BOLT #2: Protocolo de pares para la gestión de canales

El protocolo de canal de pares tiene tres fases: establecimiento, operación normal y cierre.

Índice

- BOLT #2: Protocolo de pares para la gestión de canales
- Índice
- Canal
  - Definición de `channel_id`
  - Establecimiento de canal
    - El mensaje `open_channel`
      - Tipos de canales definidos
      - Requisitos
      - Base Lógica
    - El mensaje `accept_channel`
      - Requisitos
    - El mensaje `funding_created`
      - Requisitos
      - Base Lógica
    - El mensaje `funding_signed`
      - Requisitos
      - Base Lógica
    - El mensaje `channel_ready`
      - Requisitos
      - Base lógica
  - Cierre de canal
    - Closing Initiation: `shutdown`
      - Requisitos
      - Base lógica
    - Negociación de cierre: `closing_signed`
      - Requisitos
      - Base lógica
  - Operación normal
    - Reenvío de HTLCs
      - Requisitos
      - Base lógica
    - Selección `cltv_expiry_delta`
      - Requisitos
    - Añadiendo un HTLC: `update_add_htlc`
      - Requisitos
      - Base lógica

        - Eliminando un HTLC: `update_fulfill_htlc`, `update_fail_htlc`, y `update_fail_malformed_htlc`
            - Requisitos
            - Base lógica
        - Confirmando actualizaciones hasta ahora: `commitment_signed`
            - Requisitos
            - Base lógica
        - Completar la transición al estado actualizado: `revoke_and_ack`
            - Requisitos
        - Actualizando Fees: `update_fee`
            - Requisitos
            - Base lógica
    - Retransmisión de mensaje
        - Requisitos
        - Base lógica
- Authors

Canal

Definición de `channel_id`

Algunos mensajes usan un `channel_id` para identificar el canal. Se deriva de la transacción de financiación mediante la combinación de `funding_txid` y `funding_output_index`, usando big-endian OR-exclusivo (es decir, `funding_output_index` altera los últimos 2 bytes).

Antes del establecimiento del canal, se usa un `temporal_channel_id`, que es un nonce aleatorio.

Tenga en cuenta que, dado que pueden existir `temporal_channel_id`s duplicados de diferentes pares, las API que hacen referencia a los canales por su ID de canal antes de que se cree la transacción de financiación, son intrínsecamente inseguras. El único identificador proporcionado por el protocolo para un canal antes de que se haya intercambiado la creación de fondos es la tupla (source_node_id, destination_node_id, temporary_channel_id). Tenga en cuenta que las API que hacen referencia a los canales por su ID de canal antes de que se confirme la transacción de financiación, tampoco son persistentes; hasta que sepa la clave pública del script correspondiente a la salida de financiación, nada evita la duplicación de ID de canal.

Establecimiento de canal

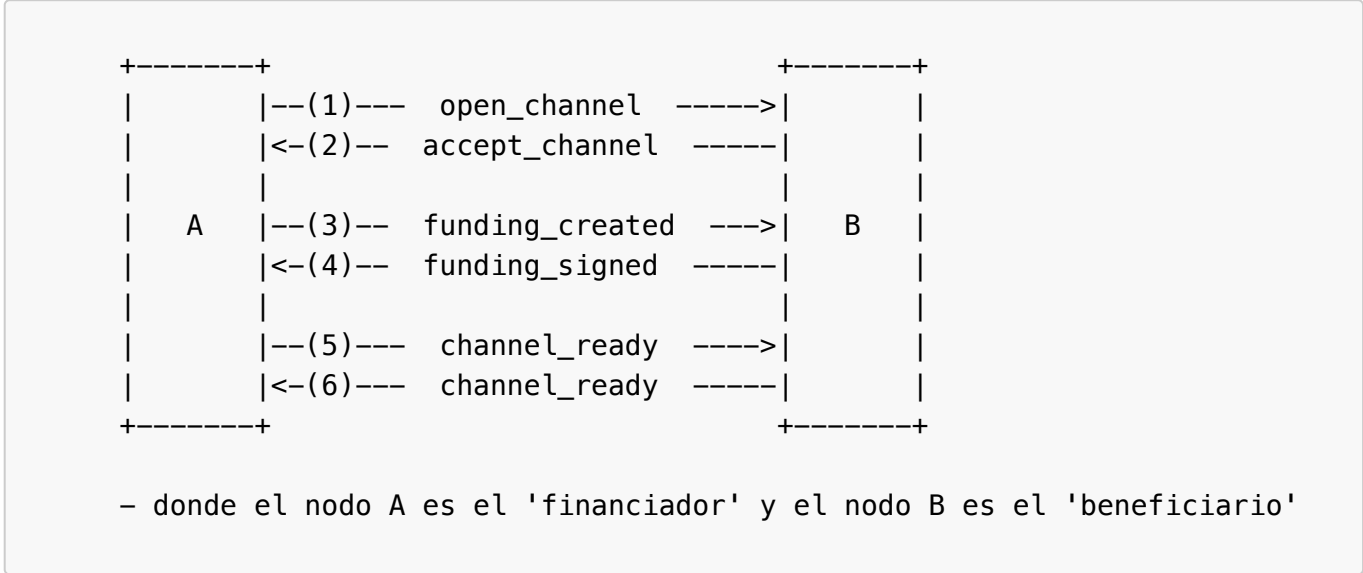
Después de autenticar e inicializar una conexión (BOLT #8 y BOLT #1, respectivamente), puede comenzar el establecimiento del canal.

Esto consiste en que el nodo de financiación (financiador) envía un mensaje `open_channel`, seguido por el nodo de respuesta (beneficiario) que envía `accept_channel`. Con los parámetros del canal bloqueados, el financiador puede crear la transacción de financiación y ambas versiones de la transacción de compromiso, como se describe en BOLT #3.

Luego, el financiador envía el punto de partida de la salida de financiación con el mensaje `funding_created`, junto con la firma de la versión del beneficiario de la transacción de compromiso. Una

vez que el beneficiario conoce el punto de salida de la financiación, puede generar la firma para la versión del financiador de la transacción de compromiso y enviarla mediante el mensaje `funding_signed`.

Una vez que el financiador del canal recibe el mensaje `funding_signed`, debe transmitir la transacción de financiación a la red de Bitcoin. Después de enviar/recibir el mensaje `funding_signed`, ambas partes deben esperar a que la transacción de financiación ingrese a la cadena de bloques y alcance la profundidad especificada (número de confirmaciones). Después de que ambos lados hayan enviado el mensaje `channel_ready`, el canal se establece y puede comenzar la operación normal. El mensaje `channel_ready` incluye información que se utilizará para construir pruebas de autenticación de canal.



Si esto falla en cualquier etapa, o si un nodo decide que los términos del canal ofrecidos por el otro nodo no son adecuados, el establecimiento del canal falla.

Tenga en cuenta que varios canales pueden operar en paralelo, ya que todos los mensajes del canal se identifican mediante un `temporal_channel_id` (antes de que se cree la transacción de financiación) o un `channel_id` (derivado de la transacción de financiación).

El mensaje `open_channel`

Este mensaje contiene información sobre un nodo e indica su deseo de configurar un nuevo canal. Este es el primer paso para crear la transacción de financiación y ambas versiones de la transacción de compromiso.

1. type: 32 (`open_channel`)
2. data:
   - [`chain_hash`:`chain_hash`]
   - [`32*byte`:`temporary_channel_id`]
   - [`u64`:`funding_satoshis`]
   - [`u64`:`push_msat`]
   - [`u64`:`dust_limit_satoshis`]
   - [`u64`:`max_htlc_value_in_flight_msat`]
   - [`u64`:`channel_reserve_satoshis`]
   - [`u64`:`htlc_minimum_msat`]

    - [u32:feerate_per_kw]
    - [u16:to_self_delay]
    - [u16:max_accepted_htlcs]
    - [point:funding_pubkey]
    - [point:revocation_basepoint]
    - [point:payment_basepoint]
    - [point:delayed_payment_basepoint]
    - [point:htlc_basepoint]
    - [point:first_per_commitment_point]
    - [byte:channel_flags]
    - [open_channel_tlvs:tlvs]

3. tlv_stream: open_channel_tlvs

4. types:

    1. type: 0 (upfront_shutdown_script)
    2. data:
        - [...*byte:shutdown_scriptpubkey]
    3. type: 1 (channel_type)
    4. data:
        - [...*byte:type]

El valor chain_hash indica la cadena de bloques exacta en la que residirá el canal abierto. Este suele ser el hash de génesis de la respectiva cadena de bloques.

La existencia de chain_hash permite a los nodos abrir canales a través de muchas cadenas de bloques distintas, así como tener canales dentro de múltiples cadenas de bloques abiertas al mismo par (si es compatible con las cadenas de destino).

El temporary_channel_id se usa para identificar este canal basado en pares hasta que se establece la transacción de financiación, momento en el que se reemplaza por el channel_id, que se deriva de la transacción de financiación.

funding_satoshis es la cantidad que el remitente está poniendo en el canal. push_msat es una cantidad de fondos iniciales que el remitente entrega incondicionalmente al receptor. dust_limit_satoshis es el umbral por debajo del cual no se deben generar salidas para el compromiso de este nodo o las transacciones de HTLC (es decir, los HTLC por debajo de esta cantidad más las tarifas de transacción de HTLC no se pueden aplicar en la cadena). Esto refleja la realidad de que las salidas pequeñas no se consideran transacciones estándar y no se propagarán a través de la red Bitcoin. channel_reserve_satoshis es la cantidad mínima que el otro nodo debe mantener como pago directo. htlc_minimum_msat indica el valor HTLC más pequeño que aceptará este nodo.

max_htlc_value_in_flight_msat es un límite en el valor total de los HTLC pendientes, lo que permite que un nodo limite su exposición a los HTLC; De manera similar, max_accepted_htlcs limita la cantidad de HTLC pendientes que puede ofrecer el otro nodo.

feerate_per_kw indica la tasa de tarifa inicial en satoshi por 1000-weight (es decir, 1/4 de los 'satoshi por 1000 vbytes' que se usan normalmente) que este lado pagará por el compromiso y las transacciones HTLC,

como se describe en [BOLT #3] (03-transactions.md#fee-calculation) (esto se puede ajustar más adelante con un mensaje `update_fee`).

`to_self_delay` es el número de bloques que las salidas `to-self` del otro nodo deben ser retardadas, usando los retrasos `OP_CHECKSEQUENCEVERIFY`; este es el tiempo que tendrá que esperar en caso de fallo antes de redimir sus propios fondos.

`funding_pubkey` es la clave pública en el script 2 de 2 del multisig resultado de la transacción de financiación.

Los diversos campos `_basepoint` se utilizan para derivar claves únicas como se describe en BOLT #3 para cada transacción de compromiso. La variación de estas claves garantiza que el ID de transacción de cada transacción de compromiso sea impredecible para un observador externo, incluso si se ve una transacción de compromiso; esta propiedad es muy útil para preservar la privacidad cuando se externalizan transacciones de sanción a terceros.

