BOLT #7: Descubrimiento de canales y nodos P2P

Esta especificación describe mecanismos simples de descubrimiento de nodos, descubrimiento de canales y actualización de canales que no dependen de un tercero para difundir la información.

El descubrimiento de nodos y canales tiene dos propósitos diferentes:

- El descubrimiento de nodos permite a los nodos transmitir su ID, host y puerto, para que otros nodos puedan abrir conexiones y establecer canales de pago con ellos.
- El descubrimiento de canales permite la creación y el mantenimiento de una vista local de la topología de la red, de modo que un nodo pueda descubrir rutas a los destinos deseados.

Para soportar el descubrimiento de canales y nodos, se admiten tres *mensajes de chismes*:

- Para el descubrimiento de nodos, los pares intercambian mensajes `node_announcement`, que proporcionan información adicional sobre los nodos. Puede haber varios mensajes `node_announcement` para actualizar la información del nodo.

- Para el descubrimiento de canales, los pares en la red intercambian mensajes `channel_announcement` que contienen información sobre nuevos canales entre los dos nodos. También pueden intercambiar mensajes `channel_update`, que actualizan la información sobre un canal. Solo puede haber un `channel_announcement` válido para cualquier canal, pero se esperan al menos dos mensajes `channel_update`.

Table of Contents

- Definición del `short_channel_id`
- El mensaje `announcement_signatures`
- El mensaje `channel_announcement`
- El mensaje `node_announcement`
- El mensajae `channel_update`
- Mensajes de consulta
- Sincronización Inicial
- Retransmisión
- Tarifa de HTLC
- Podando la de Vista de Red
- Recomendaciones para el enrutamiento

Definición del `short_channel_id`

El `short_channel_id` es la descripción única de la transacción de financiación. Se construye de la siguiente manera:

1. los 3 bytes más significativos: indicando la altura del bloque
2. los siguientes 3 bytes: indicando el índice de transacción dentro del bloque
3. los 2 bytes menos significativos: indica el índice de salida que paga al canal.

El formato legible por humanos estándar para `short_channel_id` se crea imprimiendo los componentes anteriores, en el orden: altura del bloque, índice de transacción e índice de salida. Cada componente está impreso como un número decimal y separados entre sí por la letra minúscula `x`. Por ejemplo, un `short_channel_id` podría escribirse como `539268x845x1`, lo que indica un canal en la salida 1 de la transacción en el índice 845 del bloque en la altura 539268.

Racional

El formato legible para humanos del `short_channel_id`, está diseñado para que al hacer doble clic o tocarlo dos veces, se seleccione la ID completa en la mayoría de los sistemas. Los humanos prefieren el decimal cuando leen números, por lo que los componentes de identificación se escriben en decimal. Se usa la letra minúscula `x` ya que en la mayoría de las fuentes, la `x` es visiblemente más pequeña que los dígitos decimales, lo que facilita agrupar visiblemente cada componente de la identificación.

El mensaje `announcement_signatures`

Este es un mensaje directo entre los dos extremos de un canal y sirve como un mecanismo de suscripción para permitir el anuncio del canal al resto de la red. Contiene las firmas necesarias, por parte del remitente, para construir el mensaje `channel_announcement`.

1. tipo: 259 (`announcement_signatures`)
2. datos:
    - [`channel_id`:`channel_id`]
    - [`short_channel_id`:`short_channel_id`]
    - [`signature`:`node_signature`]
    - [`signature`:`bitcoin_signature`]

La voluntad del nodo iniciador de anunciar el canal se señala durante la apertura del canal configurando el bit `announce_channel` en `channel_flags` (ver BOLT #2).

Requisitos

El mensaje `announcement_signatures` se crea construyendo un mensaje `channel_announcement`, correspondiente al canal recién establecido, y firmándolo con los secretos que coinciden con el `node_id` y la `bitcoin_key` de un punto final. Una vez firmado, se puede enviar el mensaje `announcement_signatures`.

Un nodo:

- si el mensaje `open_channel` tiene el bit `announce_channel` establecido Y no se ha enviado un mensaje `shutdown`:
    - DEBE enviar el mensaje `announcement_signatures`.
        - NO DEBE enviar mensajes de `announcement_signatures` hasta `channel_ready` se ha enviado y recibido Y la transacción de financiación tiene al menos seis confirmaciones.
- de lo contrario:
    - NO DEBE enviar el mensaje `announcement_signatures`.
- tras la reconexión (una vez que se hayan cumplido los requisitos de tiempo anteriores):

        - DEBE responder al primer mensaje `announcement_signatures` con su propio mensaje `announcement_signatures`.
        - si NO ha recibido un mensaje `announcement_signatures`:
            - DEBERÍA retransmitir el mensaje `announcement_signatures`.

Un nodo receptor:

- si `short_channel_id` NO es correcto:
    - DEBERÍA enviar una `warning` y cerrar la conexión, o enviar un `error` y falla el canal.
- si `node_signature` O `bitcoin_signature` NO es correcto:
    - PUEDE enviar una 'advertencia' y cerrar la conexión, o enviar un 'error' y fallar el canal.
- si ha enviado Y recibido un mensaje `announcement_signatures` válido:
    - DEBERÍA poner en cola el mensaje `channel_announcement` para sus pares.
- si no ha enviado `channel_ready`:
    - PUEDE diferir el manejo de las firmas de anuncio hasta que haya enviado `channel_ready`
    - de lo contrario:
        - DEBE ignorarlo.

Racional

La razón para permitir el aplazamiento de un `announcement_signatures` prematuro es que una versión anterior de la especificación no requería esperar a que se bloqueara la recepción de la financiación: aplazarlo en lugar de ignorarlo permite la compatibilidad con este comportamiento.

El mensaje `channel_announcement`

Este mensaje de chismes contiene información sobre la propiedad de un canal. Vincula cada clave de Bitcoin en cadena a la clave de nodo Lightning asociada, y viceversa. El canal no se puede utilizar prácticamente hasta que al menos un lado haya anunciado sus niveles de tarifas y su vencimiento, usando `channel_update`.

Probar la existencia de un canal entre `node_1` y `node_2` requiere:

1. demostrar que la transacción de financiación paga a `bitcoin_key_1` y `bitcoin_key_2`
2. probar que `node_1` posee `bitcoin_key_1`
3. probar que `node_2` posee `bitcoin_key_2`

Suponiendo que todos los nodos conocen los resultados de las transacciones no gastadas, la primera prueba se logra cuando un nodo encuentra el resultado proporcionado por `short_channel_id` y verifica que, de hecho, es un resultado de transacción de financiación P2WSH para esas claves especificadas en [BOLT #3](03 -transactions.md#funding-transaction-output).

Las dos últimas pruebas se logran a través de firmas explícitas: `bitcoin_signature_1` y `bitcoin_signature_2` se generan para cada `bitcoin_key` y se firma cada uno de los `node_id` correspondientes.

También es necesario demostrar que `node_1` y `node_2` están de acuerdo con el mensaje de anuncio: esto se logra al tener una firma de cada `node_id` (`node_signature_1` y `node_signature_2`) firmando el mensaje.

