TCP Traversal a través de NATs y Firewalls)

En [49] se da una caracterización y estadística de TCP Traversal a través de NATs y *Firewalls.*

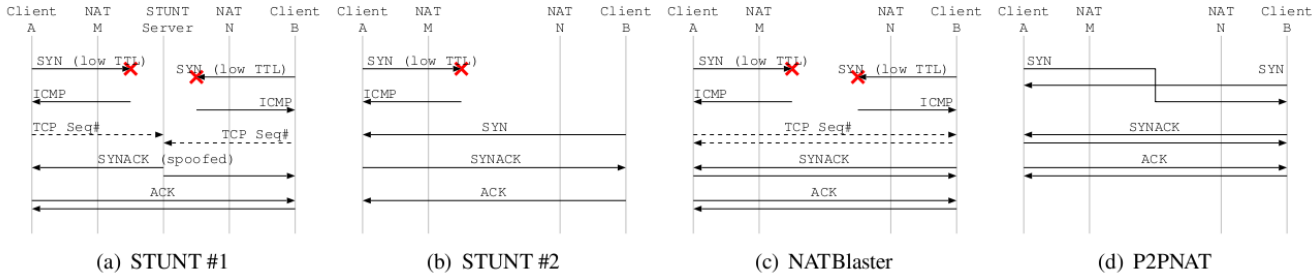

Figura 3.7: Paquetes generados por varios métodos de TCP NAT-traversal. Las líneas sólidas son paquetes TCP/IP o ICMP correspondientes a un intento de conexión. Las líneas punteadas son mensajes de control mandados por un canal fuera de banda.

3.6.1. Problemas de STUNT #1

- Cuando los NATs más exteriores tienen una interfaz en común, no se puede encontrar un TTL adecuado para que funcione el método
- Un error ICMP TTL-excedido puede ser generado en respuesta al paquete SYN y ser interpretado por el NAT como un error fatal
- El NAT puede cambiar el número de secuencia de TCP del SYN inicial de una manera que el SYNACK *spoofed* basado en el número de secuencia original se vea como un paquete fuera de ventana cuando arriba a el NAT
- Requiere que un tercero modifique un paquete para una IP arbitraria, lo que puede ser

descartado por varios filtros de ingreso y egreso de la red

3.6.2. Problemas de STUNT #2

- Tiene el mismo problema con el ICMP TTL-excedido que STUNT #1
- Dificultad de elegir un TTL adecuado
- Requiere que el NAT acepte un SYN entrante luego de un SYN saliente, que es una secuencia de paquetes poco frecuente

3.6.3. Dificultades encontradas para los distintos tipos de NAT

Método	NAT/Firewall	Linux	Windows
STUNT #1	• Determinar TTL • Error ICMP • TCP Seq# cambia • Modificación de IP	• Privilegios de superusuario	• Privilegios de superusuario • Fijar TTL
STUNT #2	• Determinar TTL • Error ICMP • SYN-out SYN-in		• Fijar TTL
NATBlaster	• Determinar TTL • Error ICMP • TCP Seq# cambia • SYN-out SYNACK-out	• Privilegios de superusuario	• Privilegios de superusuario • Fijar TTL • RAW sockets (> WinXP SP2)
P2PNAT	• Apertura simultanea de TCP • Inundación de paquetes		• TCP simultaneous open (< WinXP SP2)
STUNT #1 sin TTL	• Error RST • TCP Seq# cambia • Spoofing	• Privilegios de superusuario	• Privilegios de superusuario • TCP simultaneous open (< WinXP SP2)
STUNT #2 sin TTL	• Error RST • SYN-out SYN-in		
NATBlaster sin TTL	• Error RST • TCP Seq# cambia • SYN-out SYNACK-out	• Privilegios de superusuario	• Privilegios de superusuario • RAW sockets (> WinXP SP2) • TCP simultaneous open (< WinXP SP2)

3.6.4. Caracterización de los distintos tipos de NAT

Los siguientes datos se pueden tomar como referencia pero hay que tener en cuenta que la industria

cambia a un ritmo del 47% por año. Según este estudio se puede clasificar los NATs según varias características:

Característica	Valor
Nat Mapping	• Independent • IP δ • Puerto δ • IP y Puerto δ • Conexión δ
Filtrado	• Independent • IP • Puerto • IP y Puerto
Respuesta	• Descarte • TCP RST • ICMP
TCP Seq#	• Preserva • No preserva
Temporizadores	• Conservador • Agresivo

En la siguiente tabla se muestran los tipos de NAT mapping observados para un conjunto de 16 NATs en el laboratorio, y el estimado para un conjunto de 87 NATs de hogares.

NAT mapping	Lab	Hogares
NB:Independent	9	70,10%
NB:Address and Port 1	3	23,50%
NB:Connection 1	3	3,90%
NB:Port 1	0	2,10%
NB:Address δ	0	0,00%
NB:Connection R	1	0,50%

Tabla que clasifica por tipo de filtrado

Tipo de filtrado	Lab	Hogares
Address and Port	12	81.9%
Address	1	12.3%
Independent	3	5.8%

Filtrado de secuencias extrañas:

Secuencia	Filtrada
SYN-out SYNACK-in	0,00%
SYN-out SYN-in	13,60%
SYN-out ICMP-in SYNAK-in	6,90%

Secuencia	Filtrada
SYN-out ICMP-in SYN-in	22,40%
SYN-out RST-in SYNACK-in	25,60%
SYN-out RST-in SYN-in	28,10%

3.6.5. Nomenclatura de filtrado y mapeo de NATs

FC es equivalente a *(NB:Independent, EF:Independent),* AR es equivalente a *(NB:Independent, EF:Address* y *Port)* y PAR es equivalente a *(NB:Independent,* EF:*Port*)

3.6.6. Filtrado de la Respuesta

Cuando un paquete entrante es filtrado por un NAT, se puede elegir descartar silenciosamente o notificar al que lo envía. Se estima que 91.8% de los NATs descartan sin avisar. El resto avisa mandando un TCP RST ACK para dicho paquete.

3.6.7. Modificación de los paquetes

Los NATs cambian la IP y puerto origen de los paquetes salientes, y la IP y puerto destino de los paquetes entrantes. Necesitan traducir la IP y puerto de los paquetes encapsulados en la carga de ICMP de una manera que los *host* destino puedan saber a qué *socket* de transporte corresponde cada ICMP. Todos los NATs en el conjunto de prueba modifican correctamente los ICMP o los descartan. Estos ICMP no son siempre generados. Algunos NATs cambian los números de secuencia TCP sumándole una constante por flujo a la secuencia de paquetes salientes y restando la misma cantidad a los *ACK numbers* de los paquetes entrantes. Un 8.4% de los NATs cambian los números de secuencia de TCP. De esta manera, los métodos de TCP NAT-traversal que requieren el número de secuencia inicial del paquete que sale de el NAT no pueden usar el número de secuencia del SYN que llega al *host.*

3.6.8. Temporizadores TCP

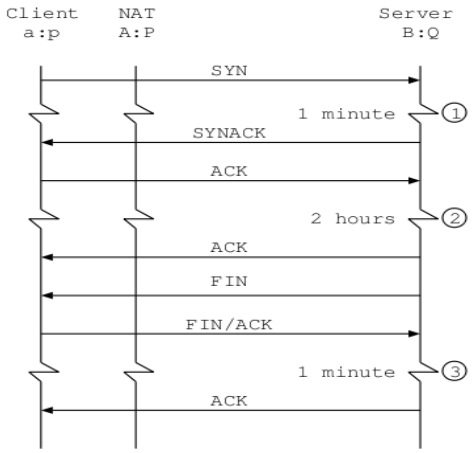

Figura 3.8: Temporizadores NAT durante una conexión TCP. (1) SYN_SENT, (2) Conexión establecida, (3) TIME_WAIT

Los NATs y firewalls no mantienen el estado indefinidamente porque esto los haría vulnerables a ataques. Aparte monitorean las banderas de TCP y recuperan el estado de conexiones explícitamente cerradas mediante FIN/FINACK o RST. En la Figura 3.8 se muestran los 3 tiempos que se usan. Dos temporizadores cortos (1 y 3) para expirar las conexiones no establecidas o conexiones que han sido cerradas respectivamente. En 2 (conexiones establecidas inactivas) los NATs usan un temporizador más largo. Según el RFC 1122 cada dos o más horas se debe mandar un paquete TCP-*Keepalive* en las conexiones inactivas. Para cada uno de estos tres casos se probó si los NATs respetaban las convenciones. Si las respetan se les denomina *conservative* y si no lo hacen se les llama *aggressive*. 27.3% de los NATs son *conservative* para los tres casos. 35.8% son conservativos para el caso 2. 21.8% son extremadamente agresivos para el caso 2, expirando la conexión antes de los 15 minutos de inactividad.

3.6.9. Predicción de puertos (Port Prediction)

88.9% de los NATs NB:Independent preservan los puertos. En el 81.9% de los casos el puerto se predice bien. Esto incluye a todos menos uno de los NATs NB:Independent y 37.5% de los NATs no NB:Independent. Para el restante 62.5%, al menos una de cada 60 veces otro host o aplicación robó el mapeo que el cliente de prueba había predecido para si mismo. En el 94% de los casos, más de 3/4 de las predicciones fueron correctas. Por lo tanto ante un intento fallido, si la aplicación reintenta la conexión, es probable que tenga éxito.

3.6.10. Problemas

Port Prediction tiene varios casos borde dónde puede fallar. En la Figura 3.9 si **A** usa el servidor STUNT **T** para predecir una IP y puerto cuando está intentando establecer conexión con **C**, va a

terminar descubriendo la IP externa del NAT **M** en vez de la del NAT **O**. *Port Prediction* requiere que la línea de NATs entre el cliente y el servidor STUNT sea la misma que la que está entre el cliente y el NAT más externo del *host* con el que se quiere conectar. **A** precisa descubrir **S** para poder conectarse con **C**. Si en cambio, **A** quisiera comunicarse con **B** y ambos usaran el servidor STUNT **S**, sus intentos de *Port Prediction* podrían colisionar haciendo que ninguno de los dos predijera bien los puertos. Ambos terminaran usando la IP externa de **O**. Este escenario se llama *"hairpin transaltion"* ya que el SYN de **A** dirigido a la IP y puerto predecido de **B** (la IP externa de **O** en este caso) va a ser dirigido a **O**, que necesita mandarlo devuelta a una interfaz interna. No todos los NATs manejan bien el *hairpin*, se estima que cerca del 72.8% no lo manejan bien.

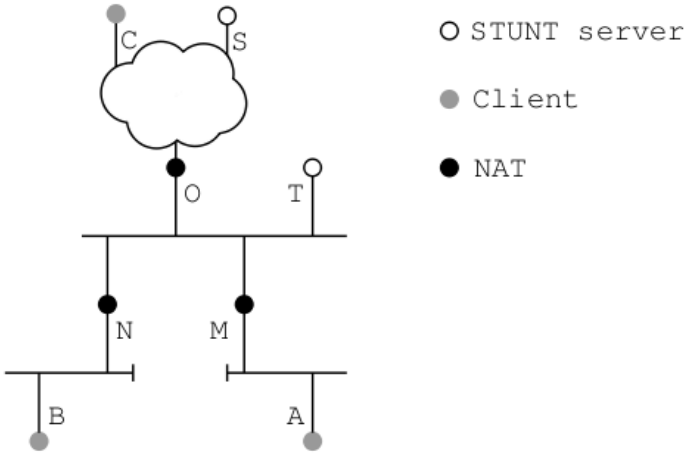

Figura 3.9: Escenario problemático para Port Prediction

3.6.11. Establecimiento de conexión TCP

Según el estudio realizado, para P2PNAT hay un 44.6% de éxito mientras que para STUNT #1 hay un 73.4% de éxito. El STUNT #2 llega a un éxito de 86%. Si se le agrega *Port Prediction* a P2PNAT le permite manejar NATs simétricos con lo que llega a un 84.3% de éxito. Sorpresivamente, modificar todos los métodos para que no usen TTLs cortos los mejora en un 5%. Eliminar las condiciones de carrera logra éxitos de 89.1% y 88.7% para los métodos STUNT modificados y 85.2% para *NATBlaster* modificado.

3.6.12. Implementación

Para la implementación se eligieron los métodos STUNT #2 y P2PNAT ya que STUNT #1 requiere muchos servidores que modifiquen los paquetes y *NATBlaster* requiere la funcionalidad de *socket RAW* que fue removida de Windows XP. El STUNT #2 es más robusto en establecer conexiones TCP. Tiene éxito en un 88% de los casos y no requiere *spoofing*, *RAW sockets* o permisos de súper usuario. Funciona con el *stack* TCP legado que no soporta *TCP simultaneous open*, mientras toma ventaja del *TCP simultaneous open* en los SOs modernos para llegar a un 100% de efectividad en los escenarios típicos. Es el más simple de implementar y funciona en Linux y Windows.

3.6.13. Limitaciones del estudio

- No se probaron todos los NATs disponibles en el mercado
- La muestra de NATs tomada es pequeña y no es representativa de los NATs existentes
- Se probaron solamente NATs domésticos. En NATs empresariales el *Port Prediction* fallará más
- No se probaron *firewalls* separados de NATs ni los *gateway* que traducen entre IPv4 e IPv6.

4 Soluciones existentes

En la primer sección de este capítulo se analizan distintas librerías para posibilitar NAT traversal. En la segunda sección se analizan algunos clientes P2P que integran alguna solución al problema de NAT traversal. A partir del análisis de la información recabada, en la última sección se eligen dos librerías para integrar al proyecto GoalBit.

4.1 Librerías

4.1.1. PJNATH

Licencia: GNU GPL v2

Documentación: Excelente

Último release: 1.8.10 (2010/12/7)

Protocolo: UDP utilizando ICE, STUN y TURN

Lenguaje: C

Plataforma: UNIX/Windows/MacOS

URL: http://www.pjsip.org/pjNATh/docs/html/index.htm

Popularidad: Excelente (http://www.pjsip.org/apps.htm)

4.1.2. JINGLE lib

Licencia: Berkeley style (http://code.google.com/apis/talk/libjingle/license.html)

Documentación: Excelente

Último release: 0.5.2 (2011/01/11)

Protocolo: UDP utilizando ICE, STUN y TURN

Lenguaje: C++

Plataforma: UNIX/Windows

URL: http://code.google.com/apis/talk/libjingle/index.html

Popularidad: *Libjingle* es un conjunto de componentes provisto por Google para interoperar con Google Talk permitiendole comunicación punto a punto y llamadas de voz.

