Δ: Solving the Bridging Trilemma

Ryan Zarick Bryan Pellegrino Caleb Banister

Abstract

Despite continuously growing demand to transfer assets between blockchains, there still does not exist an adequate solution for bridging native assets between chains. Existing cross-chain bridges have three key problems: (1) they rely on intermediate or wrapped tokens, (2) they can only support a small, limited network of chains, and (3) they cannot be composed with smart contracts on the destination chain. The use of intermediate tokens necessitates an additional swap on the destination chain to exchange those intermediate tokens for native tokens, and the limited scale causes inconvenience to users who must often bridge assets multiple times across different bridges to reach their final destination chain. To make matters worse, the additional swap on the destination chain cannot be composed with the original transfer, creating unnecessary work for users who must initiate the final swap manually. In this paper, we present the Delta (Δ) algorithm, a novel resource balancing algorithm that leverages *cross-chain liquidity* to enable a new class of cross-chain bridge (ΔBridge) that deals purely in native assets while still maintaining instant guaranteed finality. With Δ, large networks of blockchains connect to enable the quick and easy transfer of native assets from chain to chain.

1 Introduction

Transferring liquidity between chains is a common and important task in today's blockchain ecosystem, with users constantly moving liquidity to take advantage of opportunities on different chains. This movement of liquidity is made possible by *bridges*, services that facilitate cross-chain token swaps [3]. Unfortunately, prior work in cross-chain bridging must make a compromise in functionality due to the *bridging trilemma*, giving up one of the following: *Instant guaranteed finality* (Section 2.1), *unified liquidity* (Section 2.3), or *native asset transactions*. In practice, currently available bridges choose to forego *native asset transactions*, instead relying on lock-and-mint semantics where they lock assets on the source chain and mint a synthetic asset on the destination chain [8]. This ultimately results in a poor user experience, as users are forced to manually swap the intermediate tokens for native assets in a separate transaction on the destination chain.

However, the advent of omnichain communication protocols such as LayerZero [4] offers the opportunity to solve the bridging trilemma. Omnichain communication protocols enable reliable bidirectional inter-chain communication in densely connected networks of chains, making it possible to directly bridge native assets without sacrificing instant guaranteed finality. In this paper, we present the Delta (Δ) algorithm, a novel resource balancing algorithm which, in conjunction with omnichain communication, solves the bridging trilemma by enabling unified liquidity without compromising instant guaranteed finality. Bridges built on the Δ algorithm (Δ*Bridge*s) conduct native asset transfers through unified pools of liquidity while achieving instant guaranteed finality, the combination of which enables cross-chain composability (Section 2.2), and scalability. By eliminating overheads associated with lock-and-mint, ΔBridges provide benefit to both users and liquidity providers (LPs)–users no longer have to bridge assets multiple times to acquire native assets on the destination chain, and LPs can achieve high capital efficiency by staking into a single-sided asset pool while collecting fees from all incoming transfers regardless of the source chain. In addition to this, unified liquidity allows ΔBridges to easily scale to vast networks of chains, overcoming a key limitation of existing bridges. All of these features are enabled by the Δ algorithm, described in Section 3.

2 Background

In this paper, we refer to a *network* or *chain network* as the collection of chains that participate in the cross-chain asset exchange protocol. All chains in a network are *connected* to all other chains in the network via a pair of unidirectional "connections", over which native assets can be transferred directly. Each connection is backed by liquidity on the receiving end to facilitate withdrawals by the user as part of the transfer protocol–the amount of available liquidity on the receiving chain can be thought of as the "bandwidth" of the connection.

2.1 Instant guaranteed finality

Instant guaranteed finality is the guarantee that any transfer request that is committed on the source chain will be successfully committed on the destination chain as well, with the key implication being that enough liquidity is available on the destination chain to facilitate the transfer. Most existing bridges, such as Avalanche Bridge [2] and AnySwap [9], achieve instant guaranteed finality by locking the user's asset on the source chain and minting a corresponding synthetic asset on the destination chain–this mechanism is functionally equivalent to an unbounded liquidity pool, eliminating the possibility of insufficient liquidity on the destination chain. It is essential that a bridge provides instant guaranteed finality, as the absence of any guarantees of finality necessitates a transaction reversion mechanism which will negatively affect the user experience or profitability of the service. If a transaction must be reverted, the bridge has to either (1) let the user manually revert, (2) collect sufficient gas for reversion from the user upfront, or (3) finance the reversion costs itself. Pushing the responsibility of reverting the transaction onto the user is cumbersome and expensive, and collecting enough gas to conduct *and* revert the transaction upfront is not only inefficient, but also difficult to predict accurately due to gas price fluctuations. Financing the reversion costs as part of the service does provide a seamless user experience, but exposes the provider to an attack vector of spammed transactions to an endpoint with an empty liquidity pool.

However, instant guaranteed finality by itself is insufficient. For example, AnySwap can provide instant guaranteed finality by minting ANY tokens on the destination chain in response to a user request to transfer assets. At first glance, this may seem reasonable, but it does not include the actual swap from ANY to the final desired token, instead relying on the user to conduct another swap on the destination chain. ΔBridges, eliminate the use of this intermediate token and provide instant guaranteed finality of the entire *end-to-end* transfer of the native asset from the source chain to the destination chain.