`first_per_commitment_point` es el punto por compromiso que se utilizará para la primera transacción de compromiso,

Solo el bit menos significativo de `channel_flags` está definido actualmente: `announce_channel`. Esto indica si el iniciador del flujo de fondos desea anunciar este canal públicamente en la red, como se detalla en BOLT #7 .

`shutdown_scriptpubkey` permite que el nodo enviador se comprometa a dónde irán los fondos en el cierre mutuo, lo que el nodo remoto debe hacer cumplir incluso si un nodo se ve comprometido más adelante.

`option_support_large_channel` es una función que se usa para que todos sepan que este nodo aceptará `funding_satoshis` mayores o iguales a 2^24.

Dado que se transmite en el mensaje `node_announcement`, otros nodos pueden usarlo para identificar a los pares dispuestos a aceptar un canal grande incluso antes de intercambiar el mensaje `init` con ellos.

Tipos de canales definidos

Channel types are an explicit enumeration: for convenience of future definitions they reuse even feature bits, but they are not an arbitrary combination (they represent the persistent features which affect the channel operation).

Los tipos de canales son una enumeración explícita: para conveniencia de futuras definiciones, reutilizan los `feature bits` pares, pero no son una combinación arbitraria (representan las características persistentes que afectan la operación del canal).

Los tipos básicos definidos actualmente son:

- sin `features` (sin definir bits)
- `option_static_remotekey` (bit 12)
- `option_anchor_outputs` y `option_static_remotekey` (bits 20 y 12)
- `option_anchors_zero_fee_htlc_tx` y `option_static_remotekey` (bits 22 y 12)

Cada tipo básico tiene las siguientes variaciones permitidas:

- `option_scid_alias` (bit 46)
- `option_zeroconf` (bit 50)

Requisitos

El nodo emisor:

- DEBE asegurar que el valor `chain_hash` identifique la cadena en la que desea abrir el canal.
- DEBE asegurarse de que `temporary_channel_id` sea único de cualquier otro ID de canal con el mismo par.
- Si ambos nodos anunciaran `option_support_large_channel`::
  - PUEDE establecer `funding_satoshis` mayor o igual a 2^24 satoshi.
- De lo contrario:
  - DEBE establecer `funding_satoshis` a menor que 2^24 satoshi.
- DEBE configurar `push_msat` igual o menor que 1000 * `funding_satoshis`.
- DEBE establecer `funding_pubkey`, `revocation_basepoint`, `htlc_basepoint`, `payment_basepoint` y `delayed_payment_basepoint` en claves públicas secp256k1 válidas en formato comprimido.
- DEBE establecer `first_per_commitment_point` en el punto `per-commitment` que se usará para la transacción de compromiso inicial, derivado como se especifica en BOLT #3.
- DEBE establecer `channel_reserve_satoshis` mayor o igual que `dust_limit_satoshis`.
- DEBE establecer bits indefinidos en `channel_flags` a 0.
- si ambos nodos anunciaran la función `option_upfront_shutdown_script`:
  - DEBE incluir `upfront_shutdown_script` con una `shutdown_scriptpubkey` válida según requiera `shutdown scriptpubkey`, o `shutdown_scriptpubkey` de longitud cero (por ejemplo `0x0000`).
- de lo contrario:
  - DEBE incluir `upfront_shutdown_script`.
- si incluye `open_channel_tlvs`:
  - DEBE incluir `upfront_shutdown_script`.
- si se negocia `option_channel_type`:
  - DEBE establecer `channel_type`
- si incluye `channel_type`:
  - DEBE establecerlo en un tipo definido que represente el tipo que desea.
  - DEBE utilizar el mapa de bits más pequeño posible para representar el tipo de canal.
  - NO DEBE configurarlo en un tipo que contenga una `feature` que no se negoció.
  - si `announce_channel` es `true` (no `0`):
    - NO DEBE enviar `channel_type` con el bit `option_scid_alias` establecido.

El nodo emisor DEBE:

- Establecer `to_self_delay` suficiente para garantizar que el remitente pueda gastar irreversiblemente una salida de transacción de compromiso, en caso de mala conducta por parte del receptor.
- establezca `feerate_per_kw` al menos a la tasa que estima que haría que la transacción se incluyera inmediatamente en un bloque.
- establezca `dust_limit_satoshis` en un valor suficiente para permitir que las transacciones de compromiso se propaguen a través de la red de Bitcoin.

- establezca `htlc_minimum_msat` en el valor mínimo que HTLC está dispuesto a aceptar de este par.

El nodo receptor DEBE:

- ignorar los bits no definidos en `channel_flags`.
- si la conexión se ha restablecido después de recibir un `open_channel`, PERO antes de recibir un mensaje `funding_created`:
  - aceptar un nuevo mensaje `open_channel`.
  - descartar el mensaje `open_channel` anterior.

El nodo receptor PUEDE fallar en el canal si:

- Se negoció `option_channel_type` pero el mensaje no incluye un `channel_type`
- `announce_channel` es `falso` (`0`), pero desea anunciar públicamente el canal.
- `funding_satoshis` es demasiado pequeño.
- considera `htlc_minimum_msat` demasiado grande.
- considera `max_htlc_value_in_flight_msat` demasiado pequeño.
- considera `channel_reserve_satoshis` demasiado grande.
- considera `max_accepted_htlcs` demasiado pequeño.
- considera `dust_limit_satoshis` demasiado grande.

El nodo receptor DEBE fallar el canal si:

- el valor `chain_hash` se establece en un hash de una cadena que es desconocida para el receptor.
- `push_msat` es mayor que `funding_satoshis` * 1000.
- `to_self_delay` es excesivamente grande.
- `max_accepted_htlcs` es mayor que 483.
- considera `feerate_per_kw` demasiado pequeño para el procesamiento oportuno o irrazonablemente grande.
- `funding_pubkey`, `punto_base_revocación`, `punto_base_htlc`, `punto_base_pago` o `punto_base_pago_retrasado` no son claves públicas secp256k1 válidas en formato comprimido.
- `dust_limit_satoshis` es mayor que `channel_reserve_satoshis`.
- `dust_limit_satoshis` es menor que `354 satoshis` (ver BOLT 3).
- el monto del financiador para la transacción de compromiso inicial no es suficiente para el [pago de la tarifa] completo (03-transactions.md#fee-payment).
- Los montos `to_local` y `to_remote` para la transacción de compromiso inicial son menores o iguales a `channel_reserve_satoshis` (ver BOLT 3).
- `funding_satoshis` es mayor o igual a 2^24 y el receptor no admite `option_support_large_channel`.
- Admite `channel_type` y `channel_type` se configuró:
  - si `type` no es adecuado.
  - si `type` incluye `option_zeroconf` y no confía en que el remitente abra un canal no confirmado.

El nodo receptor NO DEBE:

- considerar los fondos recibidos, utilizando `push_msat`, como recibidos hasta que la transacción de financiación haya alcanzado la profundidad suficiente.

Base Lógica

El requisito de que `funding_satoshis` sea inferior a 2^24 satoshi fue un límite autoimpuesto temporal, mientras que las implementaciones aún no se consideraban estables, se puede eliminar anunciando `option_support_large_channel`.

La *reserva de canal* está especificada por `channel_reserve_satoshis` del par: se sugiere el 1% del total del canal. Cada lado de un canal mantiene esta reserva, por lo que siempre tiene algo que perder si intentara transmitir una transacción de compromiso anterior revocada. Inicialmente, esta reserva puede no ser satisfecha, ya que solo un lado tiene fondos; pero el protocolo asegura que siempre se avance hacia el cumplimiento de esta reserva, y una vez cumplida, se mantiene.

El remitente puede dar incondicionalmente fondos iniciales al receptor utilizando un `push_msat` distinto de cero, pero incluso en este caso nos aseguramos de que el financiador tenga suficientes fondos restantes para pagar las tarifas y que una parte tenga alguna cantidad que pueda gastar (lo que también implica hay al menos una salida sin polvo). Tenga en cuenta que, como cualquier otra transacción en cadena (`on-chain`), este pago no es seguro hasta que la transacción de financiación se haya confirmado lo suficiente (con el peligro de un doble gasto hasta que esto ocurra) y puede requerir un método separado para probar el pago a través de la confirmación `on-chain`.

La `tarifa_por_kw` generalmente solo le preocupa al remitente (quien paga las tarifas), pero también existe la tarifa pagada por las transacciones HTLC; por lo tanto, las tarifas excesivamente elevadas también pueden penalizar al destinatario.

Separar `htlc_basepoint` de `payment_basepoint` mejora la seguridad: un nodo necesita el secreto asociado con `htlc_basepoint` para producir firmas HTLC para el protocolo, pero el secreto para `payment_basepoint` puede estar almacenado en frío (`cold storage`).

El requisito de que `channel_reserve_satoshis` no se considere polvo según `dust_limit_satoshis` elimina los casos en los que todas las salidas sería eliminado como polvo. Los requisitos similares en `accept_channel` asegura que ambos lados `channel_reserve_satoshis` están por encima de ambos `dust_limit_satoshis`.

El receptor no debe aceptar grandes `dust_limit_satoshis`, ya que esto podría ser utilizado en ataques de duelo (`griefing attacks`), donde el compañero publica su compromiso con mucho de polvo htlcs, que efectivamente se convierten en tarifas mineras.

Los detalles sobre cómo manejar una falla de canal se pueden encontrar en [BOLT 5: Falla de un canal] (05-onchain.md#failing-a-channel).

El mensaje `accept_channel`

Este mensaje contiene información sobre un nodo e indica su aceptación del nuevo canal. Este es el segundo paso hacia la creación de la transacción de financiación y ambas versiones de la transacción de compromiso.

1. type: 33 (`accept_channel`)
2. data:
    - [`32*byte`:`temporary_channel_id`]

    - [u64:dust_limit_satoshis]
    - [u64:max_htlc_value_in_flight_msat]
    - [u64:channel_reserve_satoshis]
    - [u64:htlc_minimum_msat]
    - [u32:minimum_depth]
    - [u16:to_self_delay]
    - [u16:max_accepted_htlcs]
    - [point:funding_pubkey]
    - [point:revocation_basepoint]
    - [point:payment_basepoint]
    - [point:delayed_payment_basepoint]
    - [point:htlc_basepoint]
    - [point:first_per_commitment_point]
    - [accept_channel_tlvs:tlvs]

3. tlv_stream: accept_channel_tlvs

4. types:

    1. type: 0 (upfront_shutdown_script)
    2. data:
        - [...*byte:shutdown_scriptpubkey]
    3. type: 1 (channel_type)
    4. data:
        - [...*byte:type]

Requisitos

El temporary_channel_id DEBE ser el mismo que el temporary_channel_id en el mensaje open_channel.

