L1 to L2 messaging

In theTransaction Lifecycle section, we covered how users interact with L2 contracts. They submit transactions by putting messages into the chain's inbox or having a full nodeSequencer or aggregator do so on their behalf.

L1 and L2 chains run asynchronously from each other, so it is not possible to make a cross-chain call that produces a result within the sameTransaction as the caller. Instead, cross-chain calls must be asynchronous, meaning that the caller submits the call at some point in time, and the call runs later. Consequently, a cross-chain contract-to-contract call can never produce a result available to the calling contract (except for acknowledgment of a successful call submitted for later execution).

In this section, we will discuss how contracts interact between L1 and L2, how an L1 contract is called an L2 contract, and vice versa.

Retryable Tickets

Retryable tickets areArbitrum 's canonical method for creating L1 to L2 messages, i.e., L1 transactions that initiate a message to be executed on L2. A retryable can be submitted for a fixed cost (dependent only on its calldata size) paid at L1; itssubmission on L1 is separable / asynchronous with itsexecution on L2. Retryables provide atomicity between the cross chain operations; if the L1 transaction to request submission succeeds (i.e. does not revert) then the execution of the Retryable on L2 has a strong guarantee to ultimately succeed as well.

Retryable Tickets Lifecycle

Here we walk through the different stages of the lifecycle of aRetryable Ticket ; (1) submission, (2) auto-redemption, and (3) manual redemption.

Submission

1. Creating a retryable ticket is initiated with a call (direct or internal) to thecreateRetryableTicket
2. function of theinbox contract
3. . A ticket is guaranteed to be created if this call succeeds. Here, we describe parameters that need to be carefully set. Note that, this function forces the sender to provide areasonable
4. amount of funds (at least enough to submitting, andattempting
5. to executing the ticket), but that doesn't guarantee a successful auto-redemption.

Parameter Description l1CallValue (also referred to as deposit) Not a real function parameter, it is rather the callValue that is sent along with the transaction address to The destination L2 address uint256 l2CallValue The callvalue for retryable L2 message that is supplied within the deposit (l1CallValue) uint256 maxSubmissionCost The maximum amount of ETH to be paid for submitting the ticket. This amount is (1) supplied within the deposit (l1CallValue) to be later deducted from sender's L2 balance and is (2) directly proportional to the size of the retryable's data and L1 basefee address excessFeeRefundAddress The unused gas cost and submssion cost will deposit to this address, formula is:(gasLimit x maxFeePerGas - execution cost) + (maxSubmission - (autoredeem ? 0 : submission cost)) . (Note: excess deposit will transfer to the alias address of theParent chain tx'smsg.sender rather than this address) address callValueRefundAddress The L2 address to which the l2CallValue is credited if the ticket times out or gets cancelled (this is also called thebeneficiary , who's got a critical permission to cancel the ticket) uint256 gasLimit Maximum amount of gas used to cover L2 execution of the ticket uint256 maxFeePerGas The gas price bid for L2 execution of the ticket that is supplied within the deposit (l1CallValue) bytes calldata data The calldata to the destination L2 address 1. Sender's deposit must be enough to make the L1 submission succeed and for the L2 execution to beattempted 2. . If provided correctly, a new ticket with a uniqueTicketID 3. is created and added to retryable buffer. Also, funds (submissionCost 4. +l2CallValue 5.) are deducted from the sender and placed into the escrow for later use in redeeming the ticket. 6. Ticket creation causes theArbRetryableTx 7. precompile to emit aTicketCreated 8. event containing theTicketID 9. on L2.