2.2 Cross-chain composability

Traditionally, "composability" has described the ability to combine multiple smart contracts in a single transaction on a single chain. However, this term is insufficient to describe the capabilities of Δ; ΔBridges are *cross-chain* composable, which allows a cross-chain transfer to be composed with smart contracts on the destination chain, a feature not present on any existing bridge. Currently, bridges can only provide traditional composability, allowing composition with other smart contracts on the source chain but not the destination chain. However, a cross-chain asset transfer through a ΔBridge can be composed with other smart contracts on both the source chain and the destination chain, maximizing the degree of flexibility. For example, an asset can be swapped on the source chain, bridged to the destination chain using a ΔBridge, then swapped on the destination chain, all within the same cross-chain transaction. Cross-chain composability provides a level of convenience far exceeding any existing solution, and opens up a multitude of new opportunities for cross-chain applications.

2.3 Unified liquidity

In contrast to existing cross-chain bridges which mint their own tokens on the destination chain in response to the user locking up assets on the source chain, ΔBridges conduct transfers without involving any intermediate tokens. This direct use of native assets is what enables ΔBridges to scale better than other cross-chain bridges, and is made possible by a pool of liquidity maintained on every chain in the Δ network. Users "deposit" their assets into the liquidity pool on the source chain and "withdraw" the corresponding assets from the liquidity pool on the destination chain. All deposits and withdrawals occur purely with native assets, and no intermediate tokens are minted at any point during the bridging process.

The simplest way to manage this liquidity would be to keep a wholly separate pool for each pairwise connection, eliminating the possibility of race conditions and thus trivially providing guaranteed finality. We term this approach *fractured* liquidity, with each connection given exclusive access to deposit and withdraw from its assigned pool of liquidity. However, the fractured approach to liquidity requires that when a new chain is added to the network, a new liquidity pool must be created on every existing chain. In other words, the required number of liquidity pools scales quadratically in relation to the number of chains in the network. Due to this rapid growth in total required liquidity, fractured liquidity is undesirable and impractical for networks of any non-trivial size, and the scalability of any bridge that uses fractured liquidity is severely limited. In addition to this,

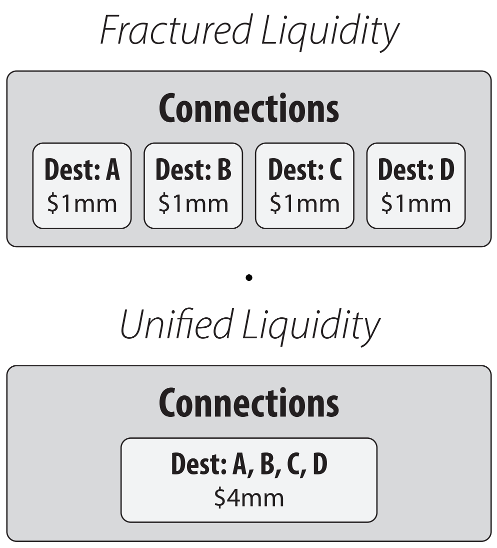

Figure 1: Fractured versus unified liquidity. ΔBridges use unified liquidity to provide high scalability and reduced overhead for LPs.

fractured liquidity requires LPs to stake into each connection's liquidity pool separately, requiring LPs to separately fund multiple liquidity pools to collect fees for the corresponding connections.

To solve these problems, it is necessary to use *unified* liquidity, where all connections deposit and withdraw from a single pool of liquidity. We illustrate the difference between fractured and unified liquidity in Figure 1: in a fractured scheme, each chain maintains a separate liquidity pool per connection, whereas in a unified scheme, all connections share a single pool. Unified liquidity does not come free, however, as it creates an additional challenge to providing instant guaranteed finality: if multiple concurrent transactions withdraw from the same liquidity pool, care must be taken to ensure that the liquidity pool is never exhausted before all transactions can complete. The Δ algorithm solves this problem, enabling unified liquidity while maintaining instant guaranteed finality.

3 Δ Algorithm Design

In this section, we describe the Δ algorithm in the context of a hypothetical ΔBridge. We begin with a simple intuitive explanation of the algorithm, then present the local state and parameters that are stored locally on each chain in Tables 1 and 2 respectively. We then present the Δ algorithm in Figure 4, and describe in detail each step in Section 3.2. Finally, we formally prove that the Δ algorithm provides instant guaranteed finality in Section 3.3.

To provide an intuitive explanation of the Δ algorithm, we define the concept of *cross-chain liquidity*; with cross-chain liquidity, each chain in the network maintains a single pool of liquidity that is soft-partitioned into slices that belong to each of the remote chains in the network. For example, in a network consisting of chains X, Y, and Z, \$100 of liquidity available locally on chain X would be *soft-partitioned* into \$50 belonging to chain Y and \$50 belonging to chain Z; this is illustrated in Figure 2. Our key insight is that it is possible to borrow and return liquidity between these soft partitions if care is taken to prevent overdrafts or race conditions, allowing the Δ algorithm to keep these partitions balanced in the face of imbalanced transaction volume. Each of the partitions represents the bandwidth available on the unidirectional channel connecting chain Y or Z to chain X, and we refer to the channel having a *deficit* if the bandwidth falls below its initial value, and a *surplus* if it exceeds its initial value.

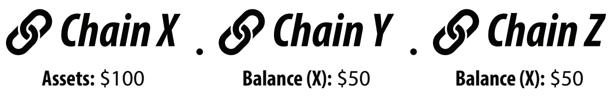

Figure 2: Δ soft-partitions a unified liquidity pool between remote chains, allowing ΔBridges to take advantage of unified liquidity without compromising guaranteed finality.

