Deposit Flow

This guide explains the deposit flow process for L2 deposit transactions, triggered by transactions or events on L1. In Optimism terminology, "deposit transaction " refers to any L2 transaction that is triggered by a transaction or event on L1.

The process is somewhat similar to the way most networking stacks work(opens in a new tab). Information is encapsulated in lower layer packets on the sending side and then retrieved and used by those layers on the receiving side while going up the stack to the receiving application.

L1 Processing

1. An L1 entity, either a smart contract or an externally owned account (EOA), sends a deposit transaction to L1CrossDomainMessenger (opens in a new tab)
2. , using sendMessage (opens in a new tab)
3. .
4. This function accepts three parameters:
5.
    - _target
6.
    - , target address on L2.
7.
    - _message
8.
    - , the L2 transaction's calldata, formatted as per the ABI(opens in a new tab)
9.
    - of the target account.
10.
    - _minGasLimit
11.
    - , the minimum gas limit allowed for the transaction on L2. Note that this is aminimum
12.
    - and the actual amount provided on L2 may be higher (but never lower) than the specified gas limit.
13.
    - Note that the actual amount provided on L2 will be higher, because the portal contract on L2 needs to do some processing before submitting the call to_target
14.
    - .
15. The L1 cross domain messenger calls its own _send function(opens in a new tab)
16. .
17. It uses these parameters:
18.
    - _to
19.
    - , the destination address, is the messenger on the other side.
20.
    - In the case of deposits, this is always 0x4200000000000000000000000000000000000007 (opens in a new tab)
21.
    - .
22.
    - _gasLimit
23.
    - , the gas limit.
24.
    - This value is calculated using thebaseGas function(opens in a new tab)
25.
    - .
26.
    - _value
27.
    - , the ETH that is sent with the message.
28.
    - This amount is taken from the transaction value.
29.
    - _data
30.
    - , the calldata for the call on L2 that is needed to relay the message.
31.
    - This is an ABI encoded(opens in a new tab)
32.
    - call to relayMessage (opens in a new tab)
33.
    - .
34. _sendMessage (opens in a new tab)
35. calls the portal's depositTransaction function(opens in a new tab)
36. .
37. Note that other contracts can also call depositTransaction (opens in a new tab)
38. directly.
39. However, doing so bypasses certain safeguards, so in most cases it's a bad idea.
40. ThedepositTransaction function(opens in a new tab)
41. runs a few sanity checks, and then emits a TransactionDeposited (opens in a new tab)
42. event.

L2 Processing

1. Theop-node
2. component looks for TransactionDeposited events on L1(opens in a new tab)
3. .
4. If it sees any such events, it parses(opens in a new tab)
5. them.
6. Next,op-node
7. converts(opens in a new tab)
8. thoseTransactionDeposited
9. events into deposit transactions(opens in a new tab)
10. .
11. In most cases user deposit transactions call the relayMessage (opens in a new tab)
12. function of L2CrossDomainMessenger (opens in a new tab)
13. .
14. relayMessage
15. runs a few sanity checks and then, if everything is good, calls the real target contract with the relayed calldata(opens in a new tab)
16. .

Denial of service (DoS) prevention

As with all other L1 transactions, the L1 costs of a deposit are borne by the transaction's originator. However, the L2 processing of the transaction is performed by the Optimism nodes. If there were no cost attached, an attacker could be able to submit a transaction that had high costs of run on L2, and that way perform a denial of service attack.

To avoid this DoS vector, depositTransaction (opens in a new tab) , and the functions that call it, require a gas limit parameter. This gas limit is encoded into the []TransactionDeposited event(opens in a new tab) , and used as the gas limit for the user deposit transaction on L2.

This L2 gas is paid for by burning L1 gas here(opens in a new tab) .

Replaying messages

Deposits transactions can fail due to several reasons:

- Not enough gas provided.
- The state on L2 does not allow the transaction to be successful.

It is possible to replay a failed deposit, possibly with more gas,

Replays in action

To see how replays work, you can usethis contract on OP Sepolia(opens in a new tab).

