The following has been released early to allow for open and collaborative research. Please tell me if you find issues with this or if you have ideas on how to improve the proposal.

1. Introduction

Application developers need to be able to create programs on the Ethereum 2 platform that have contracts on different shards in different execution environments. The application developers need to be able to use the synchronous atomic composable programming techniques that they are accustomed to using with the Ethereum 1 platform. This post proposes a technique that could allow for this.

In the diagram below, for example, a Cross Shard call is used to get the value of an oracle. If the value returned is below a certain amount, then a Cross Shard call is used to buy a commodity.

[

Example Cross Shard Function Call

2654×1224 319 KB

](https://ethresear.ch/uploads/default/original/2X/d/d9c62395c2ad2a714a3d491da52565232f1aeacd.png)

The technology described herein leverages many of the ideas from Atomic Crosschain Transactions (seeAtomic Crosschain Transactions White Paper).

This Cross Shard Transaction technology relies on the following functionality being added to Execution Environments (EE):

- System Event messages that are added to transaction receipts, that can be referenced from other EEs on other shards via Beacon Chain Crosslinks. In an EE, when the function call that is the target of a transaction ends, the EE produces a System Event message. System Event messages are different from application event messages that could be produced by contract code. Contract code can not produce events that forge System Event messages.
- Live parameter checking: when contract code calls makes a cross shard function call, check that the actual shard, EE, contract, function and parameters match those that are expected to be called.
- Contract lockability: When contracts are deployed they need to be specified as able to be locked (lockable) or not able to be locked (nonlockable). When a Transaction Segment executes, any contract that has a state update needs to be locked. If a contract was deployed as nonlockable, then it can not be locked and the transaction fails.
- Provisional state storage and contract locking: when a contract is updated as part of a Cross Shard transaction, its updated state is stored in provisional storage and the contract is locked. If the Cross Shard transaction is committed, the provisional state replaces the contract state and the contract is unlocked. If the Cross Shard transaction is ignored, the provisional state is discarded and the contract is unlocked.
- New Transaction Types: The EE needs to support the transaction types described later in this proposal.

2. Example

The best way of understanding this technology is to work through an example. Imagine the call graph shown below. The Segment Transactions with (no updates) written below them are function calls that read state and return a value. The Segment Transactions with (updates) written below them are function calls that write state and return a value.

[

Example Call Graph

2236×1098 241 KB

](https://ethresear.ch/uploads/default/original/2X/2/28052026320a82b07f3d080d92e50f6b6a09f04f.png)

The transactions to execute this complex Cross Shard Transaction are shown in the diagram below. In the diagram, blocks are represented by square boxes and transactions by rounded cornered boxes. The signed state root for Shard blocks for block number N-1 feed into Beacon Chain blocks for block number N and are known as Cross Links. The Cross Links for all Shard blocks in a Beacon Chain block for block number N are available for use by Shard blocks for block number N. The dashed lines indicate the submission of Cross Links into Beach Chain blocks from Shard blocks and their availability for use in Shard blocks from Beacon Chain blocks. The solid lines indicate transactions submitted by the application into transaction pools and then included into Shard blocks.

[

Example Transations and Blocks

2690×1446 539 KB

](https://ethresear.ch/uploads/default/original/2X/4/419657f530ff60b7451983c1b27b7a24d6e55f71.jpeg)

Walking through the processing:

- The application submits the Start transaction to Shard 1. This reserves the Cross Shard Transaction Id for the Cross Shard Transaction and specifies the call graph of the Root Transaction and Transaction Segments that will make up the Cross Shard Transaction.
- The application submits leaf Transaction Segments to Shard 3, 5, 6, and 7. The transactions execute function calls that could update state and could return a value. The EEs that execute the transactions emit System Events that include information about the transaction call and function call, including the error status and return result. For the purposes of the example, assume that only Shard 7 has any state updates.
- The application waits for a Beacon Chain block to be produced that includes a Cross Link for the Shard block that includes the submitted transactions.
- The application submits a Transaction Segment to Shard 2 to execute a function call. The transaction includes the System Event information from the transactions that executed on Shard 3 and 5 and Merkle Proofs showing that information relates to transactions that are parts of blocks that executed on the shards. The Merkle Root can be compared with the Cross Link in the Beacon Block for the shard. For the purposes of this example, assume that the transaction on Shard 2 causes no state updates.
- In parallel with submitting the Transaction Segment to Shard 2, the application submits a Transaction Segment to Shard 4 to execute a function call. The transaction includes the System Event information and Merkle Proofs for the transactions that executed on shard 6 and shard 7. For the purposes of this example, assume that this transaction causes state updates on Shard 4.
- The application waits for a Beacon Chain block to be produced that includes a Cross Link for the Shard block that includes the submitted transactions.
- The application submits the Root Transaction, along with System Event information and Merkle Proofs for the Transaction Segments on Shard 2 and 4. System Event information is emitted that indicates the entire Cross Shard Transaction should be committed.
- The application waits for a Beacon Chain block to be produced that includes a Cross Link for the Shard block that includes the submitted transactions.
- The application submits Signalling transactions on Shard 4 and 7 to unlock the contracts on Shard 4 and 7.
- The application submits a Clean transaction on Shard 1 to remove the Cross Shard Transaction Id from the list of outstanding unique ids.

3. Transaction Types

Cross Shard Transactions consist of multiple transaction types.

3.1 Start Transaction

Start transactions indicate the start of a Cross Shard Transaction. This transaction is used to reserve a per-shard and per-EE “unique” Cross Shard Transaction Id, and to register the overall call graph of Root Transaction and Transaction Segments.

Transaction fields include:

- Cross Shard Transaction Id
- Time-out block number.
- Root Transaction information (shard, EE, function call & parameters).
- For each Transaction Segment called from the code executed as a result of this transaction:
- Log message with Shard, EE, Contract Address, Function Name, Parameter Values of function called
- Return result

- Whether there was a state update and the transaction has resulted in a locked contract.
- Merkle Proof to a specified block to a crosslink.
- Note that the list of transactions must be in the order that the functions are expected to be called.
- Log message with Shard, EE, Contract Address, Function Name, Parameter Values of function called
- Return result
- Whether there was a state update and the transaction has resulted in a locked contract.
- Merkle Proof to a specified block to a crosslink.
- Note that the list of transactions must be in the order that the functions are expected to be called.

Transaction processing:

1. Fail if the Cross Shard Transaction Id is in use.
2. Check that the time-out block number is a suitable value.
3. Register the Cross Shard Transaction Id
4. Emit a Start System Event containing:
5. Tx Origin / Msg Sender: The account that signed the transaction.
6. Cross Shard Transaction Id
7. Hierarchical call graph of cross shard function calls / Transaction Segments starting at the Root Transaction. For each call include:
8. Shard
9. EE
10. Contract address
11. Data: Function name and Parameters
12. Shard
13. EE
14. Contract address
15. Data: Function name and Parameters
16. Tx Origin / Msg Sender: The account that signed the transaction.
17. Cross Shard Transaction Id
18. Hierarchical call graph of cross shard function calls / Transaction Segments starting at the Root Transaction. For each call include:
19. Shard
20. EE
21. Contract address
22. Data: Function name and Parameters
23. Shard
24. EE
25. Contract address
26. Data: Function name and Parameters

3.2 Root Transaction

The Root Transaction contains the function call that is the entry point for the overall call graph of the Cross Shard transaction. This transaction type indicates that the code should execute as per a normal Transaction Segment (see below), and all locked contracts on all shards should be committed or ignored. If the transaction completes successfully prior to the time-out block, all state updates as a result of all Transaction Segments that are part of the Cross Shard Transaction should be committed. If any of the Transaction Segments have failed, or if the transaction is submitted after the time-out, then all Transaction Segment updates should be discarded.