A transfer consists of a deposit on the source chain and a corresponding withdrawal on the destination chain. Upon receiving a transfer request from chain X to chain Y, the Δ algorithm follows the following simple rules:

1. If *any* channel on chain X has a deficit, distribute all or part of the newly deposited funds to close the deficit.

2. Any remaining funds after closing all deficits is distributed across all channels based on the associated weight.

In addition to the above rules, the Δ algorithm carefully manages local state to enable cross-chain liquidity while incurring the minimum number of cross-chain messages that must be sent–this is achieved by keeping track of all distributed funds in locally stored *credits*, which are then opportunistically piggy-backed onto user transactions to notify the associated remote chain. On the other side of the transaction, each source chain lazily tracks an estimate of the channel bandwidth with every other chain in the network, referred to as *balance* in Figure 1. By ensuring that this *balance* never exceeds the actual channel bandwidth, the Δ algorithm is able to guarantee sufficient liquidity for every transfer, and by extension instant guaranteed finality.

Name	Notation	Function
Liquidity provided	lp_s	(Initial) assets deposited
Assets	a_s	Size of liquidity pool (on local chain S)
Balance	$b_{s,d}$	Local allocation of funds for transfers from S to D
Last known balance	$lkb_{d,s}$	The last known value of $b_{d,s}$ observed by chain S
Credits	$c_{s,d}$	Funds to be sent in next transfer from S to D

Table 1: The Δ algorithm requires the above state variables to be stored on each chain. s **and** d **are the source (i.e., local) and destination chains, respectively. For variables that depend on** d**, a copy is stored for each destination chain. See Figure 3 for an illustrative example.**

Name	Notation	Function
Weights	$w_{s,d}$ $0 \leq w_{s,d} \leq 1$ $\sum_d w_{s,d} = 1$	The proportion of liquidity on chain s to be allocated to chain d

Table 2: In a Δ network, each connected chain has a single configurable weight parameter for every other chain in the network.

3.1 Notation

This section describes the notation used throughout the rest of this paper. For simplicity, in the case of a liquidity transfer from a local chain to a remote chain, we refer to the local chain as chain S (for source), and the remote chain as chain D (for destination). Subscript s (e.g. a_s) describes chain S, and subscript d (e.g. b_d) describes chain D.

Table 1 describes the state that is stored on each chain:

Liquidity provided (LP) describes the amount of liquidity staked into the liquidity pool on each chain without a corresponding withdrawal on any destination chain, typically by the service provider. For simplicity, LP can also be thought of as the initial size of the liquidity pool on the local chain, corresponding to the asset deposits by the service provider during initialization of a ΔBridge. lp_s is only stored on chain S.

Assets is the size of the liquidity pool on a given chain. This changes as user deposit and withdraw liquidity as part of transactions.

Balance describes the portion of the assets on chain D that can be used to facilitate transfers *from* chain S, essentially describing the maximum bandwidth of the connection from S to D. In other words, the balance $b_{s,d}$ stored on chain S describes the maximum amount of funds that can be transferred from chain S to chain D. The balance $b_{s,d}$ decreases when funds are transferred from chain S to chain D, but, as we explain in Section 3.2, may or may not increase when funds are transferred from chain D to chain S.

Initially, the sum of the remote balances $b_{s,d}$ for all destination chains in the network is equal to the assets on chain S. However, this will not be the case as transactions occur due to credits. It is important to note that the sum of all remote balances will always be less than or equal to the assets on the source chain, preventing a situation where the transferred funds exceed the available assets.

Last known balance (LKB) describes the last known value of the balance $b_{d,s}$ from the perspective of chain S. Put differently, the LKBs stored on chain S describe a (possibly outdated) view of how its assets are partitioned between all of the other chains in the network. As we prove in Section 3.3, LKBs are always larger or equal to the corresponding balance, making them more conservative than the true global state; in other words, a remote chain cannot have a larger balance than the local chain is aware of, preventing a situation where a transfer does not have sufficient assets to commit.

Credits is the amount of funds to be sent to a particular remote chain the next time communication occurs with that remote chain. As we describe in Section 3.2, the Δ algorithm transforms each user request into many smaller transfers to all chains in the network. The Δ algorithm uses credits as a method of batching many of these smaller transfers to reduce the operational overheads of the system, accumulating credits over multiple transactions and opportunistically communicating the total accumulated credits to the destination chain whenever a user transfers assets to that destination chain.

Table 2 describes the Δ algorithm's configurable parameters:

Weights are the only parameter that needs to be configured for the Δ algorithm. For each pair of chains S and X in the network, there are two weight parameters that must be set: $w_{s,x}$ and $w_{x,s}$, with $w_{s,x}$ stored on chain S and $w_{x,s}$ stored on chain X. The weight $w_{s,x}$ describes the proportion of LP to be allocated for transfers from chain S to chain X, allowing the protocol to allocate a higher proportion of the liquidity pool to particular chain pairs. This can be done to provide a larger liquidity buffer, allowing ΔBridges to deal with the higher transaction volume for those connections.

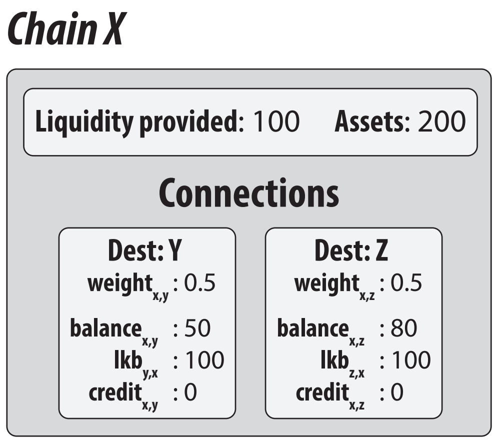

Figure 3: In a network consisting of chains X, Y, and Z, a ΔBridge stores on chain X local state corresponding to chains Y and Z. Values are purely illustrative.