1. type: 256 (`channel_announcement`)
2. data:
    - [`signature`:`node_signature_1`]
    - [`signature`:`node_signature_2`]
    - [`signature`:`bitcoin_signature_1`]
    - [`signature`:`bitcoin_signature_2`]
    - [`u16`:`len`]
    - [`len*byte`:`features`]
    - [`chain_hash`:`chain_hash`]
    - [`short_channel_id`:`short_channel_id`]
    - [`point`:`node_id_1`]
    - [`point`:`node_id_2`]
    - [`point`:`bitcoin_key_1`]
    - [`point`:`bitcoin_key_2`]

Requisitos

El nodo de origen:

- DEBE establecer `chain_hash` en el hash de 32 bytes que identifica de forma única la cadena en la que se abrió el canal:
    - para la *cadena de bloques de Bitcoin*:
        - DEBE establecer el valor `chain_hash` (codificado en hexadecimal) igual a `6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000`.
- DEBE establecer `short_channel_id` para hacer referencia a la transacción de financiación confirmada, como se especifica en BOLT #2.
    - Nota: la salida correspondiente DEBE ser un P2WSH, como se describe en BOLT #3.
- DEBE establecer `node_id_1` y `node_id_2` en las claves públicas de los dos nodos que operan el canal, de modo que `node_id_1` sea lexicográficamente menor de las dos claves comprimidas ordenadas en orden lexicográfico ascendente.
- DEBE establecer `bitcoin_key_1` y `bitcoin_key_2` en `node_id_1` y `node_id_2` respectivamente `funding_pubkey`s.
- DEBE calcular el hash doble-SHA256 `h` del mensaje, comenzando en el desplazamiento 256, hasta el final del mensaje.
    - Nota: el hash omite las 4 firmas, pero aplica un hash al resto del mensaje, incluidos los campos futuros añadidos al final.
- DEBE establecer `node_signature_1` y `node_signature_2` en firmas válidas del hash `h` (usando los respectivos secretos de `node_id_1` y `node_id_2`).
- DEBE establecer `bitcoin_signature_1` y `bitcoin_signature_2` en firmas válidas del hash `h` (utilizando los respectivos secretos de `bitcoin_key_1` y `bitcoin_key_2`).
- DEBE establecer `features` en función de las funciones que se negociaron para este canal, de acuerdo con BOLT #9
- DEBE establecer `len` en la longitud mínima requerida para contener los `feature bits` que establece.

El nodo receptor:

- DEBE verificar la integridad Y la autenticidad del mensaje verificando las firmas.

- si hay un bit par desconocido en el campo `features`:
  - NO DEBE intentar enrutar mensajes a través del canal.
- si la salida de `short_channel_id` NO corresponde a un P2WSH (usando `bitcoin_key_1` y `bitcoin_key_2`, como se especifica en BOLT #3) O la salida se gasta:
  - DEBE ignorar el mensaje.
- si el `chain_hash` especificado es desconocido para el receptor:
  - DEBE ignorar el mensaje.
- de lo contrario:
  - si `bitcoin_signature_1`, `bitcoin_signature_2`, `node_signature_1` O `node_signature_2` no son válidos O NO son correctos:
    - DEBERÍA enviar una `warning`.
    - PUEDE cerrar la conexión.
    - DEBE ignorar el mensaje.
  - de lo contrario:
    - si `node_id_1` O `node_id_2` están en la lista negra:
      - DEBE ignorar el mensaje.
    - de lo contrario:
      - si la transacción a la que se hace referencia NO fue previamente anunciada como canal:
        - DEBERÍA poner en cola el mensaje para su retransmisión.
        - PUEDE elegir NO para mensajes más largos que la longitud mínima esperada.
    - si ha recibido previamente un `channel_announcement` válido, para la misma transacción, en el mismo bloque, pero para un `node_id_1` o `node_id_2` diferente:
      - DEBE incluir en la lista negra `node_id_1` y `node_id_2` del mensaje anterior, así como este `node_id_1` y `node_id_2` Y olvidar cualquier canal conectado a ellos.
    - de lo contrario:
      - DEBE almacenar este `channel_announcement`.
- una vez que su producción de fondos se haya gastado O reorganizado:
  - DEBE olvidar un canal después de un retraso de 12 bloques.

Racional

Se requiere que ambos nodos firmen para indicar que están dispuestos a enrutar otros pagos a través de este canal (es decir, ser parte de la red pública); Requerir sus firmas de bitcoin demuestra que controlan el canal.

La lista negra de los nodos conflictivos no permite múltiples anuncios diferentes. Tales anuncios conflictivos nunca deben ser transmitidos por ningún nodo, ya que esto implica que las claves se han filtrado.

Si bien los canales no deben anunciarse antes de que sean lo suficientemente profundos, el requisito contra el retransmisión solo se aplica si la transacción no se ha movido a un bloque diferente.

Para evitar almacenar mensajes excesivamente grandes, pero aún así permitir una expansión futura razonable, los nodos pueden restringir la retransmisión (tal vez estadísticamente).

Las nuevas características del canal son posibles en el futuro: las características compatibles hacia atrás (u opcional) tendrán `feature bits` *impares*, mientras que las características incompatibles tendrán `feature bits` *pares* ("It's OK to be odd!").

Se utiliza un retraso de 12 bloques al olvidar un canal en la `funding output` para permitir que un nuevo `channel_announcement` propague que indica que este canal fue unido.

El mensaje `node_announcement`

Este mensaje de chismes permite que un nodo indique datos adicionales asociados con él, además de su clave pública. Para evitar ataques triviales de denegación de servicio, se ignoran los nodos no asociados con un canal ya conocido.

1. type: 257 (`node_announcement`)
2. data:
    - [`signature`:`signature`]
    - [`u16`:`flen`]
    - [`flen*byte`:`features`]
    - [`u32`:`timestamp`]
    - [`point`:`node_id`]
    - [`3*byte`:`rgb_color`]
    - [`32*byte`:`alias`]
    - [`u16`:`addrlen`]
    - [`addrlen*byte`:`addresses`]

`timestamp` permite ordenar los mensajes, en el caso de múltiples anuncios. `rgb_color` y `alias` permiten que los servicios de inteligencia asignen colores a los nodos como el negro y apodos geniales como 'IRATEMONK' y 'WISTFULTOLL'.

`addresses` permite que un nodo anuncie su voluntad de aceptar conexiones de red entrantes: contiene una serie de `address descriptor` para conectarse al nodo. El primer byte describe el tipo de dirección y es seguido por el número apropiado de bytes para ese tipo.

