cChannel Generalized State Channel Specification

Celer Network Core Team

May 27, 2018

cChannel of the Celer Network is a generalized state channel and sidechain suite that provides a framework to support fast and flexible state transitions with generic dependency DAG on networked channels. This short paper describes the specification and features of cChannel's generalized state channel.

1. Key Idea and a Simple Example

The concept of payment channel was initially proposed by the Lightning Network [2] to support high throughput off-chain transactions. Its major limitation is the lack of support for generalized state transitions. The need for off-chain support of generalized state transition comes with the rise of smart contract platforms such as Ethereum. Smart contract enables asynchronous value transfer based on arbitrary contractual logic. To improve the scalability of such blockchains, on-chain state transitions could be put into off-chain state channels and the corresponding value transfer should be made aware of such state transitions.

We use a simple example of *conditional payment* to illustrate the key idea about how to transform on-chain states transition to off-chain states transitions. Let's say Alice and Carl want to play a board game while betting on the result of such game in a trust-free manner: Alice will pay Carl $1 if Carl wins and vice versa.

This is a simple logic to implement on-chain. One could build a smart contract that holds Alice's and Carl's payment before the game starts. Alice and Carl will just play that game by calling on-chain smart contract's functions. When one of them loses the game, surrender or simply timeout, the winner gets the loser's deposit. The deposits can be seen as payments that are conditionally issued (i.e. condition on that the counterparty wins). Unfortunately, on-chain smart contract operations are extremely slow and expensive as every transaction involved an on-chain transaction.

To significantly improve scalability while maintaining the same semantic, off-chain

state channel design pattern can be used. Assume there is a payment channel between Alice and Carl, to enable the above semantic, we need to expand the functionality of channel's state proof to include a conditional lock that depends on the game's winner state. Alice can then send Carl an off-chain payment effectively saying: "I will pay Carl $1 if the game contract's *who_is_winner* function says Carl wins". The game state transitions can be also moved off-chain. A straightforward way is to have an on-chain contract governing the rule of the board game and that contract's address is referenced in the conditional payment. All the state transitions are happening off-chain via mutually signed game states that can be injected into the on-chain contract.

But in fact, since there is no requirement for any kind of value bond for program states, the entire game contract and the associated states can all stay off-chain as long as every party is collaborative. The only requirement is that the relevant game states are on-chain verifiable when need to be. An on-chain verifiable state means other contracts or objects can refer to it with no ambiguity. To realize that, we need to have a reference translator contract that translates off-chain references (e.g., the hash of contract code, constructor parameters, and nounce) to on-chain references (e.g., contract address). With these constructs, the game between Alice and Carl involves only one on-chain contract and no on-chain operation.

The example above is the only a simple and special example of off-chain design patterns, and it can be much more sophisticated. The conditional payment can be more complicated than just simple Boolean conditions, and can be designed in a way to redistribute locked up liquidity based on arbitrary contractual logic. In fact, *conditional payments* is just a special case of generalized *conditional state transition*. The channel dependency can also be more complicated than one-to-one dependency, and can realize the common pattern of multi-hop state relays. We detail out the technical specification in the followings sections.

2. Design Goals

Our top goal is to achieve fast, flexible and trust-free off-chain interactions. We expect in most cases off-chain state transitions will stay off-chain until final resolution. Therefore, we aim to optimize commonly used off-chain patterns into succinct interactions with built-in support from on-chain components. This does not sacrifice generality but

does promote a rich-feature on-chain layer.

Our second goal is to design data structure and object interaction logic that works for different blockchains. Celer Network aims to build a blockchain agnostic platform and to run on different blockchains that support smart contracts. Therefore, a common data structure schema and a certain layer of indirection are required.

Besides these two highlighted goals, we plan to employ formal specification of channel state machines and verify the security properties along with the communication protocols that alters those states. We also aim to provide an efficient on-chain resolution mechanism whenever possible.