Automatic Redemption

1. It is very important to note that the submission of a ticket on L1 is separable / asynchronous from its execution on L2, i.e., a successful L1 ticket creation does not guarantee a successful redemption. Once the ticket is successfully created, the two following conditions are checked: (1) if the user's L2 balance is greater than (or equal to)maxFeePerGas * gasLimit
2. and
3. (2) if themaxFeePerGas
4. (provided by the user in the ticket submission process) is greater than (or equal to) thel2Basefee
5. . If these conditions are both met, ticket's submission is followed by an attempt to execute it on L2 (i.e., anauto-redeem
6. using the supplied gas, as if theredeem
7. method of theArbRetryableTx

8. precompile had been called). Depending on how much gas the sender has provided in step 1, ticket's redemption can

either (1) immediately succeed or (2) fail. We explain both situations here:

9. If the ticket is successfully auto-redeemed, it will execute with the sender, destination, callvalue, and calldata of the original submission. The submission fee is refunded to the user on L2 (excessFeeRefundAddress
10.). Note that to ensure successful auto-redeem of the ticket, one could use the Arbitrum SDK which provides a convenience function
11. that returns the desired gas parameters when sending L1-L2 messages.
12. If a redeem is not done at submission or the submission's initial redeem fails (for example, because the L2 gas price has increased unexpectedly), the submission fee is collected on L2 to cover the resources required to temporarily keep the ticket in memory for a fixed period (one week), and only in this case, a manual redemption of the ticket is required (see next section).

Manual Redemption

1. At this point,anyone
2. can attempt to manually redeem the ticket again by callingArbRetryableTx
3. 'sredeem
4. precompile method, which donates the call's gas to the next attempt. Note that the amount of gas is NOT limited by the original gasLimit set during the ticket creation.ArbOS
5. willenqueue the redeem
6. , which is its own specialArbitrumRetryTx
7. type, to its list of redeems that ArbOSguarantees to exhaust
8. before moving on to the next non-redeem transaction in the block its forming. In this manner redeems are scheduled to happen as soon as possible, and will always be in the same block as the tx that scheduled it. Note that the redeem attempt's gas comes from the call to redeem, so there's no chance the block's gas limit is reached before execution.
9. If the fixed period (one week) elapses without a successful redeem, the ticketexpires
10. and will beautomaticallydiscarded
11. , unless some party has paid a fee tokeep the ticket alive
12. for another full period. A ticket can live indefinitely as long as it is renewed each time before it expires.

Avoid Losing Funds! If a ticket expires after 7 days without being redeemed or re-scheduled to a future date, any message and value (other than the escrowedcallvalue) it carries could be lost without possibility of being recovered. On success, theTo address receives the escrowed callvalue, and any unused gas is returned to ArbOS's gas pools. On failure, the callvalue is returned to the escrow for the future redeem attempt. In either case, the network fee was paid during the scheduling tx, so no fees are charged and no refunds are made.

Note that during redemption of a ticket, attempts to cancel the same ticket, or to schedule another redeem of the same ticket, will revert. In this manner retryable tickets are not self-modifying.

If a ticket with a callvalue is eventually discarded (cancelled or expired), having never successfully run, the escrowed callvalue will be paid out to acallValueRefundAddress account that was specified in the initial submission (step 1).

Important Notes: If a redeem is not done at submission or the submission's initial redeem fails, anyone can attempt to redeem the retryable again by callingArbRetryableTx 'sredeem precompile method, which donates the call's gas to the next attempt. ArbOS willenqueue the redeem , which is its own specialArbitrumRetryTx type, to its list of redeems that ArbOSguarantees to exhaust before moving on to the next non-redeem transaction in the block its forming. In this manner redeems are scheduled to happen as soon as possible, and will always be in the same block as the transaction that scheduled it. Note that the redeem attempt's gas comes from the call toredeem , so there's no chance the block's gas limit is reached before execution.

- One can redeem live tickets using theArbitrum Retryables Transaction Panel
- The calldata of a ticket is saved on L2 until it is redeemed or expired
- Redeeming cost of a ticket will not increase over time, it only depends on the current gas price and gas required for execution

Receipts

In the lifecycle of a retryable ticket, two types of L2 transaction receipts will be emitted:

- Ticket Creation Receipt
- : This receipt indicates that a ticket was successfully created; any successful L1 call to theInbox
- 'screateRetryableTicket
- method is guaranteed to create a ticket. The ticket creation receipt includes aTicketCreated
- event (fromArbRetryableTx
-), which includes aticketId
- field. ThisticketId
- is computable via RLP encoding and hashing the transaction; seecalculateSubmitRetryableId
- .
- Redeem Attempt

- : A redeem attempt receipt represents the result of an attempted L2 execution of a ticket, i.e, success / failure of that specific redeem attempt. It includes aRedeemScheduled
- event fromArbRetryableTx
- , with aticketId
- field. At most, one successful redeem attempt can ever exist for a given ticket; if, e.g., the auto-redeem upon initial creation succeeds, only the receipt from the auto-redeem will ever get emitted for that ticket. If the auto-redeem fails (or was never attempted — i.e., the provided L2 gas limit * L2 gas price = 0), each initial attempt will emit a redeem attempt receipt until one succeeds.

