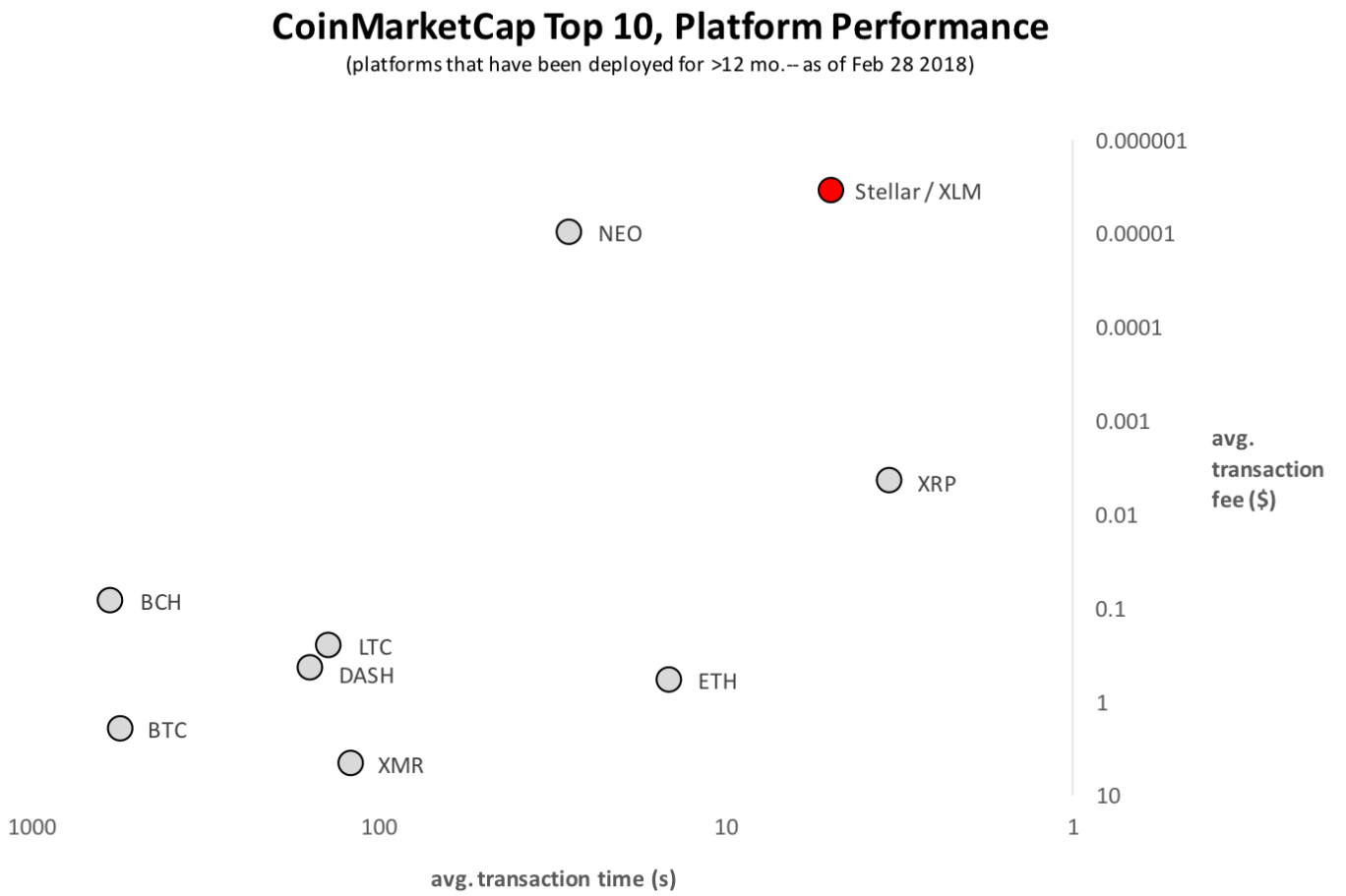
Lightning on Stellar: Technical Spec and Roadmap

Christian, March 19, 2018

We want Stellar to become the world's digital payment rail. We're already the **most deployment-ready** of the major platforms (see the below chart), but given the scale of the future we see for Stellar, we know we need to keep pushing our technology forward.



Scalability—namely, how best to achieve it—has been at the center of some of the most bitter disagreements in blockchain. We've tried to approach the problem with an open mind. To the extent an idea improves what our users care about—speed, throughput, privacy—we will explore it, and since a typical Lightning payment:

– can be confirmed instantly
– has negligible fees
– doesn't have to become public

the protocol has always interested us. As we said in our 2018 Roadmap it's now clear that Lightning is the right way forward for Stellar.