Transaction fields include:

- Cross Shard Transaction Id
- Shard, EE, Contract, Function and Parameters to be called.
- Start Transaction Log message & Merkle Proof (note: includes time-out block number)
- For each Transaction Segment called from the code executed as a result of this transaction:
- Log message with Shard, EE, Contract Address, Function Name, Parameter Values of function called
- Return result
- Whether there was a state update and the transaction has resulted in a locked contract.
- Merkle Proof to a specified block to a crosslink.
- Note that the list of transactions must be in the order that the functions are expected to be called.
- Log message with Shard, EE, Contract Address, Function Name, Parameter Values of function called
- Return result
- Whether there was a state update and the transaction has resulted in a locked contract.
- Merkle Proof to a specified block to a crosslink.
- Note that the list of transactions must be in the order that the functions are expected to be called.

Transaction processing:

1. Verify the log information for the Start Transaction and each Transaction Segment, checking the Merkle proof up to the Shard Crosslink.
2. If the block number is after the timeout, or if any of the Transaction Segment logs indicate an error, emit a System Event indicating the entire Cross Shard Transaction should be aborted.
3. Check that the transaction has been submitted by the same entity as is indicated by the Start Transaction Log.
4. Execute the code, using cached return values for the Transaction Segments.
5. Note: only lockable contracts can have state changes.
6. Note: only lockable contracts can have state changes.
7. Check that the Transaction Segments called by the code matches what was expected to be called in the Start Transaction Log.
8. The state updates are committed.
9. When the entry point function call for this shard completes, emit a System Event Message. This includes:
10. Tx Origin / Msg Sender: The account that signed the transaction.
11. Cross Shard Transaction Id.
12. Contract address.
13. Data: Function name and parameter values specified in the transaction.
14. Indication if the transaction was successful or if an error was thrown.
15. If an error is indicated: Some error code.
16. If success:

17. List of addresses of locked contracts.
18. The return value of the function call.
19. List of addresses of locked contracts.
20. The return value of the function call.
21. Note: the shard id and EE id do not need to be included as they will be implied by the Merkle Tree required to prove the log message is valid.
22. Tx Origin / Msg Sender: The account that signed the transaction.
23. Cross Shard Transaction Id.
24. Contract address.
25. Data: Function name and parameter values specified in the transaction.
26. Indication if the transaction was successful or if an error was thrown.
27. If an error is indicated: Some error code.
28. If success:
29. List of addresses of locked contracts.
30. The return value of the function call.
31. List of addresses of locked contracts.
32. The return value of the function call.
33. Note: the shard id and EE id do not need to be included as they will be implied by the Merkle Tree required to prove the log message is valid.

3.3 Transaction Segments

These transactions execute on shards to run function calls. They may update state and / or return function values. The code may call one or more functions that may return values from a different shard and may update state on a different shard. These are nested Transaction Segments.

Transaction fields include:

- Cross Shard Transaction Id
- Shard, EE, Contract, Function and Parameters to be called.
- Start Transaction Log message & Merkle Proof connecting the log message to the cross link (note: includes time-out block number).
- Depth and offset within the call graph specified in the Start Transaction Event message.
- For each Transaction Segment called from the code executed as a result of this transaction:
- Log message with Shard, EE, Contract Address, Function Name, Parameter Values of function called
- Return result
- Whether there was a state update and the transaction has resulted in a locked contract.
- Merkle Proof to a specified block to a crosslink.
- Note that the list of transactions must be in the order that the functions are expected to be called.
- Log message with Shard, EE, Contract Address, Function Name, Parameter Values of function called
- Return result
- Whether there was a state update and the transaction has resulted in a locked contract.
- Merkle Proof to a specified block to a crosslink.

- Note that the list of transactions must be in the order that the functions are expected to be called.

Transaction processing:

1. Verify the log information for the Start Transaction and each Transaction Segment, checking the Merkle proof up to the Shard Crosslink.
2. If the block number is after the timeout, or if any of the Transaction Segment logs indicate an error, emit a System Event indicating an error.
3. Check that the transaction has been submitted by the same entity as is indicated by the Start Transaction Log.
4. Verify that the call graph from the Start Transaction Event matches the Transaction Segment calls that were passed in.
5. Execute the code, using cached return values for the Transaction Segments.
6. Note: only lockable contracts can have state changes.
7. Note: only lockable contracts can have state changes.
8. Check that the Transaction Segments called by the code matches what was expected to be called in the Start Transaction Log.
9. The state updates are put into provisional storage and the contracts are locked.
10. When the entry point function call for this shard completes, emit a System Event Message that is the same as the one described for the Root Transaction.