Alternative "unsafe" Retryable Ticket Creation

TheInbox.createRetryableTicket convenience method includes sanity checks to help minimize the risk of user error: the method will ensure that enough funds are provided directly from L1 to cover the current cost of ticket creation. It also will convert the providedcallValueRefundAddress andexcessFeeRefundAddress to theirAddress Alias (see below) if either is a contract (determined by if the address has code during the call), providing a path for the L1 contract to recover funds. A power-user may bypass these sanity-check measures via theInbox 'sunsafeCreateRetryableTicket method; as the method's name desperately attempts to warn you, it should only be accessed by a user who truly knows what they're doing.

Eth deposits

A special message type exists for simple Eth deposits; i.e., sending Eth from L1 to L2. Eth can be deposited via a call to theInbox 'sdepositEth method. If the L1 caller is EOA, the Eth will be deposited to the same EOA address on L2; the L1 caller is a contract, the funds will deposited to the contract's aliased address (see below).

Note that depositing Eth viadepositEth into a contract on L2 willnot trigger the contract's fallback function.

In principle, retryable tickets can alternatively be used to deposit Ether; this could be preferable to the special eth-deposit message type if, e.g., more flexibility for the destination address is needed, or if one wants to trigger the fallback function on the L2 side.

Transacting via the Delayed Inbox

While retryables and Eth depositsmust be submitted through theDelayed Inbox , in principle,any message can be included this way; this is a necessary recourse to ensure theArbitrum chain preserves censorship resistance even if the Sequencer misbehaves (seeThe Sequencer and Censorship Resistance). However, under ordinary/happy circumstances, the expectation/recommendation is that clients use the delayed inbox only for Retryables and Eth deposits, and transact via the Sequencer for all other messages.

Address Aliasing

Unsigned messages submitted via the Delayed Inbox get their sender's addressed "aliased": when these messages are executed on L2, the sender's address —i.e., that which is returned bymsg.sender — willnot simply be the L1 address that sent the message; rather it will be the address's "L2 Alias." An address's L2 alias is its value increased by the hex value0x1111000000000000000000000000000000001111 :

L2_Alias

L1_Contract_Address +

0x1111000000000000000000000000000000001111 Try it out The Arbitrum protocol's usage of L2 Aliases for L1-to-L2 messages prevents cross-chain exploits that would otherwise be possible if we simply reused the same L1 addresses as the L2 sender; i.e., tricking an L2 contract that expects a call from a given contract address by sending retryable ticket from the expected contract address on L1.

If for some reason you need to compute the L1 address from an L2 alias on chain, you can use ourAddressAliasHelper library:

modifier

onlyFromMyL1Contract ( ) override { require (AddressAliasHelper . undoL1ToL2Alias (msg . sender)

== myL1ContractAddress ,

"ONLY_COUNTERPART_CONTRACT") ; _ ; }

Signed Messages

The delayed inbox can also accept messages that include a signature. In this case, the message will execute with themsg.sender address equal to the address that produced the included signature (i.e.,not its alias). Intuitively, the signature proves that the sender address is not a contract, and thus is safe from cross-chain exploit concerns described above. Thus, it can safely execute from signer's address, similar to a transaction included in a Sequencer'sBatch . For these messages, the address of the L1 sender is effectively ignored at L2.

These signed messages submitted through the delayed inbox can be used to execute messages that bypass the Sequencer and require EOA authorization at L2, e.g., force-including an Ether withdrawal (see"withdraw eth tutorial"). Edit this page
Last updatedonJan 27, 2025 Previous Gas and fees Next L2 to L1 messaging