How Lightning Works

Lightning is a scaling solution for distributed payment networks, originally proposed for the Bitcoin blockchain. Lightning is designed to allow users to make off-chain payments through routers and hubs. Lightning even has the potential to support cross-protocol payments, such as a payment where the sender sends Bitcoins on the Bitcoin network and the recipient receives lumens on the Stellar network, without having to trust any parties in between.

Lightning is constructed from building blocks known as *payment channels*. The concept behind payment channels is simple but powerful. They allow users to open a channel off-chain and transact there instead of on the public ledger. Because they're off-chain, transactions in the channel can be extremely fast and cheap, but similar to on-chain transactions, there's no counterparty risk. When the channel participants are ready to go their separate ways, they close the channel and settle back to the public ledger. No matter what happened in-channel, the rest of the world only sees that final transaction. It's like showing someone the last frame of a movie; from that one still, there's no way to unpack the rest of the film.

Developers have begun working on payment channel designs and implementations for several chains and ledgers beyond just Bitcoin, including Ethereum and Zcash. Each platform's channels are unique and depend on the nuances of the platform, but as a rule, any implementation will support a few basic requirements:

– No transaction submitted to the network, except when parties disengage
– No loss of funds caused by cheating parties
– No vulnerability to third-party interference
– No channel-side speed bottlenecks

Stellar supports a more flexible generalization of payment channels called *state channels*, meaning that any operation you can execute on the Stellar network (such as not only payments, but also creating, deleting, or changing permissions on accounts), you can execute within a payment channel.

Stellar's state channel implementation relies on the fact that every Stellar transaction specifies a

source account and a **sequence number**. We've figured out how to use those sequence numbers as a natural versioning mechanism for off-chain payments; it's similar to how your bank gets alerted for out-of-order cheques. To do the versioning, we're taking advantage of a new operation, BUMP_SEQUENCE, which we'll describe in complete detail below.

Our release timeline for Lightning on Stellar is:

Apr 1	BUMP_SEQUENCE pushed to a testnet
Aug 1	State channels beta implementation
Oct 1	State channels on Stellar livenet + Lightning Network beta
Dec 1	Lightning Network on Stellar livenet

Stellar's creator, Jed McCaleb, first explored Lightning back in 2015; our 2018 implementation still reflects the cleverness of his original plan, but Jeremy Rubin, with the support of Nicolas Barry and David Mazières from SDF, has added the necessary improvements to make Lightning right for us. The explanation that follows is theirs.

State Channels on Stellar

This post describes how state channels can be implemented on Stellar. In future posts, we will show how these state channels can be chained together using Hashed Timelock Contracts (HTLCs), to enable multi-hop payments and interoperability with Lightning Network implementations on other chains (to allow atomic cross-chain trades of Bitcoin for lumens, for example). This design is not finalized, and we strongly encourage feedback from other researchers and the community as we work

toward a production-ready specification and implementation.

A state channel is an arrangement among n users, $u_1\ldots u_n$, who wish to perform off-chain transactions that settle back as side-effects (net payments, but also account creations/deletions, etc). The users collaborate to create a series of "snapshot transactions"—sequences of side-effects, $T_1, T_2, \ldots, T_k$, such that only the last sequence, T_k, will ever be executed on the public ledger. To ensure that T_j cannot be executed once users create T_{j+1}, the protocol makes a synchrony assumption: it assumes that all participants can observe and respond to the ledger—including overcoming any downtime or DoS attacks—within some bounded delay D, such as a week.

To implement state channels on Stellar, we take advantage of the fact that every Stellar transaction specifies a source account and a sequence number. A transaction's sequence number must match the monotonically increasing sequence number of its source account. Our approach will be to assign successively higher ranges of sequence numbers on an escrow account R to the transactions in each sequence T_j. The sequence T_j cannot initially execute because its sequence numbers are too high. However, once all users have signed T_j, they go on to sign a second set of "ratchet transactions," V_j, that raise account R's sequence number to the point at which T_j can execute. Raising R's sequence number also permanently invalidates the snapshot transactions T_i for i < j? This is where the synchrony assumption plays in. Transactions in sets V_j and T_j are given time bounds such that the earliest time at which T_j can execute is at least D delay after the latest time at which V_j can execute. This delay allows other users to notice that V_j has been submitted and counter by submitting V_k, thereby ensuring T_k can be executed and T_j cannot.