El remitente:

- si channel_type incluye option_zeroconf:
    - DEBE establecer minimum_ depth en cero.
- de lo contrario:
    - DEBERÍA establecer minimum_ depth en un número de bloques que considere razonable para evitar el doble gasto de la transacción de financiación.
- DEBE configurar channel_reserve_satoshis mayor o igual que dust_limit_satoshis del mensaje open_channel.
- DEBE configurar dust_limit_satoshis menor o igual que channel_reserve_satoshis del mensaje open_channel.
- si se negoció option_channel_type:
    - DEBE establecer channel_type en channel_type de open_channel

El receptor:

- si minimum_depth es irrazonablemente grande:
    - PUEDE rechazar el canal.

- si `channel_reserve_satoshis` es menor que `dust_limit_satoshis` dentro del mensaje `open_channel`:
  - DEBE rechazar el canal.
- si `channel_reserve_satoshis` del mensaje `open_channel` es menor que `dust_limit_satoshis`:
  - DEBE rechazar el canal.
- si se establece `channel_type`, y `channel_type` se estableció en `open_channel`, y no son tipos iguales:
  - DEBE rechazar el canal.
- si se negoció `option_channel_type` pero el mensaje no incluye un `channel_type`:
  - PUEDE rechazar el canal.

Otros campos tienen los mismos requisitos que sus contrapartes en `open_channel`.

El mensaje `funding_created`

Este mensaje describe el punto de salida que el financiador ha creado para las transacciones de compromiso inicial. Después de recibir la del compañero firma, a través de `funding_signed`, transmitirá la transacción de financiación.

1. type: 34 (`funding_created`)
2. data:
   - [`32*byte`:`temporary_channel_id`]
   - [`sha256`:`funding_txid`]
   - [`u16`:`funding_output_index`]
   - [`signature`:`signature`]

Requisitos

El remitente DEBE establecer:

- `temporary_channel_id` igual que `temporary_channel_id` en el mensaje `open_channel`.
- `funding_txid` al ID de transacción de una transacción no maleable,
  - y NO DEBE transmitir esta transacción.
- `funding_output_index` al número de salida de esa transacción que corresponde a la salida de la transacción de financiación, como se define en BOLT #3.
- `signature` a la firma válida usando su `funding_pubkey` para la transacción de compromiso inicial, como se define en BOLT #3.

El remitente:

- al crear la transacción de financiación:
  - DEBE usar solo entradas BIP141 (testigo segregado o `Segregated Witness`).
  - DEBERÍA asegurarse de que la transacción de financiación se confirme en los próximos 2.016 bloques.

El receptor:

- si la `signature` es incorrecta O no cumple con la regla LOW-S-estándar [LOWS]:

- DEBE enviar un `warning` y cerrar la conexión, o enviar un `error` y fallar el canal.

Base Lógica

`funding_output_index` solo puede tener 2 bytes, ya que así es como se empaqueta en el `channel_id` y se usa en todo el protocolo de chismes. El límite de 65.535 salidas no debería ser demasiado oneroso.

Una transacción con todas las entradas de Testigo Segregado no es maleable, de ahí la recomendación de transacción de financiación.

El financiador puede usar CPFP en una salida de cambio para garantizar que la transacción de financiamiento se confirme antes de 2.016 bloques, de lo contrario, el beneficiario puede olvidar ese canal.

El mensaje `funding_signed`

Este mensaje le da al financiador la firma que necesita para la primera transacción de compromiso, por lo que puede transmitir la transacción sabiendo que los fondos puede ser canjeado, si es necesario.

Este mensaje introduce el `channel_id` para identificar el canal. Se deriva de la transacción de financiación mediante la combinación de `funding_txid` y `funding_output_index`, usando big-endian OR-exclusivo (es decir, `funding_output_index` altera los últimos 2 bytes).

1. type: 35 (`funding_signed`)
2. data:
    - [`channel_id`:`channel_id`]
    - [`signature`:`signature`]

Requisitos

Ambos `peers`:

- si `channel_type` estaba presente tanto en `open_channel` como en `accept_channel`:
    - Este es el `channel_type` (deben ser iguales, requerido arriba)
    - de lo contrario:
        - si se negoció `option_anchors_zero_fee_htlc_tx`:
            - el `channel_type` es `option_anchors_zero_fee_htlc_tx` y `option_static_remotekey` (bits 22 y 12)
        - de lo contrario, si se negoció `option_anchor_outputs`:
            - el `channel_type` es `option_anchor_outputs` y `option_static_remotekey` (bits 20 y 12)
        - de lo contrario, si se negoció `option_static_remotekey`:
            - el `channel_type` es `option_static_remotekey` (bit 12)
        - de lo contrario:
            - el `channel_type` está vacío
    - DEBE usar ese `channel_type` para todas las transacciones de compromiso.

El remitente DEBE establecer:

- `channel_id` por OR exclusivo del `funding_txid` y el `funding_output_index` del mensaje `funding_created`.

- `signature` a la firma válida, utilizando su `funding_pubkey` para la transacción de compromiso inicial, como se define en BOLT #3.

El receptor:

- si la `signature` es incorrecta O no cumple con la regla LOW-S-estándar[LOWS]:
  - DEBE enviar una `warning` y cerrar la conexión, o enviar un `error` y falla el canal.
- NO DEBE transmitir la transacción de financiación antes de recibir un `funding_signed` válido.
- al recibir un `funding_signed` válido:
  - DEBE transmitir la transacción de financiación.

Base Lógica

Decidimos por `option_static_remotekey`, `option_anchor_outputs` o`option_anchors_zero_fee_htlc_tx` en este punto cuando primero tenemos que generarla transacción de compromiso. Los bits de características que se comunicaron en elintercambio de mensajes `init` para la conexión actual determinar el canalformato de compromiso para el tiempo de vida total del canal. Incluso si es más tardela reconexión no negocia este parámetro, este canal continuaráuse `option_static_remotekey`, `option_anchor_outputs` o`option_anchors_zero_fee_htlc_tx`; no admitimos la "rebaja de categoría".`option_anchors_zero_fee_htlc_tx` se considera superior a`option_anchor_outputs`, que de nuevo se considera superior a`option_static_remotekey`, y se favorece la superior si hay más de unase negocia.

El mensaje `channel_ready`

Este mensaje (que solía llamarse `funding_locked`) indica que la transacción de financiación tiene suficientes confirmaciones para el uso del canal. Una vez que ambos nodos han enviado esto, el canal entra en modo de funcionamiento normal.

Tenga en cuenta que el abridor es libre de enviar este mensaje en cualquier momento (ya que presumiblemente confía en sí mismo), pero el nodo que acepta normalmente esperaría hasta que la financiación haya alcanzado la `minimum_depth` solicitada en `accept_channel`.

1. type: 36 (`channel_ready`)
2. data:
   - [`channel_id`:`channel_id`]
   - [`point`:`second_per_commitment_point`]
   - [`channel_ready_tlvs`:`tlvs`]
3. `tlv_stream`: `channel_ready_tlvs`
4. types:
   1. type: 1 (`short_channel_id`)
   2. data:
      - [`short_channel_id`:`alias`]

Requisitos

El remitente:

- NO DEBE enviar `channel_ready` a menos que sea el punto de salida proporcionado por `funding_txid` y `funding_output_index` en el mensaje `funding_created` paga exactamente `funding_satoshis` al scriptpubkey especificado en BOLT #3.
- si no es el nodo que abre el canal:
  - DEBERÍA esperar hasta que la transacción de financiación haya alcanzado la `minimum_depth` antes de enviar este mensaje.
- DEBE establecer `second_per_commitment_point` en el punto `per-commitment` que se utilizará para la transacción de compromiso #1, derivada como se especifica en BOLT #3.
- si se negoció `option_scid_alias`:
  - DEBE configurar `short_channel_id` `alias`.
- de lo contrario:
  - PUEDE establecer `short_channel_id` `alias`.
- si establece `alias`:
  - si el bit `announce_channel` se estableció en `open_channel`:
    - DEBERÍA establecer inicialmente `alias` en un valor no relacionado con el `short_channel_id` real.
  - de lo contrario:
    - DEBE establecer `alias` en un valor no relacionado con el `short_channel_id` real.
  - NO DEBE enviar el mismo `alias` para múltiples pares o usar un alias que colisiona con un `short_channel_id` de un canal en el mismo nodo.
  - Siempre DEBE reconocer el `alias` como un `short_channel_id` para los HTLC entrantes a este canal.
  - si `channel_type` tiene `option_scid_alias` establecido:
    - NO DEBE permitir HTLC entrantes a este canal usando el `short_channel_id` real
  - PUEDE enviar múltiples mensajes `channel_ready` al mismo par con diferentes valores `alias`.
- de lo contrario:
  - DEBE esperar hasta que la transacción de financiación haya alcanzado la "profundidad_mínima" antes de enviar este mensaje.

Un nodo no financiador (`fundee`):

- DEBE olvidar el canal si no ve la financiación correcta transacción después de un tiempo de espera de 2.016 bloques.

El receptor:

- PUEDE usar cualquiera de los `alias` que recibió, en los campos BOLT 11 `r`.
- si `channel_type` tiene `option_scid_alias` establecido:
  - NO DEBE utilizar el `short_channel_id` real en los campos `r` de BOLT 11.

Desde el punto de espera de `channel_ready` en adelante, cualquiera de los nodos PUEDEenvía un 'error' y falla el canal si no recibe una respuesta requerida delotro nodo después de un tiempo de espera razonable.

Base lógica

El que no financia puede simplemente olvidar que el canal existió alguna vez, ya que los fondos no están en riesgo. Si el beneficiario recordara el canal para siempre, esto crearía un riesgo de Denegación de Servicio; por lo tanto, se recomienda olvidarlo (incluso si la promesa de `push_msat` es significativa). Si el beneficiario olvida el canal antes de que se confirme, el financiador deberá transmitir la transacción de compromiso para recuperar sus fondos y abrir un nuevo canal. Para evitar esto, el financiador debe asegurarse de que la transacción de financiamiento se confirme en los próximos 2.016 bloques.

El `alias` aquí se requiere para dos casos de uso distintos. El primero es para enrutar pagos a través de canales que aún no están confirmados (ya que el `short_channel_id` real se desconoce hasta la confirmación). El segundo es proporcionar uno o más alias para usar en canales privados (incluso una vez que esté disponible un `short_channel_id` real).