Utilizado por GoogleTalk y Pidgin entre otros

4.1.3. libnice

Licencia: GNU LESSER GPL Version 2.1, February 1999

Documentación: Poca

Último release: 0.1.3 (2012/09/14)

Protocolo: UDP utilizando ICE, STUN y permite pseudo TCP sobre un layer de UDP

Lenguaje: C

Plataforma: UNIX/Windows (Gstreamer project)

URL: http://nice.freedesktop.org/wiki/

Popularidad: Librería distribuida junto con varios distros de Linux

4.1.4. mojop2p

Licencia: GNU GPL v3

Documentación: Poca

Último release: No disponible

Protocolo: XMPP/HTTP

Lenguaje: No disponible

Plataforma: UNIX/MacOS

URL: http://code.google.com/p/mojop2p/

Popularidad: Proyecto abandonado

4.1.5. gnunet

Licencia: GNU GPL v3

Documentación: Excelente

Último release: Framework 0.9.0pre2 (2010/12/23)

Protocolo: UPnP/NAT-PMP

Lenguaje: C

Plataforma: UNIX/Windows

URL: https://gnunet.org/

Popularidad: ?

4.1.6. XSTUNT Library

Licencia: GNU GPL v2

Documentación: Aceptable

Último release: 1.0 (2006/05/12)

Protocolo: TCP STUNT

Lenguaje: C/C++

Plataforma: UNIX/Windows

URL: http://www.cis.nctu.edu.tw/~gis87577/xDreaming/XSTUNT/index.html

Popularidad: Baja

4.1.7. libutp

Licencia: http://www.opensource.org/licenses/mit-license.php

Documentación: Código bien comentado

Último release: 2011/04/06

Protocolo: Implementación similar a TCP de LEDBAT. UTP provee entrega confiable y ordenada, manteniendo retardos extra al mínimo. Está implementada sobre UDP para ser independiente de la plataforma y funcional. Como resultado, uTP es el principal transporte usado en las conexiones P2P de uTorrent.

Lenguaje: C/C++

Plataforma: UNIX/Windows/MacOS

URL: https://github.com/BitTorrent/libutp

Popularidad: uTorrent, BitTorrent

4.2 Clientes P2P con solución de Nat Traversal interna

4.2.1. Gnutella

Es una gran red P2P que fue la primera red P2P descentralizada de su tipo [50].

Lista de implementaciones:

Cliente	UPnP port mapping	NAT Traversal	NAT port mapping
BearShare	Si	Si	Si
GiFT (core & plug-ins)	?	?	?
GnucDNA	No	No	No
Gtk-gnutella	Si	Si	Si
LimeWire	Si	Si	Si
Phex	No	Si	No
Shareaza	Si	Si	Si

4.2.2. BitComet

Es un cliente BitTorrent implementado en C++ que tiene su propia solución al NAT Traversal. Dicha solución tiene licencia comercial por lo que no esta abierta al público su implementación.

4.2.3. Shareaza

Cliente P2P para Windows que permite conectarse a varias redes P2P. Implementa una solución a NAT Traversal, pero la documentación está orientada al usuario y no al desarrollador por lo que no es útil para nuestro trabajo. También posee una wiki pero fue atacada por lo que tampoco se puede obtener documentación de la misma. Se puede acceder al código de este programa debido a su licencia GPLv3.

4.3 Evaluación

De la evaluación de las distintas alternativas de implementaciones para la solución al problema de NAT Traversal se decidió utilizar la librería PJNATH para una solución orientada a UDP y dentro de lo posible integrar el protocolo de manejo de congestión uTP. Se eligió PJNATH por su popularidad y excelente documentación. Otros clientes P2P no documentan sus soluciones al NAT Traversal por lo que llevaría un esfuerzo grande extraer la solución e investigarla.

Para una solución orientada a TCP que es lo que actualmente maneja GoalBit la librería a seleccionar es libnice, la cual no genera una conexión TCP pero nos provee de un pseudo TCP sobre UDP lo cual nos garantiza orden y confiabilidad.

5 Solución Propuesta

5.1 Arquitectura

A continuación explicaremos los elementos de la arquitectura que son parte de la solución para las dos librerías que seleccionamos.

En la Figura 5.1 se muestran los distintos componentes de la arquitectura de la solución. Los componentes son: *broadcaster*/*superpeer, normal peers*, servidor STUN, servidor Rendezvous y servidor Tracker. Estos se explicaran en detalle en las siguientes secciones.

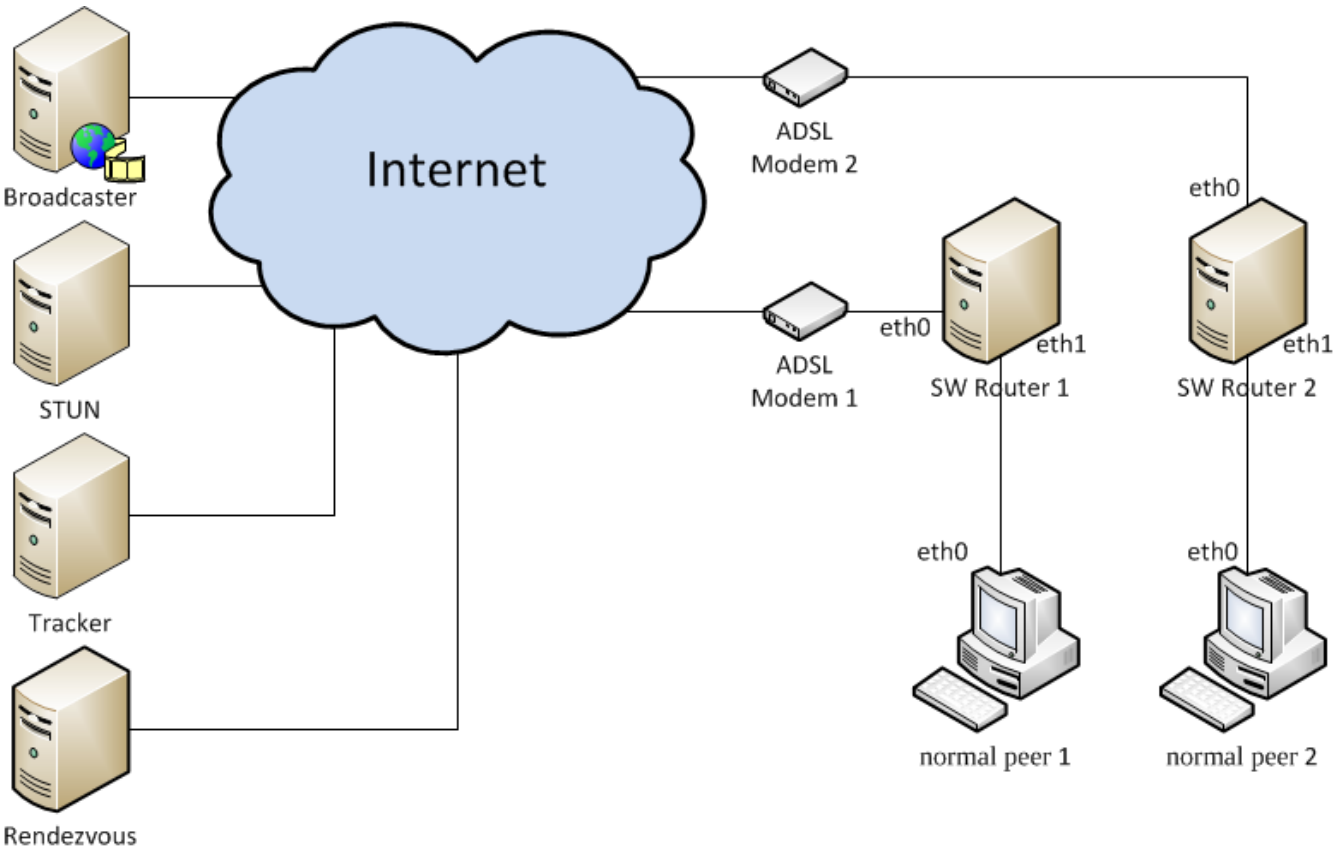

Figura 5.1: Componentes de la arquitectura

La solución propuesta integra las siguientes herramientas:

- GoalBit Media Player v0.7.7
- GoalBit Starter Suite v0.1
- GoalBit2File v0.1
- libgoalbit v0.1

En [51] están las instrucciones para la instalación del GoalBit Media Player (GMP).

Starter suite se descarga de [52] junto con las instrucciones para su instalación y manejo. Starter Suite esta compuesto por el tracker con una aplicación web que nos permite el monitoreo del tráfico en los canales, con diferentes tipos de usuario y niveles de acceso.

Para compilar e instalar libGoalbit ejecutar los siguientes comandos:

```
sudo apt-get install libssl-dev
mkdir -p ~/git/goalbit
cd ~/git/goalbit
git clone git://goalbit.git.sourceforge.net/gitroot/goalbit/libgoalbit
cd ~/git/goalbit/libgoalbit
./autogen.sh
./configure
make
sudo make install
```

Dentro de libGoalbit encontraremos GoalBit2File, la cual es una utilidad para prueba que nos permite tener un cliente GoalBit limitado el cual descarga el contenido en un archivo sin la necesidad del reproductor.

Pasos a seguir para configurar la arquitectura:

1) Crear el archivo .goalbit desde el GMP haciendo lo siguiente:
   Ir a la pestaña “Broadcast”. Llenar el formulario ingresando el “Video source”, “Output parameters” (se debe deseleccionar trackerless Kademlia e ingresar la IP de nuestro tracker) , “Channel identifier” y “Channel Name” y podemos obtener de “Generate GoalBit file” el archivo .goalbit
2) Submit file en el *tracker* y seleccionar el archivo .goalbit que recién creamos
3) Iniciar el *broadcast* desde el GMP
4) Obtener el archivo *broadcasteado* mediante el GoalBit2File especificándole la IP del *tracker,* el nombre del canal y tipo de archivo que esperamos recibir.

La Figura 5.1 muestra un diagrama del entorno de prueba donde se utilizan dos *sw routers.* También se realizaron pruebas utilizando dos *hw routes* (Linksys WRT54G2 y TP-Link N750). *hw router* significa un enrutador comercial diseñado y fabricado con *hardware* especifico para cumplir esa tarea, y *sw router* significa un enrutador hecho con una PC con dos tarjetas de red *ethernet*, el sistema operativo GNU/Linux Ubuntu 10.04, e *Iptables* que es un programa para filtrar y configurar reglas de paquetes IPv4.

Los *sw routers* no poseen UPnP, la cual es una característica a notar pues GoalBit trata de utilizar esta funcionalidad si está disponible. Se logra emular tres tipos de NAT (SYM, FC y PAR) utilizando *iptables* (como se explica en [53]) en *sw router 1* y *sw router 2*.

Configuración de iptables

Para lograr la emulación de los NAT en los sw routers se utilizaron los siguientes comandos para configurar las iptables. Primero que nada debemos deshacernos de las reglas existentes en el momento. Esto se hace de la siguiente manera:

```
sudo echo "1" > /proc/sys/net/ipv4/ip_forward
sudo iptables -F
sudo iptables -X
sudo iptables -t nat -F
sudo iptables -t nat -X
sudo iptables -t mangle -F
sudo iptables -t mangle -X
sudo iptables -P INPUT ACCEPT
sudo iptables -P FORWARD ACCEPT
sudo iptables -P OUTPUT ACCEPT
```

Después del limpiado de las iptables, pasamos a configurar de acuerdo al comportamiento esperado, En el *sw router* ppp0 esta sobre eth0 y es la interfaz generada por la conexión al módem ADSL y eth1 es la interfaz conectada por *ethernet* al *host peer* con su eth0. <*IP conexión a internet*> es la dirección IP de la interfaz ppp0 en el *sw router* así como <*IP ethernet host peer*> es la dirección IP de la red *ethernet* de eth0 del *host peer.*

Symmetric:

```
sudo iptables -t nat -A POSTROUTING -o ppp0 -j MASQUERADE --random
sudo iptables -A FORWARD -i ppp0 -o eth1 -m state --state RELATED,ESTABLISHED -j ACCEPT
sudo iptables -A FORWARD -i eth1 -o ppp0 -j ACCEPT
```

Restricted Cone:

```
sudo iptables -t nat -A POSTROUTING -o eth0 -p tcp -j SNAT --to-source <IP conexión a internet>
sudo iptables -t nat -A POSTROUTING -o eth0  -p udp -j SNAT --to-source <IP conexión a internet>
sudo iptables -t nat -A PREROUTING -i eth0 -p tcp -j DNAT --to-destination <IP ethernet host peer>
sudo iptables -t nat -A PREROUTING -i eth0 -p udp -j DNAT --to-destination <IP ethernet host peer>
sudo iptables -A INPUT -i eth0 -p tcp -m state --state ESTABLISHED,RELATED -j ACCEPT
sudo iptables -A INPUT -i eth0 -p udp -m state --state ESTABLISHED,RELATED -j ACCEPT
sudo iptables -A INPUT -i eth0 -p tcp -m state --state NEW -j DROP
sudo iptables -A INPUT -i eth0 -p udp -m state --state NEW -j DROP
```

Par (Port Restricted Cone):

```
sudo iptables -t nat -A POSTROUTING -o ppp0 -j SNAT --to-source <IP conexión a internet>
```

Full Cone:

```
sudo iptables -t nat -A POSTROUTING -o ppp0 -j SNAT --to-source <IP conexión a internet>
sudo iptables -t nat -A PREROUTING -i ppp0 -j DNAT --to-destination <IP ethernet host peer>
```

Nota: como podemos ver por el comando sudo que precede cada linea, se debe tener permisos root o sudoer para realizar estos cambios.

5.1.1. Modificación a los peers de GoalBit

En la Figura 5.2 se muestra la arquitectura lógica del *peer* modificado. La idea general de la solución es que se realice una conexión Nice y luego se establezca una conexión “local” con libgoalbit. Al recibir esta conexión “local” libgoalbit empieza a mandar los paquetes correspondientes al protocolo GoalBit.