Figure 3 shows an example of how state is stored on a given chain. Given a network containing chains X, Y, and Z, the local chain X keeps track of a balance, LKB, and credit for each remote chain (Y and Z). Weights are assigned during initial configuration, and can be modified as the system operates (e.g., when a new chain is added to the network). The exact mechanism by which new chains are added to the network can vary by implementation and is beyond the scope of this paper.

3.2 The Δ algorithm

In this section, we present the Δ algorithm in full and describe the intuition behind each step. Figure 4 gives the pseudocode for the full algorithm. The fundamental goal of the Δ algorithm is to maintain the proportional relationship between each remote balance and its respective asset pool while also keeping each balance at or above its initial value. Put more precisely, the goal is to maintain the following condition:

$$b_{d,s} \approx a_s \times w_{s,d}$$

The algorithm begins by the source chain S receiving a user request to transfer t units of liquidity to chain D. We start by checking whether there are sufficient funds on chain D to facilitate the transfer, rejecting the transfer if the balance between chain S and chain D ($b_{s,d}$) is smaller than the transfer size t (lines 1–3). Provided the balance is sufficient, we update the assets (a_s) and balance ($b_{s,d}$) to reflect the deposit from the user and the user's intent to withdraw from chain D, respectively (lines 4–5).

Following the balance and asset update, we begin the main algorithm on line 6. The quantity diff$_{s,x}$ is calculated for each remote chain X that chain S is connected

Input: Transaction amount t, destination chain ID d

```
    # On the source chain:
 1: if $b_{s,d} < t$ then
 2:     Reject the transfer
 3: end if
 4: $a_s \leftarrow a_s + t$
 5: $b_{s,d} \leftarrow b_{s,d} - t$
 6: for $x \neq s$ do
 7:     $\text{diff}_{s,x} \leftarrow \max(0,\ lp_s \times w_{s,x} - (lkb_{x,s} + c_{s,x}))$
 8: end for
 9: Total $\leftarrow \sum_x \text{diff}_{s,x}$
10: for $x \neq s$ do
11:     $\text{diff}_{s,x} \leftarrow min(Total, t) \times \frac{\text{diff}_{s,x}}{Total}$
12: end for
13: $t' \leftarrow$ t - min$(Total, t)$
14: for $\forall x$ do
15:     $c_{s,x} \leftarrow c_{s,x} + \text{diff}_{s,x} + t' \times w_{s,x}$
16: end for
17: $msg = (t, c_{s,d})$
18: $lkb_{d,s} \leftarrow lkb_{d,s} + c_{s,d}$
19: $c_{s,d} \leftarrow 0$
20: Send $msg$ to chain $d$

    # On the destination chain:
21: Receive $(t,\ c_{s,d})$ from chain $s$
22: $a_d \leftarrow a_d - t$
23: $b_{d,s} \leftarrow b_{d,s} + c_{s,d}$
24: $lkb_{s,d} \leftarrow lkb_{s,d} - t$
```

Figure 4: The Δ algorithm. Lines 1–20 describe the transaction on the source chain and lines 21–24 describe the transaction on the destination chain.

to (lines 6–8). Assets in the amount of diff$_{s,x}$ must be allocated to chain X to return it to its original balance. I.e., diff$_{s,x}$ is the difference between this target balance and the current balance on the remote chain X. The max() function serves to prevent diff$_{s,x}$ from being negative. Intuitively, this step is calculating, for every chain X that chain S is connected to, whether chain X has enough balance to continue facilitating transactions to chain S. If the balance on chain X is too low, then the Δ algorithm redistributes some of its assets in an attempt to restore chain X's balance to its initial value based on the LP of chain S and the weight corresponding to chain X. The balance on the remote chain X is calculated as the sum of the last known balance on chain X and any credits to chain X. This does not necessarily reflect the immediate value of $b_{x,s}$, as credits are only known to chain S. However, the credit information will eventually be propagated to chain X by piggybacking on a user transfer request from S to X.

Next, we calculate $Total$, which is simply the total amount of assets that would need to be redistributed to restore each remote balance to its initial value (line 9). Instead of directly redistributing assets, the Δ algorithm allocates a portion of the current transfer amount t to each remote chain. If t is at least as large as $Total$, then the current transaction is large enough to fully rebalance the remote balances. However, if t is smaller than $Total$, then it is only possible to partially rebalance the remote balances. In this case, we adjust the $\text{diff}_{s,x}$ values to sum to t instead (lines 10–12). If the total rebalance amount is less than the current transaction (i.e., $Total < t$), then line 11 has no effect. We calculate t', which may be 0, to be the portion of t that remains after using $Total$ to rebalance the remote chains (line 13).

The most important steps in the Δ algorithm are lines 14–16. Here, we add credits to each chain in an attempt to restore the remote balance to its initial value. To the existing credits for a particular remote chain, we add two values: (1) the rebalance amount ($\text{diff}_{s,x}$) for that remote chain and (2) a proportional amount of the remaining transfer balance (t'), weighted according to the supplied weight parameter ($w_{s,x}$). In more detail, we first add a credit based on the $\text{diff}_{s,x}$ that we calculated on lines 7 and 11, bringing the sum of the credit and LKB as close as possible to the initial value of the LKB. If there is any transfer balance remaining (i.e., $t' > 0$), then each remote chain is issued a credit based on the size of the remaining transfer balance and the weight corresponding to that chain.

Lines 17 crafts a message to send to the destination chain D. As we are notifying chain D of any outstanding credits ($c_{s,d}$), we update the LKB for chain D ($lkb_{s,d}$) to reflect the credits and then reset the credits for chain D to 0 (lines 18–19). Finally, chain D is notified of the transfer amount and outstanding credits (line 20), and the transaction is committed on chain S.