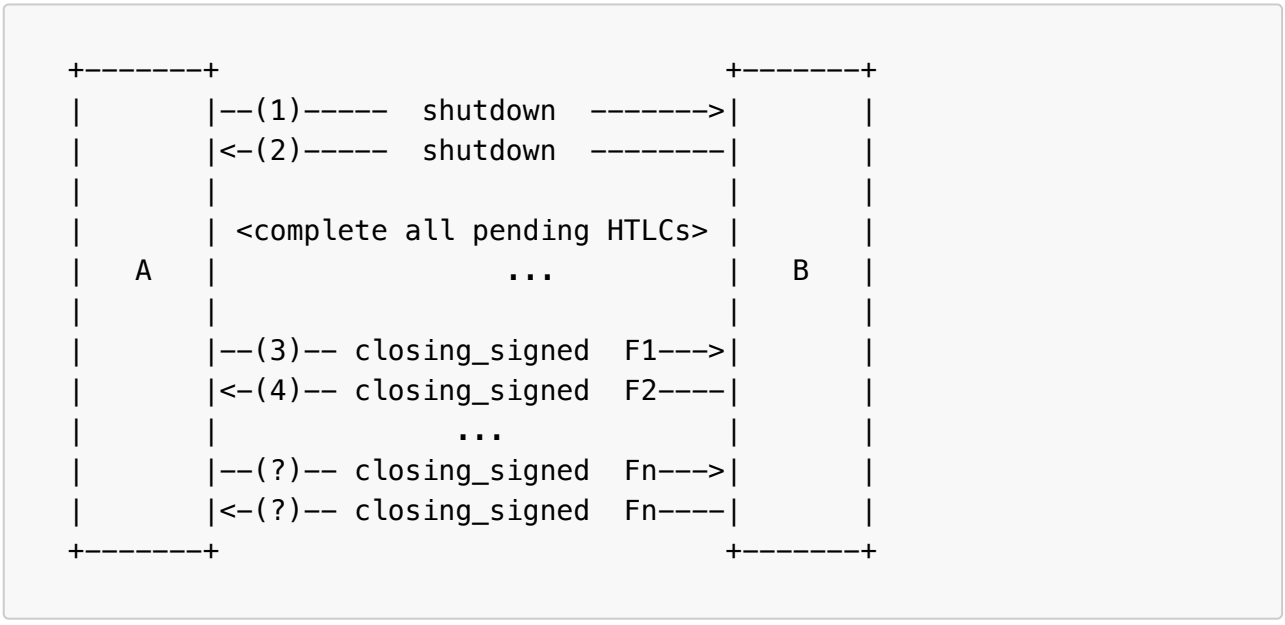
Si bien un nodo puede enviar múltiples `alias`, debe recordar todos los que ha enviado para poder usarlos en caso de que los HTLC entrantes los soliciten. El destinatario solo necesita recordar uno, para usar en las sugerencias de ruta `r` en las facturas BOLT 11.

Cierre de canal

Los nodos pueden negociar un cierre mutuo de la conexión, que a diferencia de un cierre unilateral, les permite acceder a sus fondos inmediatamente y puede negociarse con tarifas más bajas.

El cierre ocurre en dos etapas:

1. un lado indica que quiere borrar el canal (y por lo tanto no aceptará nuevos HTLC)
2. una vez que se resuelven todos los HTLC, comienza la negociación final de cierre del canal.



Closing Initiation: `shutdown`

Cualquiera de los nodos (o ambos) puede enviar un mensaje de `shutdown` para iniciar el cierre, junto con la `scriptpubkey` a la que se le quiere pagar.

1. type: 38 (`shutdown`)
2. data:
    - [`channel_id`:`channel_id`]

- [u16:len]
- [len*byte:scriptpubkey]

Requisitos

Un nodo emisor:

- si no ha enviado un funding_created (si es un financiador) o funding_signed (si es un financiador):
  - NO DEBE enviar un shutdown
- PUEDE enviar un shutdown antes de channel_ready, es decir, antes de que la transacción de financiación haya alcanzado la minimum_depth.
- si hay actualizaciones pendientes en la transacción de compromiso del nodo receptor:
  - NO DEBE enviar un shutdown.
- NO DEBE enviar múltiples mensajes de shutdown.
- NO DEBE enviar un update_add_htlc después de un shutdown.
- si no quedan HTLC en ninguna de las transacciones de compromiso (incluidos los HTLC en polvo) y ninguna de las partes tiene pendiente revoke_and_ack para enviar:
  - NO DEBE enviar ningún mensaje de actualización después de ese punto.
- DEBERÍA fallar al enrutar cualquier HTLC agregado después de haber enviado shutdown.
- si envió un shutdown_scriptpubkey de longitud distinta de cero en open_channel o accept_channel:
  - DEBE enviar el mismo valor en scriptpubkey.
- DEBE establecer scriptpubkey en una de las siguientes formas:
  1. OP_0 20 20 bytes (versión 0 paga para presenciar hash de clave pública), O
  2. OP_0 32 32 bytes (versión 0 pago para presenciar hash de secuencia de comandos), O
  3. si (y solo si) se negocia option_shutdown_anysegwit:
  - OP_1 a OP_16 inclusive, seguido de una sola pulsación de 2 a 40 bytes (programa testigo versiones 1 a 16)

Un nodo receptor:

- si no ha recibido un funding_signed (si es un financiador) o un funding_created (si es un beneficiario):
  - DEBERÍA enviar un error y fallar el canal.
- si scriptpubkey no está en una de las formas anteriores:
  - DEBERÍA enviar un warning.
- si aún no ha enviado un channel_ready:
  - PUEDE responder a un mensaje de 'apagado' con un 'apagado'

- una vez que no haya actualizaciones pendientes en el par, A MENOS QUE ya haya enviado un `shutdown`:
  - DEBE responder a un mensaje `shutdown` con `shutdown`
- si ambos nodos anunciaron la característica `option_upfront_shutdown_script`, y el nodo receptor recibió un `shutdown_scriptpubkey` de longitud distinta de cero en `open_channel` o `accept_channel`, y `shutdown_scriptpubkey` no es igual a `scriptpubkey`:
  - PUEDE enviar un `warning`.
  - DEBE fallar la conexión.

Base lógica

Si el estado del canal es siempre "limpio" (sin cambios pendientes) cuando uncomienza el apagado, se evita la cuestión de cómo comportarse si no fuera así:el remitente siempre envía primero un `commitment_signed`.Como el cierre implica un deseo de terminar, implica que no hay nuevosSe agregarán o aceptarán HTLC. Una vez que se liquidan los HTLC, no hay compromisospor el cual se debe una revocación, y todas las actualizaciones se incluyen tanto en el compromisotransacciones, el par puede comenzar inmediatamente a cerrar la negociación, por lo que prohibimos másactualizaciones de la transacción de compromiso (en particular, `update_fee` seríaposible de otra manera). Sin embargo, aunque hay HTLC en la transacción de compromiso,el iniciador puede considerar deseable aumentar la tarifa ya que puede haberHTLC en el compromiso que podría expirar.

Los formularios `scriptpubkey` incluyen solo formularios segwit estándar aceptados porla red Bitcoin, lo que garantiza que la transacción resultantepropagar a los mineros. Sin embargo, los nodos antiguos pueden enviar scripts no segwit, quepuede ser aceptado para la compatibilidad con versiones anteriores (con una advertencia para forzar el cierresi esta salida no cumple con los requisitos del relé de polvo).

La función `option_upfront_shutdown_script` significa que el nodoquería comprometerse previamente con `shutdown_scriptpubkey` en caso de que fueracomprometido de alguna manera. Este es un compromiso débil (un malévoloimplementación tiende a ignorar especificaciones como esta!), peroproporciona una mejora gradual en la seguridad al requerir la cooperacióndel nodo receptor para cambiar la `scriptpubkey`.

El requisito de respuesta `shutdown` implica que el nodo envía `commitment_signed` para confirmar cualquier cambio pendiente antes de responder; sin embargo, teóricamente podría volver a conectarse, lo que simplemente borraría todos los cambios no confirmados pendientes.

Negociación de cierre: `closing_signed`

Una vez que se completa el cierre, el canal está vacío de HTLC, no hay compromisospara los cuales se debe una revocación, y todas las actualizaciones están incluidas en ambos compromisos,las transacciones finales de compromiso actual no tendrán HTLC y la tarifa de cierrecomienza la negociación. El financiador elige una tarifa que cree que es justa, yfirma la transacción de cierre con los campos `scriptpubkey` delmensajes de `shutdown` (junto con la tarifa elegida) y envía la firma;el otro nodo luego responde de manera similar, utilizando una tarifa que considera justa. Esteel intercambio continúa hasta que ambos acuerdan la misma tarifa o cuando uno de los lados fallael canal.

En el método moderno, el financiador envía su rango de tarifas permisible y elel no financiador tiene que elegir una tarifa en este rango. Si el no financiador elige el mismovalor, la negociación se completa después

de dos mensajes, de lo contrario, el financiadorresponder con el mismo valor (completando después de tres mensajes).

1. type: 39 (`closing_signed`)
2. data:
    - [`channel_id`:`channel_id`]
    - [`u64`:`fee_satoshis`]
    - [`signature`:`signature`]
    - [`closing_signed_tlvs`:`tlvs`]
3. `tlv_stream`: `closing_signed_tlvs`
4. types:
    1. type: 1 (`fee_range`)
    2. data:
        - [`u64`:`min_fee_satoshis`]
        - [`u64`:`max_fee_satoshis`]

Requisitos

The funding node:

- after `shutdown` has been received, AND no HTLCs remain in either commitment transaction:
    - SHOULD send a `closing_signed` message.

The sending node:

- SHOULD set the initial `fee_satoshis` according to its estimate of cost of inclusion in a block.
- SHOULD set `fee_range` according to the minimum and maximum fees it is prepared to pay for a close transaction.
- if it doesn't receive a `closing_signed` response after a reasonable amount of time:
    - MUST fail the channel
- if it is not the funder:
    - SHOULD set `max_fee_satoshis` to at least the `max_fee_satoshis` received
    - SHOULD set `min_fee_satoshis` to a fairly low value
- MUST set `signature` to the Bitcoin signature of the close transaction, as specified in BOLT #3.

The receiving node:

- if the `signature` is not valid for either variant of closing transaction specified in BOLT #3 OR non-compliant with LOW-S-standard rule[LOWS]:
    - MUST send a `warning` and close the connection, or send an `error` and fail the channel.
- if `fee_satoshis` is equal to its previously sent `fee_satoshis`:
    - SHOULD sign and broadcast the final closing transaction.
    - MAY close the connection.
- if `fee_satoshis` matches its previously sent `fee_range`:
    - SHOULD use `fee_satoshis` to sign and broadcast the final closing transaction

    - SHOULD reply with a `closing_signed` with the same `fee_satoshis` value if it is different from its previously sent `fee_satoshis`
    - MAY close the connection.
- if the message contains a `fee_range`:
    - if there is no overlap between that and its own `fee_range`:
        - SHOULD send a warning
        - MUST fail the channel if it doesn't receive a satisfying `fee_range` after a reasonable amount of time
    - otherwise:
        - if it is the funder:
            - if `fee_satoshis` is not in the overlap between the sent and received `fee_range`:
                - MUST fail the channel
            - otherwise:
                - MUST reply with the same `fee_satoshis`.
        - otherwise (it is not the funder):
            - if it has already sent a `closing_signed`:
                - if `fee_satoshis` is not the same as the value it sent:
                    - MUST fail the channel
            - otherwise:
                - MUST propose a `fee_satoshis` in the overlap between received and (about-to-be) sent `fee_range`.
- otherwise, if `fee_satoshis` is not strictly between its last-sent `fee_satoshis` and its previously-received `fee_satoshis`, UNLESS it has since reconnected:
    - SHOULD send a `warning` and close the connection, or send an `error` and fail the channel.
- otherwise, if the receiver agrees with the fee:
    - SHOULD reply with a `closing_signed` with the same `fee_satoshis` value.
- otherwise:
    - MUST propose a value "strictly between" the received `fee_satoshis` and its previously-sent `fee_satoshis`.