Se definen los siguientes tipos de `address descriptor`:

- `1`: ipv4; data = `[4:ipv4_addr][2:port]` (longitud 6)
- `2`: ipv6; data = `[16:ipv6_addr][2:port]` (longitud 18)
- `3`: Obsoleto (longitud 12). Se utiliza para contener servicios de cebolla Tor v2.
- `4`: Tor v3 onion service; data = `[35:onion_addr][2:port]` (longitud 37)
    - version 3 (prop224) direcciones de servicio de cebolla; Codifica: `[32:32_byte_ed25519_pubkey] || [2:checksum] || [1:version]`, where `checksum = sha3(".onion checksum" || pubkey || version)[:2]`.
- `5`: nombre del host DNS; data = `[1:hostname_len][hostname_len:hostname][2:port]` (longitud hasta 258)
    - `hostname` los bytes DEBEN ser caracteres ASCII.
    - Los caracteres que no son ASCII DEBEN codificarse con Punycode: https://en.wikipedia.org/wiki/Punycode

Requisitos

El nodo de origen:

- DEBE configurar `timestamp` para que sea mayor que cualquier `node_announcement` anterior que haya creado anteriormente.
  - PUEDE basarse en una marca de tiempo UNIX.
- DEBE establecer `signature` en la firma de doble-SHA256 del paquete restante completo después de `firma` (utilizando la clave proporcionada por `node_id`).
- PUEDE establecer `alias` Y `rgb_color` para personalizar su apariencia en mapas y gráficos.
  - Nota: el primer byte de `rgb_color` es el valor rojo, el segundo byte es el valor verde y el último byte es el valor azul.
- DEBE establecer `alias` en una cadena UTF-8 válida, con cualquier byte final de `alias` igual a 0.
- DEBE llenar `addresses` con un descriptor de dirección para cada dirección de red pública que espera conexiones entrantes.
- DEBE establecer `addrlen` en el número de bytes en `addresses`.
- DEBE colocar los descriptores de direcciones en orden ascendente.
- NO DEBE colocar ningún descriptor de dirección de tipo cero en ninguna parte.
- DEBERÍA usar la ubicación solo para alinear campos que siguen a `addresses`.
- NO DEBE crear un descriptor de dirección `type 1`, `type 2` o `type 5` con `port` igual a 0.
- DEBERÍA asegurarse de que `ipv4_addr` Y `ipv6_addr` sean direcciones enrutables.
- DEBE configurar `features` de acuerdo con BOLT #9
- DEBERÍA configurar `flen` a la longitud mínima requerida para contener los bits de `features` que establece.
- NO DEBERÍA anunciar un servicio de cebolla Tor v2.
- NO DEBE anunciar más de un nombre de host DNS `type 5`.

El nodo receptor:

- si `node_id` NO es una clave pública comprimida válida:
  - DEBERÍA enviar una `warning`.
  - PUEDE cerrar la conexión.
  - NO DEBE seguir procesando el mensaje.
- si `signature` NO es una firma válida (usando `node_id` del doble-SHA256 de todo el mensaje que sigue al campo `signature`, incluido cualquier campo futuro añadido al final):
  - DEBERÍA enviar una `warning`.
  - PUEDE cerrar la conexión.
  - NO DEBE seguir procesando el mensaje.
- si el campo `features` contiene *bits pares desconocidos*:
  - NO DEBE conectarse al nodo.
  - A menos que pague una factura BOLT #11 que no tenga los mismos bits establecidos, NO DEBE intentar enviar pagos *al* nodo.
  - NO DEBE enrutar un pago *a través* del nodo.
- DEBE ignorar el primer 'descriptor de dirección' que NO coincide con los tipos definidos anteriormente.
- si `addrlen` es insuficiente para contener los descriptores de dirección de los tipos conocidos:
  - DEBERÍA enviar una `warning`.
  - PUEDE cerrar la conexión.
- si `port` es igual a 0:

    - DEBE ignorar `ipv6_addr` O `ipv4_addr` O `hostname`.
- si `node_id` NO se conoce previamente de un mensaje `channel_announcement`, O si `timestamp` NO es mayor que el último `node_announcement` recibido de este `node_id`:
    - DEBE ignorar el mensaje.
- de lo contrario:
    - si `timestamp` es mayor que el último `node_announcement` recibido de este `node_id`:
        - DEBERÍA poner en cola el mensaje para su retransmisión.
        - PUEDE elegir NO poner en cola mensajes más largos que la longitud mínima esperada.
- PUEDE usar `rgb_color` Y `alias` para hacer referencia a los nodos en las interfaces.
    - DEBERÍA insinuar sus orígenes autofirmados.
- DEBE ignorar los servicios de cebolla de Tor v2.
- si se anuncia más de una dirección `type 5`:
    - DEBE ignorar los datos adicionales.
    - NO DEBE reenviar el `node_announcement`.

Racional

Las nuevas funciones de nodo son posibles en el futuro: las compatibles con versiones anteriores (u opcionales) tendrán *bits* de `feature` *impares*, las incompatibles tendrán _bits de `feature` *par*. Estos se propagarán normalmente; Los bits de características incompatibles aquí se refieren a los nodos, no al mensaje `node_announcement` en sí.

Es posible que se agreguen nuevos tipos de direcciones en el futuro; como los descriptores de direcciones deben ordenarse en orden ascendente, los desconocidos pueden ignorarse con seguridad. En el futuro también se pueden agregar campos adicionales más allá de `addresses`, con relleno opcional dentro de `addresses`, si requieren cierta alineación.

Consideraciones de Seguridad para los Alias de Nodo

Los alias de nodo son definidos por el usuario y proporcionan una vía potencial para ataques de inyección, tanto durante el proceso de representación como durante la persistencia.

Los alias de nodos siempre deben desinfectarse antes de mostrarse en contextos HTML/Javascript o cualquier otro marco de representación interpretado dinámicamente. De manera similar, considere usar declaraciones preparadas, validación de entrada, y escapar carácteres para proteger contra vulnerabilidades de inyección y motores de persistencia que admiten SQL u otros lenguajes de consulta interpretados dinámicamente.

- Stored and Reflected XSS Prevention
- DOM-based XSS Prevention
- SQL Injection Prevention

No seas como la escuela de Little Bobby Tables.

El mensajae `channel_update`

Una vez que se ha anunciado inicialmente un canal, cada lado anuncia de forma independiente las tarifas y el delta de vencimiento mínimo que requiere para retransmitir los HTLC a través de este canal. Cada uno usa el shortid del canal de 8 bytes que coincide con el `channel_announcement` y el campo

`channel_flags` de 1 bit para indicar en qué extremo del canal está (origen o final). Un nodo puede hacer esto varias veces para cambiar las tarifas.

Tenga en cuenta que el mensaje de chismes `channel_update` solo es útil en el contexto de *retransmitir* pagos, no al *enviar* pagos. Al realizar un pago `A` -> `B` -> `C` -> `D`, solo `channel_update` está relacionado con los canales `B` -> `C` (anunciado por `B`) y `C` -> `D` (anunciado por `C`) entrará en juego. Al construir la ruta, las cantidades y los vencimientos de los HTLC deben calcularse hacia atrás desde el destino hasta el origen. El valor inicial exacto de `amount_msat` y el valor mínimo de `cltv_expiry`, que se utilizarán para el último HTLC de la ruta, se proporcionan en la solicitud de pago (ver BOLT #11).

1. type: 258 (`channel_update`)
2. data:
    - [`signature`:`signature`]
    - [`chain_hash`:`chain_hash`]
    - [`short_channel_id`:`short_channel_id`]
    - [`u32`:`timestamp`]
    - [`byte`:`message_flags`]
    - [`byte`:`channel_flags`]
    - [`u16`:`cltv_expiry_delta`]
    - [`u64`:`htlc_minimum_msat`]
    - [`u32`:`fee_base_msat`]
    - [`u32`:`fee_proportional_millionths`]
    - [`u64`:`htlc_maximum_msat`]

El campo de bits `channel_flags` se utiliza para indicar la dirección del canal: identifica el nodo desde el que se originó esta actualización y señala varias opciones relacionadas con el canal. La siguiente tabla especifica el significado de sus bits individuales:

Bit de posición	Nombre	Significado
0	`direction`	Dirección a la que se refiere esta actualización.
1	`disable`	Deshabilitar el canal.