To support state channels as well as some other applications, Stellar is adding a new operation, BUMP_SEQUENCE. The new operation enables transactions to arbitrarily increase the sequence number of a target account. You can see the proposed semantics of BUMP_SEQUENCE here.

We begin the protocol specification with the presumption of a set of users and accounts such that:

$$
\begin{equation*}
\begin{split}
N &=& \text{ number of users}\\
sk_{i, a} &=& \text{ secret key known by user i for account a}\\
G &=& \text{ aggregation group, for example a Schnorr group}\\
A_i &=& \text{ i^{th} account with key $sk_{i, A_i} \times G$}\\
$$

q_i &=& \text{ sequence number of i^{th} account at start of protocol}\\
u_i &=& \text{ software node for user i maintaining state}\\
&&T_{next},v_{max},
T_{max},
A_i,
sk_{i, A_i},
sk_{i, R}
\\
\end{split}
\end{equation*}
\]

Our state channel is set up using an escrow account \(R\).

\[
\begin{equation*}
\begin{split}
R_{sk} &= \text{secret key for R (e.g., a aggregated key from $sk_{*, R}$ of all participants)}\\
R_{pk} &= R_{sk} \times G\\
R_q &= \text{sequence number of account R at start of protocol}\\
R_{q'} &= \text{current sequence number of account } R\\
\end{split}
\end{equation*}
\]

While this specification describes an escrow account that uses a single aggregated public key from the private keys of the participants, an alternative (used by the example implementation described below) would be to use an N-of-N multisignature account, with one key for each of the participants.

The channel state will be updated in rounds, for which \(M_j\) is the start time of round \(j\).

First we create a transaction sequence \(T_1\) that disburses the contents of \(R\).

\[
\begin{equation*}
\begin{split}
T_1 &=& \text{ }t_{1, 1}\ldots t_{1, n}\\
t_{1, i} &=& \text{ transaction in 1^{st} round of agreement on behalf of participant i with}\\

$$
&& minTime = M_1 + \text{timeout_claim} + \text{timeout_claim_delay} \\
&& maxTime = \infty \\
&& operations = \ldots \small \text{// undo deposit }P\\
&& sequence = R_q+n+i\\
&& source = R
\end{split}
\end{equation*}
$$

Note that because of the sequence selected, it is not immediately usable. We must first create a set of transactions V_1 to bump the sequence number of R to the appropriate value.

$$
\begin{equation*}
\begin{split}
V_1 &=& \text{ set of transactions for 1^{st} round of agreement for sequence bumping }R\\
v_{j, i} &=&\text{ transaction in jth round of on A_i which is}\\
&&minTime = M_1\\\
&&maxTime = M_1\ + \text{timeout_claim}\\
&&operations = \text{sequence_bump } R_q \to R_{q+n}\\
&&signers = R_{pk}, A_i\\
&&sequence = q_i\\
&&source = A_i\\
\end{split}
\end{equation*}
$$

Then users can sign and submit a compound transaction jointly funding R.