Estos pasan a través de Nice, hasta llegar al otro extremo donde se tiene también una conexión “local” con libgoalbit. De este modo ambos extremos libgoalbit tienen la ilusión de tener conexiones libgoalbit “puras” sin la librería Nice en el medio.

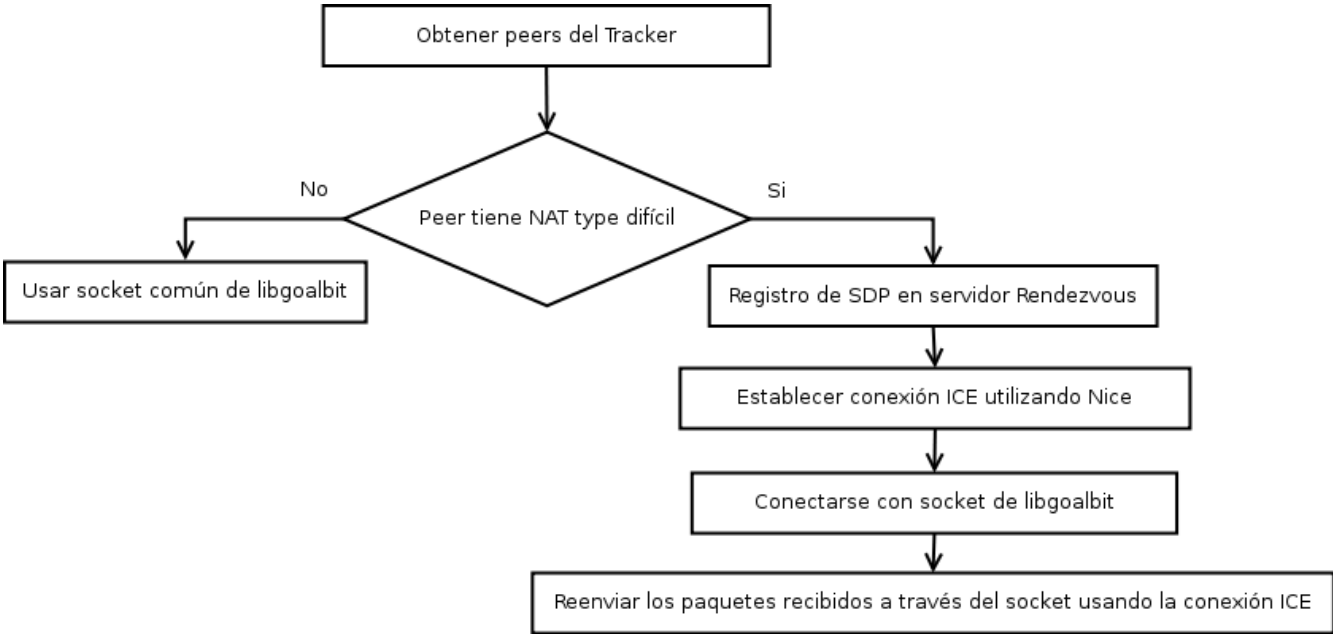

Figura 5.2: Arquitectura lógica del peer modificado

Para la librería PJNATH es la misma arquitectura, sólo que en vez de realizar la conexión ICE con Nice se utiliza PJNATH. La librería PJNATH sería funcional en una implementación de GoalBit que utilice UDP pero no para la implementación actual de GoalBit en TCP, por lo que no se completó dicha integración. Una vez que llegamos a un momento de la integración de PJNATH en el que vimos que se perdían paquetes y que el orden no era siempre el mismo optamos por abandonar dicha librería y enfocarnos en la librería Nice.

5.1.2. Servidor STUN

Para utilizar la implementación del protocolo ICE se debe contar con un servidor STUN que cumpla con RFC 3489 [8]. Este puede ser un servidor público o un servidor propio.

Los servidores STUN públicos que usamos como stunserver.org y stun.pjsip.org usan la implementación [54] de STUN. No usamos un servidor propio debido a que se necesitan dos interfaces con distinta dirección IP pública que no poseemos.

Modo de uso del servidor

```
./server [-v] [-h] [-h IP_Address] [-a IP_Address] [-p port] [-o port] [-m mediaport]
```

El servidor STUN necesita dos IPs y dos puertos, estos pueden ser especificados utilizando los siguientes parámetros:

-h fija la IP primaria

-a fija la IP secundaria

-p fija el puerto primario, por defecto 3478

-o fija el puerto secundario, por defecto 3479

-b hace que el servidor corra en *background*

-m indica que empezará en el puerto m

-v corre en modo *verbose*

Observación

La utilización de STUN hace más compleja la pila de protocolos y no está tan optimizada como TCP para obtener alto rendimiento y amigabilidad ante congestiones [49].

5.1.3. Servidor de Rendezvous

Para intercambiar los SDP se dispone de un Servidor Rendezvous. Este recibe información de los clientes que quieren iniciar una sesión ICE con otros. La lógica del cliente se incorpora al código de los clientes GoalBit. Luego de que ambos *peers* están conectados ambos pueden recibir y enviar datos a través de la conexión ICE.

Servidor Rendezvous

El servidor Rendezvous accede a una base de datos para registrar los Id y SDP de los distintos clientes, y el peer en el que está interesado el cliente. En caso de que ya se cuente con el cliente correspondiente, se le devolverá el SDP del mismo y se registrará como pendiente entregarle el SDP del nuevo cliente al cliente que se registró primero, en su próxima consulta.

Como se ve en la Figura 5.3 el servidor Rendezvous está todo el tiempo esperando recibir un paquete. Cuando recibe un paquete analiza su tipo y según cuál sea realiza ciertas tareas con la base de datos y envía una respuesta al cliente. La respuesta con tipo=**n** se usa para indicar que no se encontró un cliente que le sirva al cliente que envío el paquete al servidor. En caso de que si se encuentre un Id que le sirva al cliente, se envía el SDP del mismo al cliente y se borra el registro de la base para indicar que ese SDP ya fue utilizado.

En caso de recibir un paquete con tipo=**d** borra los datos asociados al *id* del paquete ya que el tipo=**d** indica que el cliente ya logró establecer conexión con otro *peer* por lo que los SDP que estaban registrados en el servidor ya no son validos.

La implementación en C++ del servidor está basada en el servidor Fork-N presentado en [55]. Fork-N es un ejemplo de un servidor que atiende un *socket* y luego de que alguien se conecte a el hace un *fork* para atender al pedido en un proceso hijo. Para implementar el servidor Rendezvous agregamos la lógica de procesamiento de mensaje donde el mensaje es, según el tipo del mismo, desmembrado en sus componentes para responder de la manera esperada al pedido. Los pedidos que no pueden ser satisfechos en el momento son almacenados en una base de datos MySQL la cual es mantenida eliminando las entradas que no fueron renovadas en un periodo configurable de tiempo. Un semáforo

mutex mantiene el procesamiento de los mensajes mutuamente excluidos entre cada uno de los procesos hijos, haciendo que el procesamiento de las diferentes operaciones en la base y respuesta sea una operación atómica.

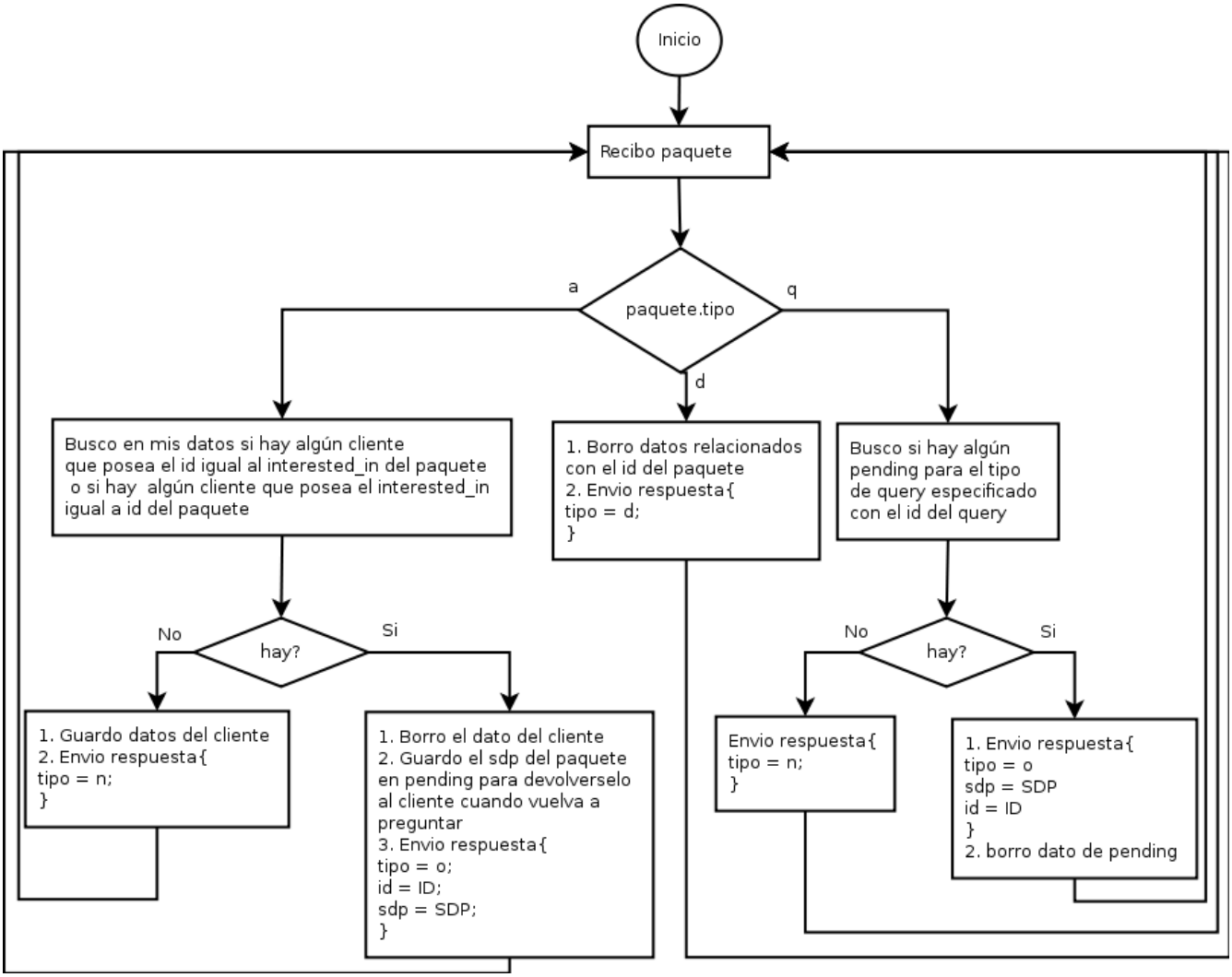

Figura 5.3: Servidor Rendezvous

5.1.4. Modificaciones al Tracker de GoalBit

Agregar Id de los pares (*peers*)

Como vimos en la sección anterior el servidor de *rendezvous* necesita un identificador del *peer*, éste

identificador es generado aleatoriamente por los clientes y transferido al *tracker.* El *tracker* no comunica el identificador de los *peers* a los *peers,* por lo que un *peer* no puede comunicarle al *rendezvous server* el identificador de los *peers* en los cuales está interesado. Para hacer que sea posible modificamos el *tracker* de GoalBit para que se incluya el identificador de los *peers* junto con la información que le transmite a los *peers* acerca de otros *peers.*

Agregar tipo de NAT (NAT type) de los pares (peers)

Se agrega el NAT *type* (que corresponde con el resultado de STUN) de los *peers* para distinguir en qué caso se debe usar la nueva forma de conexión (mediante ICE). De esta forma sólo se usa ICE cuando realmente es necesario debido al NAT *type* de los clientes que se quieren conectar y además se permite compatibilidad con versiones anteriores de GoalBit en la que no está disponible la solución mediante ICE.

Enviar información según el Id de los pares (peers)

En el *tracker* se revisa el **Id** del peer. En la primer parte del **Id** hay un código asociado a la versión del cliente *libgoalbit*. De esta manera podemos distinguir quienes implementan la solución con ICE para poder brindar compatibilidad con versiones anteriores de libGoalBit *(peers* con **Id** "viejo") en las que no está disponible la solución mediante ICE (se les debe enviar la información del *tracker* como se venía haciendo hasta ahora). A los *peers* con **Id** "nuevo" se les debe enviar la información con las modificaciones para que sepan si el peer con el que se quieren contactar es compatible o no con ICE.

5.2 Solución con librería Nice

Según [56] Libnice es una implementación de ICE (RFC 5245) y de STUN (RFC 5389 [42]). Provee una librería basada en Glib (libnice) y una librería libre de Glib (libstun) así como elementos Gstreamer.

ICE es útil para programas que quieren establecer *streams* P2P. Automatiza el proceso de atravesar NATs y provee seguridad contra algunos ataques. También permite que los programas creen *streams* confiables usando una capa TCP sobre UDP. Esto es sumamente útil para poder usar la implementación actual de GoalBit en TCP.

Algunos estándares que usan ICE son SIP y XMPP Jingle.

5.2.1. Funcionamiento

Para crear un NiceAgent en modo confiable se utiliza `nice_agent_new_reliable`, que usa el `PseudoTcpSocket` para asegurar la confiabilidad de los mensajes. El `PseudoTcpSocket` implementa un *Pseudo* TCP *Socket* sobre UDP el cual es un subconjunto del stack TCP para permitir transporte confiable sobre *sockets* no confiables (como UDP). NiceAgent (el objeto principal de libnice) se encarga de todo lo concerniente a ICE.

Al ser libnice una librería que utiliza glib, el modo de manejo de la misma es a través de eventos que se encuentran ligados a un *main loop*. Este *loop* al correr se encarga de verificar los eventos a los cuales se "attacharon" funciones para hacer el manejo de los eventos.

Para que el agente pueda obtener una estimación de la topología a la cual se enfrenta hay que *setearle* un *stun server*, esto se logra invocando a `g_object_set(G_OBJECT(data.agent), "stun-server", STUN_ADDR, "stun-server-port", STUN_PORT, NULL);`

También debemos conectar señales del agente a funciones, esto se logra invocando a `g_signal_connect(G_OBJECT(data.agent), SENIAL, G_CALLBACK(funcion), &data);`, de esta manera cuando el agente emita una señal llamará a la función asignada a la misma con *data* como dato.