El campo de bits `message_flags` se utiliza para proporcionar detalles adicionales sobre el mensaje:

Bit de posición	Nombre
0	`must_be_one`
1	`dont_forward`

El `node_id` para la verificación de la firma se toma del `channel_announcement` correspondiente: `node_id_1` si el bit menos significativo de las banderas es 0 o `node_id_2` en caso contrario.

Requisitos

El nodo de origen:

- NO DEBE enviar un `channel_update` creado antes de que se haya recibido `channel_ready`.

- PUEDE crear un `channel_update` para comunicar los parámetros del canal al compañero del canal, aunque el canal aún no se haya anunciado (es decir, el bit `announce_channel` no se ha establecido).
    - DEBE establecer `short_channel_id` en un `alias` que haya recibido del par, o en el canal real `short_channel_id`.
    - DEBE establecer `dont_forward` en 1 en `message_flags`
    - NO DEBE reenviar tal `channel_update` a otros pares, por razones de privacidad.
    - Nota: tal `channel_update`, uno que no esté precedido por un `channel_announcement`, no es válido para ningún otro par y sería descartado.
- DEBE establecer `firma` en la firma del doble-SHA256 del paquete restante completo después de `signature`, utilizando su propio `node_id`.
- DEBE configurar `chain_hash` Y `short_channel_id` para que coincida con el hash de 32 bytes Y el ID de canal de 8 bytes que identifica de forma única el canal especificado en el mensaje `channel_announcement`.
- si el nodo de origen es `node_id_1` en el mensaje:
    - DEBE establecer el bit de `direction` de `channel_flags` en 0.
- de lo contrario:
    - DEBE establecer el bit de `direction` de `channel_flags` en 1.
- DEBE establecer `htlc_maximum_msat` en el valor máximo que enviará a través de este canal para un solo HTLC.
    - DEBE establecerlo en un valor inferior o igual a la capacidad del canal.
    - DEBE establecer esto en un valor menor o igual a `max_htlc_value_in_flight_msat` que recibió del par.
- DEBE establecer `must_be_one` en `message_flags` en 1.
- DEBE establecer bits en `channel_flags` y `message_flags` que no tienen asignado un significado a 0.
- PUEDE crear y enviar un `channel_update` con el bit `disable` establecido en 1, para señalar la indisponibilidad temporal de un canal (p. ej., debido a una pérdida de conectividad) O la indisponibilidad permanente (p. ej., antes de una liquidación en cadena).
    - DEBIÓ enviar un `channel_update` posterior con el bit `disable` establecido en 0 para volver a habilitar el canal.
- DEBE establecer `timestamp` en un valor mayor que 0 Y mayor que cualquier `channel_update` enviado previamente para este `short_channel_id`.
    - DEBERÍA basar `timestamp` en una marca de tiempo UNIX.
- DEBE establecer `cltv_expiry_delta` en el número de bloques que restará de `cltv_expiry` de un HTLC entrante.
- DEBE establecer `htlc_minimum_msat` en el valor mínimo de HTLC (en milisatoshi) que aceptará el par del canal.
- DEBE establecer `fee_base_msat` en la tarifa base (en millisatoshi) que cobrará por cualquier HTLC.
- DEBE establecer `fee_proportional_millionths` en la cantidad (en millonésimas de satoshi) que cobrará por satoshi transferido.
- NO DEBERÍA crear `channel_update`s redundantes
- Si crea un nuevo `channel_update` con parámetros de canal actualizados:
    - DEBE seguir aceptando los parámetros del canal anterior durante 10 minutos

El nodo receptor:

- si `short_channel_id` NO coincide con un `channel_announcement` anterior, O si el canal ha sido cerrado mientras tanto:
  - DEBE ignorar `channel_update`s que NO correspondan a uno de sus propios canales.
- DEBERÍA aceptar `channel_update`s para sus propios canales (incluso si no son públicos), para aprender los parámetros de reenvío de los nodos de origen asociados.
- si `signature` no es una firma válida, usando `node_id` del doble-SHA256 de todo el mensaje que sigue al campo `signature` (incluyendo campos desconocidos después de `fee_proportional_millionths`):
  - DEBERÍA enviar una `warning` y cerrar la conexión.
  - NO DEBE seguir procesando el mensaje.
- si el valor `chain_hash` especificado es desconocido (lo que significa que no está activo en la cadena especificada):
  - DEBE ignorar la actualización del canal.
- si la `timestamp` es igual a la última `channel_update` recibida para este `short_channel_id` Y `node_id`:
  - si los campos debajo de `timestamp` difieren:
    - PUEDE incluir en la lista negra este `node_id`.
    - PUEDE olvidar todos los canales asociados con él.
  - si los campos debajo de `timestamp` son iguales:
    - DEBE ignorar este mensaje
- si `timestamp` es inferior a la última `channel_update` recibida para este `short_channel_id` Y para `node_id`:
  - DEBE ignorar el mensaje.
- de lo contrario:
  - si la `timestamp` está irrazonablemente lejos en el futuro:
    - PUEDE descartar `channel_update`.
  - de lo contrario:
    - DEBERÍA poner en cola el mensaje para su retransmisión.
    - PUEDE elegir NO para mensajes más largos que la longitud mínima esperada.
- si `htlc_maximum_msat` es mayor que la capacidad del canal:
  - PUEDE incluir en la lista negra este `node_id`
  - DEBE ignorar este canal durante las consideraciones de ruta.
- de lo contrario:
  - DEBE considerar `htlc_maximum_msat` al enrutar.

Racional

Los nodos utilizan el campo `timestamp` para eliminar `channel_update`s que están demasiado lejos en el futuro o no se han actualizado en dos semanas (`pruning`); por lo que tiene sentido que sea una marca de tiempo UNIX (es decir, segundos desde UTC 1970-01-01). Sin embargo, esto no puede ser un requisito estricto, dado el posible caso de dos `channel_update` en un solo segundo.

Se supone que más de un mensaje `channel_update` cambiando los parámetros del canal en el mismo segundo puede ser un intento de DoS y, por lo tanto, el nodo responsable de firmar dichos mensajes puede estar en la lista negra. Sin embargo, un nodo puede enviar un mismo mensaje `channel_update` con una firma diferente (cambiando el nonce en la firma cuando está firmando) y, por lo tanto, los campos aparte de la firma se verifican para ver si los parámetros del canal han cambiado para la misma marca de tiempo.

También es importante tener en cuenta que las firmas ECDSA son maleables. Entonces, un nodo intermedio que recibió el mensaje `channel_update` puede retransmitirlo simplemente cambiando el componente `s` de la firma con `-s`. Sin embargo, esto no debería dar como resultado la incluisión en la lista negra del `node_id` del nodo que originó el mensaje.

La recomendación contra `channel_update`s redundantes minimiza el spam en la red, sin embargo, a veces es inevitable. Por ejemplo, un canal con un compañero al que no se puede acceder eventualmente generará una `channel_update` para indicar que el canal está deshabilitado, y otra actualización volverá a habilitar el canal cuando el compañero restablezca el contacto. Debido a que los mensajes de chismes se procesan por lotes y reemplazan a los anteriores, el resultado puede ser una única actualización aparentemente redundante.

