Force-Move Games

Tom Close Andrew Stewart

Magmo Research

research@magmo.com

June 21, 2018

Abstract

The current scalability limitations of decentralized crypto-currencies, like Bitcoin and Ethereum, are a major barrier to their wide-spread adoption. Without a central authority, it is a challenge to maintain distributed consensus while also achieving the throughput necessary to support everyday financial transactions. State channels help to reduce the volume of on-chain transactions by enabling trustless, off-chain interactions between a fixed set of participants. In this paper, we present a state channel framework capable of running a restricted set of n-party state channel applications. This restricted set is general enough to encompass many of the common state channel applications, while keeping the framework simple enough to be readily amenable to immediate development. As a proof of concept, we implement the framework for the 2-party case on Ethereum.

Contents

1 Introduction

The recent rise in the popularity of cryptocurrencies has highlighted their current scalability limitations. The requirement to keep block processing and transmission times small, relative to the block creation time, leads to an inherent limit to the number of transactions that a block can contain, in both Bitcoin and Ethereum. In Ethereum, the maximum transaction rate is approximately 15 transactions per second across the entire network [1]. When the volume of transactions surpasses the network capacity, the consequences can be disastrous for those trying to use the cryptocurrency for everyday transactions, as was seen with Bitcoin in Dec 2017, when the average transaction fee exceeded $50 [2] and average confirmation times were longer than an hour [3]. When benchmarked against the visa card network, which can handle over 50,000 transactions per second [4], it becomes clear that scaling solutions are needed if Ethereum is to achieve widespread adoption.

Payment channels and state channels represent one solution to this scalability problem. Both these approaches reduce the volume of on-chain transactions, by enabling trustless, off-chain interactions between a fixed set of participants.

1.1 State channel overview

A state channel can be thought of as a protocol for dividing a predetermined set of assets between a predetermined set of participants, using the blockchain to avoid requiring mutual trust or a trusted third party. Although the blockchain is integral to the security of the setup, the protocol is designed so that the participants are able and incentivized to perform the majority of the operations off-chain.

In order for state channel interactions to be trustless, the assets in question must be locked in an on-chain contract, and only accessible through the outcome of the state channel. These locked assets are often referred to as the *state deposit* in the literature.

Once the state deposit is in place, the interaction proceeds with participants exchanging cryptographically signed messages off-chain. Under normal operation, one player proposes a state update by broadcasting a signed copy of their desired state transition. The state progresses when all other participants sign the new state.

When the interaction is over one or more participants can present what they claim to be the latest agreed state to the blockchain. This typically starts a timeout period to allow other participants to submit a later state, if one exists. Once the timeout period has elapsed, the blockchain divides the assets according to the final state.

1.2 Disputes and resolutions

A *dispute* occurs when one or more players refuse to sign the new state. When a dispute occurs, the *resolution process* can be started on-chain. The process starts with a submission phase, where participants are invited to submit data that will be used by the adjudicator when resolving the dispute. There is typically a predefined duration for this phase. If the dispute is not resolved during the submission phase, the adjudicator releases funds according to the *resolution* specified by the most recently known state at the time of the dispute.

The security of a state channel system comes from the ability of the players to reason about how the blockchain will resolve their disputes: if you know that the signed agreements you hold will allow you to enforce a particular outcome via the blockchain, then you can consider the current state to be in some sense equivalent to that outcome. This sort of reasoning forms the basis for the *counterfactual* techniques, introduced in [5].

The on-chain dispute resolution behaviour is highly dependent on the kind of disputes that it must handle. Here we categorise the disputes into 3 broad classes, each of which capable of preventing the transition to a new agreed state:

> *External state* disputes can occur when state-channel transitions are dependent on properties external to the state that change with time. As long as an interaction proceeds off-chain, the only notion of time, and subsequently time-dependent properties, is the time which all participants agree on. If Alice claims that, at the time she sent Bob her state update, the ETH-USD exchange rate was x and he refuses to sign the new state, she has no choice but to go to the chain - and to go to the chain quickly before the value of x changes significantly.
>
> *Conflicting move* disputes occur when players send conflicting updates. For example, in a payment channel Alice might try to update the state to $(4, 6)$ at exactly the same time that Bob tries to update it to $(6, 4)$. As discussed in the previous point, the lack of an undisputed clock means that they cannot tell which of these updates was first. In the general case, the resolution function needs to be able to accept a set of conflicting updates and decide how the state should progress.
>
> *Inactivity* can be viewed a dispute in the sense that it prevents the overall state from progressing. This case also covers sending invalid state updates, which are viewed by the framework as equivalent to not sending any update. Inactivity covers a range of different situations that cannot in general be distinguished by the blockchain. At one end of the range, inactivity could be due to a lack of connectivity or the loss of the keys needed to sign states. At the other inactivity could be a deliberate strategy to avoid signing updates that are not in the player's interest. In both cases it is important that the state can progress on-chain, so that the funds of the active players are not locked indefinitely.