5.2.2. Señales manejadas

- `candidate-gathering-done`: se produce cuando se termina de descubrir los candidatos locales, los que son interfaces para intentar crear el *stream* de datos con un *peer.*
- `component-state-changed`: se produce cuando el agente cambia de estado, los estados posibles son:
  - **NICE_COMPONENT_STATE_DISCONNECTED**: el agente se desconectó
  - **NICE_COMPONENT_STATE_GATHERING**: el agente comenzó a obtener los candidatos locales.
  - **NICE_COMPONENT_STATE_CONNECTING**: estableciendo la conexión
  - **NICE_COMPONENT_STATE_CONNECTED**: poseemos al menos un par de candidatos (uno local y otro del *peer*) a través de los cuales se pueden intercambiar datos
  - **NICE_COMPONENT_STATE_READY**: el componente se encuentra listo para transmitir y recibir información a través del canal creado
  - **NICE_COMPONENT_STATE_FAILED**: la creación del canal falló
- `reliable-transport-writable`: se produce cuando tenemos un *reliable agent*, y este se encuentra listo para transmitir. En caso de que el mismo no se encuentre listo la operación de escritura para el agente `(nice_agent_send)` nos devolverá **-1** o enviará los datos que pueda y nos devolverá un entero (el número de datos que se pudieron enviar) menor que la cantidad de datos que le mandamos para enviar. En estos casos debemos esperar a que vuelva a suceder este evento para volver a llamar a `nice_agent_send`.
- `new-selected-pair`: se produce cuando un par de candidatos (uno local y otro del *peer*) fueron seleccionados para transferir datos por un componente del *stream.*

5.2.3. Recibir y transmitir datos

El evento de llegada de datos al *stream* es anotado con `nice_agent_attach_recv`, la función *attacheada* a este evento será llamada cada vez que el *stream* reciba datos.

Para que nuestro agente esté pronto para empezar a transmitir datos debemos realizar la siguiente secuencia:

1. Crear un *main loop* de `glib(g_main_loop_new)`.
2. Crear un agente de NICE del tipo *reliable* pasandole el *main loop* que acabamos de crear (`nice_agent_new_reliable`).
3. Setear el STUN *server* y el TURN *server*, en nuestro caso no se utiliza servidor TURN.
4. Conectar las funciones que se quiere que manejen las señales.
5. Agregar un *stream* al agente.
6. *Attachear* una función como *receiver* (llamada cuando se posee datos en un *stream*).
7. Indicar al agente que comience a recoger candidatos.
8. Inicializar el *main loop*.

Cuando el agente dispara la señal de que tiene todos los candidatos, se intercambia esta información con el *peer* con el que se está interesado en establecer conexión, para esto se utiliza el servidor de rendezvous.

Una vez que se obtienen los candidatos del *peer* remoto y se los pasa al agente, este comenzará a tratar de generar conexiones entre pares de ellos. En caso de que la conexión entre al menos un par de candidatos se pueda establecer se pasa al estado **CONNECTED**.

Luego de que se agotaron las combinaciones de candidatos y se han establecido todas la conexiones que se pudieron entre los candidatos se pasa al estado **READY**. Aquí es cuando se comienza a enviar datos. Los dos *peers* no entran en estado **READY** al mismo tiempo por lo tanto uno de ellos puede recibir datos antes de estar en estado **READY** lo cual se debe manejar.

5.2.4. Integración Nice-Libgoalbit

Dentro de la función `NewPeer` que es la que se llama cada cierto tiempo para chequear si hay nuevos *peers* se crea una conexión Nice si ambos *peers* tienen integrada la solución NICE (esto se hace para mantener compatibilidad hacia atrás con las viejas versiones de GoalBit que no tenían NICE).

En el caso de que se use NICE se crea un hilo con la función `establish_nice` a la que se le pasa el parámetro `args` que contendrá nuestro identificador y el del *peer* en el que se está interesado.

Lo que hace `establish_nice` es crear la conexión NICE y dejar el main_loop corriendo. Una vez realizado esto empieza todo el negociado de NICE. Cuando el negociado termina (se llega al estado NICE_COMPONENT_STATE_READY) o recibimos datos por el canal (caso posible por la llegada al estado NICE_COMPONENT_STATE_READY de uno de los *peers* antes que el otro), se invoca a la función `createChannel` que crea un *socket* conectado a libgoalbit que se guarda en `dta->sock_ext`. Para ello invoca a `connectToLib` y utiliza una funcionalidad de la glib para *attachar* un evento por tiempo al `mainloop`, para que cada 100 milisegundos se lea del *socket* que está creando a ver si hay datos. Análogamente, cuando se reciben datos a través de NICE, estos se escriben al `socket dta->sock_ext` en la función `cb_nice_recv`.

Esta solución se implementó con éxito. Los resultados obtenidos pueden verse en la sección 7.1.

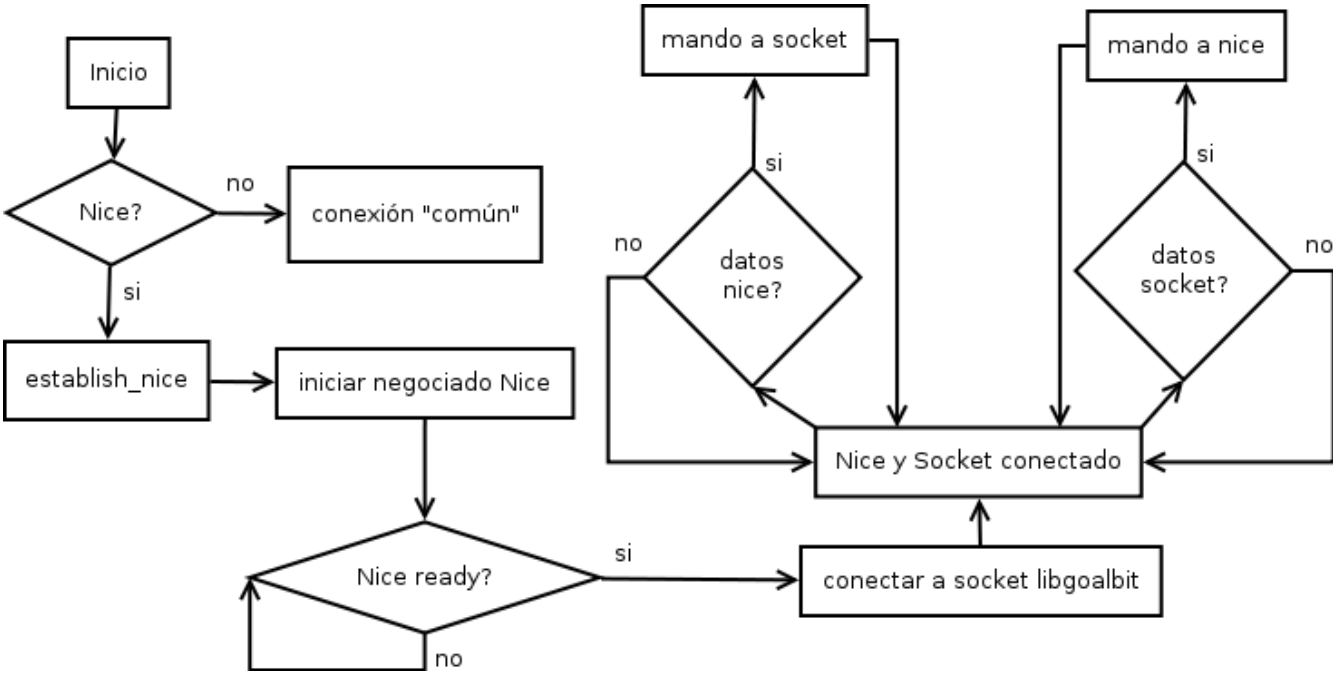

Figura 5.4: Arquitectura de integración NICE-Libgoalbit

5.3 Solución con librería *PJNATH*

Durante las pruebas de la integración de PJNATH a GoalBit detectamos que no se enviaban los paquetes siempre en el mismo orden o la misma cantidad. Al consultar la documentación y la lista de correo vimos que no era confiable la conexión por lo que debimos buscar una alternativa: libnice. De lo que habíamos realizado para PJNATH se reutiliza para la integración de libnice el servidor *Rendezvous*, el conocimiento de libgoalbit y la modificación al *tracker* de GoalBit.

Esta librería sería funcional en una implementación de GoalBit que utilice UDP pero no para la implementación actual de GoalBit en TCP ya que tiene perdida de paquetes y no garantiza el orden de recepción de los paquetes, por lo que no se completó su integración a libgoalbit. De todas formas se vio muy útil su estudio puesto que la amplia mayoría de soluciones para NAT Traversal suponen una comunicación UDP, y entre otras razones esto se debe a que se logran mejores resultados. Por tanto no debe descartarse a futuro que la mejor solución sea agregar comunicación UDP a libgoalbit, siendo ésta la razón que motiva este estudio.

La librería PJNATH (PJSIP NAT *Traversal Helper*) [57] contiene varios objetos para asistir a los programas en la tarea de atravesar un NAT, usando protocolos basados en estándares como STUN, TURN, y ICE. El objeto de más bajo nivel de esta librería es el ICE *stream transport* (ice_strans), dónde se envuelven las funcionalidades de STUN, TURN, y ICE en un sólo objeto y se provee a las aplicaciones con una API para mandar y recibir datos, así como realizar un manejo de sesión ICE.

Desde el punto de vista del diseño de la librería, todas las funcionalidades en PJNATH se implementan

en dos capas, la capa independiente del transporte o de sesión, y la capa de transporte. La capa de sesión contiene sólo la lógica para manejar la sesión correspondiente (por ejemplo, STUN, TURN, y sesión ICE). La capa de transporte une a la capa de sesión con *sockets* para que estén listos para usar objetos de transporte.

Se asume que el programa quiere usar la capa de transporte ICE, y no la capa de sesión ICE.

5.3.1. Prototipo de ICE

La implementación del PJSIP que utilizamos es la 1.10, y como prototipo utilizamos el ICEdemo que se encuentra en el mismo paquete (pjproject-1.10/pjsip-apps/src/samples/icedemo.c). Esta aplicación muestra como utilizar ICE en PJNATH sin tener que utilizar el protocolo SIP.

Este demo se utiliza con las siguiente instrucciones:

- Debemos Poseer al menos un servidor STUN
- Un servidor TURN es opcional, en nuestro caso no lo utilizamos, pero podemos encontrar uno dentro del package del PJSIP (pjproject-1.10/pjnath/src/pjturn-srv)
- Una topología de red donde se pueda verificar el funcionamiento correcto del protocolo ICE, en nuestro caso poseíamos dos clientes cada uno detrás de su propio NAT

Los parámetros a especificar en la linea de comando son los siguientes:

Opciones generales de ICEdemo

--comp-cnt, -c N	Cuenta de componentes (default=1)
--nameserver, -n IP	Configurar *nameserver* para activar la resolución de DNS SRV
--max-host, -H N	Establecer la cantidad máxima de *hosts* candidatos a N
--regular, -R	Usar nominación regular (*default* agresiva)
--log-file, -L FILE	Salvar salida en un *log* FILE
--help, -h	Desplegar esta pantalla.

Opciones de ICEdemo relacionadas con STUN

--stun-srv, -s HOSTDOM	Habilitar candidato srflx resolviendo STUN *server*. HOSTDOM puede ser "host_or_ip[:port]" o un nombre de dominio si se usa un DNS SRV.

Opciones de ICEdemo relacionadas con TURN

--turn-srv, -t HOSTDOM	Habilitar candidato por *relay* utilizando un TURN *server*. HOSTDOM puede ser de la forma "host_or_ip[:port]" o un nombre de dominio si se usa DNS SRV
--turn-tcp, -T	Utilizar TCP para conectarse al TURN *server*
--turn-username, -u UID	*Setear* UID como nombre de usuario TURN
--turn-password, -p PWD	*Setear* PWD como *password* de la credencial

--turn-fingerprint, -F Utilizar *fingerprint* para pedidos salientes de TURN

Entendiendo el flujo de ICE

1. crear instancia
2. repetir estos pasos cuantas veces se quiera
    1. iniciar sesión como oferente o contestador
    2. desplegar nuestro SDP
    3. "enviar" nuestro SDP desde la salida de "s" al servidor remoto, copiando y pegando el SDP en la otra aplicación icedemo*
    4. procesar el SDP de la otra instancia de icedemo, pegando el SDP generado por la otra instancia de icedemo*
    5. iniciar la negociación de ICE desde ambos extremos**
    6. la negociación de ICE se llevará a cabo,y el resultado será desplegado en pantalla
    7. enviar datos al *host* remoto
    8. finalizar/detener la sesión ICE
3. destruir instancia

* El intercambio de SDP que se realiza en el icedemo a través de copiar y pegar. Se hará en GoalBit a través del Servidor Rendezvous, siendo este el responsable de comunicar la información de los interesados en generar una conexión P2P.

** Tenemos aproximadamente 7 segundos para iniciar la negociación. En GoalBit se iniciará la negociación en tanto obtengamos el SDP del otro cliente.

Cliente

El cliente *Answerer* al principio crea un SDP, y obtiene del *Tracker* su **Id** junto con una lista de clientes que tienen datos que le interesan. Luego selecciona un cliente con **Id** interested_in de la lista de clientes y envía su **Id**, SDP e interested_in al servidor Rendezvous y espera una respuesta.

Si la respuesta tiene tipo=**o** obtiene el SDP de la respuesta e inicia la conexión con el otro cliente.