3. Channel Specification

In this section, we provide specifications for the core components of cChannel's generalized state channel with a top-down approach, and describes the **Common State Channel Interface** that applies to any state channel with value transfer and arbitrary contractual logic. There could be extensive specialization and optimization for different concrete use cases, but the principles stay the same.

Before the detailed specification of a generalized state channel, we first introduce several important notations and terms that will be used throughout this section.

- (**State**). Denote by s the state of a channel. For example, for a bi-party payment channel, s represents the available balances of the two parties; for a multi-party board game, s represents the board state.
- (**State Proof**). State proof serves as a bridge data structure between on-chain contracts and off-chain communication protocols. A state proof sp contains the following fields

$$sp = \{\Delta s, seq, merkle\_root, sigs\}, \tag{1}$$

where Δs denotes the most recent accumulative state updates. Note that given a base state s_0 and a state update Δs, we can uniquely produce a new channel state s. For example, in a bi-party payment channel, the base state s_0 corresponds to deposits of the two parties and the state update Δs is a mapping that indicates the amount of tokens transferred from one participant to the other participant. seq is the sequence number for the state proof. A state proof with a

higher sequence number will invalid state proofs with lower sequence numbers. *merkle_root* is the root of the Merkle tree of all pending conditions and is crucial for creating a conditional dependency between states in cChannel. Finally, *sigs* represents the signature from all parties on this state proof. State proof is valid only if all parties signatures are present.

- (**Condition**). Condition *cond* is the data structure representing the basic unit of conditional dependency and this is where the conditional dependency DAGs are weaved. A condition can be specified as follows.

$$cond = \big\{timeout, *\textsc{IsFinalized}(args), *\textsc{QueryResult}(args)\big\} \tag{2}$$

  Here, *timeout* is the timeout after which the condition expires. For example, for a condition that depends on the results of a board game, *timeout* may correspond to the maximum duration of the board game (e.g., approximately 20 minutes in block time). Boolean function pointer $\textsc{IsFinalized}(args)$ is used to check whether the condition has been resolved and settled before the condition timeout. The arguments for this function call are application-specific. For example, in the board game, the arguments could be as simple as $args = [blocknumber]$ querying whether the game winner has been determined before *blocknumber*. In addition, $\textsc{QueryResult}(args)$ is a result query function pointer that returns arbitrary bytes as the condition's resolved result. For example, in the board game, the arguments could be $args = [player1]$ querying whether $player1$ is the winner (boolean condition); in the second-price auction, the arguments could be $args = [participant1, participant2, \cdots, participantN]$ querying who is the winner and the amount of money each participant should pay (generic condition). The resolution process for a condition is to first perform $\textsc{IsFinalized}(args)$ and then perform result query $\textsc{QueryResult}(args)$.

- (**Condition Group**). Condition group *cond_group* is a higher-level abstraction for a group of conditions to express generalized state dependencies. A condition group can be specified as follows.

$$cond\_group = \big\{\Lambda, \textsc{ResolveGroup}(cond\_results)\big\}, \tag{3}$$

  where Λ denotes a set of conditions contained in this condition group. Each condition $cond \in \Lambda$ resolves to an arbitrary bytes array (i.e., the output of

cond.QUERYRESULT(*args*)). These bytes array are handled by a group resolving function RESOLVEGROUP(*cond_results*) which takes the resolving results of all conditions as inputs and returns a state update Δs. For a payment channel, each condition group corresponds to a conditional payment. For example, a conditional payment saying that "A pays B $1 if B wins the game" corresponds to a condition group that contains two conditions: the Hashed Time Lock condition (for multi-hop relay) and the game condition ("B wins"). The RESOLVEGROUP function simply returns a transfer of $1 from A to B if both conditions are true.