As we will explain shortly, the force-move game framework manages to completely rule out the first two types of dispute here, by restricting the set of applications that can run on the framework.

1.3 The force-move game approach

In contrast to fully general state channel frameworks, force-move games have the property that the outcome of the resolution in any state is straightforward to calculate and does not change with time. To accomplish this we make two important restrictive choices in the framework. Firstly we store the resolution properties on the channel states themselves, which removes disagreements due to external states.[1] In doing this, we naturally introduce significant restrictions on the types of application that the framework will support.

Secondly, we specify that participants must take turns signing states. This removes any disagreements that could arise due to conflicting moves, as it is always completely defined whose move it is. This again likely introduces some restrictions on the types of application that the framework will support, but in practice it seems as though these can mostly be mitigated by adapting the design of applications.

Once we rule out the first two types of agreement, we're left with the inactivity case. We handle this with the *force-move* operation, which the framework is named after. If an opponent is inactive, a participant can issue a force-move operation on-chain, which will wait a predetermined length of time for the opponent to respond and otherwise terminates the game so that the assets can be withdrawn.

1.4 Related work

The most well-known payment channel implementation is the Bitcoin lightning network [6], which recently launched on the mainnet. The network handles the routing of multi-hop payments across a distributed network of nodes, secured using the hashed time-locked contracts (HTLCs) approach [7]. The Raiden network [8] is working on bringing the same idea to Ethereum.

The enhanced smart contract capabilities of the Ethereum blockchain open up additional capabilities when compared to Bitcoin. The sprites paper [9] used this fact to increase the efficiency of HTLC multi-hop payment channels on Ethereum. The Perun protocol [10, 11] introduces the concept of virtual channels, an alternative approach to enabling unconnected parties to interact through one or more intermediaries, which readily supports state channel interactions. The work is key to the efficacy of state channels as a scaling solution, by allowing a large proportion of state channels to be opened and closed without an on-chain transaction.

[1] This is semantically equivalent to having a *pure* resolution function.

1.5 Notation

Due to the inherent complexity in dealing with on-chain and off-chain state, we will be using the following pseudocode notation throughout the paper, which allows us to go beyond functions and specify `Structs` and `Types`:

Pseudocode notation

```
// Struct definition
struct StructName contains
    memberName: MemberType
    otherMemberName: OtherMemberType
end

// Method definition
StructName::methodName: input ↦ output

// Function definition
function functionName(arg: Type1, arrayArg: Type2[])
    returns (returnType)
    Require(someCondition)
    // Implementation
    return returnValue
end function
```

As you can see, we allow our `Structs` to have convenience methods defined on them. These methods are all simple "getter" methods, returning properties that can be easily

calculated from the attributes stored on the `Struct`. All state-modifying actions will be performed by `Functions`.

We use the convention that `Types` and `Structs` are capitalized, while `attributes`, `methods`, and `functions` are not.

This paper starts with an informal introduction to the force-move game framework in Section 2, which uses rock-paper-scissors [12] as an example. Sections 3 and 4 give a formal description of the framework, and then the final sections cover approaches for *funding* the games.

2 Informal introduction to force-move games

The force-move game framework prescribes a format for the off-chain agreements and specifies the rules surrounding how these agreements will be interpreted by the on-chain adjudicator. It also specifies the interface that application developers should adhere to when writing the bespoke code that will power their application.

As a way of introducing the framework, we will start by presenting an example application and detailing some of the main interactions that can happen between the two players playing the game. For the example, we will demonstrate how to implement the well-known game of rock-paper-scissors [12] as a force-move game.

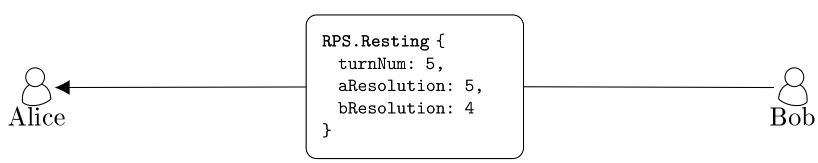
2.1 Collaborative play

In this section, we will show how the game progresses during *collaborative play* - when both players behave cooperatively and exchange signed agreements off-chain.

The real-world game involves two players simultaneously picking a move. In the force-move game implementation, we will use a commit-reveal strategy to simulate this, where one player commits to a value beforehand and then only reveals their choice once the other player's choice is known.