Lines 21–24 describe the transaction on the destination chain, with the destination assets (a_d) and LKB ($lkb_{s,d}$) updated to reflect the transfer of size t. The balance between chain D and chain S ($b_{s,d}$) is increased to reflect any credits received. Note that it is possible for the received credits to be 0 even if the transaction size is greater than zero. This concludes the Δ algorithm.

3.3 Proof of Instant Guaranteed Finality

We provide a proof that the Δ algorithm achieves instant guaranteed finality. To recall, we define instant guaranteed finality as the guarantee that a transfer which is not rejected by the source chain S is guaranteed to be successfully committed on the destination chain D, with the primary implication being that the liquidity pool on chain D is guaranteed to have enough assets to facilitate the transfer. Importantly, transactions are atomic, meaning lines 1–17 occurs atomically. This prevents the case where two (or more) concurrent transactions of sizes t_1 and t_2 such that $t_1 + t_2 > b_{s,d}$ get accepted.

For a transaction T transferring t tokens from chain S to chain D, we define the committability of transaction T on chain D as:

$$committable(T) \implies a_d > t$$

Instant guaranteed finality is the guarantee that any transaction not rejected by chain S will always be committable on chain D. We define and prove three theorems to help prove instant guaranteed finality for the Δ algorithm:

Theorem 1. *The remote balance $b_{d,s}$ can never exceed the last known balance $lkb_{d,s}$.*

$$b_{d,s} \leq lkb_{d,s}$$

Proof. We use a proof by contradiction. Suppose that $b_{d,s} > lkb_{d,s}$

$b_{d,s}$ and $lkb_{d,s}$ start with the same initial value by the definition of the algorithm.

The balance $b_{d,s}$ is only updated in line 23 of the algorithm, and $b_{d,s}$ can only be updated as a result of a transaction from chain S:

$$b'_{d,s} = b_{d,s} + c_{s,d}$$

$c_{s,d}$ can only be received by chain d if it was sent in line 20 of the algorithm.

Line 20 of the algorithm cannot execute unless line 18 of the algorithm also executed, because the receiver-side transaction cannot start until the sender-side transaction is committed.

$$lkb'_{d,s} = lkb_{d,s} + c_{s,d}$$

If at any point $b_{d,s} > lkb_{d,s}$, then there must be some transaction where $b'_{d,s} > lkb'_{d,s}$

If $b'_{d,s} > lkb'_{d,s}$, this implies one of two possibilities:

(1) There exists a transaction where:

$$b_{d,s} + c_{s,d} > lkb_{d,s} + c_{s,d} \implies c_{s,d} > c_{s,d}$$

However, this is impossible, as $c_{s,d} = c_{s,d}$.

(2) Line 24 is executed before its matching execution of line 5

This is impossible, as the destination chain transaction cannot start until the sender chain transaction is committed.

Therefore, the remote balance $b_{d,s}$ must be strictly less than or equal to the last known balance $lkb_{d,s}$.

□

Theorem 2. *The sum of all LKBs and credits on a given chain can never exceed the local assets on that chain.*

$$\sum_{d \in D} lkb_{s,d} + \sum_{d \in D} c_{s,d} \leq a_s$$

Proof. We define the set D to describe all of the destination chains for a given source chain.

By definition, the initial state is such that:

$$\sum_{d \in D} lkb_{s,d} + \sum_{d \in D} c_{s,d} = a_s$$

$lkb_{d,s}$ is only updated on line 18, and $c_{s,d}$ is only updated on lines 15 and 19. Note that $lkb_{d,s}$ and $c_{s,d}$ are only ever changed by transactions originating on chain S, meaning all mutations of this particular state happen in atomic transactions.

For the purposes of this proof, lines 18 and 19 can be ignored as they do not change the sum of $lkb_{d,s}$ and $c_{s,d}$.

Let a', $lkb'_{s,d}$, and $c'_{s,d}$ represent the assets, LKB, and credits after executing the algorithm once for a particular transaction of size t.

We prove by contradiction that:

$$\sum_{d \in D} lkb'_{s,d} + \sum_{d \in D} c'_{s,d} \leq a'_s$$

Suppose:

$$\sum_{d \in D} lkb'_{s,d} + \sum_{d \in D} c'_{s,d} > a'_s$$

$$\implies$$

$$\sum_{d \in D} lkb_{s,d} + \sum_{d \in D} c_{s,d} + \sum_{d \in D} \text{diff}_{s,d} + \sum_{d \in D} (t' \times w_{s,d}) > a_s + t$$

This can be simplified to:

$$\sum_{d \in D} \text{diff}_{s,d} + \sum_{d \in D} (t' \times w_{s,d}) > t$$

By definition (Table 2), $\sum_{d \in D} w_{s,d} = 1$, so we can further simplify:

$$\sum_{d \in D} \text{diff}_{s,d} + t' > t$$

By the definition of $Total$ in line 9, we can see that:

$$\sum_{d \in D} \frac{\text{diff}_{s,d}}{Total} = \sum_{d \in D} \frac{\text{diff}_{s,d}}{\sum_x^x \text{diff}_{s,x}} = 1$$

By line 11, we observe:

$$\sum_{d \in D} \text{diff}_{s,d} = min(t, Total)$$

Given this, we can simplify further:

$$min(t, Total) + t' > t \implies t' > t - min(t, Total)$$

However, by line 13 of the algorithm:

$$t' = t - min(Total, t)$$

Therefore, the sum of all LKBs and credits on a given chain can never exceed the local assets on that chain. □

Theorem 3. $lkb_{s,d} \geq 0$

Proof. Let $lkb'_{s,d}$ be the value of $lkb_{s,d}$ after the destination chain transaction is committed (lines 21–24.