1. CallstopChanges
2. , using this Foundry command:
3. PRIV_KEY
4. =<
5. your
6. private
7. key
8. her
9. e
10.

11. export
12. ETH_RPC_URL
13. =
14. https://sepolia.optimism.io
15. GREETER
16. =
17. 0xEF60cF6C6D0C1c755be104843bb72CDa3D778630
18. cast
19. send
20. --private-key
21. PRIV_KEY GREETER
22. "stopChanges()"
23. Verify thatgetStatus()
24. returns false, meaning changes are not allowed, and see the value ofgreet()
25. using Foundry.
26. Note that Foundry returns false as zero.
27. cast
28. call
29. GREETER
30. "greet()"
31. |
32. cast
33. --to-ascii
34. ;
35. cast
36. call
37. GREETER
38. "getStatus()"
39. Get the calldata.
40. You can use this Foundry command:
41. cast
42. calldata
43. "setGreeting(string)"
44. "testing"
45. Or just use this value:
46. 0xa4136862000000000000000000000000000000000000000000000000000000000000002000000000000000000000000000000000000000000000000000000000000000774657374696e670000
47. Send a greeting change as a deposit.
48. Use these commands:
49. L1_RPC
50. =
51. https://sepolia.optimism.io
52. L1XDM_ADDRESS
53. =
54. 0x5086d1eef304eb5284a0f6720f79403b4e9be294
55. FUNC
56. =
57. "sendMessage(address,bytes,uint32)"
58. CALLDATA
59. =
60. `
61. cast
62. calldata
63. "setGreeting(string)" "testing"`
64. cast
65. send
66. --rpc-url
67. L1_RPC
68. --private-key
69. PRIV_KEY L1XDM_ADDRESS FUNC GREETER CALLDATA
70. 10000000
71. The transaction will be successful on L1, but then emit a fail event on L2.
72. The next step is to find the hash of the failed relay.
73. The easiest way to do this is to look inthe internal transactions of the destination contract(opens in a new tab)
74. , and select the latest one that appears as a failure.
75. It should be a call to L2CrossDomainMessenger at address0x420...007
76. . This is the call you need to replay.
77. If the latest internal transaction is a success, it probably means your transaction hasn't relayed yet. Wait until it is, that may take a few minutes.
78. Get the transaction information using Foundry.
79. TX_HASH
80. =<
81. transaction
82. hash
83. from
84. Ethersca
85. n
86.

87. L2XDM_ADDRESS
88. =
89. 0x4200000000000000000000000000000000000007
90. REPLAY_DATA
91. =
92. `
93. cast
94. tx
95. TX_HASH

input
`
CallstartChanges()
to allow changes using this Foundry command:
cast
send
--private-key
PRIV_KEY GREETER
"startChanges()"
⚠
Don't do this prematurely
If you callstartChanges()
too early, it will happen from the message is relayed to L2, and then the initial deposit will be successful and there will be no need to replay it.
Verify thatgetStatus()
returns true, meaning changes are not allowed, and see the value ofgreet()
.
Foundry returns true as one.
cast
call
GREETER
"greet()"
|
cast
--to-ascii
;
cast
call
GREETER
"getStatus()"
Now send the replay transaction.
cast
send
--private-key
PRIV_KEY
--gas-limit
10000000
L2XDM_ADDRESS REPLAY_DATA
Why do we need to specify the gas limit?
The gas estimation mechanism tries to find the minimum gas limit at which the transaction would be successful.
However,L2CrossDomainMessenger
does not revert when a replay fails due to low gas limit, it just emits a failure message.
The gas estimation mechanism considers that a success.
To get a gas estimate, you can use this command:
cast
estimate
--from
0x0000000000000000000000000000000000000001
L2XDM_ADDRESS REPLAY_DATA
That address is a special case in which the contract does revert.
Verify the greeting has changed:
cast
call
GREETER
"greet()"
|
cast
--to-ascii
;
cast
call
GREETER
"getStatus()"

Debugging

To debug deposit transactions you can ask the L2 cross domain messenger for the state of the transaction.

1. Look on Etherscan to see theFailedRelayedMessage
2. event.
3. SetMSG_HASH
4. to that value.
5. To check if the message is listed as failed, run this:
6. cast
7. call
8. L2XDM_ADDRESS
9. "failedMessages(bytes32)"
10. MSG_HASH
11. To check if it is listed as successful, run this:
12. cast
13. call
14. L2XDM_ADDRESS
15. "successfulMessages(bytes32)"
16. MSG_HASH

Smart Contract Overview Transaction Flow