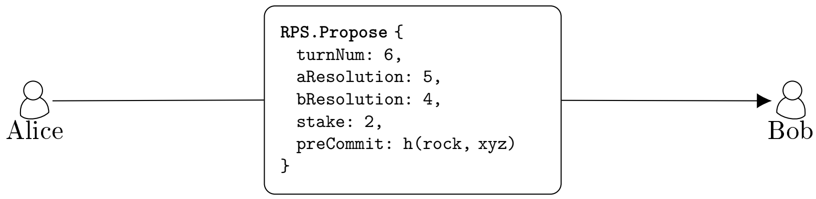
In our implementation, each round of the game will pass through four different stages: `RPS.Resting`, `RPS.Propose`, `RPS.Accept`, and `RPS.Reveal`. The states for each stage will have different attributes. Instead of trying to define all the different attributes, we will give an example of the states that would be exchanged during a round of the game played between Alice and Bob.

We will assume that we start in a position where Bob has just signed and sent the `RPS.Resting` state to Alice. It could be that Alice and Bob have just entered the game from the setup phase, which we'll cover later, or they might have just finished a previous round. To arrive in this position, Alice and Bob will have deployed an on-chain adjudicator contract and will each have deposited a sufficient amount to fund the game.

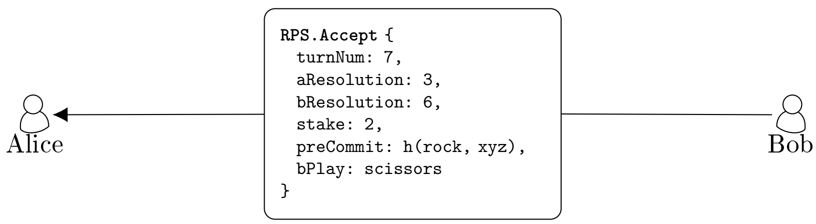


The states we show here are slightly simplified. In particular, we omit the *framework attributes* that appear in every single state, which we will cover in detail in Section 3.1.3. In the state above `aResolution` and `bResolution` represent the funds that Alice and Bob would respectively receive if the game were to end in the current state. The `turnNum` increases as each move is played. Note that the `turnNum` starts at 5 in this example, to reflect the fact that previous moves were necessary to get Alice and Bob into the starting position for our game.

Alice kicks off the round by signing and sending the `RPS.Propose` state to Bob. In doing this, she chooses a `stake` that the winner will receive from the loser. The resolution does not update at this point, as Bob has not yet agreed to the new round. She also provides the `preCommit`, which she calculates by hashing her choice, `rock`, with a random string, `xyz`:

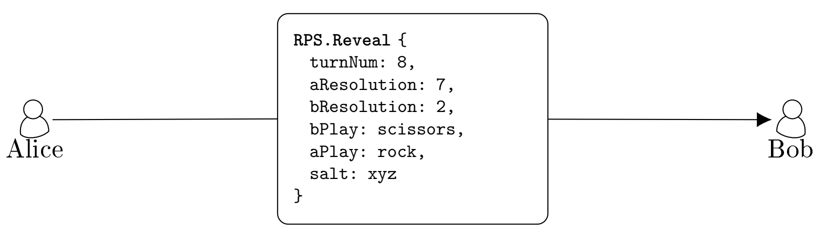


Bob then decides whether to accept the round or not. If he did not want to accept, he would sign and send back the same resting state as he sent in the beginning, apart from an increased `turnNum`. If he does want to accept, he signs the `RPS.Accept` state, providing his choice, in this case `scissors`:



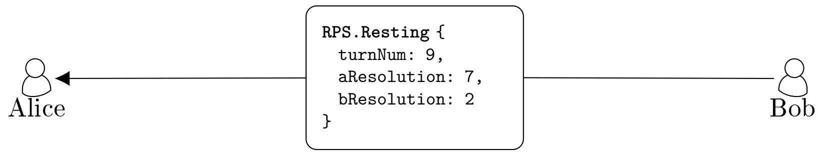
Note that, at part of this transition, Bob has updated the resolution as though he had won: removing an amount of `stake` from `aResolution` and applying it to `bResolution`. This change is specified by the rules of the game and is a crucial part of making the game game-theoretically sound. The risk here is that once Alice receives this state, she knows whether she has won or not but no-one else does. Without the added incentive of Bob being the default winner in this position, it could be in Alice's interest to end the game at this point, by refusing to reveal the outcome.

The next step is for Alice to reveal her value. To do this she signs the `RPS.Reveal` state, which reveals her choice. She also provides the `salt` used in the pre-commit, so that Bob can verify that she has not changed her choice:



Alice has also updated the `resolution` to reflect the fact that she is the winner of the round.

Bob then completes the round by signing the `RPS.Resting` state.