3.4 Signalling Transactions

These transactions are used to unlock a contract locked by a Transaction Segments.

This transaction includes:

- Root transaction log information and Merkle Proofs showing the Root Transaction has completed successfully or has failed / timed-out.
- Transaction Segment for this shard log and Merkle Proof, showing which contracts were locked.

Transaction processing:

1. If the Root transaction log indicates “Commit”, apply the provisional updates and unlock the contracts
2. If the Root transaction log indicates “Ignore”, discard provisional updates and unlock the contracts.

These transactions should be either free, or perhaps even an incentive should be paid to ensure participants submit this transaction when there was a failure / if the entity that locked the contract stops participating. Invalid Signalling transactions should penalise users. The penalty should not be onerous, as invalid Signalling transactions may occur as a result of Beacon Chain reorganisations.

3.5 Clean Transaction

Removes the unique id from the list of outstanding unique ids.

The transaction fields include:

- Start transaction log and proof (indicating the call graph of Cross Shard Transactions),
- Root Transaction log and proof
- Transaction Segments logs and Merkle Proofs (indicating which if any contracts were locked),
- Signalling Transactions logs and Merkle Proofs (indicating that the appropriate Signalling Transactions were called).

Valid Clean transactions should be incentivized to ensure participants submitted this transaction. However, invalid Clean transactions should penalise users. The penalty should not be onerous, as invalid Clean transactions may occur as a result of Beacon Chain reorganisations.

3.6 Other Transactions

In addition to the transaction types described above, other transactions that the EE needs to support are:

- Deploy a lockable contract.
- Deploy a nonlockable contract.
- For certain scenarios it might be possible to create specialised transaction types that remove the need for the Start and Clean transactions. For instance, if there is a Root Transaction and a single Transaction Segment that executes a cross shard read and does not update state, it might be possible to create a specialised Root Transaction that does not require the Start and Clean transactions. This is currently being considered further.

4. Cross Shard Processing

To do a Cross Shard Transaction, the application would do the following. Note that, the user's application does not do much, and most of the complexity would be handled by a library wrapper, like Web3J.

1. Determine call graph and parameter values for transactions using dynamic program analysis (code simulation).
2. Submit the Start Transaction. This could be done in parallel with the next step, or this could be done in one slot prior to the next step.
3. Submit leaf Transaction Segments. A "leaf" transaction is a transaction who's function calls do not call other cross-shard functions.
4. Wait a block for cross links to be published.
5. Submit Transaction Segments that contain functions that call other Transaction Segments. Repeat for each "layer" of nested transactions.
6. Wait a block for cross links to be published.
7. Submit the Root Transaction
8. Wait a block for cross links to be published.
9. Submit all Signalling transactions on all Shards on which Transaction Segments were executed that updated state.

5. Live Parameter Checking

Live parameter checking is used to ensure that as contract code executes, the actual value of parameters passed into cross shard function calls matches the parameter values that are in the signed Transaction Segments. Recall that the parameter values for Transaction Segments are published in System Events when the transactions execute, and that the application feeds this information into the Transaction Segments or Root Transactions that call the functions that they represent. This means that the EE has access to the expected parameter values for cross shard function calls from Transaction Segments as the code executes.

The diagram below shows example contract code that calls a function on another shard that returns a value (sh2.ee2.conC.funcC()), and then, depending on the value of state1, might call a function on another shard that does not return a value (sh1.ee4.conD.funcD()).

[

Live Parameter Checking

2688×1078 186 KB

](https://ethresear.ch/uploads/default/original/2X/b/bc464b19898cdce5ea2c9e0e438ea25620c54b6a.png)

When an application is creating the Transaction Segments for the calls to funcC and funcD, it needs to simulate the code execution. For example, if _param1 is going to be 5, and state1 is 2 and state2 is 3, and funcC will return 10 given the parameter is 5, then Transaction Segments need to be created with parameter values 5 passed into funcB, 5 passed into funcC, and 13 passed into funcD. Note that if state1 was 1, then Transaction Segments would only be constructed for the call to funcB and funcC, as funcD would never be called.