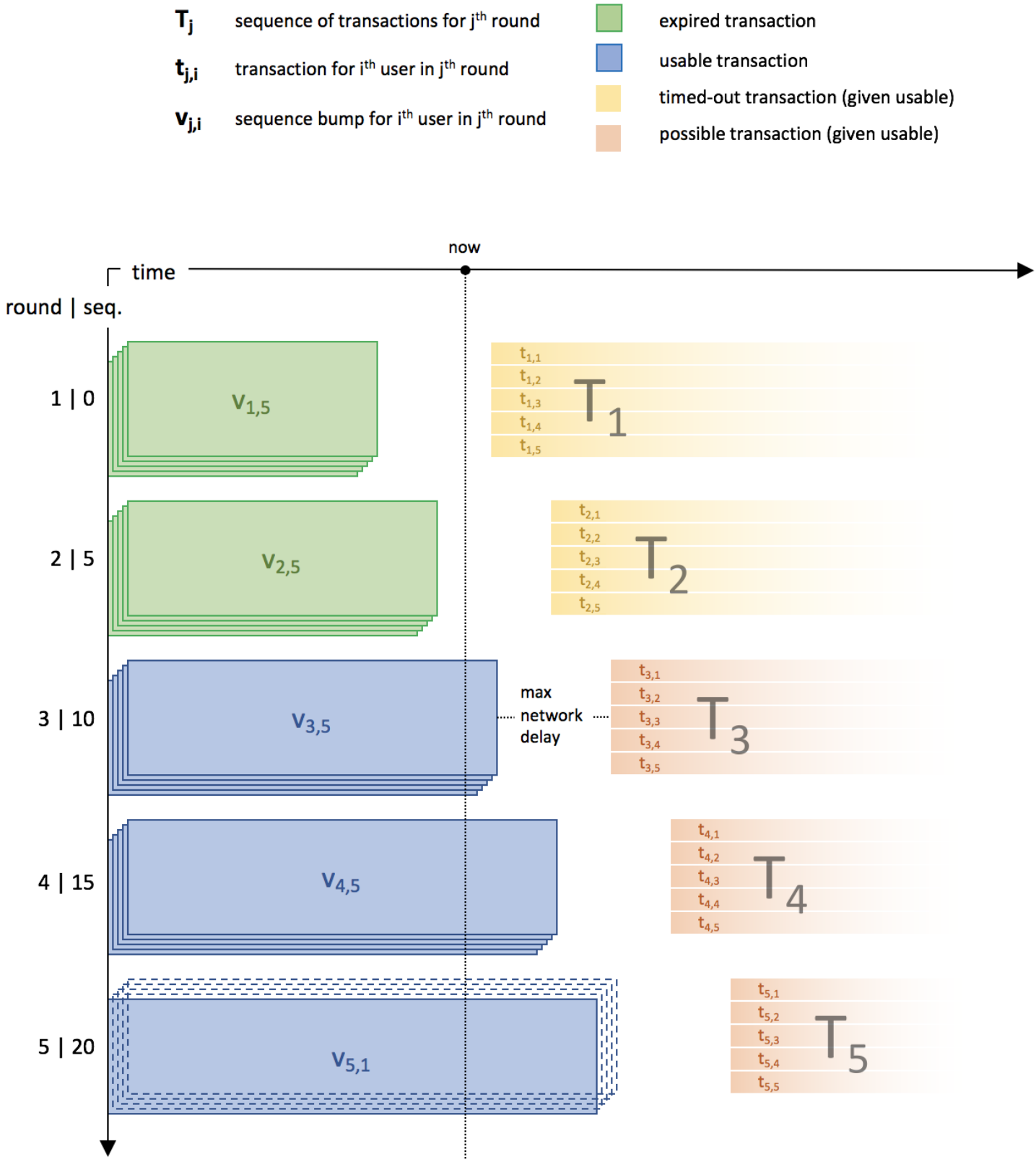
$$
P = \text{joint deposit into } R
$$

Each off-chain payment then consists of creating a new sequence of transactions T_j and V_j disbursing the funds in R so as to effect net settlement of the first j off-chain transactions.

$$

$$
\begin{equation*}
\begin{split}
T_j &=& \text{ sequence of transactions for j^{th} round of agreement settling out of }R\\
t_{j,i} &=& \text{ transaction in j^{th} round of agreement on behalf of participant i with}\\
&&minTime = M_j\ + \text{timeout_claim_delay} + \text{timeout_claim}\\
&&maxTime = \infty\\
&&operations = c_{j,i}\\
&&sequence = R_q+j*(n+1)+i\\
&&source = R\\
c_{j,i} &=& \text{ operations that satisfy u_i’s claims over the assets in }R\\
\\
V_j &=& \text{ set of transactions for j^{th} round of agreement for sequence bumping }R\\
v_{j, i} &=& \text{ transaction in j^{th} round of on A_i which is}\\
&&minTime = M_j\\
&&maxTime = M_j + \text{timeout_claim}\\
&&operations = \text{sequence_bump } R \to R_{q+j*(n+1)}\\
&&signers = R_{pk}, A_i\\
&&sequence = q_i\\
&&source = A_i\\
\end{split}
\end{equation*}
$$

To help illustrate this, here is a visualization of a Stellar lightning channel in the process of updating from round 4 to round 5.



Update Rules

snapshot update \(j\):

\(T_{next} = [] \)
for \(i \in 1\ldots k\) {
  generate \(t_{j,i}\)
  for each \(k \in 1…n \) {
    \(t_{j, i} \gets signed(t_{j, i}, sk_{k, R}) \)
  }
  \(T_{next} \gets T_{next} + [t_{j,i}]\)
}
for each \(u_i \in U\){
  \(T_{next}^{u_i} \gets T_{next}\)
}
for each \(i \in 1…n\) {
  generate \(v_{j, i}\)
  for each \(k \in \{ 1…n \} – \{ i \} \) {
    \(v_{j, i} \gets signed(v_{j, i}, sk_{k, i}) \)
    atomically {
      \(T_{max}^{u_i} \gets T_{next}^{u_i}\)
      \(v_{max}^{u_i} \gets v_{j, i}\)
    }
  }
}