Now they are back in the `RPS.Resting` state, Alice is free to propose another round if she wishes. As it stands, this is all she can do. We'll talk about how to add a way to conclude the game in Section 3.2.3.

2.2 Defining game rules

For each of the interactions described above, the framework must be able to judge whether the message sent was valid or not. In order meet this requirement, the application developer deploys an on-chain library containing the rules of the game, which specifies which transitions are valid. This library only needs to be deployed once (and not once per game played) and the address it is deployed at can be used to unambiguously define the game being played in a channel.

A lot of the transition rules are fairly straight-forward. In the rock-paper-scissors case they would include making sure that players do not change their plays or the `stake`, and that the `resolution` updates appropriately, according to the state. For example, the transition rules for the `RPS.Propose` $\mapsto$ `RPS.Accept` transition are as follows:

- `aResolution := aResolution - stake`
- `bResolution := bResolution + stake`
- `stake` shouldn't change
- `preCommit` shouldn't change
- `bPlay` is one of `rock/paper/scissors`

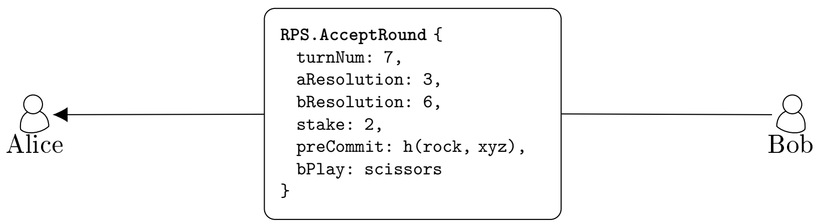
In order to specify these rules, the game library must have a `validTransition` function that takes two states and returns true if the transition from one state to the other is valid.

2.3 Force-move and resolution

So far we have only looked at the case where Alice and Bob behave cooperatively. In general, we cannot assume that this will be the case. For example, a player might not be able to cooperate due to a loss of their internet connection or signing key. As we have already seen, it could also be that a player is incentivised not to cooperate, as a given transition is not in their economic interest. In both these cases, we need to make sure that the other player has the capability to reclaim the fair amount of funds according to the current state of the game.

As an example, we'll look at the `RPS.Accept` $\mapsto$ `RPS.Reveal` transition. In our example above, the revealer (Alice) knew that she had won the game, so it was clearly in her interest to reveal that fact to claim her winnings from Bob. If Alice hadn't won, she might have been tempted to stall the game, preventing Bob from claiming his winnings.

We'll look at the case where Bob has just sent the following `RPS.Accept` state to Alice but Alice has not responded in some time:



In this case, Bob can go to the blockchain to force Alice to continue the game. To do this he calls the `forceMove` operation on the on-chain adjudicator contract, which exists from the game setup phase:

```
Adjudicator.forceMove(RPS.Propose{...}, RPS.Accept{...})     (1)
```

Note that Bob provides both the last state signed by Alice and the last state he signed. This is in accordance with the idea that state channels only progress by complete consent of the participants: Bob needs signatures from both parties to launch a challenge. In calling this method, he lays down the following challenge to Alice:

Bob: Alice, you moved to `RPS.Propose`, after which I moved to `RPS.Accept`. Now it's your turn to move!

When Bob plays the force-move, a deadline is set for Alice to respond. If this deadline expires before Alice responds, the `resolution` stored on the challenge state s will decide how the funds should be split:

```
s.resolution #=> {aResolution: 3, aResolution: 6}     (2)
```

Not that the state resolves as though Bob has played the winning move. As discussed earlier, this is because the only fair resolution in the `RPS.Accept` state is to award all the stake to the non-revealer – otherwise the revealer would always be incentivised to stall when they had not won.

A *game diagram* is a useful way of bringing together all the pieces of the game that an application developer must specify. Fig 1 shows the game state diagram for the rock-paper-scissors game.

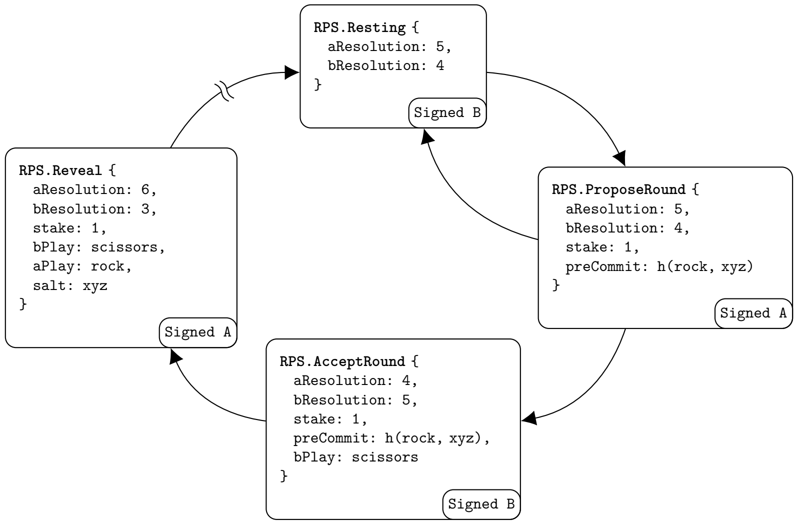


Figure 1: Game diagram for a basic 2-player force-move game formulation of rock-paper-scissors. Allowed transitions are shown with solid arrows.