6. Safety & Liveness

The following walks through a set of possible failure scenarios and describes how the scenario is handled.

6.1 Root Transaction Fails

If the Root Transaction fails for any reason, a System Event is created that indicates that all updates for all Transaction Segments should be discarded. Based on this System Event, Signalling Transactions can be used to unlock all contracts on all Shards in all EEs. The Root Transaction could fail because:

- The code in the Root Transaction could throw an error.
- The System Event messages passed into the Root Transaction may indicate that an error occurred with one of the Transaction Segments.
- The parameters that a cross shard function is called with do not match the parameter values in the System Event message for the Transaction Segment for the function call.
- The Root Transaction is submitted after the Cross Shard Transaction block time-out.

6.2 Transaction Segment Fails

If a Transaction Segment fails for any reason, a System Event is created that indicates that it failed. This System Event can be passed up to the Root Transaction to cause the entire Cross Shard Transaction to fail. A Transaction Segment could fail because:

- The code in the Transaction Segment could throw an error.
- The System Event messages passed into the Transaction Segment may indicate that an error occurred with one of the subordinate Transaction Segments.
- The parameters that a cross shard function is called with do not match the parameter values in the System Event message for the Transaction Segment for the function call.
- The Transaction Segment is submitted after the Cross Shard Transaction block time-out.

6.3 Invalid Merkle Proof or Invalid System Event

The application could submit an invalid Merkle Proof or an invalid System Event message such that the hashing of the System Event message combined with the Merkle Proof does not yield a Merkle Root that matches the Cross Link's state root. In this case, the transaction in question would fail.

6.4 Application does not submit a Transaction Segment

The application could see that a subordinate Transaction Segment has failed, and decide to not submit any further Transaction Segments, the Root Transaction, or Signalling Transactions. To address this, another user could wait for the Cross Shard Transaction to time-out, and submit a Root Transaction to fail the overall Cross Shard Transaction (this would be free), and submit Signalling Transactions (would reward the caller) and the Clean Transaction (would reward the caller) to unlock all locked contracts and remove the Cross Shard Transaction Id.

6.5 Application does not submit a Root Transaction

The application could see that a subordinate Transaction Segment has failed, and decide to not submit the Root Transaction, or Signalling Transactions. To address this, another user could wait for the Cross Shard Transaction to time-out, and submit a Root Transaction to fail the overall Cross Shard Transaction (this would be free), and submit Signalling Transactions (would reward the caller) and the Clean Transaction (would reward the caller) to unlock all locked contracts and remove the Cross Shard Transaction Id.

6.6 Replay Attacks

The Root Transaction and Transaction Segments are tied to the account used to sign the Start Transaction, except in failure cases when the Cross Shard time-out has expired. The transactions for the account will have an account nonce, that will prevent replay. Other users that try to replay Root Transaction and Transaction Segments will fail as they will not be able to sign the transactions with the private key that signed the Start Transaction.

6.7 Denial of Service (DOS) Attacks

An attacker could submit Clean transactions that include correct data, with a single incorrect Merkle Proofs. The attacker could submit the transaction repeatedly in an attempt to cause lots of processing to occur on the Ethereum Clients. A large call graph would result in the Ethereum Clients needing to process many Merkle Proofs. This behaviour is discouraged by penalising invalid Clean transactions. Additionally, any user could receive a small reward for submitting a valid Clean transaction.

6.8 Beacon Chain Forks

If the Beacon Chain forks, transactions for shard blocks on all shards will need to be replayed, based on the revised Cross Links. Some of the replayed transactions might expect Cross Links that no longer exist, and hence rather than passing, these transactions will fail. In this scenario, the Cross Shard Transaction would fail and the provisional state for any state updates would be discarded.