Now we are ready to specify the interface for a state channel. A state channel $\mathcal{C}$ can be specified as the following tuple:

$$\mathcal{C} = \{p, s_0, sp, s, \mathcal{F}, \tau\}, \tag{4}$$

$p = \{p_1, p_2, ..., p_n\}$ is the set of participants in this channel. s_0 is the on-chain base state for this channel (e.g., initial deposits for each participant in a payment channel). *sp* represents the most updated known state proof for the channel. s is the updated channel state after state proof *sp* is fully settled. τ is the settlement timeout increment for the state proof that will be specified later. $\mathcal{F}$ contains a set of standard functions that should be implemented by every state channel:

- RESOLVESTATEPROOF(*sp*, *cond_groups*). This function updates the current state proof by resolving attached condition groups.
- GETUPDATEDSTATE(*sp*, s_0). This function is used to get the most updated state based on off-chain state proof *sp* and on-chain base states s_0.
- UPDATESTATE(s). This function allows on-chain updates of state channel's currently resolved state s.
- INTENDSETTLE(*new_sp*). This function opens a challenge period before the settlement timeout. During the challenge period, this function takes a state proof as input and update the current state proof if the input is newer.
- CONFIRMSETTLE(*sp*). This function validates and confirms the current state proof as fully settled given the current time has exceeded the settlement timeout.
- ISFINALIZED(*args*) and QUERYRESULT(*args*) are the entry points for resolving conditional dependency. It accepts outside queries with necessary arguments for the querying contract to interpret accordingly. In fact, some patterns are used

frequently enough, in cChannel's implementation, we separate them into predefined function interfaces.

- CloseStateChannel(s). This function terminates the life cycle of the state channel and distributes necessary states (e.g., account balances) out according to the latest settled state s.

Settlement timeout is determined based on time of last called ResolveStateProof or SettleStateProof and the settlement timeout increment is τ.

Dependency Constraints.. When we creating dependencies among different state channels, some constraints need to be enforced in order to guarantee a proper resolution of the dependency DAG. Suppose state channel $\mathcal{C}_1$ depends on state channel $\mathcal{C}_2$. Then it is required that the participants of $\mathcal{C}_1$ should be a subset of the participants of $\mathcal{C}_2$ such that the participants of $\mathcal{C}_1$ have the necessary information resolving its dependency on $\mathcal{C}_1$.

4. Common Utilities

The above abstraction defines the common pattern for generalized state channel construction. In different blockchains, the actual implementation might be different. For example, cross-contract calls contains return value in Ethereum but only trigger registered callbacks in Dfinity. After reviewing multiple blockchain implementations on state transition VMs, we identify two common utilities that are essential for the operation of generalized state channel in practice as followings:

- **Off-chain Address Translator (OAT).** In the above abstraction, condition and condition group are associated with different functions. These functions should be the reference of on-chain contract's functions, but since program (smart contract) states are not inherently bound to constraints on the blockchain, there should be no fundamental requirement to have an on-chain presence. The only barrier to moving them entirely off-chain is the possible ambiguity of reference for functions such as IsFinalized and QueryResult. To resolve this ambiguity, we can define an on-chain rule set to map off-chain references to on-chain references. Off-chain Address Translator is built for that. An off-chain contract can be referenced by a unique identifier generated through its contract code, ini-

tial states, and a certain nonce. We call such unique identifier off-chain address. When resolving conditions on-chain, the referenced contracts need to be deployed and the corresponding functions (e.g., IsFinalized and QueryResult) should be able to be translated from off-chain addresses to on-chain addresses. To realize such functionality, OAT needs to be able to deploy contract code and initial states on-chain to get an on-chain contract and establish the mapping from off-chain address to on-chain address.

- **Hash Time Lock Registry(HTLR).** Hash Time Lock is commonly used in the scenario where transactions involving multiple state channels need to happen atomically. For example, multi-hop relayed payment (unconditional or conditional), atomic swap between different tokens, cross-chain bridges and more. HTL can be implemented entirely off-chain, but as Sprite [1] has pointed out, this is an over-optimization that actually limits the off-chain scalability. Therefore, Sprite [1] proposes a central registry where all locks can refer to. We extend and modify Sprite to fit in cChannel's common model. Effectively, HTLR provides dependency endpoints (IsFinalized, QueryResult) for conditions that act as locks. IsFinalized takes a hash and block number and returns true if the corresponding pre-image has been registered before the block number. QueryResult takes a hash and returns true if the pre-image of the hash is registered. These two functions can be simplified further into one, but for the sake of generality, we can simply keep them as two separate functions. Note that HTLR, and associated IsFinalized and QueryResult, is always on-chain.