Cuando un nodo crea una nueva `channel_update` para cambiar los parámetros de su canal, tardará un tiempo en propagarse a través de la red y los pagadores pueden usar parámetros más antiguos. Se recomienda seguir aceptando parámetros anteriores durante al menos 10 minutos para mejorar la latencia y la confiabilidad de los pagos.

El campo `must_be_one` en `message_flags` se usaba anteriormente para indicar la presencia del campo `htlc_maximum_msat`. Este campo ahora debe estar siempre presente, por lo que `must_be_one` es un valor constante e ignorado por los receptores.

Mensajes de consulta

Negociar la opción `gossip_queries` a través de `init` habilita una serie de consultas extendidas para la sincronización de chismes. Estos solicitan explícitamente qué chismes deben recibirse.

Hay varios mensajes que contienen una gran variedad de `short_channel_id`s (llamados `encoded_short_ids`), por lo que incluimos un byte de codificación que permite definir diferentes esquemas de codificación en el futuro, si brindan un beneficio.

Tipos de codificación:

- `0`: matriz sin comprimir de tipos `short_channel_id`, en orden ascendente.
- `1`: utilizado anteriormente para la compresión zlib, esta codificación NO DEBE utilizarse.

Esta codificación también se usa para arreglos de otros tipos (marcas de tiempo, banderas, ...), y se especifica con un prefijo `encoded_`. Por ejemplo, `encoded_timestamps` es una matriz de marcas de tiempo con un prefijo `0`.

Los mensajes de consulta se pueden ampliar con campos opcionales que pueden ayudar a reducir la cantidad de mensajes necesarios para sincronizar las tablas de enrutamiento al habilitar:

- Filtrado basado en la marca de tiempo de los mensajes `channel_update`: solo solicite mensajes `channel_update` que sean más nuevos que los que ya tiene.
- Filtrado basado en suma de verificación de mensajes `channel_update`: solo solicite mensajes `channel_update` que contengan información diferente a la que ya tiene.

Los nodos pueden indicar que admiten consultas de chismes extendidas con el bit de función `gossip_queries_ex`.

The `query_short_channel_ids`/`reply_short_channel_ids_end` Messages

1. type: 261 (`query_short_channel_ids`) (`gossip_queries`)
2. data:
    - [`chain_hash`:`chain_hash`]
    - [`u16`:`len`]
    - [`len*byte`:`encoded_short_ids`]
    - [`query_short_channel_ids_tlvs`:`tlvs`]
3. `tlv_stream`: `query_short_channel_ids_tlvs`
4. types:
    1. type: 1 (`query_flags`)
    2. data:
        - [`byte`:`encoding_type`]
        - [`...*byte`:`encoded_query_flags`]

`encoded_query_flags` es una matriz de campos de bits, uno de tamaño grande por campo de bits, un campo de bits para cada `short_channel_id`. Los bits tienen el siguiente significado:

Posición de bits	Significado
0	El remitente quiere `channel_announcement`
1	El remitente quiere `channel_update` para el nodo 1
2	El remitente quiere `channel_update` para el nodo 2
3	El remitente quiere `node_announcement` para el nodo 1
4	El remitente quiere `node_announcement` para el nodo 2

Los indicadores de consulta deben codificarse mínimamente (`minimally encoded`), lo que significa que un indicador se codificará con un solo byte.

1. type: 262 (`reply_short_channel_ids_end`) (`gossip_queries`)
2. data:
    - [`chain_hash`:`chain_hash`]
    - [`byte`:`full_information`]

This is a general mechanism which lets a node query for the `channel_announcement` and `channel_update` messages for specific channels (identified via `short_channel_id`s). This is usually used either because a node sees a `channel_update` for which it has no `channel_announcement` or because it has obtained previously unknown `short_channel_id`s from `reply_channel_range`.

Este es un mecanismo general que permite que un nodo consulte los mensajes `channel_announcement` y `channel_update` para canales específicos (identificados a través de `short_channel_id`s). Esto generalmente se usa porque un nodo ve una `channel_update` para la cual no tiene `channel_announcement` o porque ha obtenido `short_channel_id`s previamente desconocidos de `reply_channel_range`.

Requisitos

El remitente:

- NO DEBE enviar `query_short_channel_ids` si envió un `query_short_channel_ids` anterior a este par y no recibió `reply_short_channel_ids_end`.
- DEBE establecer `chain_hash` en el hash de 32 bytes que identifica de forma única la cadena a la que se refieren los `short_channel_id`s.
- DEBE establecer el primer byte de `encoded_short_ids` en el tipo de codificación.
- DEBE codificar un número entero de `short_channel_id`s a `encoded_short_ids`
- PUEDE enviar esto si recibe un `channel_update` para un `short_channel_id` para el cual no tiene `channel_announcement`.
- NO DEBE enviar esto si el canal al que se hace referencia no es una salida no utilizada.
- PUEDE incluir `query_flags` opcional. En ese caso:
    - DEBE establecer `encoding_type`, como `encoded_short_ids`.
    - Cada indicador de consulta es un tamaño grande `minimally-encoded`.
    - DEBE codificar un indicador de consulta por `short_channel_id`.

El receptor:

- si el primer byte de `encoded_short_ids` no es un tipo de codificación conocido:
    - PUEDE enviar una `warning`.
    - PUEDE cerrar la conexión.
- si `encoded_short_ids` no se decodifica en un número entero de `short_channel_id`:
    - PUEDE enviar una `warning`.
    - PUEDE cerrar la conexión.
- si no ha enviado `reply_short_channel_ids_end` a un `query_short_channel_ids` recibido previamente de este remitente:
    - PUEDE enviar una `warning`.
    - PUEDE cerrar la conexión.
- si el mensaje entrante incluye `query_short_channel_ids_tlvs`:
    - si `encoding_type` no es un tipo de codificación conocido:
        - PUEDE enviar una `warning`.
        - PUEDE cerrar la conexión.
    - si `encoded_query_flags` no se decodifica en exactamente una bandera por `short_channel_id`:
        - PUEDE enviar una `warning`.
        - PUEDE cerrar la conexión.
- DEBE responder a cada `short_channel_id` conocido:
    - si el mensaje entrante no incluye `encoded_query_flags`:
        - con un `channel_announcement` y la última `channel_update` para cada extremo
        - DEBE seguir con cualquier `node_announcement` para cada `channel_announcement`
    - de lo contrario:
        - Definimos `query_flag` para el enésimo `short_channel_id` en `encoded_short_ids` para que sea el enésimo tamaño grande de los `encoded_query_flags` decodificados.
        - si se establece el bit 0 de `query_flag`:
            - DEBE responder con un `channel_announcement`

            - si el bit 1 de `query_flag` está establecido y ha recibido una `channel_update` de `node_id_1`:
                - DEBE responder con la última `channel_update` para `node_id_1`
            - si el bit 2 de `query_flag` está establecido y ha recibido una `channel_update` de `node_id_2`:
                - DEBE responder con la última `channel_update` para `node_id_2`
            - si el bit 3 de `query_flag` está establecido y ha recibido un `node_announcement` de `node_id_1`:
                - DEBE responder con el último `node_announcement` para `node_id_1`
            - si el bit 4 de `query_flag` está establecido y ha recibido un `node_announcement` de `node_id_2`:
                - DEBE responder con el último `node_announcement` para `node_id_2`
        - NO DEBE esperar a la siguiente descarga de chismes salientes para enviarlos.
- DEBERÍA evitar enviar `node_announcements` duplicados en respuesta a un solo `query_short_channel_ids`.
- DEBE seguir estas respuestas con `reply_short_channel_ids_end`.
- si no mantiene actualizada la información del canal para `chain_hash`:
    - DEBE establecer `full_information` en 0.
- de lo contrario:
    - DEBERÍA establecer `full_information` en 1.