Si la respuesta tiene tipo=**n** espera un TIMEOUT y vuelve a consultar al servidor, esta vez enviando un paquete con tipo=**q** y su **Id**. Repite esto último hasta que obtiene una respuesta positiva del servidor. Una vez que recibe esta respuesta positiva obtiene el SDP de la respuesta e inicia conexión con el otro cliente. Finalmente envía un paquete con tipo=**d** para indicarle al servidor que ya logró conectarse con un *peer.*

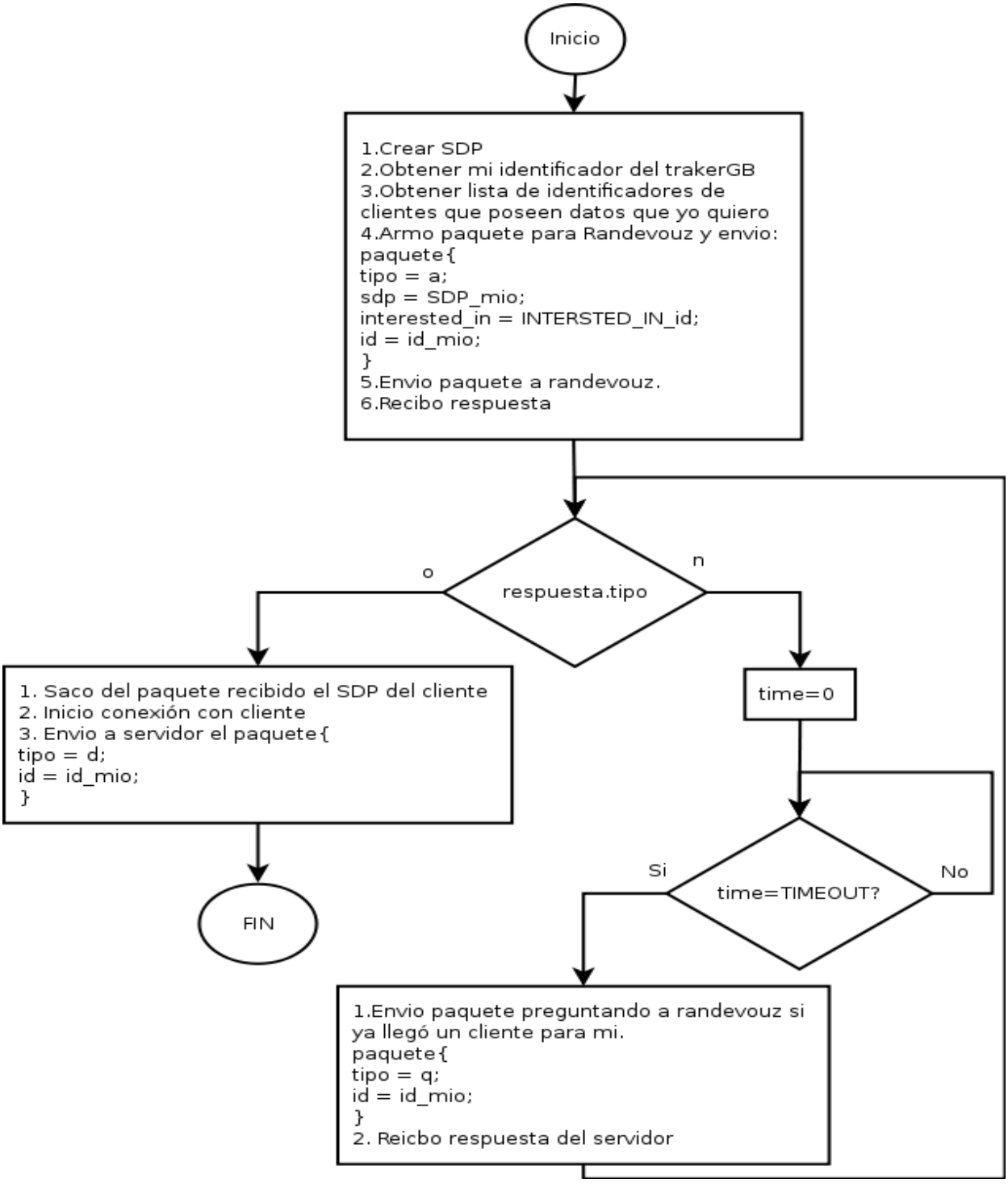

Figura 5.5: Cliente Answerer

5.3.2. Definiciones

`ice_strans`: Es el nombre de clase de PJNATH para el transporte que implementa la negociación ICE.

`ICE session`: Una sesión multimedia (por ejemplo una sesión de llamada) entre dos extremos ICE,

dentro de `ice_strans`. Un objeto `ice_strans` puede ser reusado para facilitar múltiples sesiones ICE (pero no simultáneamente).

`ICE endpoint`: El programa que implementa ICE.

5.3.3. Incorporando ICE en un programa

Para usar ICE, el programa debe reemplazar sus *sockets send* y *receive* por el objeto ice_strans. Una vez que la sesión ICE en `ice_strans` está levantada y corriendo, el programa usa `pj_ice_strans_sendto()` para mandar datos, y registra `on_rx_data()` en la estructura `pj_ice_strans_cb` para recibir datos.

Para el uso de SIP/SDP[58], un `ice_strans` es bueno para un media stream, y un media stream puede contener más de un media transport, o componente invocado en términos de ICE (por ejemplo: un RTP y un RTCP). Cada componente generalmente va a ser provisto con más de un candidato, por ejemplo: candidatos locales, candidato STUN, y candidato TURN. Será luego tarea de ICE descubrir qué par de candidatos usar para la sesión.

Para usos no SIP, será decisión de diseño del programa si crear un `ice_strans` con muchos componentes, o muchos `ice_strans` con un componente cada uno. Usar la primera de las opciones será más simple ya que sólo se debe trabajar con una sesión, pero la segunda opción tiene la ventaja de una negociación más rápida (por decenas o cientos de mili segundos) ya que ambos `objetos ice_strans` pueden hacer la negociación en paralelo.

5.3.4. Preparativos

Antes de usar el ICE de PJNATH se deben realizar varios pasos.

La librería PJNATH depende de las siguientes librerías, por lo tanto deben ser compiladas (o adaptadas si es necesario) y agregadas a las especificaciones de linkedición del programa:

- PJLIB (para accesos a memoria, temporizador, *network* I/O, así como la estructura de datos básicos),
- PJLIB-UTIL (para los algoritmos de encriptación necesarios para STUN)

Varios objetos PJLIB deben ser preparados por los programas:

- como mínimo una instancia de `memory pool factory (MPF)` es requerida para todos los programas basados en PJLIB,
- como mínimo una pila de temporizador para administrar los temporizadores,

- como mínimo una instancia de `ioqueue` para administrar los eventos de I/O de red.

Un objeto de cada uno es suficiente, aunque el programa puede crear más para mejorar el rendimiento (limitando el número de objetos que cada uno maneja) o por otras razones.

Una vez que estos objetos son creados, debe haber algo que consulte la pila del `temporizador` y de `ioqueue`. Generalmente los programas crean como mínimo un *thread* para realizar este *polling*.

Estas son tareas básicas que son requeridas para todos los programas de red basados en PJLIB, en los ejemplos se pueden ver código de cómo usarlo (por ejemplo: `turn-client`).

5.3.5. Ciclo de vida básico

Lo que sigue es un pequeño panorama del ciclo de vida de ice_strans. Cada uno de los pasos será explicado en las siguientes secciones:

- crear `ice_strans`,
- esperar a que el relevamiento de candidatos se complete,
- iniciar la sesión ICE:
  - crear la sesión ICE,
  - intercambiar la información de ICE con el remoto (usuario, contraseña, lista de candidatos),
  - iniciar la negociación ICE,
  - esperar a que la negociación termine,
  - intercambiar datos entre los extremos,
  - destruir la sesión ICE,
  - repetir los pasos de arriba para empezar una nueva sesión ICE,
- destruir `ice_strans`.

Creando el ICE stream transport

- Para crear el `ice_strans`:
- inicializar el `pj_ice_strans_cfg`. Entre otras cosas, esta estructura contiene las configuraciones requeridas para habilitar y usar STUN y TURN, así como instancias de la MPF, temporizador *heap* e *ioqueue* en el campo `stun_cfg`.
- Invocar a `pj_ice_strans_create()`
- esperar a que el callback de `on_ice_complete()` del `pj_ice_strans_cb` sea invocado con *op argument* de PJ_ICE_STRANS_OP_INIT, para indicar el estado del proceso de relevamiento de candidatos (por ejemplo: el resultado del pedido de STUN *binding* y las operaciones de reserva de TURN). El estado de este proceso de relevamiento de candidatos va a

ser indicado en el argumento de status del *callback*, con PJ_SUCCESS indicando operación exitosa.

- Una vez que el `ice_strans` es creado, puede ser usado para crear sesiones de ICE. Una sesión ICE representa una sesión multimedia entre extremos (o sea una sesión de llamada). Después de que una sesión se completa, la misma `ice_strans` puede ser usada para facilitar siguientes sesiones. Sólo una sesión puede estar activa en un `ice_strans` al mismo tiempo.

Trabajando con la sesión

Los pasos para usar la sesión generalmente son los siguientes:

Creación de sesión

- Crear la sesión invocando a `pj_ice_strans_init_ice()`, especificando el rol inicial del extremo (ICE) y opcionalmente, el usuario y contraseña local.

El rol afecta el comportamiento de la negociación ICE, especialmente para determinar que extremo es el controlador. Mientras ICE provee resolución de conflicto en su proceso de negociación, siempre es recomendado proveer el valor inicial correcto para evitar *round-trips* innecesarios para resolver el conflicto de roles (en la solución que se implementó se optó porque siempre se usara el mismo rol y ICE resolviera el conflicto de roles ya que esto va en concordancia con la forma en que GoalBit trata a las conexiones, no importando si el cliente se está conectando o el cliente está recibiendo la conexión).

Intercambiando información ICE con el extremo remoto

Antes de que empiece la negociación ICE, cada extremo ICE necesitará saber la información ICE del otro extremo. En usos SIP/SDP, esto sucederá cuando los programas intercambien SDPs entre si. Para usos no SIP, el programa decidirá como intercambiar la información (así como cómo codificarla).

La siguiente información debe ser enviada al extremo remoto ICE:

- el usuario y contraseña de la sesión ICE local (*user fragment* y contraseña)
- la lista de candidatos de cada uno y todos los componentes ICE. La función `pj_ice_strans_enum_cands()` es usada para listar los candidatos del componente ICE. Para cada candidato, la siguiente información debe ser intercambiada:
  - ID del componente
  - tipo de candidato (i.e. *host*, srflx, o *relay*)
  - *foundation* ID
  - prioridad
  - tipo de transporte (sólo soporta UDP por ahora)
  - IP de transporte (familia, IP y puerto)
  - IPs relacionadas opcionales (ejemplo: para candidatos srflx/STUN, la IP relacionada es la IP local desde donde se manda el pedido STUN). Sólo se usará en caso de tener que corregir

problemas.

- Opcionalmente la IP del candidato por defecto para cada componente ICE. Si el extremo remoto no soporta ICE, puede mandar los datos a esta IP. El programa también puede usar esta IP para intercambiar datos mientras la negociación ICE está ocurriendo. Para candidato por defecto debe elegirse el que tenga más posibilidades de tener éxito, ejemplo TURN, STUN, o uno de los candidatos locales, en ese orden. El programa puede usar la función `pj_ice_strans_get_def_cand()` para obtener el candidato por defecto de `ice_strans`.

Cómo codificar/descodificar así como intercambiar la información de arriba en contextos no-SIP dependerá del programa/uso que se le vaya a dar. En el ejemplo de uso de PJSIP dónde ICE es integrado con *media transport*, la tarea de codificar/descodificar la información de arriba es hecha por el ICE *transport* de PJMEDIA (pjmedia/transport_ice.c), y la información va a ser intercambiada mediante oferta/respuesta SDP. A continuación hay un ejemplo de SDP generado por PJSIP que contiene información ICE, con los atributos relevantes para ICE en negrita:

```
v=0
o=- 3423381096 3423381096 IN IP4 81.178.x.y
s=pjmedia
c=IN IP4 81.178.x.y
t=0 0
a=X-nat:5
m=audio 4808 RTP/AVP 103 102 104 117 3 0 8 9 101
a=rtcp:4809 IN IP4 81.178.x.y
a=rtpmap:103 speex/16000
a=rtpmap:102 speex/8000
a=rtpmap:104 speex/32000
a=rtpmap:117 iLBC/8000
a=fmtp:117 mode=30
a=sendrecv
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=ice-ufrag:2b2c6196
a=ice-pwd:06ea0fa8
a=candidate:Sc0a8000e 1 UDP 1694498815 81.178.x.y 4808 typ srflx raddr 192.168.0.14 rport 4808
a=candidate:Hc0a8000e 1 UDP 2130705151 192.168.0.14 4808 typ host
a=candidate:Sc0a8000e 2 UDP 1694498814 81.178.x.y 4809 typ srflx raddr 192.168.0.14 rport 4809
a=candidate:Hc0a8000e 2 UDP 2130705150 192.168.0.14 4809 typ host
```

(Nota: las líneas c= y a=rtcp contienen la IP del candidato por defecto en ICE para los componentes RTP y RTCP respectivamente. Las IPs públicas han sido alteradas en el SDP de arriba por razones de seguridad).

El ice_strans también necesitará recibir la información de arriba antes de que pueda empezar la negociación ICE.

Empezando la negociación ICE

- Una vez que los extremos ICE han mandado/recibido la información ICE a/desde el extremo remoto, pueden empezar la negociación ICE llamando a `pj_ice_strans_start_ice()`. Esta función necesitará la información ICE de arriba como parámetro. Cada extremo necesitará llamar a esta función para que la negociación tenga éxito.

La negociación ICE luego empezará.

- El tiempo en el que cada extremo invoca a la función `pj_ice_strans_start_ice()` no tiene que ser absolutamente simultáneo, aunque cuanto más sincronizado mejor para acelerar la negociación, y también hay un límite de 7 a 8 segundos antes de que la negociación ICE termine en estado *timeout*.

Obteniendo el resultado de la negociación ICE

- El programa será notificado del resultado en el (nuevo) *callback* `on_ice_complete()` del `pj_ice_strans_cb`, aunque en este momento el parámetro op será PJ_ICE_STRANS_OP_NEGOTIATION. El estado de la operación será indicado en el parámetro de estado del *callback*, dónde PJ_SUCCESS indica negociación exitosa.
- Es posible que el número de componentes entre los dos extremos ICE sea distinto, por ejemplo: un extremo soporta RTCP pero el otro no. La función `pj_ice_strans_get_running_comp_cnt()` puede ser usada (después de que la negociación ICE termina) para encontrar cuántos componentes han sido negociados por ICE. De todas maneras, el programa siempre puede deducir esta información comparando su lista de candidatos locales con la lista de candidatos en el otro extremo.