2.4 Responding to a force-move

If Alice wants to avoid ending the game according to the challenge state's resolution, she can respond to the force-move operation. There are several ways to respond to a force-move, which will be covered comprehensively in Section 4.4. In this case, we'll cover the most straight-forward of these options: the `respondWithMove` response.

In responding with a move, Alice answers Bob's force-move as follows:

Bob: Alice, you moved to `RPS.Propose`, after which I moved to `RPS.Accept`. Now it's your turn to move!

Alice: Ok, Bob. Here's my move to the `RPS.Reveal` state.

Alice performs this action by calling the `respondWithMove` method on the on-chain adjudicator.

```
Adjudicator.respondWithMove(RPS.Reveal{...})          (3)
```

The adjudicator will check that Alice's response represents a valid transition according to the rules of the game and, if it does, cancel Bob's outstanding force-move challenge.

By responding with a move, Alice has provided the exact state that she would have done if she had sent the move directly to Bob off-chain. Alice and Bob are therefore now in the exact same situation as if the force-move had not happened and Alice had behaved cooperatively off-chain. They can therefore now continue to play the game off-chain.

2.5 Payment channels

The payment channel is an important example for any state channel framework. In this section, we show how to implement a payment channel as a force-move game.

The payment channel game is very simple, involving only the `resolution` fields, which record how much each player will receive if the game ends in a given position. The transition rules are designed to implement the rule that you should never unilaterally be able to take funds from your opponent, but you should be able to unilaterally give funds to your opponent.

Figure 2 shows the game diagram for a 2-player force-move implementation of a payment channel with capacity 2 wei. The 2 wei capacity was chosen to make it possible to easily enumerate all the possible outcomes when producing the game diagram. The game library is specified in Specification 1. The code for a payment channel is significantly more succinct than the diagram!

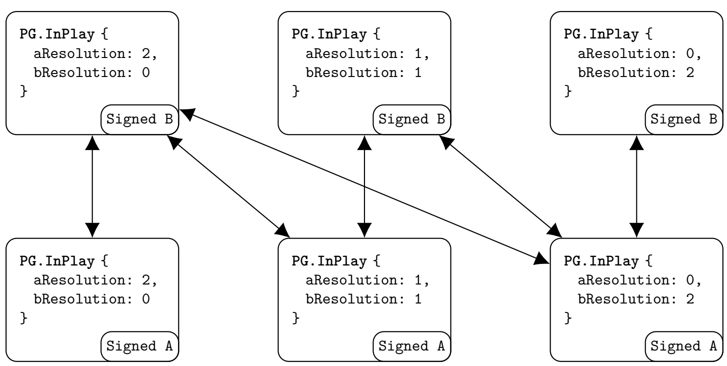


Figure 2: Game diagram for a 2-player force-move game formulation of a payment channel with a capacity of 2 wei. The rules of the payment game have been designed to respect the principle that you should never unilaterally be able to take funds from your opponent, but you should be able to unilaterally give funds to your opponent. For example, from the state in the top-left A has three choices: (1) give 0 to B, (2) give 1 to B, (3) give 2 to B. In all of these cases, B gains the ability to make the next move. By contrast if the game is in the top-right, A has only one option: to give 0 to B. Each state stores the current values of `aResolution` and `bResolution`. Additional state attributes, such as `turnNum`, have been omitted for brevity.

Specification 1 Two-player payment game

```
function PaymentGame.validTransition((s_1: State, s_2: State))
    returns (Boolean)
    Require(sum(s_1.resolution) == sum(s_2.resolution))
    i := s_1.indexOfMover
    Require(s_1.resolution[i] ≥ s_2.resolution[i])
end function
```

3 Force-move games when collaborating

In this section we will describe the framework that specifies n-player force-move games, treating the situation when the players are behaving collaboratively, so that the game can progress off-chain.

3.1 Game objects

3.1.1 Channels

A channel, γ, can be thought of as a container in which a force-move game is played. Every move made will contain a reference to a channel, and it is through the channel that we identify them as being moves made in the same instance of a given game. Note that a channel is completely unrelated to the method of communication used by the parties.