Initially,

$$lkb_{s,d} = b_{s,d} \geq 0$$

Suppose

$$lkb'_{s,d} < 0$$

By theorem 1,

$$b_{s,d} \leq lkb_{s,d}$$

$$\implies lkb'_{s,d} < 0 \implies b'_{s,d} < 0$$

$$\implies b'_{s,d} < 0 \implies t > b_{s,d}$$

This is invalid, as lines 1–3 do not permit this.

Note that we do not need to consider race conditions in this case because the sender-side part of the transaction (which updates $b_{s,d}$) happens as an atomic transaction. □

Using these three theorems, we prove our claim that the Δ algorithm provides instant guaranteed finality.

Proof. We start by defining a *history* as the complete, temporally ordered sequence of non-rejected transfer requests that occurred up to a given point in time. We claim that no history can ever result in a negative value of a_d for any chain D in the chain network, and prove this by induction.

We start with a history composed of one transaction T_0 of size t_0.

By lines 1–3, if $b_{s,d} < t_0$ then the transaction is rejected, meaning that any non-rejected transaction must fulfill the condition $b_{s,d} > t_0$.

By Theorem 1:

$$b_{s,d} \leq lkb_{s,d} \implies lkb_{s,d} \geq t_0$$

By Theorem 2:

$$a_d \geq lkb_{s,d} \geq t_0 \implies a_d \geq t_0$$

a_d for any chain D in the network is only ever reduced in line 22, meaning a_d can only (potentially) become negative after executing this line.

We observe that this history cannot result in any negative value of a_d because $a_d \geq t \implies a_d - t \geq 0$.

Now, given a history $\mathbf{H}$, we define $\mathbf{H}'$ to be the resulting history when an additional non-rejected transfer T_i from chain S to chain D is appended to the end of $\mathbf{H}$.

Let a'_d, $b_{d,s}$, and $lkb'_{d,s}$ be the values of a_d, $b_{d,s}$ and $lkb_{d,s}$ respectively after executing all transfers in $\mathbf{H}$.

We observe that $b'_{d,s} \geq t_i$, otherwise the transfer T_i would have been rejected.

By Theorem 2, we observe that $b'_{d,s} \leq lkb'_{d,s}$.

$$\implies b'_{d,s} + \sum_{x \in X, x \neq s} lkb'_{d,x} \leq \sum_{x \in X} lkb'_{d,x}$$

By Theorem 1, we observe that $\sum_{x \in X} lkb'_{d,x} \leq a'_d$.

$$b'_{d,s} + \sum_{x \in X, x \neq s} lkb'_{d,x} \leq \sum_{x \in X} lkb'_{d,x} \leq a'_d$$

$$\sum_{x \in X, x \neq s} lkb'_{d,x} \leq a'_d - b'_{d,s}$$

By Theorem 3, $lkb_{d,x}$ can never be negative, therefore:

$$0 \leq \sum_{x \in X, x \neq s} lkb'_{d,x} \leq a'_d - b'_{d,s} \implies b'_{d,s} \leq a'_d$$

$$t_i \leq b'_{d,s} \leq a'_d \implies t_i \leq a'_d$$

Therefore, there are sufficient assets on chain D to serve the transaction T_i.

Note that reordering transactions (in a valid manner) within any given history does not affect the correctness of this proof, as the resulting reordered history is still a valid history of finite size–this implies that there is no possibility of a race condition resulting in insufficient assets. □

3.4 Discussion

As we proved in the previous section, the Δ algorithm is able to achieve instant guaranteed finality, and does so while using a soft-partitioned unified liquidity pool to achieve scalability. On top of this, ΔBridges deal purely in native assets, not minting or issuing any intermediate or wrapped tokens, which results in a single seamless transfer from the native asset on the source chain to the native asset on the destination chain. To our knowledge, no existing cross-chain bridge is capable of providing this functionality, as all existing bridges fundamentally rely on lock-and-mint semantics. Furthermore, ΔBridge transactions are composable across chains: for example, a user can, in one step, initiate a single cross-chain transaction to convert ETH [10] to USDC [7], bridge the USDC to Solana [5] using a ΔBridge, and swap the transferred USDC with SOL. This enables a unified, streamlined experience for multichain applications (e.g. SushiSwap [6] or Abracadabra [1]). Each of these three features—instant guaranteed finality with native assets, capital efficiency through unified liquidity pools, and cross-chain composability—is unique to ΔBridges and not present on any other existing cross-chain bridge.

However, there is one shortcoming of the Δ algorithm as shown above that we must address: It is possible for user transactions to exhaust the available balance for a particular source-destination chain pair. To solve this problem in a cost-effective manner, we introduce *equilibrium fees*, which are transaction fees designed to incentivize users to transfer liquidity in a manner that attempts to keep all balances above their initial value. The key idea is to charge an extra fee for users transferring assets from a source chain S to a destination chain D when the $lkb_{d,s}$ is high, and pay those fees back to users for conducting transfers when $lkb_{d,s}$ is low. This results in an incentive to deposit assets when the available liquidity is low, leading users to replenish the liquidity pool.

While the exact details of how to optimally set and scale such fees is beyond the scope of this paper, we provide an example of a simple fee schedule that encourages equilibrium:

$$fee_{s,d} = (lkb_{d,s} + c_{s,d}) - lp_s \times w_{s,d}$$

This formula is similar to the calculation of $\text{diff}_{s,x}$ in Figure 4, line 7, except negated. Using this formula, one could hypothetically charge a fee proportional to how far the remote balance is from its initial value. In short, the larger an LKB, the higher the fee charged to the users for transfers in that particular direction. Conversely, if an LKB is below the initial value for a particular pair of chains, then there is an arbitrage opportunity to bridge assets in that direction and pay a negative fee (i.e., collect a reward) for doing so.