The receiving node:

- if one of the outputs in the closing transaction is below the dust limit for its `scriptpubkey` (see BOLT 3):
    - MUST fail the channel

Base lógica

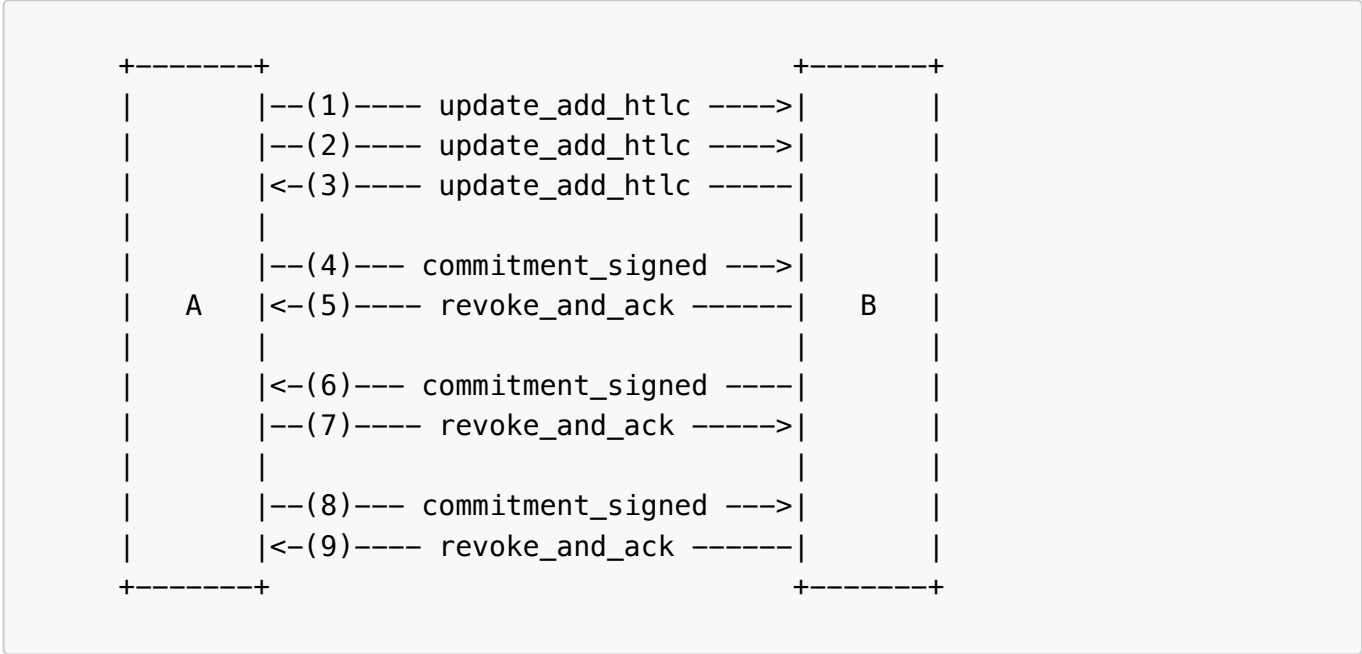
Cuando no se proporciona `fee_range`, el requisito "estrictamente entre" garantizaese progreso hacia adelante se hace, aunque solo sea por un solo satoshi a la vez. Para evitar mantener el estado y manejar el caso de la esquina, donde las tarifas se han desplazadoentre la desconexión y la reconexión, la negociación se reinicia en la reconexión. Tenga en cuenta que existe un riesgo limitado si la transacción de cierre esretrasado, pero será emitido muy pronto; por lo que normalmente no hayrazón para pagar una prima por un procesamiento rápido. Tenga en cuenta que el no financiador no está pagando la tarifa, por lo que no hay razón para ellotener una tarifa máxima. Sin embargo, puede querer una tarifa mínima para garantizarque la transacción se propaga. Siempre puede usar CPFP más tarde para acelerarconfirmación si

es necesario, por lo que el mínimo debe ser bajo. Puede suceder que la transacción de cierre no cumpla con el relé predeterminado de bitcoinpolíticas (por ejemplo, cuando se usa un script de apagado que no es segwit para una salida por debajo de 546satoshis, lo cual es posible si `dust_limit_satoshis` está por debajo de 546 satoshis). No hay fondos en riesgo cuando eso sucede, pero el canal debe cerrarse a la fuerza comola transacción de cierre probablemente nunca llegará a los mineros.

Operación normal

Una vez que ambos nodos hayan intercambiado `channel_ready` (y opcionalmente `announcement_signatures`), el canal se puede usar para realizar pagos a través de `Hashed Time Locked Contracts`.

Los cambios se envían por lotes: uno o más mensajes `update_` se envían antes de un mensaje `commitment_signed`, como en el siguiente diagrama:



Contrariamente a la intuición, estas actualizaciones se aplican a los *otros nodos* transacción de compromiso; el nodo solo agrega esas actualizaciones a su propio transacción de compromiso cuando el nodo remoto reconoce que ha los aplicó a través de `revoke_and_ack`.

Por lo tanto, cada actualización atraviesa los siguientes estados:

1. pendiente en el receptor
2. en la última transacción de compromiso del receptor
3. ... y la transacción de compromiso anterior del receptor ha sido revocada, y la actualización está pendiente en el remitente
4. ... y en la última transacción de compromiso del remitente
5. ... y se ha revocado la transacción de compromiso anterior del remitente

Como las actualizaciones de los dos nodos son independientes, los dos compromisos las transacciones pueden estar desincronizadas indefinidamente. Esto no es preocupante: lo que importa es si ambas partes se han comprometido irrevocablemente a un actualización particular o no (el estado final, arriba).

Reenvío de HTLCs

En general, un nodo ofrece HTLC por dos razones: para iniciar un pago propio, o para reenviar el pago de otro nodo. En el caso de reenvío, se debe tener cuidado para garantizar que el HTLC *saliente* no se pueda redimir a menos que el *entrante* HTLC se puede redimir. Los siguientes requisitos aseguran que esto siempre sea cierto.

La respectiva **adición/eliminación** de un HTLC se considera *irrevocablemente comprometida* cuando:

1. La transacción de compromiso **con/sin** está comprometida por ambos nodos, y cualquier transacción de compromiso anterior **sin/con** se ha revocado, O
2. La transacción de compromiso **con/sin** se ha comprometido de forma irreversible la cadena de bloques

Requisitos

Un nodo:

- until an incoming HTLC has been irrevocably committed:
  - MUST NOT offer the corresponding outgoing HTLC (`update_add_htlc`) in response to that incoming HTLC.
- until the removal of an outgoing HTLC is irrevocably committed, OR until the outgoing on-chain HTLC output has been spent via the HTLC-timeout transaction (with sufficient depth):
  - MUST NOT fail the incoming HTLC (`update_fail_htlc`) that corresponds to that outgoing HTLC.
- once the `cltv_expiry` of an incoming HTLC has been reached, OR if `cltv_expiry` minus `current_height` is less than `cltv_expiry_delta` for the corresponding outgoing HTLC:
  - MUST fail that incoming HTLC (`update_fail_htlc`).
- if an incoming HTLC's `cltv_expiry` is unreasonably far in the future:
  - SHOULD fail that incoming HTLC (`update_fail_htlc`).
- upon receiving an `update_fulfill_htlc` for an outgoing HTLC, OR upon discovering the `payment_preimage` from an on-chain HTLC spend:
  - MUST fulfill the incoming HTLC that corresponds to that outgoing HTLC.

Base lógica

En general, un lado del intercambio debe tratarse antes que el otro. Cumplir con un HTLC es diferente: el conocimiento de la preimagen es, por definición, irrevocable y el HTLC entrante debe cumplirse lo antes posible para reducir la latencia.

Un HTLC con un vencimiento injustificadamente largo es un vector de denegación de servicio y por lo tanto no está permitido. Tenga en cuenta que el valor exacto de "irrazonable" actualmente no está claro y puede depender de la topología de la red.

Selección `cltv_expiry_delta`

Una vez que se agota el tiempo de espera de un HTLC, puede cumplirse o agotarse; se debe tener cuidado con esta transición, tanto para los HTLC ofrecidos como para los recibidos.

Considere el siguiente escenario, donde A envía un HTLC a B, quien reenvía a C, quien entrega los bienes tan pronto como el pago es recibió.

1. C necesita asegurarse de que el HTLC de B no puede expirar, incluso si B se vuelve insensible; es decir, C puede cumplir con el HTLC entrante en la cadena antes de que B pueda tiempo de espera en la cadena.

2. B necesita estar seguro de que si C cumple con el HTLC de B, puede cumplir con el HTLC entrante de A; es decir, B puede obtener la preimagen de C y cumplir con la entrada HTLC en la cadena antes de que A pueda desconectarlo en la cadena.

La configuración crítica aquí es `cltv_expiry_delta` en BOLT #7 y el `min_final_cltv_expiry_delta` relacionado en BOLT #11. `cltv_expiry_delta` es la mínima diferencia en los tiempos de espera de HTLC CLTV, en la caja de reenvío (B). `min_final_cltv_expiry_delta` es la mínima diferencia entre el tiempo de espera de HTLC CLTV y la altura de bloque actual, para el caja de terminales (C).

Tenga en cuenta que un nodo está en riesgo si acepta un HTLC en un canal y ofrece un HTLC en otro canal con una diferencia demasiado pequeña entre los tiempos de espera de CLTV. Por esta razón, el `cltv_expiry_delta` para el canal *saliente* se usa como delta a través de un nodo.

El número de bloques en el peor de los casos entre la salida y la resolución HTLC entrante se puede derivar, dadas algunas suposiciones:

- bloques `R` de profundidad de reorganización en el peor de los casos
- un período de gracia 'G' se bloquea después del tiempo de espera de HTLC antes de darse por vencido un compañero que no responde y cae a la cadena
- un número de bloques `S` entre la transmisión de la transacción y la transacción incluida en un bloque

El peor de los casos es para un nodo de reenvío (B) que tarda más tiempo posible para detectar el cumplimiento de HTLC saliente y también toma el mayor tiempo posible para canjearlo en la cadena:

1. El B->C HTLC expira en el bloque `N`, y B espera los bloques `G` hasta que deja de esperar a que C. B o C se comprometa con la cadena de bloques,y B gasta HTLC, lo que requiere que se incluyan bloques `S`.
2. Mal caso: C gana la carrera (justo) y cumple el HTLC, B solo ve esa transacción cuando ve el bloque `N+G+S+1`.
3. En el peor de los casos: hay una reorganización `R` profunda en la que C gana y cumple B solo ve la transacción en `N+G+S+R`.
4. B ahora necesita cumplir con el HTLC A->B entrante, pero A no responde: B espera 'G' más bloques antes de abandonar la espera de A. A o B se compromete con la cadena de bloques.
5. Mal caso: B ve la transacción de compromiso de A en el bloque `N+G+S+R+G+1` y tiene para gastar la salida de HTLC, que requiere bloques `S` para ser extraídos.
6. En el peor de los casos: hay otra reorganización `R` profunda que A usa para gastar la transacción de compromiso, por lo que B ve el compromiso de A transacción en el bloque `N+G+S+R+G+R` y tiene que gastar la salida HTLC, que toma bloques `S` para ser extraídos.
7. El gasto de HTLC de B debe tener una profundidad de al menos "R" antes de que se agote el tiempo deespera, de lo contrario otra reorganización podría permitir que A agote el tiempo de espera de la transacción.