6.9 Finality

The Cross Shard Transaction would be final once the Checkpoint after the last Signalling Transaction has been finalised. This is usually the first beacon block in an epoch. Once a Checkpoint is final, all prior blocks are final, and implicitly all prior Cross Links will be final, and hence all Shard blocks will be final.

7. Hotel Train Example (including ERC20)

The Hotel and Train problem is an example of a scenario involving a travel agent that needs to ensure the atomicity of a complex multi-contract transaction that crosses three shards. The travel agent needs to ensure that they either book both the hotel room and the train seat, or neither, so that they avoid the situation where a hotel room is successfully booked but the train reservation fails, or vice versa. Additionally, payment via ERC 20 tokens needs to only occur if the reservations are made. A final requirement is that the transactions need to occur in such a way that other users can book hotel rooms and train seats, and pay for them, simultaneously.

Imagine there are three shards involved: the travel agency operates on Shard 1, the hotel on Shard 3 and the train on Shard 4. Also have a second travel agency that operates on Shard 2. This is shown diagrammatically below.

[

Hotel Train Example

2648×1462 455 KB

](https://ethresear.ch/uploads/default/original/2X/a/ac48e5515da9ef235b21f6338fb51a5a5318be85.png)

The hotel is represented as a nonlockable router contract and multiple lockable hotel room contracts. The hotel issues ERC 20 tokens for travel agencies to pay for hotel rooms. The ERC 20 contract consists of a router contract and one or more payment slot contracts for each account. Similarly, the train is represented by a nonlockable router contract and mutiple lockable train seat contracts.

To book a room, the travel agency creates cross shard function calls (Transaction Segments) that book a hotel room and a train seat and pay for them. The Hotel Router contract works by finding an appropriate room that is available on the requested day that is not currently booked, and then booking the room. When searching for a room, the code skips room contracts that are currently locked. Similarly, when paying for a room, the ERC router contract needs to transfer money from the travel agency's account to the hotel's account. It does this by finding a payment slot contract for the hotel that is not locked and paying into that.

Given the ability to determine programmatically which contracts are locked, and thus avoid them, the hotel, train, and ERC 20 contracts can be written such that both travel agencies can simultaneously execute bookings without encountering a locked contract.

8. Other Considerations

Account Nonces: Account nonces are deemed to be outside lockable state. As such, when an account submits a transaction, and the transaction nonce value increases, the account does not get locked.

Ether Transfer: Currently, this scheme is just focused on function calls. Ether transfer between shards may be possible with this technique. It just has not been considered yet.

Pay for Gas on All Shards: It would be great if a user could have Ether on one shard, and use it to pay for gas on all of the shards that the Cross Shard Transaction executes on. This technique ca not do that at present. Perhaps when Ether Transfer is resolved, this will be possible.

Transaction size: Transactions in this proposal often include multiple Merkle Proofs and other data. The probable effects on transaction size need to be analysed.

Ethereum 1.x: Assuming Ethereum 1.x is in an EE on a shard, all existing contracts could be marked as nonlockable. The EE could support the features described in this post. Additional EVM opcodes could be added to allow cross shard function calls.

Application code: I hear you say, “There sounds like a lot of complexity in the application. Wasn’t this technique supposed to make application development simpler?” The answer to this is that the vast majority of the complexity will be absorbed into libraries such as Web3J. Contract code and other application code should be straight forward.

9. Acknowledgements

This research has been undertaken whilst I have been employed full-time at ConsenSys. I acknowledge the support of University of Queensland where I am completing my PhD, and in particular the support of my PhD supervisor Dr Marius Portmann.

I acknowledge the co-authors of theoriginal Atomic Crosschain Transaction paper and the co-authors of theAtomic Crosschain Transaction white paper @Raghavendra-PegaSys (Dr Raghavendra Ramesh), Dr Sandra Johnson, John Brainard, Dr David Hyland-Wood and Roberto Saltini for their help creating the technology upon which this post is based. I thank @cat (Dr Catherine Jones) for reviewing this paper and providing astute feedback. I thank@benjaminion (Ben Edgington) for answering lots of questions about Eth2.