Mandando y recibiendo datos

- Usar `pj_ice_strans_sendto()` para mandar datos al extremo ICE remoto. Los datos entrantes se reportaran en el *callback* `on_rx_data()` de la estructura `pj_ice_strans_cb`.

Terminando la sesión

- Una vez que la sesión terminó (por ejemplo la llamada terminó), invocar a `pj_ice_strans_stop_ice()` para liberar los recursos reservados por la sesión.
- El programa puede reusar la misma instancia de `ice_strans` para iniciar una nueva sesión repitiendo los de Creación de sesión que están más arriba.

Destruyendo el ICE stream transport

- Usar `pj_ice_strans_destroy()` para destruir el ICE *stream transport*. Esto iniciará el procedimiento de liberación de TURN (si TURN es usado), y por último cerrará los *sockets* así como todos los recursos reservados por la instancia `ice_strans`.
- Notar que la destrucción de `ice_strans` no se completará inmediatamente si TURN es usado (ya que necesita esperar al proceso de liberación), por lo tanto es importante que el *polling* del temporizador *heap* y de *ioqueue* se siga haciendo. El programa no será notificado cuando la destrucción de `ice_strans` termine, puede asumir que el objeto `ice_strans` no será usado más tan pronto como se llame a `pj_ice_strans_destroy()`.

La información aplica al release 1.1 de PJSIP (16/03/2009).

Mantenimiento de vida (Keep-alive)

- Una vez que el `ice_strans` es creado, el *keep-alive* de STUN y TURN se hará automáticamente e internamente. El período por defecto de *keep-alive* de STUN es **15** segundos (PJ_STUN_KEEP_ALIVE_SEC), y el de TURN es también **15** segundos (PJ_TURN_KEEP_ALIVE_SEC).

Cambio en la IP

- Los cambios en la IP mapeada por STUN son manejados automáticamente por `ice_strans` a través de intercambios de *keep-alive* de STUN, aunque actualmente no hay ningún *callback* para notificar al programa de este evento. Invocando a `pj_ice_strans_enum_cands()` se puede obtener la IP actualizada.

Los cambios en las IPs de las interfaces locales no son detectados.

- Si el cambio en la IP es importante para el programa, actualmente sólo se puede recomendar al programa que implemente esta detección, y reinicie la sesión ICE o destruya/recree el `ice_strans` una vez que detecte este cambio en la IP.

Tiempo de negociación

- La negociación ICE puede tomar desde decenas a cientos de mili segundos en completar. El tiempo que demora en completar la negociación ICE depende del número de candidatos a través de todos los componentes en un `ice_strans`, el tiempo de *round-trip* entre los dos extremos ICE, así como el *round-trip* de señales ya que la información ICE es intercambiada usando señales. En una prueba pequeña, tomó cerca de 125 mili segundos en completar, en un escenario dónde dos extremos (SIP) estaban detrás de distintas conexiones ADSL (ambas en el Reino Unido), con dos componentes y 2 a 4 candidatos por componente. Es relevante mencionar que se usó un proxy SIP para la llamada (el proxy SIP estaba en EEUU), por lo que el tiempo de negociación también dependía del *round-trip* de señales SIP.

También notar que puede tomar segundos para que ICE reporte del fracaso en la negociación. ICE va a esperar hasta el *time-out* de las retransmisiones STUN (7 a 8 segundos).

6 Evaluación

En este capítulo se presentan pruebas que permitirán saber en qué medida se beneficia la aplicación GoalBit por el uso de ICE. Para ello se emula, como se explica en [53], los distintos tipos de NATs con y sin la solución ICE. De este modo se puede saber en que casos se logra conectividad entre dos *peers* utilizando y no utilizando ICE. Se obtienen datos del porcentaje de NATs de cada tipo que tienen los usuarios y así se sabe cuál es el beneficio que implica el uso de ICE. Para las pruebas se usó la solución con la librería Nice.

La Sección 6.1 contiene una evaluación de NATs hecha utilizando la herramienta NAT Check. Esta sección también explica cómo se hacen y por qué se hacen las pruebas para clasificar a los NATs. La Sección 6.2.1 contiene datos sobre NATs recabados por Technische Universität München con una herramienta Web. La sección 6.2.2 contiene datos recabados sobre nuestro entorno de prueba con esta misma herramienta Web. En la Sección 6.3 están los resultados de pruebas de libgoalbit con y sin la integración de Nice en distintas configuraciones de NAT que generamos en nuestro entorno de prueba.

6.1 Evaluación de las técnicas de NAT Traversal utilizando NAT Check

Para evaluar la robustez de las técnicas descritas de HP en TCP y UDP en varios NATs existentes, Bryan Ford[59] implementó y distribuyó un programa de prueba llamado NAT Check (http://midcom-p2p.sourceforge.net/)[36], y solicitó información a usuarios de Internet acerca de sus NATs.

El principal propósito de NAT Check es probar las dos propiedades más importantes de los NATs para permitir HP confiable en UDP y TCP: traducción de *endpoint* consistente conservando la identidad como se describe en secciones anteriores, y descartar silenciosamente los TCP SYNs entrantes en vez de rechazarlos mediante errores RST o ICMP. Además NAT Check chequea si el NAT soporta traducción de *Hairpin,* y si el NAT filtra cualquier tráfico no solicitado. Esta última propiedad no afecta el HP pero provee un indicador útil de la política del *firewall* del NAT.

NAT Check no intenta probar cada faceta relevante del comportamiento del NAT de manera individual: una amplia variedad de sutiles diferencias en el comportamiento son conocidas, algunas de las cuales son dificultosas para probar de manera confiable. En lugar de esto, NAT Check simplemente intenta responder la pregunta, "¿cómo se comportan las técnicas propuestas de HP en NATs bajo circunstancias típicas de red?"

6.1.1. Método de prueba

NAT *Check* consiste en un programa cliente que corre en una máquina detrás de un NAT a ser probado, y tres servidores con IPs bien conocidas. El cliente que opera con los tres servidores para probar el comportamiento del NAT relevante al HP para TCP y UDP. El programa cliente es pequeño y relativamente portable, corriendo actualmente en Windows, Linux, BSD, y MAC OS X. Las máquinas que sirven de *hosting* para los servidores bien conocidos corren FreeBSD.

6.1.2. Prueba de UDP

Para probar el comportamiento del NAT para UDP, el cliente abre un *socket* y se ata a un puerto UDP local, luego sucesivamente envía pedidos de *"ping"* a los servidores **1** y **2**, como se muestra en la Figura 6.1. Cada servidor responde al *ping* del cliente con una respuesta que incluye el *endpoint* UDP público del cliente: la IP del cliente y el puerto UDP observado por el servidor. Si los dos servidores reportan el mismo *endpoint* público para el cliente, NAT Check asume que el NAT preserva correctamente la identidad del *endpoint* del cliente, satisfaciendo la precondición primaria para un UDP HP confiable.

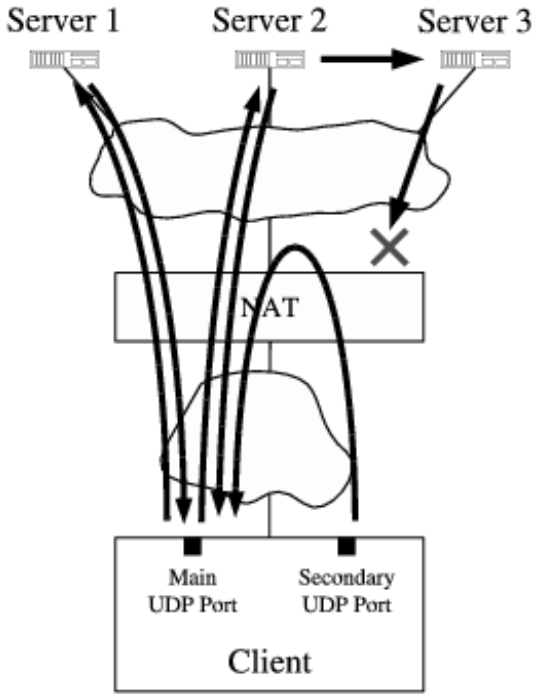

Figura 6.1: Método de prueba para UDP de NAT Check

Cuando el servidor **2** recibe un pedido UDP del cliente, además de responder directamente al cliente envía el pedido al servidor **3**, el cual a su vez responde al cliente desde su propia IP. Si el *firewall* del NAT filtra apropiadamente el tráfico entrante no solicitado en cada sesión, entonces el cliente nunca ve las respuestas del servidor **3**, incluso cuando son dirigidas al mismo puerto público que las de los servidores **1** y **2**.

Para probar el soporte de traducción de *Hairpin,* el cliente simplemente abre un segundo *socket* UDP en un puerto local diferente y lo utiliza para enviar mensajes al *endpoint* público representando el primer *socket* UDP del cliente, como fue reportado por el servidor **2**. Si estos mensajes llegan al primer *endpoint* privado del cliente, el NAT soporta traducción de *Hairpin.*

6.1.3. Prueba de TCP

La prueba de TCP sigue un patrón similar al de UDP. El cliente utiliza un sólo puerto local TCP para iniciar una sesión saliente con el servidor **1** y **2**, y verifica si los *endpoint* públicos reportados por el servidor **1** y **2** son los mismos, que es la primera precondición para un TCP HP confiable.

La respuesta del NAT a solicitudes de conexión entrantes no solicitadas también impacta la velocidad y confiabilidad del TCP HP, por esto NAT Check también prueba este comportamiento. Cuando el servidor **2** recibe el pedido del cliente, en vez de responderle inmediatamente envía el mismo pedido al servidor **3** y espera que el servidor **3** le envíe una señal de que siga adelante. Cuando el servidor **3** recibe el pedido reenviado, intenta iniciar una conexión entrante con el *endpoint* público del cliente. El servidor **3** espera hasta **5** segundos para que esta conexión tenga éxito o falle, y si el intento de conexión está todavía en progreso después de **5** segundos, el servidor **3** responde al servidor **2** con una señal de siga adelante y continua esperando hasta **20** segundos. Una vez que el cliente finalmente recibe la respuesta del servidor **2** (que el servidor **2** demoró en mandar porque estaba esperando la señal de siga adelante del servidor **3**), el cliente intenta una conexión saliente con el servidor **3**, causando una apertura simultánea de TCP con el servidor **3**.

Lo que sucede durante esta prueba depende del comportamiento del NAT como se detalla a continuación. Si el NAT descarta los paquetes SYN entrantes no solicitados del servidor **3**, entonces nada pasa en el *socket* del cliente que está escuchando durante el período de **5** segundos antes de que el servidor **2** responda al cliente. Cuando el cliente finalmente inicia su propia conexión con el servidor **3**, abriendo un agujero en el NAT, el intento triunfa inmediatamente. Si por otro lado, el NAT no descarta el SYN entrante no solicitado del servidor **3** pero les permite pasar (lo que está bien para HP pero no es ideal por seguridad), entonces el cliente recibe una conexión TCP entrante en el *socket* que estaba escuchando antes de recibir la respuesta del servidor **2**. Finalmente, si el NAT rechaza activamente el paquete SYN entrante no solicitado del servidor **3** enviando como respuesta un paquete TCP RST, entonces el servidor **3** se rinde y los intentos siguientes del cliente de conectar con el servidor **3** fallan.

Para probar la traducción de *Hairpin* para TCP, el cliente simplemente utiliza un puerto TCP local secundario para intentar conectarse con el *endpoint* público correspondiente a su puerto TCP primario, del mismo modo que para UDP.

6.1.4. Resultados de las pruebas

Los datos del NAT Check que recogimos consisten en 380 muestras cubriendo una variedad de enrutadores NAT de *hardware* de 68 fabricantes y del NAT incluido dentro de 8 SOs. Sólo 335 del total de muestras incluyen resultados para la traducción de *Hairpin* UDP, y sólo 286 muestras incluyen resultados para TCP, debido a que implementamos estas funcionalidades en las últimas versiones de NAT Check después de haber comenzado a recoger datos. La información se resume en la Tabla 1, esta sólo lista fabricantes de manera individual para los fabricantes que al menos tuvieron 5 muestras. La variación en los resultados de las pruebas para algunos fabricantes puede ser debida a varios factores, como ser distintos tipos de productos del mismo fabricante, distintas versiones de *software* o *firmware* en la misma implementación de NAT, distintas configuraciones, y probablemente ocasionales errores en la prueba por parte de NAT Check.

	UDP				TCP			
NAT Hardware	HP		Hairpin		HP		Hairpin	
Linksys	45/46	98%	5/42	12%	33/38	87%	3/38	8%
Netgear	31/37	84%	3/35	9%	19/30	63%	0/30	0%
D-Link	16/21	76%	11/21	52%	9/19	47%	2/19	11%
Draytek	2/17	12%	3/12	25%	2/7	29%	0/7	0%
Belkin	14/14	100%	1/14	7%	11/11	100%	0/11	0%
Cisco	12/12	100%	3/9	33%	6/7	86%	2/7	29%
SMC	12/12	100%	3/10	30%	8/9	89%	2/9	22%
ZyXEL	7/9	78%	1/8	13%	0/7	0%	0/7	0%
3Com	7/7	100%	1/7	14%	5/6	83%	0/6	0%
NATs basados en SO								
Windows	31/33	94%	11/32	34%	16/31	52%	28/31	90%
Linux	26/32	81%	3/25	12%	16/24	67%	2/24	8%
FreeBSD	7/9	78%	3/6	50%	2/3	67%	1/1	100%
All Vendors	310/380	82%	80/335	24%	184/286	64%	37/286	13%

Tabla 1: Muestras de usuarios sobre el soporte de HP UDP y TCP en NATs

De 380 muestras para UDP, en 310 (82%) el NAT consistentemente tradujo el *endpoint* privado del cliente, indicando compatibilidad básica con UDP HP. El soporte para traducción de *hairpin* es mucho menos común, de 335 muestras que incluyen resultados sobre traducción *hairpin* UDP, sólo 80 (24%) la soportan.