Por lo tanto, el peor de los casos es `3R+2G+2S`, asumiendo que `R` es al menos 1. Tenga en cuenta que elposibilidades de tres reorganizaciones en las que el otro nodo las gana todas esbaja para `R` de 2 o más. Dado que se usan tarifas altas (y los gastos de HTLC pueden usartarifas casi arbitrarias), `S` debe ser

pequeño durante el funcionamiento normal; a pesar de,dado que los tiempos de los bloques son irregulares, todavía hay bloques vacíos, las tarifas pueden variaren gran medida, y las tarifas no se pueden aumentar en las transacciones de HTLC, `S = 12` debe serconsiderado un mínimo. `S` es también el parámetro que puede variar más bajoataque, por lo que un valor más alto puede ser deseable cuando las cantidades no despreciables están enriesgo. El período de gracia `G` puede ser bajo (1 o 2), ya que se requiere que los nodos superen el tiempo de esperao cumplir lo antes posible; pero si `G` es demasiado bajo aumenta el riesgo decierre de canal innecesario debido a retrasos en la red.

Hay cuatro valores que deben derivarse:

1. el `cltv_expiry_delta` para canales, `3R+2G+2S`: en caso de duda, un `cltv_expiry_delta` de al menos 34 es razonable (R=2, G=2, S=12).

2. la fecha límite para los HTLC ofrecidos: la fecha límite después de la cual el canal debe fallará y se agotará el tiempo de espera en la cadena. Estos son los bloques `G` después de los HTLC `cltv_expiry`: 1 o 2 bloques es razonable.

3. la fecha límite para los HTLC recibidos que este nodo ha cumplido: la fecha límite después de en el que el canal tiene que fallar y el HTLC se cumplió en la cadena antes es `cltv_expiry`. Consulte los pasos 4-7 anteriores, que implican una fecha límite de `2R+G+S` bloques antes de `cltv_expiry`: 18 bloques es razonable.

4. el `cltv_expiry` mínimo aceptado para pagos terminales: el El peor de los casos para el nodo terminal C son los bloques `2R+G+S` (como, de nuevo, los pasos 1-3 anteriores no se aplican). El valor predeterminado en BOLT #11 es 18, que coincide con este cálculo.

Requisitos

An offering node:

- MUST estimate a timeout deadline for each HTLC it offers.
- MUST NOT offer an HTLC with a timeout deadline before its `cltv_expiry`.
- if an HTLC which it offered is in either node's current commitment transaction, AND is past this timeout deadline:
    - SHOULD send an `error` to the receiving peer (if connected).
    - MUST fail the channel.

A fulfilling node:

- for each HTLC it is attempting to fulfill:
    - MUST estimate a fulfillment deadline.
- MUST fail (and not forward) an HTLC whose fulfillment deadline is already past.
- if an HTLC it has fulfilled is in either node's current commitment transaction, AND is past this fulfillment deadline:
    - SHOULD send an `error` to the offering peer (if connected).
    - MUST fail the channel.

Añadiendo un HTLC: `update_add_htlc`

Cualquiera de los nodos puede enviar `update_add_htlc` para ofrecer un HTLC al otro,que se puede canjear a cambio de una preimagen de pago. Las cantidades están enmillisatoshi, aunque la aplicación en cadena solo es posible para todocantidades de satoshi mayores que el límite de polvo (en transacciones de compromiso, estos se redondean hacia abajo comoespecificado en BOLT #3).

El formato de la porción `onion_routing_packet`, que indica dónde se realiza el pago está destinado, se describe en BOLT #4.

1. type: 128 (`update_add_htlc`)
2. data:
    - [`channel_id`:`channel_id`]
    - [`u64`:`id`]
    - [`u64`:`amount_msat`]
    - [`sha256`:`payment_hash`]
    - [`u32`:`cltv_expiry`]
    - [`1366*byte`:`onion_routing_packet`]
3. `tlv_stream`: `update_add_htlc_tlvs`
4. types:
    1. type: 0 (`blinding_point`)
    2. data:
        - [`point`:`blinding`]

Requisitos

A sending node:

- if it is *responsible* for paying the Bitcoin fee:
    - MUST NOT offer `amount_msat` if, after adding that HTLC to its commitment transaction, it cannot pay the fee for either the local or remote commitment transaction at the current `feerate_per_kw` while maintaining its channel reserve (see Updating Fees).
    - if `option_anchors` applies to this commitment transaction and the sending node is the funder:
        - MUST be able to additionally pay for `to_local_anchor` and `to_remote_anchor` above its reserve.
    - SHOULD NOT offer `amount_msat` if, after adding that HTLC to its commitment transaction, its remaining balance doesn't allow it to pay the commitment transaction fee when receiving or sending a future additional non-dust HTLC while maintaining its channel reserve. It is recommended that this "fee spike buffer" can handle twice the current `feerate_per_kw` to ensure predictability between implementations.
- if it is *not responsible* for paying the Bitcoin fee:
    - SHOULD NOT offer `amount_msat` if, once the remote node adds that HTLC to its commitment transaction, it cannot pay the fee for the updated local or remote transaction at the current `feerate_per_kw` while maintaining its channel reserve.
- MUST offer `amount_msat` greater than 0.

- MUST NOT offer `amount_msat` below the receiving node's `htlc_minimum_msat`
- MUST set `cltv_expiry` less than 500000000.
- if result would be offering more than the remote's `max_accepted_htlcs` HTLCs, in the remote commitment transaction:
  - MUST NOT add an HTLC.
- if the sum of total offered HTLCs would exceed the remote's `max_htlc_value_in_flight_msat`:
  - MUST NOT add an HTLC.
- for the first HTLC it offers:
  - MUST set `id` to 0.
- MUST increase the value of `id` by 1 for each successive offer.
- if it is relaying a payment inside a blinded route:
  - MUST set `blinding_point` (see Route Blinding)

`id` MUST NOT be reset to 0 after the update is complete (i.e. after `revoke_and_ack` has been received). It MUST continue incrementing instead.

Un nodo receptor:

- receiving an `amount_msat` equal to 0, OR less than its own `htlc_minimum_msat`:
  - SHOULD send a `warning` and close the connection, or send an `error` and fail the channel.
- receiving an `amount_msat` that the sending node cannot afford at the current `feerate_per_kw` (while maintaining its channel reserve and any `to_local_anchor` and `to_remote_anchor` costs):
  - SHOULD send a `warning` and close the connection, or send an `error` and fail the channel.
- if a sending node adds more than receiver `max_accepted_htlcs` HTLCs to its local commitment transaction, OR adds more than receiver `max_htlc_value_in_flight_msat` worth of offered HTLCs to its local commitment transaction:
  - SHOULD send a `warning` and close the connection, or send an `error` and fail the channel.
- if sending node sets `cltv_expiry` to greater or equal to 500000000:
  - SHOULD send a `warning` and close the connection, or send an `error` and fail the channel.
- MUST allow multiple HTLCs with the same `payment_hash`.
- if the sender did not previously acknowledge the commitment of that HTLC:
  - MUST ignore a repeated `id` value after a reconnection.
- if other `id` violations occur:
  - MAY send a `warning` and close the connection, or send an `error` and fail the channel.
- if `blinding_point` is provided:
  - MUST use the corresponding blinded private key to decrypt the `onion_routing_packet` (see Route Blinding)

The `onion_routing_packet` contains an obfuscated list of hops and instructions for each hop along the path.It commits to the HTLC by setting the `payment_hash` as associated data, i.e. includes the `payment_hash` in the computation of HMACs.This prevents replay attacks that would reuse a previous `onion_routing_packet` with a different `payment_hash`.

Base lógica

Las cantidades no válidas son una clara violación del protocolo e indican un desglose.

Si un nodo no aceptaba múltiples HTLC con el mismo hash de pago, unel atacante podría sondear para ver si un nodo tenía un HTLC existente. Esterequisito, para hacer frente a los duplicados, conduce al uso de un separadoidentificador; se supone que un contador de 64 bits nunca se ajusta.

Las retransmisiones de actualizaciones no reconocidas están explícitamente permitidas parafines de reconexión; Permitirlos en otros momentos simplifica lacódigo del destinatario (aunque una verificación estricta puede ayudar a la depuración).

`max_accepted_htlcs` está limitado a 483 para garantizar que, incluso si ambospartes envían el número máximo de HTLC, el mensaje `commitment_signed`todavía estar por debajo del tamaño máximo de mensaje. También asegura queuna sola transacción de penalización puede gastar toda la transacción de compromiso,según lo calculado en BOLT #5.

Los valores de `cltv_expiry` iguales o superiores a 500000000 indicarían un tiempo en segundos, y el protocolo solo admite una caducidad en bloques.

El nodo *responsable* de pagar la tarifa de Bitcoin debe mantener una "tarifa buffer de picos" en la parte superior de su reserva para acomodar un futuro aumento de tarifas. Sin este búfer, el nodo *responsable* de pagar la tarifa de Bitcoin puede llegar a un estado en el que no puede enviar ni recibir ningún HTLC que no sea polvo mientras manteniendo su reserva de canal (debido al aumento de peso de la transacción de compromiso), lo que resulta en un canal degradado. Ver #728 para más detalles.

Eliminando un HTLC: `update_fulfill_htlc`, `update_fail_htlc`, y `update_fail_malformed_htlc`

Para simplificar, un nodo solo puede eliminar los HTLC agregados por el otro nodo. Hay cuatro razones para eliminar un HTLC: se proporciona la preimagen de pago, se agotó el tiempo de espera, no se pudo enrutar o tiene un formato incorrecto.

Para proporcionar la preimagen:

1. type: 130 (`update_fulfill_htlc`)
2. data:
    - [`channel_id`:`channel_id`]
    - [`u64`:`id`]
    - [`32*byte`:`payment_preimage`]

Para un HTLC con tiempo de espera agotado o ruta fallida:

1. type: 131 (`update_fail_htlc`)
2. data:
    - [`channel_id`:`channel_id`]
    - [`u64`:`id`]
    - [`u16`:`len`]
    - [`len*byte`:`reason`]

El campo `reason` es un blob cifrado opaco en beneficio del iniciador HTLC original, como se define en BOLT #4; sin embargo, hay una variante de fallo de malformación especial para el caso donde el par no pudo analizarlo: en este caso, el nodo actual toma medidas, encriptando en un `update_fail_htlc` para la retransmisión.