Specification 2 Framework specification

```
struct Channel contains
    channelType: Address
    participants: Address[]
    channelNonce: Uint
end
Channel::id c ↦ h(c.gameType, c.participants, c.channelNonce)
Channel::numberOfParticipants c ↦ c.participants.length
```

where h is a cryptographic hash function, such as the `keccak256` hash used by ethereum [13]. The resulting `channelId` is designed to uniquely identify the channel.

The `channelType` specifies the rules of the game being played in the channel. In practice, the channel type would be an address of the on-chain location where the rules of the game can be found.

The `participants` is a list of parties participating in the game. In practice, this would be a list of addresses corresponding to the keys which the parties are using to sign their states. The position of the parties in the list is significant and can be used by games to assign different roles to different participants, which can affect the moves they are allowed to make.

The `channelNonce` is a value chosen so that the `channelId` is unique. In order to meet this requirement, it is necessary for participants to keep track of some information about the games they've played with each opponent. In practice, this requirement is relatively small: it is sufficient for each opponent to just store the highest channel nonce used so far[2].

All players must take responsibility for ensuring that the nonce is chosen to make the channel id unique. A failure to do so can lead to funds being lost due to replay attacks from the previous channel. If the channel id is not unique, players should refuse to join the channel.

3.1.2 Outcomes

Force-move games are typically played with some assets at stake; the whole purpose of the framework is to enable the distribution of assets to be tied to the trajectory of the game, without requiring trust between the participating parties. We will refer to the split of assets at the end of the game, as the *outcome* of the game. As we will cover in later sections, outcomes can be obtained collaboratively, off-chain, through the conclusion process, or non-collaboratively, on-chain, through the challenge process.

In the most general case, an outcome consists of a list of addresses and the assets that the game has allocated to them. We say that a force-move game, g, is *closed* if, in all possible outcomes, it only allocates funds to the participants of its channel. Otherwise, we say that the game is *open*.

For the purposes of this paper, we will assume that we are working with a closed game, on a single (fungible) asset. This allows us to specify the outcome using an array, `outcome`, of length γ.`numberOfParticipants`, where `outcome[i]` represents the amount of coin to be distributed to `participants[i]`. In general, this will not be the case though and therefore we define an `Outcome` type, to make it clear exactly where the framework can be modified to support more complicated setups.

Framework specification – cont'd

```
// for the purposes of this paper
type Outcome := Uint[]
```

[2] In the case where this total reaches the maximum allowed value, to start afresh with a new set of keys

3.1.3 Moves and states

The state of an force-move game is advanced when one participant makes a *move*.[3] A move consists of a `State` and a (cryptographic) `signature`.

Framework specification – cont'd

```
struct Move contains
   state: State
   signature: Signature
end
Move::channelId: m ↦ m.state.channelId
Move::signer: m ↦ m.signature.signer
```

From the signature you can deduce the *signer* – the participant who signed the move. The game state is advanced if the signed state they send represents a valid transition. If not, the update is ignored, leading to the general principle that *making an invalid move is equivalent to not making any move at all.*

The framework defines multiple types of `State`, which all have some attributes in common:

Framework specification – cont'd

```
type State := PreFundsetupState |
              PostFundsetupState |
              GameState |
              ConcludeState

struct *State contains
   // attributes shared by all states
   channel: Channel
   turnNum: Uint
   resolution: Outcome

   // state-type specific properties specified in Section 3.2
   // ...  omitted ...

end

State::nParticipants: s ↦ s.channel.participants.length
State::mover: s ↦ s.channel.participants[s.turnNum%s.nParticipants]
State::channelId: s ↦ s.channel.id
```

The `turnNum` introduces an ordering on the states. As explained in the following

[3] When we say a player "makes a move" or "signs a state", we implicitly mean that the player broadcasts the move, or signed state, to all other players.

section, the game rules specify that the `turnNum` must increase as the game progresses.

The `resolution` specifies the `Outcome` that would occur if the game were to end in this state. Determining the resolution in each state is an important part of game design.

Note that the definition of `State::mover` introduces an important design decision of the force-move game framework: that *the mover is fully determined by the turn number*. Informally, using the fact that s.`turnNum` must be incremented by 1, this rule states that *players must take turns in a cyclical order*.

States also have `nParticipants` and `channelId` attributes inherited from their members.

Beyond these shared attributes, different states in the same game can and will contain different sets of attributes. The different state types are covered in more detail in Section 3.2

3.1.4 Valid moves and transitions

For a move to be valid, the following conditions must hold:

Framework specification – cont'd

```
function validMove(fromMove: Move, toMove: Move)
    returns (Boolean)

    Require(validTransition(fromMove.state, toMove.state))
    Require(toMove.state.mover == toMove.signer)

    return true
end function
```

The first of these rules ensures that a valid move requires a valid state transition. The second says that the move must be signed by the moving player as determined from the state's `turnNum`.