Racional

Es posible que los nodos futuros no tengan información completa; ciertamente no tendrán información completa sobre cadenas `chain_hash` desconocidas. Si bien no se puede confiar en este campo `full_information` (anteriormente y confusamente llamado `complete`), un 0 indica que el remitente debe buscar en otro lugar para obtener datos adicionales.

El mensaje explícito `reply_short_channel_ids_end` significa que el receptor puede indicar que no sabe nada, y el remitente no necesita confiar en los tiempos de espera. También provoca una limitación natural de la tasa de consultas.

Los mensajes `query_channel_range` y `reply_channel_range`

1. type: 263 (`query_channel_range`) (`gossip_queries`)
2. data:
    - [`chain_hash`:`chain_hash`]
    - [`u32`:`first_blocknum`]
    - [`u32`:`number_of_blocks`]
    - [`query_channel_range_tlvs`:`tlvs`]
3. `tlv_stream`: `query_channel_range_tlvs`
4. types:
    1. type: 1 (`query_option`)
    2. data:

        - [bigsize:query_option_flags]

query_option_flags es un campo de bits representado como un tamaño grande minimally-encoded. Los bits tienen el siguiente significado:

Posición de bits	Significado
0	El remitente quiere marcas de tiempo
1	El remitente quiere sumas de verificación

Aunque es posible, no sería muy útil solicitar sumas de verificación sin solicitar también marcas de tiempo: el nodo receptor puede tener un channel_update más antiguo con una suma de verificación diferente, pedirlo sería inútil. Y si una suma de comprobación channel_update es realmente 0 (lo que es bastante improbable), no se consultará.

1. type: 264 (reply_channel_range) (gossip_queries)
2. data:
    - [chain_hash:chain_hash]
    - [u32:first_blocknum]
    - [u32:number_of_blocks]
    - [byte:sync_complete]
    - [u16:len]
    - [len*byte:encoded_short_ids]
    - [reply_channel_range_tlvs:tlvs]
3. tlv_stream: reply_channel_range_tlvs
4. types:
    1. type: 1 (timestamps_tlv)
    2. data:
        - [byte:encoding_type]
        - [...*byte:encoded_timestamps]
    3. type: 3 (checksums_tlv)
    4. data:
        - [...*channel_update_checksums:checksums]

Para una solo channel_update, las marcas de tiempo se codifican como:

1. subtype: channel_update_timestamps
2. data:
    - [u32:timestamp_node_id_1]
    - [u32:timestamp_node_id_2]

Donde:

- timestamp_node_id_1 es la marca de tiempo de channel_update para node_id_1, o 0 si no hubo channel_update de ese nodo.

- `timestamp_node_id_2` es la marca de tiempo de `channel_update` para `node_id_2`, o 0 si no hubo `channel_update` de ese nodo.

For a single `channel_update`, checksums are encoded as:

1. subtype: `channel_update_checksums`
2. data:
    - [`u32`:`checksum_node_id_1`]
    - [`u32`:`checksum_node_id_2`]

Donde:

- `checksum_node_id_1` es la suma de verificación de `channel_update` para `node_id_1`, o 0 si no hubo `channel_update` de ese nodo.
- `checksum_node_id_2` es la suma de verificación de `channel_update` para `node_id_2`, o 0 si no hubo `channel_update` de ese nodo.

La suma de verificación de una `channel_update` es la suma de verificación CRC32C como se especifica en RFC3720 de este `channel_update` sin su `signature` y campos `timestamp`.

Esto permite consultar canales dentro de bloques específicos.

Requisitos

El remitente de `query_channel_range`:

- NO DEBE enviar esto si ha enviado un `query_channel_range` anterior a este par y no recibió todas las respuestas de `reply_channel_range`.
- DEBE establecer `chain_hash` en el hash de 32 bytes que identifica de forma única la cadena que quiere que `reply_channel_range` se refiera
- DEBE establecer `first_blocknum` en el primer bloque para el que quiere conocer los canales
- DEBE establecer `number_of_blocks` en 1 o más.
- PUEDE agregar un `query_channel_range_tlv` adicional, que especifica el tipo de información extendida que le gustaría recibir.

El receptor de `query_channel_range`:

- si no ha enviado todo `reply_channel_range` a un `query_channel_range` recibido previamente de este remitente:
    - PUEDE enviar una `warning`.
    - PUEDE cerrar la conexión.
- DEBE responder con uno o más `reply_channel_range`:
    - DEBE configurarse con `chain_hash` igual a `query_channel_range`,
    - DEBE limitar `number_of_blocks` al número máximo de bloques cuyos resultados podrían caber en `encoded_short_ids`
    - PUEDE dividir el contenido del bloque en múltiples `reply_channel_range`.
    - el primer mensaje `reply_channel_range`:
        - DEBE establecer `first_blocknum` menor o igual que `first_blocknum` en `query_channel_range`

        - DEBE establecer `first_blocknum` más `number_of_blocks` mayor que `first_blocknum` en `query_channel_range`.
    - mensaje `reply_channel_range` sucesivo:
        - DEBE tener `first_blocknum` igual o mayor que el `first_blocknum` anterior.
    - DEBE establecer `sync_complete` en `false` si este no es el `reply_channel_range` final.
    - el mensaje final `reply_channel_range`:
        - DEBE tener `first_blocknum` más `number_of_blocks` igual o mayor que `query_channel_range first_blocknum` más `number_of_blocks`.
    - DEBE establecer `sync_complete` en `true`.

Si el mensaje entrante incluye `query_option`, el receptor PUEDE agregar información adicional a su respuesta:

- si se establece el bit 0 en `query_option_flags`, el receptor PUEDE agregar un `timestamps_tlv` que contiene las marcas de tiempo `channel_update` para todos los `short_chanel_id` en `encoded_short_ids`
- si se establece el bit 1 en `query_option_flags`, el receptor PUEDE agregar un `checksums_tlv` que contiene sumas de verificación `channel_update` para todos los `short_chanel_id` en `encoded_short_ids`.

Racional

Una sola respuesta puede ser demasiado grande para un solo paquete, por lo que es posible que se requieran múltiples respuestas. Queremos permitir que un par almacene resultados enlatados para (digamos) rangos de 1000 bloques, de modo que las respuestas puedan exceder el rango solicitado. Sin embargo, requerimos que cada respuesta sea relevante (que se superponga al rango solicitado).

Al insistir en que las respuestas sean en orden creciente, el receptor puede determinar fácilmente si las respuestas están hechas: simplemente verifique si `first_blocknum` más `number_of_blocks` es igual o excede el `first_blocknum` más `number_of_blocks` que solicitó.