Para un HTLC no analizable:

1. type: 135 (`update_fail_malformed_htlc`)
2. data:
    - [`channel_id`:`channel_id`]
    - [`u64`:`id`]
    - [`sha256`:`sha256_of_onion`]
    - [`u16`:`failure_code`]

Requisitos

Un nodo:

- SHOULD remove an HTLC as soon as it can.
- SHOULD fail an HTLC which has timed out.
- until the corresponding HTLC is irrevocably committed in both sides' commitment transactions:
    - MUST NOT send an `update_fulfill_htlc`, `update_fail_htlc`, or `update_fail_malformed_htlc`.
- When failing an incoming HTLC:
    - If `current_blinding_point` is set in the onion payload and it is not the final node:
        - MUST send an `update_fail_htlc` error using the `invalid_onion_blinding` failure code with the `sha256_of_onion` of the onion it received, for any local or downstream errors.
        - SHOULD add a random delay before sending `update_fail_htlc`.
    - If `blinding_point` is set in the incoming `update_add_htlc`:
        - MUST send an `update_fail_malformed_htlc` error using the `invalid_onion_blinding` failure code with the `sha256_of_onion` of the onion it received, for any local or downstream errors.

Un nodo receptor:

- if the `id` does not correspond to an HTLC in its current commitment transaction:
    - MUST send a `warning` and close the connection, or send an `error` and fail the channel.
- if the `payment_preimage` value in `update_fulfill_htlc` doesn't SHA256 hash to the corresponding HTLC `payment_hash`:
    - MUST send a `warning` and close the connection, or send an `error` and fail the channel.
- if the `BADONION` bit in `failure_code` is not set for `update_fail_malformed_htlc`:
    - MUST send a `warning` and close the connection, or send an `error` and fail the channel.
- if the `sha256_of_onion` in `update_fail_malformed_htlc` doesn't match the onion it sent:
    - MAY retry or choose an alternate error response.
- otherwise, a receiving node which has an outgoing HTLC canceled by `update_fail_malformed_htlc`:
    - MUST return an error in the `update_fail_htlc` sent to the link which originally sent the HTLC, using the `failure_code` given and setting the data to `sha256_of_onion`.

Base lógica

Un nodo que no agota el tiempo de espera de los HTLC corre el riesgo de fallar en el canal (ver Selección `cltv_expiry_delta`).

Un nodo que envía `update_fulfill_htlc`, antes que el remitente, también es comprometido con el HTLC y corre el riesgo de perder fondos.

Si la cebolla tiene un formato incorrecto, el nodo ascendente no podrá extraer la clave compartida para generar una respuesta, de ahí el mensaje de error especial, que hace que este nodo lo haga.

El nodo puede verificar que el SHA256 del que se queja el flujo ascendente coincide con la cebolla que envió, lo que puede permitirle detectar bits aleatorios errores Sin embargo, sin volver a verificar el paquete cifrado real enviado, no sabrá si el error fue propio o del remoto; entonces tal detección se deja como una opción.

Los nodos dentro de una ruta ciega deben usar `invalid_onion_blinding` para evitar filtrar información a los remitentes que intentan sondear la ruta ciega.

Confirmando actualizaciones hasta ahora: `commitment_signed`

Cuando un nodo tiene cambios para el compromiso remoto, puede aplicarlos, firma la transacción resultante (como se define en BOLT #3), y envía un mensaje `commitment_signed`.

1. type: 132 (`commitment_signed`)
2. data:
    - [`channel_id`:`channel_id`]
    - [`signature`:`signature`]
    - [`u16`:`num_htlcs`]
    - [`num_htlcs*signature`:`htlc_signature`]

Requisitos

A sending node:

- MUST NOT send a `commitment_signed` message that does not include any updates.
- MAY send a `commitment_signed` message that only alters the fee.
- MAY send a `commitment_signed` message that doesn't change the commitment transaction aside from the new revocation number (due to dust, identical HTLC replacement, or insignificant or multiple fee changes).
- MUST include one `htlc_signature` for every HTLC transaction corresponding to the ordering of the commitment transaction (see BOLT #3).
- if it has not recently received a message from the remote node:
    - SHOULD use `ping` and await the reply `pong` before sending `commitment_signed`.

Un nodo receptor:

- once all pending updates are applied:
    - if `signature` is not valid for its local commitment transaction OR non-compliant with LOW-S-standard rule LOWS:
        - MUST send a `warning` and close the connection, or send an `error` and fail the channel.
    - if `num_htlcs` is not equal to the number of HTLC outputs in the local commitment transaction:

        - MUST send a `warning` and close the connection, or send an `error` and fail the channel.
- if any `htlc_signature` is not valid for the corresponding HTLC transaction OR non-compliant with LOW-S-standard rule [LOWS]:
    - MUST send a `warning` and close the connection, or send an `error` and fail the channel.
- MUST respond with a `revoke_and_ack` message.

Base lógica

Tiene poco sentido ofrecer actualizaciones de spam: esto implica un bug.

El campo `num_htlcs` es redundante, pero hace que la comprobación de la longitud del paquete sea totalmente autónoma.

La recomendación de requerir mensajes recientes reconoce la realidad de que las redes no son confiables: los nodos podrían no darse cuenta de que sus pares están fuera de línea hasta después de enviar `commitment_signed`. Una vez que se envía `commitment_signed`, el remitente se considera vinculado a esos HTLC y no puede fallar los HTLC entrantes relacionados hasta que los HTLC de salida se resuelvan por completo.

Tenga en cuenta que el `htlc_signature` aplica implícitamente el mecanismo de bloqueo de tiempo en el caso de que los HTLC ofrecidos se agoten a tiempo de espera o se gasten los HTLC recibidos. Esto se hace para reducir las tarifas mediante la creación de scripts más pequeños en comparación con el establecimiento explícito de bloqueos de tiempo en las salidas HTLC.

El `option_anchors` permite que las transacciones HTLC "traigan sus propias tarifas" adjuntando otras entradas y salidas, de ahí las banderas de firma modificadas.

Completar la transición al estado actualizado: `revoke_and_ack`

Una vez que el destinatario de `commitment_signed` verifica la firma y sabe que tiene una nueva transacción de compromiso válida, responde con la preimagen de compromiso para la transacción de compromiso anterior en un mensaje `revoke_and_ack`.

Este mensaje también sirve implícitamente como un acuse de recibo del `commitment_signed`, por lo que este es un momento lógico para que el remitente `commitment_signed` aplique (a su propio compromiso) cualquier actualización pendiente que haya enviado antes de ese `commitment_signed`.

La descripción de la derivación de claves se encuentra en BOLT #3.

1. type: 133 (`revoke_and_ack`)
2. data:
    - [`channel_id`:`channel_id`]
    - [`32*byte`:`per_commitment_secret`]
    - [`point`:`next_per_commitment_point`]

Requisitos

A sending node:

- MUST set `per_commitment_secret` to the secret used to generate keys for the previous commitment transaction.
- MUST set `next_per_commitment_point` to the values for its next commitment transaction.

Un nodo receptor:

- if `per_commitment_secret` is not a valid secret key or does not generate the previous `per_commitment_point`:
  - MUST send an `error` and fail the channel.
- if the `per_commitment_secret` was not generated by the protocol in BOLT #3:
  - MAY send a `warning` and close the connection, or send an `error` and fail the channel.

Un nodo:

- MUST NOT broadcast old (revoked) commitment transactions,
  - Note: doing so will allow the other node to seize all channel funds.
- SHOULD NOT sign commitment transactions, unless it's about to broadcast them (due to a failed connection),
  - Note: this is to reduce the above risk.

Actualizando Fees: `update_fee`

El nodo que paga la tarifa de Bitcoin envía un mensaje `update_fee`. Como cualquier actualización, primero se compromete con la transacción de compromiso del receptor y luego (una vez reconocida) se confirma con la del remitente. A diferencia de un HTLC, `update_fee` nunca se cierra sino que simplemente se reemplaza.

Existe la posibilidad de una carrera, ya que el destinatario puede agregar nuevos HTLC antes de recibir el `update_fee`. En esta circunstancia, es posible que el remitente no pueda pagar la tarifa en su propia transacción de compromiso, una vez que el destinatario finalmente reconozca la `update_fee`. En este caso, la tarifa será inferior a la tasa de la tarifa, como se describe en BOLT #3.

The exact calculation used for deriving the fee from the fee rate is given in BOLT #3.

El cálculo exacto utilizado para derivar la tarifa a partir del `fee rate` se proporciona en BOLT #3.

1. type: 134 (`update_fee`)
2. data:
   - [`channel_id`:`channel_id`]
   - [`u32`:`feerate_per_kw`]

Requisitos

The node *responsible* for paying the Bitcoin fee:

- SHOULD send `update_fee` to ensure the current fee rate is sufficient (by a significant margin) for timely processing of the commitment transaction.

The node *not responsible* for paying the Bitcoin fee:

- MUST NOT send `update_fee`.

Un nodo receptor:

- if the `update_fee` is too low for timely processing, OR is unreasonably large:
  - MUST send a `warning` and close the connection, or send an `error` and fail the channel.
- if the sender is not responsible for paying the Bitcoin fee:
  - MUST send a `warning` and close the connection, or send an `error` and fail the channel.
- if the sender cannot afford the new fee rate on the receiving node's current commitment transaction:
  - SHOULD send a `warning` and close the connection, or send an `error` and fail the channel.
    - but MAY delay this check until the `update_fee` is committed.

Base lógica

Se requieren tarifas de Bitcoin para que los cierres unilaterales sean efectivos. Con `option_anchors`, `feerate_per_kw` ya no es tan crítico para garantizar la confirmación como lo era en el formato de compromiso heredado, pero aún debe ser suficiente para poder ingresar al mempool (satisfacer la tarifa mínima de retransmisión y la tarifa mínima de mempool).

Para el formato de compromiso heredado, no existe un método general para que el nodo de broadcasting use el elemento `child-pays-for-parent` para aumentar su tarifa efectiva.

Dada la variación en las tarifas y el hecho de que la transacción puede gastarse en el futuro, es una buena idea que el pagador de la tarifa mantenga un buen margen (digamos 5 veces el requisito de la tarifa esperada) para txes de compromiso heredadas; pero, debido a los diferentes métodos de estimación de tarifas, no se especifica un valor exacto.

Dado que las tarifas actualmente son unilaterales (la parte que solicitó la creación del canal siempre paga las tarifas por la transacción de compromiso), lo más simple es permitirle solo establecer niveles de tarifas; sin embargo, dado que se aplica la misma tasa de tarifa a las transacciones de HTLC, el nodo receptor también debe preocuparse por que la tarifa sea razonable.