The `validTransition` function consists of some universal rules, as well as some rules that are dependent on the types of states that are involved.

Framework specification – cont'd

```
function validTransition(fromState: State, toState: State)
  returns (Boolean)

  Require(toState.channelId == fromState.channelId)
  Require(toState.turnNum == fromState.turnNum + 1)

  // state-type specific logic specified in Section 3.2
  // ... omitted ...

end function
```

The first of these two rules specifies that no details of the channel can change within a game: the `channelType`, `channelId`, or `participants` must all remain the same.

The second states that the turn number must increment. The `turnNum` therefore introduces an ordering on the set of moves, where moves with a higher `turnNum` are recognised as later than states with a lower `turnNum`.

3.1.5 Alternative moves

Nothing in the conditions prevents a player from signing multiple valid moves with the same `turnNum`. In general, it is impossible to prevent a player from signing and transmitting more than one move if they choose to; it is therefore important to be explicit about how to handle this situation in the protocol.

In a force-move game, if there exist multiple valid moves in the game with the same turn number, we refer to these moves as *alternative moves*. In providing alternative moves, the player gives the next player the right to choose the move they want to progress from. A player therefore theoretically does not gain anything by making multiple moves, though the ability to make multiple moves can be useful in practice in the design of some games (see Fig. 3).

We refer to the set of moves with a single `turnNum` as a *turn*.

3.2 Game mechanics

In this section, we look in more detail at the different types of state, and the valid transitions between them. it is worth emphasising that everything in this section still assumes collaborative behaviour, where the game progresses off-chain; the non-collaborative, on-chain dispute process will be covered in Section 4.

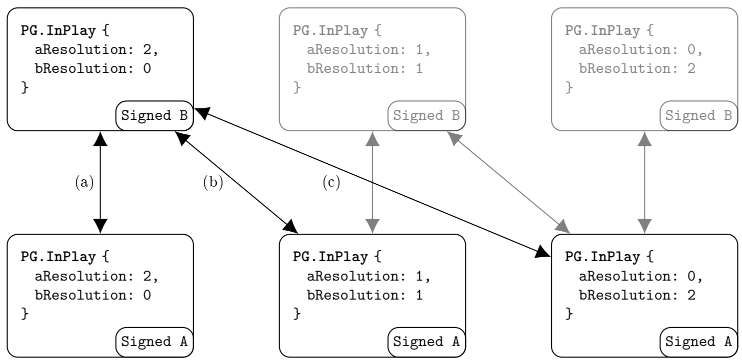


Figure 3: Payment channel with alternative moves. One example of a game where alternative moves can be useful is the payment channel. In a payment channel, it is very easy to reason about which of set of alternative moves will be accepted: we can assume that the opponent will always accept the move which leads to the biggest increase in their total, which is easy to assess when only one currency is involved. In this case, we can make the exchange more efficient by exploiting the ability to make alternative moves. For example if the game was in the $\{A : 2, B : 0\}$ with A to move, A could move to $\{A : 1, B : 1\}$ and then later move to $\{A : 0, B : 2\}$ without having to wait for B to counter-sign the $\{A : 1, B : 1\}$.

3.2.1 Game overview

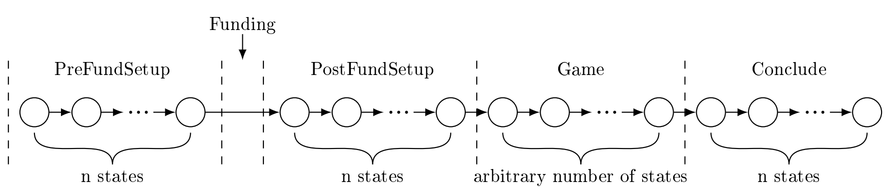


Figure 4: Overview of the stages of collaborative play in the "happy path" case. Note that the allowed transitions from `PRE/POSTFUNDSETUP` $\mapsto$ `CONCLUDE` are omitted from the diagram.

A game begins when a player, known as the *starting player*, broadcasts a `PREFUNDSETUP` state to the desired participants of the game. Each player follows in turn by signing a transition to the subsequent `PREFUNDSETUP` state, until n = `numberOfParticipants` states have been signed, completing the `PREFUNDSETUP` round.

Loosely speaking, the `PREFUNDSETUP` phase is the opening handshake that establishes that all players want to start a particular game, with a specified amount of funds and a specified starting position. By the end of the `PREFUNDSETUP` phase, every player should hold a set of n signed `PREFUNDSETUP` states – one for each player in the game. This is exactly what they need to launch a challenge on-chain to recover their funds if one of their opponents stall (see Section 4.3 for further details). Thus the `PREFUNDSETUP` rounds provides each player with the guarantees they need to be able to safely commit funds to the game.