La adición de campos de marca de tiempo y de suma de verificación permite a un par omitir la consulta de actualizaciones redundantes.

El mensaje `gossip_timestamp_filter`

1. type: 265 (`gossip_timestamp_filter`) (`gossip_queries`)
2. data:
    - [`chain_hash`:`chain_hash`]
    - [`u32`:`first_timestamp`]
    - [`u32`:`timestamp_range`]

Este mensaje permite que un nodo restrinja los futuros mensajes de chismes a un rango específico. Un nodo que quiera mensajes de chismes tendría que enviar esto, de lo contrario, la negociación `gossip_queries` significa que no se recibirían mensajes de chismes.

Tenga en cuenta que este filtro reemplaza a cualquier anterior, por lo que puede usarse varias veces para cambiar el chisme de un compañero.

Requisitos

El remitente:

- DEBE establecer `chain_hash` en el hash de 32 bytes que identifica de forma única la cadena a la que quiere que se refiera el chisme.

El receptor:

- DEBE enviar todos los mensajes de chismes cuyo `timestamp` sea mayor o igual a `first_timestamp`, y menor que `first_timestamp` más `timestamp_range`.
  - PUEDE esperar a la próxima descarga de chismes salientes para enviarlos.
- DEBERÍA enviar mensajes de chismes a medida que los genera, independientemente de la `timestamp`.
- De lo contrario (chismes transmitidos):
  - DEBERÍA restringir los futuros mensajes de chismes a aquellos cuyo `timestamp` sea mayor o igual a `first_timestamp`, y menor que `first_timestamp` más `timestamp_range`.
- Si un `channel_announcement` no tiene `channel_update`s correspondientes:
  - NO DEBE enviar el `channel_announcement`.
- De lo contrario:
  - DEBE considerar el `timestamp` del `channel_announcement` como el `timestamp` de una `channel_update` correspondiente.
  - DEBE considerar si enviar el `channel_announcement` después de recibir el primer `channel_update` correspondiente.
- Si se envía un `channel_announcement`:
  - DEBE enviar el `channel_announcement` antes de cualquier `channel_update` y `node_announcement` correspondientes.

Racional

Dado que `channel_announcement` no tiene una marca de tiempo, generamos una probable. Si no hay `channel_update` entonces no se envía en absoluto, lo que es más probable en el caso de canales podados.

De lo contrario, `channel_announcement` suele ir seguido inmediatamente de `channel_update`. Idealmente, especificaríamos que la primera (más antigua) marca de tiempo de `channel_update` se use como la hora del `channel_announcement`, pero los nuevos nodos en la red no tendrán esto, y además requerirían la primera marca de tiempo `channel_update` para ser almacenado. En su lugar, permitimos que se utilice cualquier actualización, que sea simple de implementar.

En el caso de que se pierda el `channel_announcement`, se puede usar `query_short_channel_ids` para recuperarlo.

Los nodos pueden usar `timestamp_filter` para reducir su carga de chismes cuando tienen muchos pares (por ejemplo, establecer `first_timestamp` en `0xFFFFFFFF` después de los primeros pares, en el supuesto de que la propagación es adecuada). Esta suposición de propagación adecuada no se aplica a los mensajes de chismes generados directamente por el propio nodo, por lo que deben ignorar los filtros.

Sincronización Inicial

Si un nodo requiere una sincronización inicial de mensajes de chismes, se marcará en el mensaje `init`, a través de una `featura flag` (BOLT #9).

Tenga en cuenta que la función `initial_routing_sync` se anula (y debe considerarse igual a 0) por la función `gossip_queries` si esta última se negocia a través de `init`.

Tenga en cuenta que `gossip_queries` no funciona con nodos más antiguos, por lo que el valor de `initial_routing_sync` sigue siendo importante para controlar las interacciones con ellos.

Requisitos

Un nodo:

- si se negocia la función `gossip_queries`:
  - NO DEBE transmitir ningún mensaje de chismes que no haya generado él mismo, a menos que se solicite explícitamente.
- de lo contrario:
  - si requiere una copia completa del estado de enrutamiento del par:
    - DEBERÍA establecer el indicador `initial_routing_sync` en 1.
  - al recibir un mensaje `init` con el indicador `initial_routing_sync` establecido en 1:
    - DEBERÍA enviar mensajes de chismes para todos los canales y nodos conocidos, como si fueran recién recibidos.
  - si el indicador `initial_routing_sync` se establece en 0, O si se completó la sincronización inicial:
    - DEBERÍA reanudar el funcionamiento normal, como se especifica en la siguiente sección Retransmisión.

Retransmisión

Requisitos

Un nodo receptor:

- al recibir un nuevo `channel_announcement` o `channel_update` o `node_announcement` con una `timestamp` actualizada:
  - DEBE actualizar su vista local de la topología de la red en consecuencia.
- después de aplicar los cambios del anuncio:
  - si no hay canales asociados al nodo de origen correspondiente:
    - PUEDE purgar el nodo de origen de su conjunto de nodos conocidos.
  - de lo contrario:
    - DEBE actualizar los metadatos apropiados Y almacenar la firma asociada con el anuncio.
      - Nota: esto permitirá que el nodo reconstruya el anuncio para sus pares más adelante.

Un nodo:

- si se negocia la función `gossip_queries`:
  - No DEBE enviar chismes que no haya generado él mismo, hasta que reciba `gossip_timestamp_filter`.

- DEBERÍA vaciar los mensajes de chismes salientes una vez cada 60 segundos, independientemente de la hora de llegada de los mensajes.
    - Nota: esto da como resultado anuncios escalonados que son únicos (no duplicados).
    - NO DEBERÍA reenviar mensajes de chismes a compañeros que enviaron `networks` en `init` y no especificaron el `chain_hash` de este mensaje de chismes.
- PUEDE volver a anunciar sus canales periódicamente.
    - Nota: esto no se recomienda para mantener bajos los requisitos de recursos.
- tras el establecimiento de la conexión:
    - DEBE enviar todos los mensajes `channel_announcement`, seguidos de los últimos mensajes `node_announcement` Y `channel_update`.

Racional

Una vez que se ha procesado el mensaje de chismes, se agrega a una lista de mensajes salientes, destinados a los pares del nodo de procesamiento, reemplazando cualquier actualización anterior del nodo de origen. Esta lista de mensajes de chismes se eliminará a intervalos regulares; una transmisión de este tipo con almacenamiento y reenvío retrasado se denomina *staggered broadcast* o *difusión escalonada*. Además, tal procesamiento por lotes forma un límite de tasa natural con gastos generales bajos.

El envío de todos los chismes sobre la reconexión es naif, pero simple, y permite el `bootstrapping` de nuevos nodos, así como la actualización de nodos que han estado fuera de línea durante algún tiempo. La opción `gossip_queries` permite una sincronización más refinada.

Tarifa de HTLC

Requisitos

El nodo de origen:

- DEBERÍA aceptar HTLC que paguen una tarifa igual o superior a:
    - fee_base_msat + (cantidad_a_reenviar * fee_proportional_millionths / 1000000)
- DEBERÍA aceptar HTLC que paguen una tarifa anterior, durante un tiempo razonable después de enviar `channel_update`.
    - Nota: esto permite cualquier retardo de propagación.