\(a_k\) monitor:

\(r \gets R_{q}\)
wait until \(R_{q'} > r \)
\(r \gets bumpTo(operations({v_{max}}) \))
if \(R_{q'} = r\) {
  wait until \(minTime(T_{max})\)
  publish \(T_{max}\)
} **else if** \(R_{q'} < r \) {
  publish \(v_{max}\)
  wait until \(minTime(T_{max})\)
  publish \(T_{max}\)
} **else if** \(R_{q'} >= r + n \) {
  // we are corrupted; no clear next action
} **else if** \(R_{q'} < r + n\) {

publish \(T_{max}[R_{q'}…r+n]\)
}

timeout:

loop {
wait until \(maxTime(v_{max}) – now <= D \)
try {
snapshot update to \(T'\) where \(seq(T') = seq(T_{max}) + 1\)
} **catch** {
publish \(v_{max}\)
}
}

honest_close:

if \(1 = |\{bumpTo(operations(v_{max}^{u_i})[0]) | i \in 1\ldots n\} |\) **and** corresponding \(T_{max} = T_{next} \) {
snapshot update to \(T'\) **where** \(seq(T') = seq(T_{max}) + 1\) **and** \(minTime(T') = now \)
publish \(v_{max}\)
publish \(T_{max}\)
}

Example Using JavaScript SDK

We'll use the Stellar JavaScript SDK to show how one can create a state channel between Alice and Bob. This example is simplified for educational purposes and does not implement a fully functional payment channel, nor does it precisely reflect the specification or final implementation.

The channel will have 1000 lumens deposited into it, with an initial balance of 250 for Alice and 750 for Bob. We will then have them sign transactions that update the balance to 500/500, without any of those transactions having to hit the chain. Finally, they will close the channel.

Alice and Bob need to select values for TIMEOUT_CLAIM and TIMEOUT_CLAIM_DELAY based on their payment frequency and network connectivity expectations (including the synchrony assumption for the network, \(D\)). TIMEOUT_CLAIM_DELAY should be at least \(D\), whereas TIMEOUT_CLAIM

should be at least \(D\) plus the maximum expected time between rounds. To be able to use concrete time periods in the examples below, we will pick a value of one week for \(D\), set TIMEOUT_CLAIM_DELAY to 1 week, and set TIMEOUT_CLAIM to 2 weeks. (These times are unrealistically conservative, but should be easy to track in the below examples.)

We'll start like this:

```
const moment = require('moment')
const bigInt = require('big-integer')
const {
  Account,
  Asset,
  Keypair,
  Network,
  Operation,
  Server,
  TransactionBuilder,
} = require('stellar-sdk')

const TIMEOUT_CLAIM = moment.duration(2, 'week').seconds()
const TIMEOUT_CLAIM_DELAY = moment.duration(1, 'week').seconds()

const server = new Server('https://horizon-testnet.stellar.org')
Network.useTestNetwork()

// Alice and Bob are preexisting funded accounts controlled by AliceKeypair and BobKeypai


const AliceKeypair = Keypair.fromSecret('SCIXVMGTGHIOVMHRA7B7ICJ4XWAYSQP67VNSLNXS7OYZKXDS7

const AliceKey = AliceKeypair.publicKey()
const Alice = await server.loadAccount(AliceKeypair.publicKey())

// Alice generates throwaway keys for her version account and for the ratchet account
const AliceVersionKeypair = Keypair.random()
const AliceRatchetKeypair = Keypair.random()

const AliceVersionKey = AliceVersionKeypair.publicKey()
const AliceRatchetKey = AliceRatchetKeypair.publicKey()

// Bob does the same
const BobKeypair = Keypair.fromSecret('SAJ2ISPPRUA4MPCDFOILZ6E4H3X6I4OVTMPX4QZBLXTMWMSKO5M

const BobKey = BobKeypair.publicKey()
const Bob = await server.loadAccount(BobKey)

const BobVersionKeypair = Keypair.random()
const BobRatchetKeypair = Keypair.random()

const BobVersionKey = BobVersionKeypair.publicKey()
```

```
const BobRatchetKey = BobRatchetKeypair.publicKey()

// the Ratchet account ID is Alice's ratchet key
const RatchetAccountId = AliceRatchetKeypair.publicKey()
```