5. Out-of-box Features

Additionly, we need to look into patterns that would be commonly used and enhance certain on-chain components with out-of-box features to simplify the corresponding off-chain interactions. **Generalized Payment Channel (GPC)** is a very good example of that. Generalized Payment Channel is payment channel that conforms to the general state channel specification and therefore can support various conditional payments based on further on-chain or off-chain objects.

We first make the abstract model more concrete in the context of GPC. s_0 represents the static deposit map for each party in p. s represents the final netted value

each party owns. SubmitStateProof is the function to submit a state proof and trigger a timeout challenge period before SettleStateProof can be called and confirm the state proof. IsFinalized and QueryResult are functions to check whether the state of this payment channel has been finalized and query the current balances. One may wonder why a payment channel has an interface for outside query. This is because some other payments or states may depend on s or existence of certain conditional payment locked in sp. ResolveStateProof is the most interesting part as this is where a lot of specialized optimization will happen and greatly reduce the off-chain interaction complexity. We will discuss this part later. GetUpdatedState is a straightforward function to compute the netted out payment for each party based on the initial deposit and the fully resolved sp. CloseStateChannel simply closes the channel and distributes the netted out the final balance for each party.

With this basic model, we discuss how we can further optimize the GPC constructs to enable useful out-of-box features.

- **Dynamic Deposit and Withdrawal.** A common requirement for GPC is to enable seamless on-chain transactions when the counterparty is not connected to the network. For withdrawal, we introduce a pair of functions IntendWithdraw and ConfirmWithdraw, to meet this requirement. IntendToWithdraw changes the base state s_0 with a challenge period. Counter-party can submit conflicting sp to dispute. If no dispute happens before the challenge period defined by τ, ConfirmToWithdraw is called to confirm and distribute the withdrawal. These two functions works very much like IntendSettle and ConfirmSettle. Deposit is straightforward as it only changes the base state s_0.
- **Boolean Circuit Condition Group.** We conjecture GPC's most common use case would be Boolean circuit based conditional payment. For example, "A pays B if function X or function Y return true". To optimize for such payment, we tweak the interface of condition group and condition. In particular, we can specialize function ResolveGroup to release a pre-defined conditional payment if any of the condition resolving results (or any boolean circuit of condition results) holds true. This way, we saved the trouble of creating additional objects for ResolveGroup and the corresponding multi-party communication overhead. We also specialize condition as Boolean condition so that we require the

depending objects should have an interface with the effect of "isSatisfied" that returns true or false based on the state queried.

- **Fund Assignment Condition Group.** Another more generalized use case for GPC is generalized state assignment. We implement this by introducing another different type of condition group, which only has one condition in it. QueryResult will directly return a state assignment map dictating an update for Δs. This enables a more general plug-in point for GPC. One can plugin an off-chain contract that was initialized with certain locked in liquidity. This contract can check not only who wins a game (Boolean) but also how many steps the winner took to win the game, and then assign the value transfer by carrying out certain computations (e.g, winner get more money if won with fewer steps). The involved parties can generate a Condition Group that references the check function of the off-chain contract address they mutually agree on.

There can be many more common patterns defined for different patterns, but the above example illustrates the design principle for such optimization.

References

[1] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry, *Sprites: Payment channels that go faster than lightning*, CoRR abs/1702.05812 (2017). Available at http://arxiv.org/abs/1702.05812.

[2] J. Poon and T. Dryja, *The bitcoin lightning network: Scalable off-chain instant payments*, Technical Report (draft) (2015).