Retransmisión de mensaje

Debido a que los transportes de comunicación no son confiables y es posible que deban restablecerse de vez en cuando, el diseño del transporte se ha separado explícitamente del protocolo.

No obstante, se supone que nuestro transporte es ordenado y confiable. La reconexión introduce dudas sobre lo recibido, por lo que hay reconocimientos explícitos en ese punto.

Esto es bastante sencillo en el caso del establecimiento y cierre del canal, donde los mensajes tienen un orden explícito, pero durante el funcionamiento normal, los reconocimientos de actualizaciones se retrasan hasta el intercambio `commitment_signed` / `revoke_and_ack`; por lo que no se puede asumir que se han recibido las actualizaciones. Esto también significa que el nodo receptor solo necesita almacenar actualizaciones al recibir `commitment_signed`.

Tenga en cuenta que los mensajes descritos en BOLT #7 son independientes de canales particulares; sus requisitos de transmisión están cubiertos allí, y además de ser transmitidos después de `init` (como todos los mensajes), son independientes de los requisitos aquí.

1. type: 136 (`channel_reestablish`)
2. data:

    - [channel_id:channel_id]
    - [u64:next_commitment_number]
    - [u64:next_revocation_number]
    - [32*byte:your_last_per_commitment_secret]
    - [point:my_current_per_commitment_point]

next_commitment_number: un número de compromiso es un contador incremental de 48 bits para cada transacción de compromiso; los contadores son independientes para cada par en el canal y comienzan en 0. Solo se transmiten explícitamente al otro nodo en caso de restablecimiento; de lo contrario, son implícitos.

Requisitos

Un nodo financiador:

- upon disconnection:
    - if it has broadcast the funding transaction:
        - MUST remember the channel for reconnection.
    - otherwise:
        - SHOULD NOT remember the channel for reconnection.

A non-funding node:

- upon disconnection:
    - if it has sent the funding_signed message:
        - MUST remember the channel for reconnection.
    - otherwise:
        - SHOULD NOT remember the channel for reconnection.

Un nodo:

- MUST handle continuation of a previous channel on a new encrypted transport.
- upon disconnection:
    - MUST reverse any uncommitted updates sent by the other side (i.e. all messages beginning with update_ for which no commitment_signed has been received).
        - Note: a node MAY have already used the payment_preimage value from the update_fulfill_htlc, so the effects of update_fulfill_htlc are not completely reversed.
- upon reconnection:
    - if a channel is in an error state:
        - SHOULD retransmit the error packet and ignore any other packets for that channel.
    - otherwise:
        - MUST transmit channel_reestablish for each channel.
        - MUST wait to receive the other node's channel_reestablish message before sending any other messages for that channel.

The sending node:

- MUST set next_commitment_number to the commitment number of the next commitment_signed it expects to receive.

- MUST set `next_revocation_number` to the commitment number of the next `revoke_and_ack` message it expects to receive.
- if `option_static_remotekey` applies to the commitment transaction:
  - MUST set `my_current_per_commitment_point` to a valid point.
- otherwise:
  - MUST set `my_current_per_commitment_point` to its commitment point for the last signed commitment it received from its channel peer (i.e. the commitment_point corresponding to the commitment transaction the sender would use to unilaterally close).
- if `next_revocation_number` equals 0:
  - MUST set `your_last_per_commitment_secret` to all zeroes
- otherwise:
  - MUST set `your_last_per_commitment_secret` to the last `per_commitment_secret` it received

Un nodo:

- if `next_commitment_number` is 1 in both the `channel_reestablish` it sent and received:
  - MUST retransmit `channel_ready`.
- otherwise:
  - MUST NOT retransmit `channel_ready`, but MAY send `channel_ready` with a different `short_channel_id` `alias` field.
- upon reconnection:
  - MUST ignore any redundant `channel_ready` it receives.
- if `next_commitment_number` is equal to the commitment number of the last `commitment_signed` message the receiving node has sent:
  - MUST reuse the same commitment number for its next `commitment_signed`.
- otherwise:
  - if `next_commitment_number` is not 1 greater than the commitment number of the last `commitment_signed` message the receiving node has sent:
    - SHOULD send an `error` and fail the channel.
  - if it has not sent `commitment_signed`, AND `next_commitment_number` is not equal to 1:
    - SHOULD send an `error` and fail the channel.
- if `next_revocation_number` is equal to the commitment number of the last `revoke_and_ack` the receiving node sent, AND the receiving node hasn't already received a `closing_signed`:
  - MUST re-send the `revoke_and_ack`.
  - if it has previously sent a `commitment_signed` that needs to be retransmitted:
    - MUST retransmit `revoke_and_ack` and `commitment_signed` in the same relative order as initially transmitted.
- otherwise:
  - if `next_revocation_number` is not equal to 1 greater than the commitment number of the last `revoke_and_ack` the receiving node has sent:
    - SHOULD send an `error` and fail the channel.
  - if it has not sent `revoke_and_ack`, AND `next_revocation_number` is not equal to 0:
    - SHOULD send an `error` and fail the channel.

Un nodo receptor:

- if `option_static_remotekey` applies to the commitment transaction:

    - if `next_revocation_number` is greater than expected above, AND `your_last_per_commitment_secret` is correct for that `next_revocation_number` minus 1:
        - MUST NOT broadcast its commitment transaction.
        - SHOULD send an `error` to request the peer to fail the channel.
    - otherwise:
        - if `your_last_per_commitment_secret` does not match the expected values:
            - SHOULD send an `error` and fail the channel.
- otherwise, if it supports `option_data_loss_protect`:
    - if `next_revocation_number` is greater than expected above, AND `your_last_per_commitment_secret` is correct for that `next_revocation_number` minus 1:
        - MUST NOT broadcast its commitment transaction.
        - SHOULD send an `error` to request the peer to fail the channel.
        - SHOULD store `my_current_per_commitment_point` to retrieve funds should the sending node broadcast its commitment transaction on-chain.
    - otherwise (`your_last_per_commitment_secret` or `my_current_per_commitment_point` do not match the expected values):
        - SHOULD send an `error` and fail the channel.

Un nodo:

- MUST NOT assume that previously-transmitted messages were lost,
    - if it has sent a previous `commitment_signed` message:
        - MUST handle the case where the corresponding commitment transaction is broadcast at any time by the other side,
            - Note: this is particularly important if the node does not simply retransmit the exact `update_` messages as previously sent.
- upon reconnection:
    - if it has sent a previous `shutdown`:
        - MUST retransmit `shutdown`.

Base lógica

Los requisitos anteriores aseguran que la fase de apertura sea casi atómica: si no se completa, comienza de nuevo. La única excepción es si el mensaje `funding_signed` se envía pero no se recibe. En este caso, el financiador olvidará el canal y, presumiblemente, abrirá uno nuevo al volver a conectarse; mientras tanto, el otro nodo eventualmente olvidará el canal original, debido a que nunca recibió `channel_ready` o vio la transacción de financiación en la cadena.

No hay confirmación de `error`, por lo que si se produce una reconexión, es de buena educación retransmitir antes de desconectarse de nuevo; sin embargo, no es IMPRESCINDIBLE, porque también hay ocasiones en las que un nodo puede simplemente olvidar el canal por completo.

`closing_signed` tampoco tiene acuse de recibo, por lo que debe retransmitirse en la reconexión (aunque la negociación se reinicia en la reconexión, por lo que no es necesario que sea una retransmisión exacta). El único acuse de recibo para `shutdown` es `closing_signed`, por lo que es necesario retransmitir uno u otro.

El manejo de las actualizaciones es igualmente atómico: si no se reconoce la confirmación (o no se envió), las actualizaciones se vuelven a enviar. Sin embargo, no se insiste en que sean idénticos: podrían estar en un orden diferente, implicar tarifas diferentes o incluso faltar HTLC que ahora son demasiado antiguos para agregarse. Requerir que sean idénticos significaría efectivamente una escritura en el disco por parte del remitente en cada transmisión, mientras que el esquema aquí fomenta una sola escritura persistente en el disco para cada "compromiso_firmado" enviado o recibido. Pero si necesita retransmitir tanto un `commitment_signed` como un `revoke_and_ack`, el orden relativo de estos dos debe conservarse, de lo contrario, se cerrará el canal.

Nunca se debe solicitar una retransmisión de `revoke_and_ack` después de que se haya recibido un `closing_signed`, ya que eso implicaría que se ha completado un apagado, lo que solo puede ocurrir después de que el nodo remoto haya recibido `revoke_and_ack`.

Tenga en cuenta que `next_commitment_number` comienza en 1, ya que el número de compromiso 0 se crea durante la apertura. `next_revocation_number` será 0 hasta que se envíe `commitment_signed` para el número de compromiso 1 y luego se reciba la revocación para el número de compromiso 0.

`channel_ready` se reconoce implícitamente por el inicio de la operación normal, que se sabe que comenzó después de que se recibió un `commitment_signed`; por lo tanto, la prueba para un `next_commitment_number` mayor que 1.

Un borrador anterior insistía en que el financiador "DEBE recordar... si ha transmitido la transacción de financiación, de lo contrario NO DEBE": de hecho, este era un requisito imposible. Un nodo debe, en primer lugar, persistir en disco y, en segundo lugar, transmitir la transacción o viceversa. El nuevo lenguaje refleja esta realidad: ¡sin duda es mejor recordar un canal que no ha sido transmitido que olvidar uno que sí lo ha sido! deje que el financiador lo abra mientras el beneficiario lo ha olvidado.

`option_data_loss_protect` se agregó para permitir que un nodo, que de alguna manera se haya quedado atrás (por ejemplo, se haya restaurado desde una copia de seguridad anterior), detecte que se ha quedado atrás. Un nodo retrasado debe saber que no puede transmitir su transacción de compromiso actual, lo que conduciría a la pérdida total de fondos, ya que el nodo remoto puede demostrar que conoce la preimagen de revocación. El `error` devuelto por el nodo retrasado debería hacer que el otro nodo abandone su transacción de compromiso actual en la cadena. El otro nodo debe esperar ese `error` para darle al nodo retrasado la oportunidad de corregir su estado primero (por ejemplo, reiniciando con una copia de seguridad diferente).

Si el nodo retrasado no tiene la última copia de seguridad, al menos le permitirá recuperar fondos que no sean HTLC, si `my_current_per_commitment_point` es válido. Sin embargo, esto también significa que el nodo caído ha revelado este hecho (aunque no de forma demostrable: podría estar mintiendo), y el otro nodo podría usar esto para transmitir un estado anterior.

`option_static_remotekey` elimina la clave cambiante `to_remote`, por lo que `my_current_per_commitment_point` es innecesario y, por lo tanto, se ignora (para simplificar el análisis, sigue siendo y debe ser un punto válido, sin embargo), pero la divulgación del secreto anterior aún permite la detección de retrasos. Sin embargo, una implementación puede ofrecer ambas cosas y recurrir al comportamiento `option_data_loss_protect` si `option_static_remotekey` no se negocia.

Authors

[FIXME: Insert Author List]