We then create three accounts:

```
const setupAccountsTx = new TransactionBuilder(Alice)
  .addOperation(
    Operation.createAccount({
      destination: AliceVersionKey,
      startingBalance: '1',
    })
  )
  .addOperation(
    Operation.createAccount({
      destination: BobVersionKey,
      startingBalance: '1',
    })
  )
  .addOperation(
    // set up the ratchet account
    // which initially has only Alice's ratchet key
    // the funding transaction will add Bob's key
    Operation.createAccount({
      destination: AliceRatchetKey,
      startingBalance: '2',
    })
  )
  .build()

setupAccountsTx.sign(AliceKeypair)
await server.submitTransaction(setupAccountsTx)

const AliceVersion = await server.loadAccount(AliceVersionKey)
const BobVersion = await server.loadAccount(BobVersionKey)
const Ratchet = await server.loadAccount(RatchetAccountId)
```

Alice and Bob must now prepare round 0 before funding the channel.

First, they prepare snapshot transactions reflecting their current balances, and exchange their signatures on them.

```
const Round0Time = moment().unix()
const RatchetSequenceNumber = bigInt(Ratchet.sequenceNumber())
const Ratchet0SequenceNumber = RatchetSequenceNumber.plus(3)
```

```
const Snapshot0Alice = new TransactionBuilder(
  new Account(RatchetAccountId, Ratchet0SequenceNumber.toString()),
  {
    timebounds: {
      minTime: Round0Time + TIMEOUT_CLAIM + TIMEOUT_CLAIM_DELAY,
      maxTime: 0,
    },
  }
)
  .addOperation(
    Operation.payment({
      destination: Alice.accountId(),
      asset: Asset.native(),
      amount: '250',
    })
  )
  .build()

const Snapshot0Bob = new TransactionBuilder(
  new Account(
    RatchetAccountId,
    Ratchet0SequenceNumber.plus(1).toString()
  ),
  {
    timebounds: {
      minTime: Round0Time + TIMEOUT_CLAIM + TIMEOUT_CLAIM_DELAY,
      maxTime: 0,
    },
  }
)
  .addOperation(
    // gives control over the ratchet, and its remaining 750 lumens, to Bob
    Operation.setOptions({
      signer: { ed25519PublicKey: BobKey, weight: 2 },
    })
  )
  .build()

// exchange signatures
Snapshot0Bob.sign(AliceRatchetKeypair)
Snapshot0Alice.sign(BobRatchetKeypair)
```

They then exchange their initial Ratchet transactions, which will bump the sequence number of the ratchet account to the sequence number immediately preceding the snapshot transactions. (Note that this will not work yet in the existing SDK, because the BUMP_SEQUENCE operation is not yet supported on the network.)

```
const Ratchet0Alice = new TransactionBuilder(
  new Account(AliceVersion.accountId(), AliceVersion.sequenceNumber()),
  { timebounds: { minTime: Round0Time, maxTime: Round0Time + TIMEOUT_CLAIM } }
```

```
)
  .addOperation(
    Operation.BumpSequence({
      sourceAccount: RatchetKey,
      target: Ratchet0SequenceNumber.minus(1).toString(),
    })
  )
  .build()

const Ratchet0Bob = new TransactionBuilder(
  new Account(BobVersion.accountId(), BobVersion.sequenceNumber()),
  { timebounds: { minTime: Round0Time, maxTime: Round0Time + TIMEOUT_CLAIM } }
)
  .addOperation(
    Operation.BumpSequence({
      sourceAccount: RatchetKey,
      target: Ratchet0SequenceNumber.minus(1).toString(),
    })
  )
  .build()
```

Now that the snapshot transactions and ratchet transactions are in place, either Alice or Bob will have the ability to close the channel and receive their portion of the lumens. This means it is now safe for Alice and Bob to fund the channel.

```
const fundingTx = new TransactionBuilder(Ratchet)
  .addOperation(
    Operation.payment({
      source: Alice.accountId(),
      destination: Ratchet.accountId(),
      asset: Asset.native(),
      amount: '248', // Alice has already paid in 2 lumens
    })
  )
  .addOperation(
    Operation.payment({
      source: Bob.accountId(),
      destination: Ratchet.accountId(),
      asset: Asset.native(),
      amount: '750',
    })
  )
  .addOperation(
    Operation.setOptions({
      signer: { ed25519PublicKey: BobRatchetKey, weight: 1 },
      lowThreshold: 2,
      medThreshold: 2,
      highThreshold: 2,
    })
  )
  .build()
```

```
fundingTx.sign(AliceKeypair)
fundingTx.sign(BobKeypair)
fundingTx.sign(AliceRatchetKeypair)

await server.submitTransaction(fundingTx)
```

Now the channel is fully set up. If, at this point, either Alice or Bob were to act dishonestly (e.g. by going offline or refusing to respond) either party can initiate their ratchet transaction, then the snapshot transactions, to get back to their initial state.