De 286 muestras para TCP, 184 (64%) muestran compatibilidad con HP para TCP: el NAT traduce consistentemente el *endpoint* TCP privado del cliente, y no devuelve paquetes RST en respuesta a intentos de conexión entrantes no solicitados. El soporte de traducción de *hairpin* es poco común: sólo 37 (13%) de las muestras soportan TCP para *hairpin.*

Como estas muestras fueron generadas por una comunidad de voluntarios “auto-seleccionados”, no constituyen una muestra aleatoria y por lo tanto no necesariamente representan la verdadera distribución de NATs en uso. El resultado sin embargo es alentador: parece ser que la mayoría de los NATs en funcionamiento más comunes soportan HP UDP y TCP al menos en escenarios de NAT de un sólo nivel.

6.2 Clasificación de los NAT utilizando NATAnalizer

6.2.1. Resultados del Technische Universität München

Según [60] que tiene estadísticas para 2680 NATs, obtenidos entre Diciembre de 2011 y diciembre de 2012, en forma anónima, a través de una aplicación web que realiza ciertas pruebas sobre los clientes se obtienen los siguientes resultados usando el algoritmo STUN:

- casi un 50% de los NATs son PAR,
- cerca del 12% son SYM,
- cerca del 10% son FC,
- menos de un 5% son AR.

Por lo tanto la probabilidad de las distintas combinaciones sería PAR-PAR casi 25%, PAR-SYM cerca de 5%, PAR-FC cerca de 3%, SYM-SYM cerca de 1% y PAR-AR cerca de 1%.

En las Figuras 6.2, 6.3 y 6.4 se puede ver un resumen gráfico de esta información.

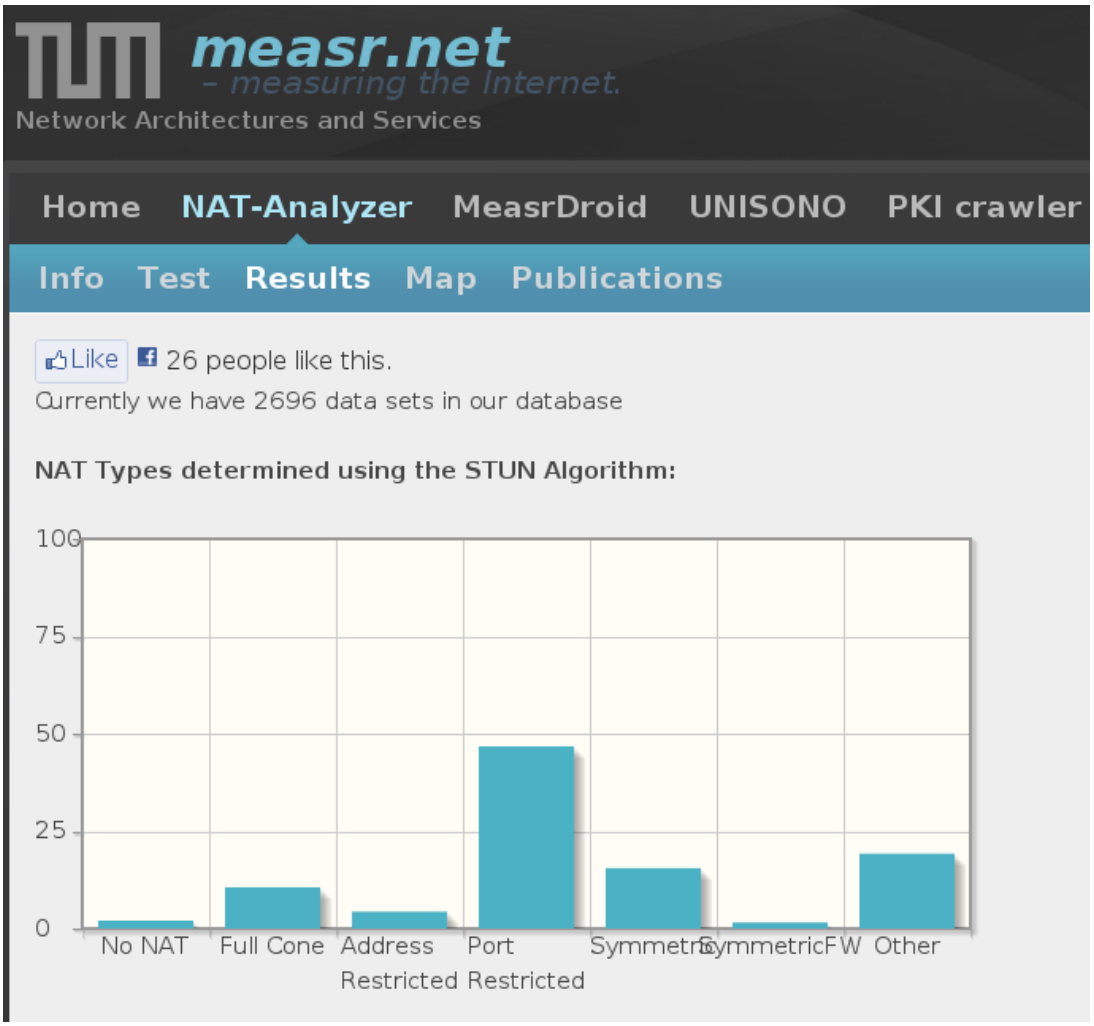

Figura 6.2: Tipo de NAT según STUN

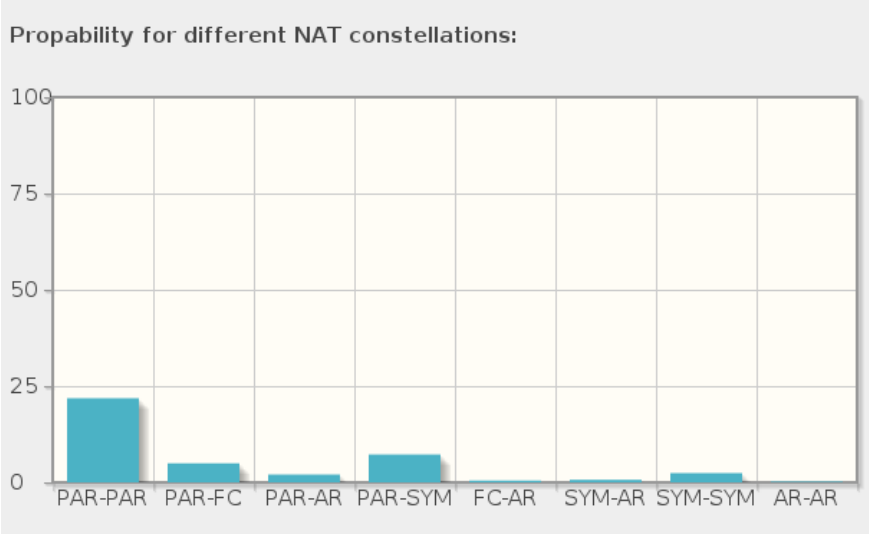

Figura 6.3: Probabilidad de distintas combinaciones de NAT

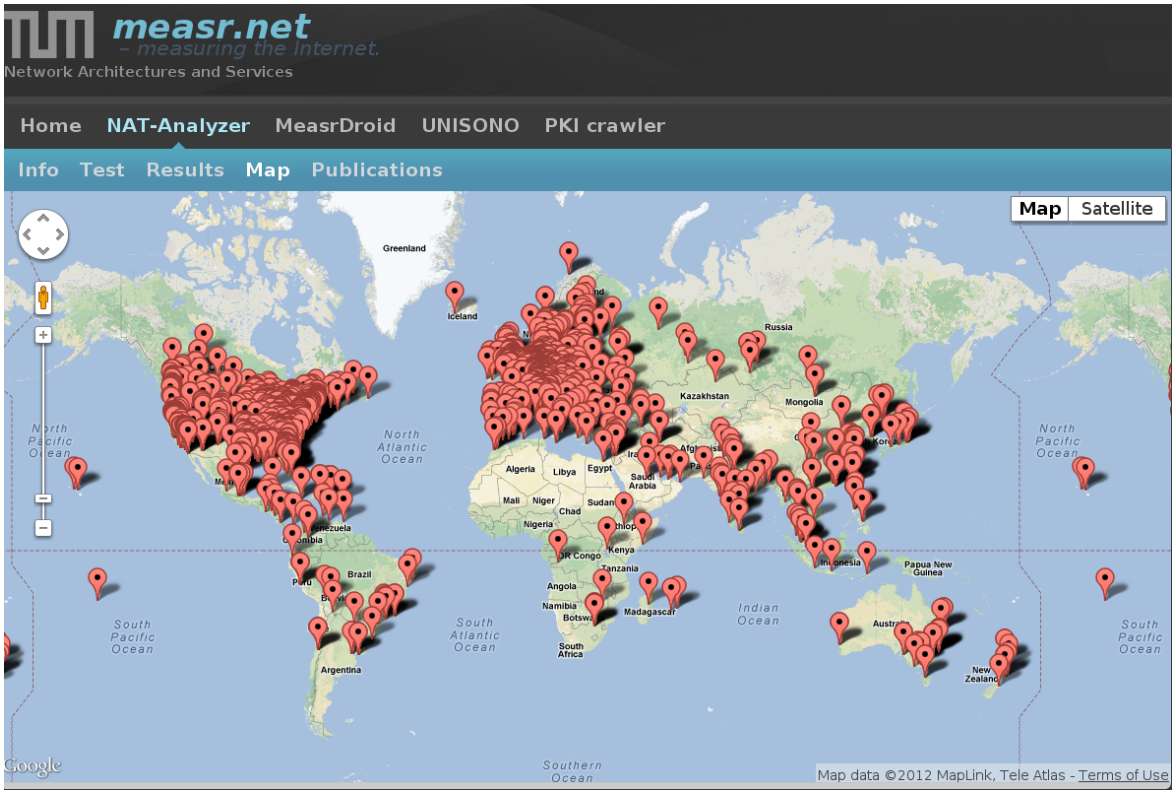

Figura 6.4: Mapa de datos de muestreo

6.2.2. Resultado para nuestro ambiente de pruebas

Utilizando la herramienta web NATAnalyzer se probó para cada uno de los *software routers* que emulamos o los *hardware routers* que disponíamos, cuáles eran los resultados que se obtenían. A partir de ello se elaboró la siguiente tabla que resume los resultados obtenidos:

	UPnP	STUN	UDP Binding	TCP Binding	UDP Mapping	TCP Mapping	SIP ALG	FTP ALG	UDP Hole Punching	TCP Hole Punching	UDP Timeout
hw Router 1 (TP-Link N750)	No	PAR	*EIB*, predicción de puerto fácil.	*EIB*, predicción de puerto fácil.	IP externa distinta de IP local. Conserva puertos de origen externos en cada conexión .	IP externa distinta de IP local. Conserva puertos de origen externos en cada conexión.	Paquete SIP INVITE inicial no fue modificado en el camino al servidor. No hay SIP ALG.	Comando de puerto FTP inicial fue modificado. Seguramente el NAT implementa FTP ALG.	Prueba TTL alto OK. Prueba TTL bajo OK. Prueba silencioso OK.	Prueba TTL alto falla.	29 segs.
sw Router PAR	No	PAR	*EIB*, predicción de puerto fácil.	*EIB*, predicción de puerto fácil.	IP externa distinta de IP local. Conserva puertos de origen externos en cada conexión .	IP externa distinta de IP local. Conserva puertos de origen externos en cada conexión.	Paquete SIP INVITE inicial no fue modificado en el camino al servidor. No hay SIP ALG.	Comando de puerto FTP inicial no fue modificado en el camino al servidor. No hay FTP ALG.	Prueba TTL alto OK. Prueba TTL bajo OK. Prueba silencioso OK.	Prueba TTL alto falla.	29 segs.
sw Router SYM	No	SYM	EDB, predicción de puertos puede ser difícil	EDB, predicción de puertos puede ser difícil	Sin resultados	Sin resultados	Paquete SIP INVITE inicial no fue modificado en el camino al servidor. No hay SIP ALG.	Comando de puerto FTP inicial no fue modificado en el camino al servidor. No hay FTP ALG.	Prueba TTL alto falla. Prueba TTL bajo falla. Prueba silencioso falla.	Prueba TTL alto falla.	29 segs.
sw Router FC	No	FC	EIB, predicción de puertos fácil	EIB, predicción de puertos fácil	IP externa distinta de IP local. Conserva puertos de origen externos en cada conexión.	IP externa distinta de IP local. Conserva puertos de origen externos en cada conexión.	No SIP ALG	No FTP ALG	Prueba TTL alto y bajo OK Prueba Silencioso falla.	Prueba TTL alto OK.	60 segs.

	UPnP	STUN	UDP Binding	TCP Binding	UDP Mapping	TCP Mapping	SIP ALG	FTP ALG	UDP Hole Punching	TCP Hole Punching	UDP Time out
hw Router (Linksys WRT54 G2)	Si	PAR	EIB, predicción de puertos fácil	EIB, predicción de puertos fácil	IP externa distinta de IP local. Conserva puertos de origen externos en cada conexión.	IP externa distinta de IP local. Conserva puertos de origen externos en cada conexión.	No SIP ALG	NAT implementa un FTP-ALG	Prueba TTL alto y bajo OK Prueba Silencioso ok.	Prueba TTL alto y bajo OK.	60 segs.

*El valor de 29 segundos para UDP *Timeout* es un valor común. Sin embargo, si los programas no envían suficientes *keep alive* puden haber problemas. Para 60 segs. No debería haber problemas.