Podando la de Vista de Red

Requisitos

Un nodo:

- DEBERÍA monitorear las transacciones de financiación en la cadena de bloques, para identificar los canales que se están cerrando.
- si se gasta la producción de fondos de un canal:
    - DEBE eliminarse de la vista de red local Y considerarse cerrado.
- si el nodo anunciado ya no tiene ningún canal abierto asociado:
    - PUEDE eliminar los nodos agregados a través de mensajes `node_announcement` desde su vista local.

- Nota: este es un resultado directo de la dependencia de un `node_announcement` precedido por un `channel_announcement`.
-

Recomendación sobre la Poda de Entradas Obsoletas

Requisitos

Un nodo:

- si la `timestamp` de la última `channel_update` en cualquier dirección tiene más de dos semanas (1209600 segundos):
  - PUEDE podar el canal.
  - PUEDE ignorar el canal.
  - Nota: esta es una política de nodo individual y NO DEBE ser aplicada por los pares de reenvío, p. cerrando canales al recibir mensajes de chismes desactualizados.

Racional

Varios escenarios pueden dar lugar a que los canales se vuelvan inutilizables y sus terminales no puedan enviar actualizaciones para estos canales. Por ejemplo, esto ocurre si ambos extremos pierden el acceso a sus claves privadas y no pueden firmar `channel_update`s ni cerrar el canal en cadena. En este caso, es poco probable que los canales formen parte de una ruta calculada, ya que se separarían del resto de la red; sin embargo, permanecerían en la vista de red local y se reenviarían a otros pares indefinidamente.

El `channel_update` más antiguo se usa para eliminar el canal, ya que ambos lados deben estar activos para que el canal sea utilizable. Al hacerlo, se eliminan los canales incluso si un lado continúa enviando `channel_update`s nuevos pero el otro nodo ha desaparecido.

Recomendaciones para el enrutamiento

Al calcular una ruta para un HTLC, se deben considerar tanto el `cltv_expiry_delta` como la tarifa: el `cltv_expiry_delta` contribuye al tiempo que los fondos no estarán disponibles en el caso de una falla en el peor de los casos. La relación entre estos dos atributos no está clara, ya que depende de la confiabilidad de los nodos involucrados.

Si una ruta se calcula simplemente enrutando al destinatario deseado y sumando los `cltv_expiry_delta`s, entonces es posible que los nodos intermedios adivinen su posición en la ruta. Conocer el CLTV del HTLC, la topología de red circundante y `cltv_expiry_delta`s le da al atacante una forma de adivinar el destinatario previsto. Por lo tanto, es muy recomendable agregar una compensación aleatoria al CLTV que recibirá el destinatario previsto, que golpea a todos los CLTV a lo largo de la ruta.

Para crear un desplazamiento plausible, el nodo de origen PUEDE iniciar una caminata aleatoria limitada en el gráfico, comenzando desde el destinatario deseado y sumando los `cltv_expiry_delta`s, y usar la suma resultante como el desplazamiento. Esto crea efectivamente una *extensión de ruta oculta* a la ruta real y proporciona una mejor protección contra este vector de ataque que simplemente elegir un desplazamiento aleatorio.

Otras consideraciones más avanzadas implican la diversificación de la selección de rutas, para evitar puntos únicos de falla y detección, y el equilibrio de los canales locales.

Ejemplo de enrutamiento

Considere cuatro nodos:

Cada uno anuncia el siguiente `cltv_expiry_delta` en su extremo de cada canal:

1. A: 10 bloques
2. B: 20 bloques
3. C: 30 bloques
4. D: 40 bloques

C también usa un `min_final_cltv_expiry_delta` de 9 (el valor predeterminado) al solicitar pagos.

Además, cada nodo tiene un esquema de tarifas establecido que utiliza para cada uno de sus canales:

1. A: 100 base + 1000 millonésimas
2. B: 200 base + 2000 millonésimas
3. C: 300 base + 3000 millonésimas
4. D: 400 base + 4000 millonésimas

La red verá ocho mensajes `channel_update`:

1. A->B: `cltv_expiry_delta` = 10, `fee_base_msat` = 100, `fee_proportional_millionths` = 1000
2. A->D: `cltv_expiry_delta` = 10, `fee_base_msat` = 100, `fee_proportional_millionths` = 1000
3. B->A: `cltv_expiry_delta` = 20, `fee_base_msat` = 200, `fee_proportional_millionths` = 2000
4. D->A: `cltv_expiry_delta` = 40, `fee_base_msat` = 400, `fee_proportional_millionths` = 4000
5. B->C: `cltv_expiry_delta` = 20, `fee_base_msat` = 200, `fee_proportional_millionths` = 2000
6. D->C: `cltv_expiry_delta` = 40, `fee_base_msat` = 400, `fee_proportional_millionths` = 4000
7. C->B: `cltv_expiry_delta` = 30, `fee_base_msat` = 300, `fee_proportional_millionths` = 3000

8. C->D: `cltv_expiry_delta` = 30, `fee_base_msat` = 300, `fee_proportional_millionths` = 3000

B->C. Si B enviara 4,999,999 millisatoshi directamente a C, no se cobraría una tarifa ni agregaría su propio `cltv_expiry_delta`, por lo que usaría el `min_final_cltv_expiry_delta` solicitado por C de 9. Presumiblemente, también agregue una *ruta de sombra* para dar un CLTV adicional de 42. Además, podría agregar deltas de CLTV adicionales en otros saltos, ya que estos valores representan un mínimo, pero elige no hacerlo aquí, en aras de la simplicidad:

- `amount_msat`: 4999999
- `cltv_expiry`: current-block-height + 9 + 42
- `onion_routing_packet`:
  - `amt_to_forward` = 4999999
  - `outgoing_cltv_value` = current-block-height + 9 + 42

A->B->C. Si A enviara 4,999,999 millisatoshi a C a través de B, debe pagar a B la tarifa que especificó en B->C `channel_update`, calculada según HTLC Fees:

```
    fee_base_msat + ( amount_to_forward * fee_proportional_millionths /
1000000 )

    200 + ( 4999999 * 2000 / 1000000 ) = 10199
```

De manera similar, necesitaría agregar `channel_update` `cltv_expiry_delta` de B->C (20), `min_final_cltv_expiry_delta` solicitado por C (9) y el costo de la *ruta oculta* (42). Por lo tanto, el mensaje `update_add_htlc` de A->B sería:

- `amount_msat`: 5010198
- `cltv_expiry`: current-block-height + 20 + 9 + 42
- `onion_routing_packet`:
  - `amt_to_forward` = 4999999
  - `outgoing_cltv_value` = current-block-height + 9 + 42

El `update_add_htlc` de B->C sería el mismo que el pago directo de B->C anterior.

A->D->C. Finalmente, si por alguna razón A elige la ruta más cara a través de D, el mensaje `update_add_htlc` de A->D sería:

- `amount_msat`: 5020398
- `cltv_expiry`: current-block-height + 40 + 9 + 42
- `onion_routing_packet`:
  - `amt_to_forward` = 4999999
  - `outgoing_cltv_value` = current-block-height + 9 + 42

Y `update_add_htlc` de D->C sería nuevamente el mismo que el pago directo de B->C anterior.