These equilibrium fees will ultimately result in users naturally working towards balancing each connection in the network, ensuring ΔBridges are able to continually service transfers without requiring the addition of more base liquidity.

4 Examples

In this section, we illustrate the Δ algorithm via a detailed explanation of the example transactions in Figure 5. This example shows the behavior of the Δ algorithm without any equilibrium fees. Each horizontal section (row) in

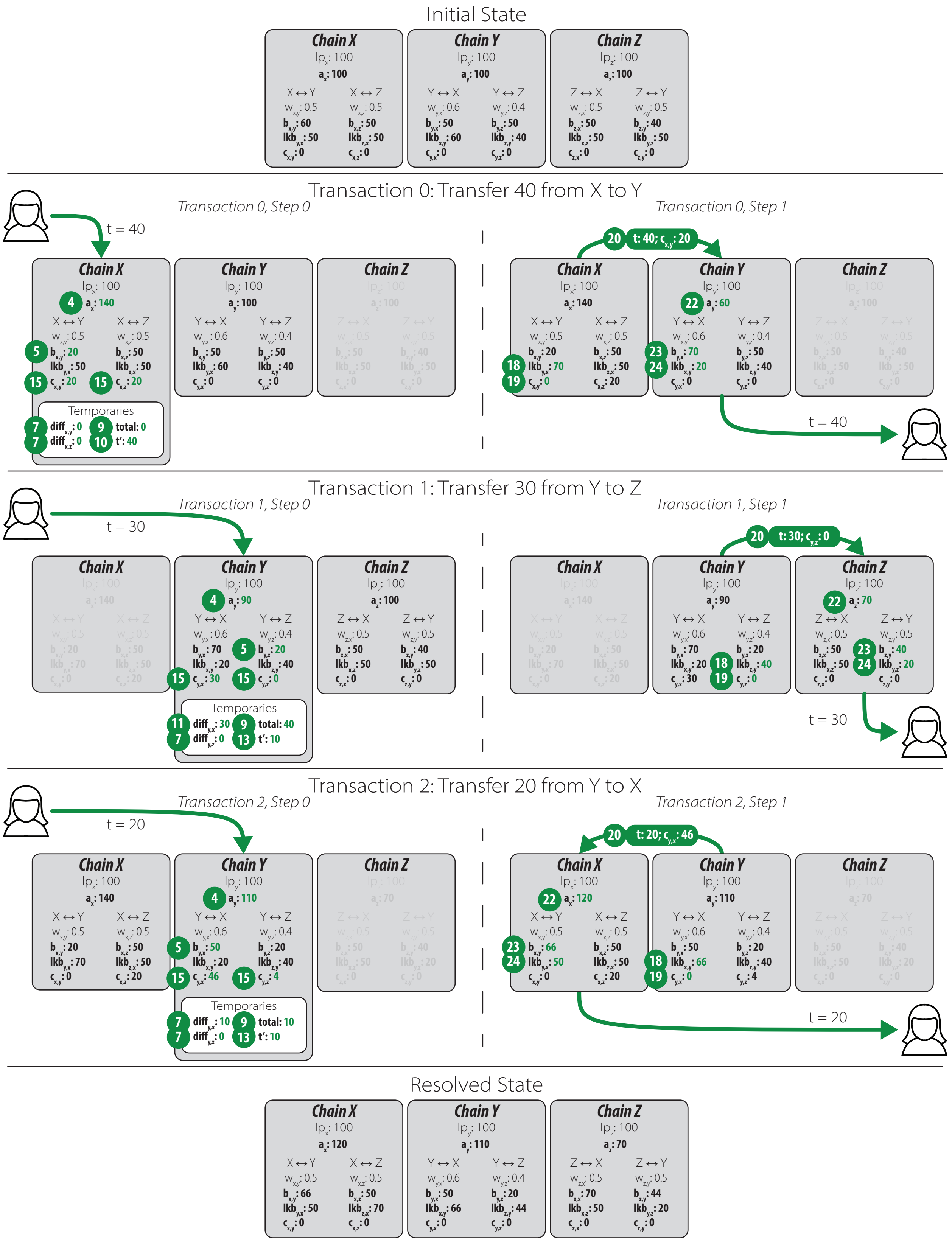

Figure 5: Three example transactions under the Δ algorithm. See Section 4 for further explanation.

the figure represents a transaction and, for simplicity, we split the execution of the protocol into two phases (columns) per transaction. The top row shows the initial state of each chain in the network, and, for illustrative purposes, the final row shows what the state of the network would be if all credits were resolved.

For each transaction, we annotate every change in local state with a number representing the line of the algorithm (Figure 4) that the change corresponds to. Mutable state is bold and intermediate values are organized in a white box labeled "Temporaries". We use three hypothetical blockchains (X, Y, and Z) in our example, and any chains not involved in a particular transaction are grayed out.

Initial state: Each chain has assets equaling their liquidity provided, a total of \$100 per chain. Chains X and Z evenly weight each of their connections with $w = 0.5$, meaning that $b_{y,x}$, $b_{z,x}$, $b_{x,z}$, $b_{y,z}$ are all equal and have the value 50. However, chain Y weights its connection with X with $w_{y,x} = 0.6$ and its connection with Z with $w_{y,z} = 0.4$. As a result, $b_{x,y} = 60$ and $b_{z,y} = 40$, meaning that chain Z can only send up to \$40 to chain Y, whereas chain X can send up to \$60 to chain Y.