*EIB: *Endpoint Independent Binding*

*EDB: *Endpoint Dependent Binding*

6.3 Evaluación de nuestra solución

Dados los resultados obtenidos en 6.2.1 entendemos que incluyendo las siguientes combinaciones de tipos de NAT, los resultados obtenidos van a ser suficientemente representativos de las combinaciones de NAT más comunes de las redes domésticas:

	Tipo de NAT del enrutador	
Caso	normal peer 1	normal peer 2
1	PAR hw	PAR sw
2	PAR hw	SYM
3	PAR hw	FC
4	SYM	PAR sw
5	SYM	SYM
6	SYM	FC
7	PAR sw	PAR sw
8	PAR sw	FC
9	FC	FC

Los resultados obtenidos fueron los siguientes:

	¿Intercambian entre pares?	
Caso	Con Nice	Sin Nice
1	Si	No
2	Si	No
3	Si	No
4	No	No
5	No	No
6	Si	No
7	Si	No
8	Si	No
9	Si	No

El tiempo entre que se levanta un peer y el otro es 15 segundos. Se logra que los pares intercambien piezas en 7 de las 9 combinaciones probadas. Las combinaciones en las que se logró que los pares intercambien piezas son las que son más frecuentes según el estudio presentado en la Sección 6.2.1. Las combinaciones en las que no intercambian piezas son SYM-FC (5% de probabilidad de ocurrencia según dicho estudio) y SYM-SYM (cercano a 1% de ocurrencia según dicho estudio). Por lo tanto concluimos que se logra una mejora sensible con la integración de la librería Nice al proyecto GoalBit.

7 Conclusiones

7.1 Resultados alcanzados

Se logró:

- Entender con mayor profundidad sobre NAT, NAT-traversal, protocolos P2P, clientes, servidores, transporte de video en red.
- Integrar PJNATH pero no se realizó la extensión de UDP de libGoalBit.
- Integrar la librería libnice al cliente libgoalbit.
- Modificar el código PHP del *tracker* y la base de datos del tracker para que comunicara los nuevos datos que eran necesarios para establecer el NAT-traversal.
- Implementación de servidor de *rendezvous* para intercambio de SDPs con algoritmo de intercambio basado en interés de un *peer* en conectarse con uno especifico y otros con conectarse con él.
- Configuración con servidor STUN propio (versión corregida para correr en ambiente 64bits) o con servidor STUN de público acceso.
- Intercambio de piezas entre *peers* normales atrás de NAT

7.2 Dificultades encontradas

Las aplicaciones eran difíciles de depurar ya que cuando había un error no sabíamos si se debía a que no estábamos ejecutando la aplicación como se esperaba (el tema del tiempo en que se ejecutaba cada paso y de los *threads* nos complicó bastante), al tipo de NAT de nuestros enrutadores u otra razón.

Fue difícil entender el código de PJNATH ya que este es un módulo de un *framework* más grande, y además utiliza conceptos avanzados de programación en C como *callbacks* a los que no estábamos acostumbrados.

7.3 Lo que se planteó hacer

Nos planteamos integrar PJNATH a libgoalbit y adecuar el protocolo de libgoalbit para que se pueda usar con UDP que es el protocolo del canal que te brinda la librería PJNATH. También nos planteamos integrar uTP a la solución final. Luego nos planteamos integrar libnice a libgoalbit.

7.4 Lo que se hizo realmente

Se integró PJNATH a libgoalbit, pero como se vio que adecuar el protocolo de libgoalbit para que use UDP era muy complicado, entonces se integró libnice a libgoalbit, y de ese modo se obtuvo un canal que es TCP sobre UDP.

7.5 Posibles extensiones al trabajo

Se podría adaptar libgoalbit para que se pueda usar mediante UDP. También se podría integrar uTP a libgoalbit. En la parte de evaluación se podría:

- investigar como lograr un AR de modo de saber con mayor exactitud la mejora que se logra utilizando Nice,
- probar con otros tipos de conexiones a Internet como Banda Ancha Móvil,
- hacer un estudio con la herramienta NATAnalyzer de qué tipo de NATs tienen los actuales usuarios de GoalBit,
- probar qué sucede cuando hay múltiples niveles de NAT,
- intercambio de piezas para *broadcaster* y *superpeer* atrás de NAT.

8 Glosario

- ALG: Application Level Gateway
- AR: Address Restricted NAT o Restricted Cone NAT
- CDN: Content Delivery Network
- CIDR: Classless Inter Domain Routing
- DNS: Domain Name System
- FC: Full Cone NAT
- GA: Global Address
- GMP: GoalBit Media Player
- GP: Global Port
- GPL: General Public License
- HP: Hole Punching
- ICE: Interactive Connectivity Establishment
- ICMP: Internet Control Message Protocol
- IANA: Internet Assigned Numbers Authority
- IETF: Internet Engineering Task Force
- IP: Internet Protocol
- ISN: Initial Sequence Number
- ISP: Internet Service Provider
- LSR: Loose Source Route
- NAPT: Network Address and Port Transaltion
- NAT: Network Address Translation
- NB: NAT Behavior
- NUTSS: Son las siglas de lo que se considera sus componentes principales en la arquitectura, NAT, URI, Tunnel, SIP, y STUNT
- P2P: Peer to Peer
- PAR: Port-restricted cone NAT
- RFC: Request For Comments
- RTCP: Real-Time Transport Control Protocol
- RTP: Real time Transport Protocol
- RTS: Request To Send
- SDP: Session Description Protocol
- SIP: Session Initiation Protocol
- SO: Sistema Operativo
- SOHO: Small Office/ Home Office
- SRTP: Secure Real time Transport Protocol
- STUN: Es un protocolo que utiliza un host para determinar cuál puerto UDP y IP global se le fue asignado por el NAT más lejano
- STUNT: Es un protocolo que extiende STUN para incluir TCP

- SYM: Symmetric NAT
- TCP: Transmission Control Protocol
- TTL: Time To Live
- TU: TCP/UDP
- TURN: Traversal Using Relay NAT
- UDP: User Datagram Protocol
- UPnP: Universal Plug and Play
- URI: Uniform Resource Identifier

Bibliografía

[1] Zhou Hu, "NAT Traversal Techniques and Peer-to-Peer Applications", HUT T-110.551 Seminar on Internetworking, 2005-04-26/27
[2] K. Egevang, P. Francis, "The IP Network Address Translator (NAT)", IETF Network Working Group, RFC1631, May 1994
[3] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris and Scott Shenker, "Middleboxes No Longer Considered Harmful", USENIX Association, OSDI ’04: 6th Symposium on Operating Systems Design and Implementation
[4] Phillipa Gill, Martin Arlitt, Zongpeng Li and Anirban Mahanti, "YouTube Traffic Characterization: A View From the Edge", IMC '07 Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, Pages 15-28, October 2007
[5] M. Boucadair, R. Penno and D. Wing, "UPnP IGD-PCP Interworking Function", IETF PCP Working Group, October 2010
[6] Andrew S. Tanenbaum, "Redes de Computadoras (TCP: establecimiento de una conexión)", 4a edición, Páginas 539-540, 2003
[7] University of Southern California, "Transmission Control Protocol", IETF for Defense Advanced Research Projects Agency, RFC793, September 1981
[8] J. Rosenberg, J. Weinberger, C. Huitema and R. Mahy, "Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)", IETF Network Working Group, RFC3489, March 2003
[9] J. Saltzer, "On the Naming and Binding of Network Destinations", IETF Network Working Group, RFC1498, August 1993
[10] V. Fuller, T. Li, J. Yu and K. Varadhan, "Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy", IETF Network Working Group, RFC1519, September 1993
[11] Paul Francis, "Is the Internet Going NUTSS?", IEEE Computer Society, IEEE INTERNET COMPUTING, November/December 2003
[12] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, E. Lear, "Address Allocation for Private Internets", IETF Network Working Group, RFC1918, February 1996
[13] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley and E. Schooler, "SIP: Session Initiation Protocol", IETF Network Working Group, RFC3261, June 2002
[14] M. Handley, V. Jacobson and C. Perkins , "SDP: Session Description Protocol", IETF Network Working Group, RFC4566, July 2006
[15] "GoalBit", http://goalbit.sourceforge.net, 2012
[16] María Elisa Bertinat, Daniel De Vera, Darío Padula, Franco Robledo Amoza, Pablo Rodríguez-Bocca, Pablo Romero and Gerardo Rubino, "GoalBit: The First Free and Open Source Peer-to-Peer Streaming Network", LANC '09 Proceedings of the 5th International Latin American Networking Conference , Pages 49-59 , September 2009
[17] Daniel De Vera, "GoalBit: la primer red P2P de distribución de video en vivo de código abierto y gratuita", Instituto de Computación, FACULTAD DE INGENIERÍA, 18 de Octubre de 2010
[18] "Especifiación de BitTorrent", http://wiki.theory.org/BitTorrentSpecification, October 2012
[19] "Video Lan Client", http://wiki.videolan.org, October 2012
[20] "Enhanced C Torrent", http://www.rahul.net/dholmes/ctorrent, October 2012
[21] "Open Tracker", http://erdgeist.org/arts/software/opentracker, October 2012

[22] Bryan Ford, Pyda Srisuresh and Dan Kegel, "Peer-to-Peer Communication Across Network Address Translators", USENIX Association, 2005 USENIX Annual Technical Conference, 2005
[23] P. Srisuresh and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations", IETF Network Working Group, RFC2663, August 1999
[24] P. Srisuresh, K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)", January 2001,http://www.ietf.org/rfc/rfc3022.txt
[25] L. Daigle, Ed., "IAB Considerations for UNilateral Self-Address Fixing (UNSAF) Across Network Address Translation", IETF Network Working Group, RFC3424, November 2002
[26] T. Hain, "Architectural Implications of NAT", IETF Network Working Group, RFC2993, November 2000
[27] Yuan Wei, Suguru Yoshida, Daisuke Yamada, Shigeki Goto, "A New Method for Symmetric NAT Traversal in UDP and TCP", New Zealand Network Research Workshop 2008, Asia Pacific Advanced Network 2008, 4-8 August 2008
[28] M. Bagnulo, P. Matthews, I. van Beijnum, "Stateful NAT64: Network Address and Protocol Translation from IPv6 Clients to IPv4 Servers", IETF, RFC6146, April 2011
[29] R. Mahy, P. Matthews, J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", IETF, RFC5766, April 2010
[30] J. Postel, "INTERNET CONTROL MESSAGE PROTOCOL", IETF Network Working Group, RFC792, September 1981
[31] D. Senie, "Network Address Translator (NAT)-Friendly Application Design Guidelines", IETF Network Working Group, RFC3235, January 2002
[32] M. Holdrege and P. Srisuresh, "Protocol Complications with NAT", IETF Network Working Group, RFC3027, January 2001
[33] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", IETF, RFC5245, April 2010
[34] R. Hinden and S. Deering, "IP Version 6 Addressing Architecture", IETF Network Working Group, RFC1884, December 1995
[35] J. Rosenberg and H. Schulzrinne, "An Offer/Answer Model with the Session Description Protocol (SDP)", IETF Network Working Group, RFC3264, June 2002
[36] P. Srisuresh, J. Kuthan, J. Rosenberg, A. Molitor and A. Rayhan , "Middlebox communication architecture and framework", IETF Network Working Group, RFC3303, August 2002
[37] M. Borella, J. Lo, D. Grabelsky and G. Montenegro , "Realm Specific IP: Framework", IETF Network Working Group, RFC3102, October 2001
[38] M. Borella, D. Grabelsky, J. Lo and K. Taniguchi , "Realm Specific IP: Protocol Specification", IETF Network Working Group, RFC3103, October 2001
[39] C. Huitema , "Real Time Control Protocol (RTCP) attribute in Session Description Protocol (SDP) ", IETF Network Working Group, RFC3605, October 2003
[40] G. Camarillo and J. Rosenberg , "The Alternative Network Address Types (ANAT) Semantics for the Session Description Protocol (SDP) Grouping Framework", IETF Network Working Group, RFC4091, June 2005
[41] G. Camarillo and J. Rosenberg , "Usage of the Session Description Protocol (SDP) Alternative Network Address Types (ANAT) Semantics in the Session Initiation Protocol (SIP)", IETF Network Working Group, RFC4092, June 2005
[42] J. Rosenberg, R. Mahy, P. Matthews and D. Wing, "Session Traversal Utilities for NAT (STUN)",

IETF Network Working Group, RFC5389, October 2008
[43] Saikat Guha, Yutaka Takeda, Paul Francis, "NUTSS: A SIP-based Approach to UDP and TCP Network Connectivity", SIGCOMM'04 Workshops, Portland, Oregon, USA, Aug. 30+Sept. 3, 2004
[44] "Digital Living Network Alliance", http://www.dlna.org/, October 2012
[45] "Application-Layer Traffic Optimization (alto)", https://datatracker.ietf.org/wg/alto/charter/, October 2012
[46] Yi-Wen Lai and KuangFu Lai, "Implementing nat traversal on bittorrent", Dept. of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
[47] Andrew Biggadike, Daniel Ferullo, Geoffrey Wilson, Adrian Perrig, "NATBLASTER: Establishing TCP Connections Between Hosts Behind NATs", SIGCOMM Asia Workshop 2005 Beijing, China, 2005
[48] Jeffrey L. Eppinger, "TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem", Institute for Software Research International, School of Computer Science, Carnegie Mellon University, January 2005
[49] Saikat Guha and Paul Francis, "Characterization and Measurement of TCP Traversal through NATs and Firewalls", IMC '05 Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement , Pages 18-18 , 2005
[50] "Gnutella", http://www8.cs.umu.se/~bergner/thesis/html/node32.html, October 2012
[51] "Compiling GoalBit", http://sourceforge.net/apps/mediawiki/goalbit/index.php?title=Compiling_GoalBit, October 2012
[52] "GoalBit: Starter Suite", http://sourceforge.net/apps/mediawiki/goalbit/index.php?title=Starter_Suite, October 2012
[53] "NAT Traversal Testing", https://wiki.asterisk.org/wiki/display/TOP/NAT+Traversal+Testing, September 2012
[54] "STUN Client and Server", http://sourceforge.net/projects/stun/, September 2012
[55] Michael J. Donahoo and Kenneth L. Calvert, "TCP/IP Sockets in C: Practical Guide for Programmers", Morgan Kaufmann, ISBN-13:978-1-55860-826-9, 2001
[56] "The GLib ICE implementation", http://nice.freedesktop.org/wiki/, September 2012
[57] "Using Standalone PJNATH's ICE in (non-SIP) Applications", https://trac.pjsip.org/repos/wiki/Using_Standalone_ICE, September 2012
[58] M. Handley and V. Jacobson, "SDP: Session Description Protocol", IETF Network Working Group, RFC2327, April 1998
[59] "Bryan Ford", http://www.brynosaurus.com/, October 2012
[60] "NAT-Analyzer Results", http://nattest.net.in.tum.de/results.php, November 2012