Critically, the redeeming party must act within the specified time range. In this case, if there are no further rounds in the channel and Bob does not cooperate in creating further rounds, Alice should attempt to close the channel within one week (to give herself at least \(D\) time to get her transaction included). She must then wait two weeks (a total of three weeks from the start time of the channel) for the snapshot transactions to become valid.

Now, Bob wants to pay Alice 250 lumens over the channel. In other words, they want to update the channel state, so the balances change from 250/750 (with Alice owning 250) to 500/500.

Alice and Bob create new snapshot transactions, reflecting the updated state, and exchange their signatures on them.

```
const Ratchet1SequenceNumber = Ratchet0SequenceNumber.plus(3)
const Ratchet1Account = new Account(
  Ratchet.accountId(),
  Ratchet1SequenceNumber.toString()
)

const Round1Time = moment().unix()

const Snapshot1Alice = new TransactionBuilder(
  new Account(RatchetAccountId, Ratchet1SequenceNumber.toString()),
  {
    timebounds: {
      minTime: Round1Time + TIMEOUT_CLAIM + TIMEOUT_CLAIM_DELAY,
      maxTime: 0,
    },
  }
```

They now can create and exchange signatures on new ratchet transactions:

```
const Ratchet1Bob = new TransactionBuilder(
  new Account(BobVersion.accountId(), BobVersion.sequenceNumber()),
  { timebounds: { minTime: Round1Time, maxTime: Round1Time + TIMEOUT_CLAIM } }
```

```
)
  .addOperation(
    Operation.BumpSequence({
      sourceAccount: RatchetKey,
      target: Ratchet1SequenceNumber.minus(1).toString(),
    })
  )
  .build()

const Ratchet1Alice = new TransactionBuilder(
  new Account(AliceVersion.accountId(), AliceVersion.sequenceNumber()),
  { timebounds: { minTime: Round1Time, maxTime: Round1Time + TIMEOUT_CLAIM } }
)
  .addOperation(
    (Operation as any).BumpSequence({
      sourceAccount: RatchetKey,
      target: Ratchet1SequenceNumber.minus(1).toString(),
    })
  )
  .build()

Ratchet1Bob.sign(AliceRatchetKeypair)
Ratchet1Alice.sign(BobRatchetKeypair)
```

This payment is now done. **Note that none of these transactions are broadcast to the network.**

However, there's now a potential problem—Alice and Bob still have valid ratchet and snapshot transactions from round 0, when their balances were different. What happens if Bob tries to submit those transactions, to close the channel at an outdated state?

Each of Alice and Bob should therefore monitor the network to detect any transactions from the other's version account. If they detect one, they should immediately submit the ratchet transaction from the latest round.

```
const streamHandler = server
  .transactions()
  .forAccount(BobVersion.accountId())
  .cursor('now')
  .stream({
    onmessage: async function(transaction) {
      if (transaction.hash !== Ratchet1Bob.hash().toString('hex')) {
        await server.submitTransaction(Ratchet1Alice)
      }
    },
  })
```

To ensure that there is enough time for Alice or Bob to challenge any invalid submission, they should

make sure that rounds happen frequently enough that the remaining time that latest ratchet transaction is valid is at least as long as D, so they will have time to respond to any submissions of stale ratchet transactions.

The parties can add as many payments as they like to the channel by creating and signing new snapshot transactions reflecting the new state of the channel, as well as ratchet transactions that set up those snapshot transactions. For each new round, the starting sequence number of the snapshot transactions is incremented by 3. None of these transactions need to be submitted to the network.