Transaction 0, step 0: A user requests to transfer \$40 from chain X to chain Y. The transaction begins with the user depositing \$40 into the liquidity pool on chain X. As the transaction is smaller than the available balance, the transaction is not rejected. Next, a_x is updated to 140 and $b_{x,y}$ is updated to 20 by lines 4 and 5, respectively. diff, $Total$, and t' are calculated in lines 6–12. In the case of transaction 0, all of the LKBs are at their initial value, so the entire transaction is split between $c_{x,y}$ and $c_{x,z}$ proportionally to $w_{x,y}$ and $w_{x,z}$ respectively. The result is that the transaction of size 40 gets split evenly into 20 for chain Y and 20 for chain Z (line 15).

Transaction 0, step 1: The Δ algorithm notifies chain Y of the transaction and any outstanding credits. To that end, we start by updating $lkb_{y,x}$ to 70 on chain X to reflect the credits (line 18) and then resetting the credits to zero (line 19). After that, we send the transaction size t along with the value of the credits before it was reset ($c_{x,y} = 20$) to chain Y. Upon receipt of this message, chain Y updates its assets and balance to reflect the asset transfer (lines 22–23). Chain Y then updates its last known value of the balance $b_{x,y}$ based on the 20 received credits (line 24). Finally, chain Y can grant \$40 to the user and commit the transaction, concluding the asset bridge process.

Transaction 1: The second transaction illustrates how the Δ algorithm works to dynamically rebalance its remote balances during each transaction. In transaction 1, a user transfers \$30 from chain Y to chain Z. The primary difference compared to transaction 0 is that the current value of $lkb_{x,y}$ is 20, reflecting the withdrawal of \$40 that occurred in transaction 0, step 1. As such, in line 7, the delta between the current value of 20 and the initial value of 60 is calculated to be 40. However, this delta is larger than the current transaction, so it is capped to the transaction amount (30) in line 11. This means that the transaction of \$30 will be fully distributed to chain X to move the remote balance $b_{x,y}$ as close as possible to its initial value of 60. Despite the fact that the destination of this transaction is chain Z, there is no credit allocated to chain Z, and the entire transaction amount is used to replenish the bandwidth of the connection from chain Y to chain X.

Transaction 2: The final transaction shows both how credits from previous transactions are piggybacked onto new user requests and how the Δ algorithm handles different weight parameterizations. In transaction 2, a user transfers \$20 from Y to X. The remote balance $lkb_{y,x}$ is still 10 below its initial value of 60, so the value $\text{diff}_{y,x}$ is calculated to be 10. As a result, the transaction is divided into two parts: (1) \$10 is used to restore $lkb_{y,x}$ back to its initial value of 60 and (2) the remaining \$10 is divided between chains X and Z in a manner proportional to their assigned weights, giving additional credits of 6 to chain X and 4 to chain Z. When chain Y notifies chain X of the transaction, it sends all of the accumulated credit (46, in this case). By piggybacking accumulated credit onto user requests, the Δ algorithm is able to conduct transfers and rebalance connections with a small number of transactions. Minimizing the number of transactions is a key part of what makes ΔBridges practical and scalable, as each transaction can be costly to commit, especially on layer 1 chains like Ethereum.

5 Conclusion

In this paper, we presented the Δ algorithm, a novel resource balancing algorithm enabling cross-chain bridges with instant guaranteed finality in native assets with unified liquidity and cross-chain composability. We presented a formalization of the algorithm and proved that it achieves instant guaranteed finality. These features of the Δ algorithm enables ΔBridges to ensure that requests are only accepted when there is a sufficient amount of liquidity in the system, precluding the possibility of any rolled-back transactions. In addition, we presented a fee scheme that allows the Δ algorithm to be automatically maintained by user transactions, ensuring continuous operation without the introduction of additional LP.

ΔBridges represent a new evolution in the cross-chain bridge ecosystem, providing three features not present in any other solution: no existing cross-chain bridge provides cross-chain composability, no existing cross-chain bridge scales to large numbers of chains, and no existing cross-chain bridge can transact using native assets. The Δ algorithm allows ΔBridges to resolve all of these shortcomings, providing unparalleled versatility and convenience.

The flexibility afforded by the Δ algorithm creates the opportunity to improve many existing applications, such as decentralized exchanges, by taking advantage of single-transaction cross-chain swapping of native assets across vast networks of chains. We envision ΔBridges as the infrastructure that enables densely-connected networks of blockchains and ushers in a new class of cross-chain applications that take advantage of fast guaranteed cross-chain native asset transfers.

References

[1] Abracadabra.money. https://abracadabra.money/.

[2] Avalanche bridge faq. https://docs.avax.network/learn/avalanche-bridge-faq.

[3] Blockchain bridges. https://medium.com/1kxnetwork/blockchain-bridges-5db6afac44f8.

[4] Layerzero: Trustless omnichain interoperability protocol. https://layerzero.network/.

[5] Solana. https://solana.com/.

[6] Sushiswap. https://sushi.com/.

[7] Usdc: the world's leading digital dollar stablecoin. https://www.circle.com/en/usdc.

[8] What are blockchain bridges and why are they important for defi. https://blog.makerdao.com/what-are-blockchain-bridges-and-why-are-they-important-for-defi/.

[9] Anyswap dex user guide. https://anyswap-faq.readthedocs.io/en/latest/index.html. Accessed: 2021-5-13.

[10] Ethereum. https://ethereum.org/en/. Accessed: 2021-5-13.