Finally, to close the channel, Alice and Bob sign and submit closing transactions to the network, using the balances from the latest snapshot transactions. These transactions are similar to the previous rounds—involving both ratchet transactions and snapshot transactions—except that that snapshot transactions do not need time bounds, and only one shared Ratchet transaction is required.

```
const CooperativeCloseSequenceNumber = Ratchet1SequenceNumber.plus(3)

const CooperativeCloseSnapshotAlice = new TransactionBuilder(
  new Account(RatchetAccountId, CooperativeCloseSequenceNumber.toString())
)
  .addOperation(
    Operation.payment({
      destination: Alice.accountId(),
      asset: Asset.native(),
      amount: '500',
    })
  )
  .build()

const CooperativeCloseSnapshotBob = new TransactionBuilder(
  new Account(
    RatchetAccountId,
    CooperativeCloseSequenceNumber.plus(1).toString()
  )
)
  .addOperation(
    Operation.setOptions({
      signer: { ed25519PublicKey: BobKey, weight: 2 },
    })
  )
  .build()

CooperativeCloseSnapshotAlice.sign(AliceRatchetKeypair)
CooperativeCloseSnapshotBob.sign(AliceRatchetKeypair)

CooperativeCloseSnapshotAlice.sign(BobRatchetKeypair)
CooperativeCloseSnapshotBob.sign(BobRatchetKeypair)
```

```
const CooperativeCloseRatchet = new TransactionBuilder(
  new Account(Ratchet.accountId(), RatchetInitialSequenceNumber.toString()),
  { timebounds: { minTime: ClosingTime, maxTime: ClosingTime + TIMEOUT_CLAIM } }
)
  .addOperation(
    Operation.BumpSequence({
      target: CooperativeCloseSequenceNumber.minus(1).toString(),
    })
  )
  .build()

CooperativeCloseRatchet.sign(AliceRatchetKeypair)
CooperativeCloseRatchet.sign(BobRatchetKeypair)

await server.submitTransaction(CooperativeCloseRatchet)
await server.submitTransaction(CooperativeCloseSnapshotAlice)
await server.submitTransaction(CooperativeCloseSnapshotBob)
```

This is one way to do a safe honest close; there are others that reveal even less information to the network.

Informal Proof

We can informally prove that at this point we have made it impossible to close at the initial channel state after the next round has completed. Our argument generalizes to any number of prior states, and also holds when Alice and Bob's roles are reversed.

Assume Bob is malicious and Alice is honest.

1. Alice is monitoring the network for activity on the BobVersion account.
2. After the second round, Bob submits Ratchet0Bob.
3. Alice now has at least \(D\) to submit a later ratchet transaction to counter Bob's. (This relies on Alice enforcing this invariant—i.e., if her latest ratchet transaction, in this case Ratchet1Alice, is less than \(D\) away, she must immediately close the channel or submit the ratchet transaction, to ensure that Bob isn't able to submit a stale ratchet transaction that leaves her too little time to respond).
4. Alice can submit Ratchet1Alice, to set up Snapshot1Alice.

5. Because all of Bob's ratchet transcations use the same sequence number, there are no other operations Bob can use to affect the Ratchet account
6. Once the minTime on Snapshot1Alice is satisfied, Alice can use it to redeem her funds.

Assume Bob disappears after Snapshot1Alice, and Alice is honest.

1. Alice requests an honest_close, but Bob is unreachable.
2. Alice submits Ratchet1Alice before it expires.
3. Alice waits until the minTime of Snapshot1Alice has been satisfied.
4. Alice submits Snapshot1Alice.

Assume Bob disappears part-way through a payment or honest close—after signing Snapshot1Alice and Snapshot1Bob, but before signing Ratchet1Alice—and Alice is honest.

1. Neither party has the applicable ratchet transaction for the second round, so that payment is never finalized.
2. Alice can use Ratchet0Alice and Alice0Snapshot to close the channel.

Assume Bob disappears part-way through a payment or honest close—after receiving Alice's signature on BobRatchet1, but before giving Alice his signature on Ratchet1Alice—and Alice is honest.

1. Alice proceeds with AliceRatchet0.
2. If Bob comes back online before BobRatchet1 expires, he can contest with BobRatchet1. After the delay period, either can submit Snapshot1Alice and Snapshot1Bob.
3. Alternatively, if Bob doesn't come back online, Alice can wait until BobRatchet1 has expired and the minTime of Alice0Snapshot has been satisfied, and submit Alice0Snapshot.

Assume Alice disappears part-way through a payment or honest close—after creating Snapshot1Alice, but before creating Ratchet1Alice—while Bob is honest.

1. Neither party has Ratchet1Alice or Ratchet1Bob.
2. Bob forces a close using Ratchet0Bob, Snapshot0Alice, and Bob0Snapshot.

Future Work

This is a simple design for payment channels on Stellar, but there is still much work to be done.

We're currently working on support for multi-hop payments, increased privacy and scalability, and interoperability with Lightning Network channels on other blockchains such as Bitcoin. If you're interested in helping us build out our protocol, join us on GitHub or StackExchange.

DEVELOPER | STELLAR NEWS | TECHNICAL POSTS