Game funding is decoupled from the game mechanics, and will be discussed in more detail in Section 5. For the purposes of this section, you can assume that the funding step involves each player making an on-chain transaction into an on-chain *adjudicator* contract, in a pre-defined order, proceeding only when they have verified that all players before them have deposited.

After the game has been funded, the starting player starts the `POSTFUNDSETUP` round, by signing the first `POSTFUNDSETUP` state. The other players respond by signing their own `POSTFUNDSETUP` states in order, thus completing the `POSTFUND-SETUP` round. In signing their `POSTFUNDSETUP` state, each player is stating that sufficient funds have been deposited to start the game. If this is not the case, the player can instead sign a `CONCLUDE` state, indicating that they no longer wish to participate in the game. We discuss this further in Section 3.2.6.

On the starting player's third turn, they must transition from an `POSTFUNDSETUP` state to a `GAME` state. Players may then continue to move to `GAME` states until the game is complete, which in general takes an arbitrary number of turns. When someone wishes to gracefully end the game, they move to a `CONCLUDE` state.

Once a player has moved into the conclude mode, each remaining player signs a `CONCLUDE` state. After each player has signed such a `CONCLUDE` state, the game is considered finished, and any player can form a *conclusion proof*. The conclusion proof may be registered on the adjudicator, as described in Section 4.4.4.

The `validTransition` function is used to enforce the structure defined above. To show how, we will now provide the complete specification of the `valid-Transition` function, which was first introduced in Section 3.1:

Framework specification – cont'd

```
function validTransition(s₁: State, s₂: State)
  returns (Boolean)
  Require(s₂.channelId == s₁.channelId)
  Require(s₂.turnNum == s₁.turnNum + 1)

  if s₁.stateType == PREFUNDSETUP then
    return validTransitionFromPrefundsetup(s₁, s₂)
  else if s₁.stateType == POSTFUNDSETUP then
    return validTransitionFromPostfundsetup(s₁, s₂)
  else if s₁.stateType == GAME then
    return validTransitionFromGame(s₁, s₂)
  else if s₁.stateType == CONCLUDE then
    return validTransitionFromConclude(s₁, s₂)
  end if
end function
```

In the rest of this section, we will dig into each phase of the game in more detail. We will abandon the chronological order as used above, starting first with the `GAME` states, as these are the most interesting part of the framework. We will then proceed with the `CONCLUDE` states, before finishing with the `PRE/POSTFUNDSETUP` states, which are the most technical.

3.2.2 The `GAME` stage

The `GAME` states have the following properties:

Framework specification – cont'd

```
struct GameState contains
  channel: Channel
  turnNum: Uint
  resolution: Outcome
  gameAttributes: Byte[]
end
GameState::stateType: s ↦ Game
```

In addition to the universal `channelId`, `turnNum` and `resolution` fields, the `GAME` states contain a `gameAttributes` field. The `gameAttributes`, along with their encoding into a `Byte[]` array, which must be specified by the *GameLibrary*.

The Game Library is an on-chain contract, which specifies the application-specific attributes and logic for a force-move game. The Game Library need only be deployed once, and the address of the deployed contract can then be used to uniquely define the game being played.

In order to conform to the framework interface, the Game Library needs to implement a single function the `GameLibrary.validTransition` function:

Framework specification – cont'd

```
function GameLibrary.validTransition(s1: State, s2: State)
    returns (Boolean)
    // written by the application developer
end function
```

As one of the simplifications in this framework, we enforce that the `GameLibrary.validTransition` function **must be pure** – it can only depend on the properties of states passed in, and cannot read from or write to the blockchain.

The `GameLibrary.validTransition` function is used by the framework's `validTransition` function as follows:

Framework specification – cont'd

```
function validTransitionFromGame(s1: State, s2: State)
    returns (Boolean)
    if s2.stateType == GAME then
        Require(GameLibrary.validTransition(s1, s2))
    else
        Require(s2.stateType == CONCLUDE)
        Require(s2.resolution == s1.resolution)
    end if
    return true
end function
```

Once the starting player has moved to the `GAME` phase, players may subsequently make `GAME` moves according to the rules specified by the game developer in `GameLibrary.validTransition`.

From each `GAME` state, s, we also allow a transition to a `CONCLUDE` state, s', where s.`resolution` $==$ s'.`resolution`. This is a pragmatic decision: if a player wanted to end the game on their turn, they always have the option of stalling, forcing another player to play a force-move and ultimately ending the game on-chain, resulting in an outcome of the the current state's `resolution`. Given that this possibility always exists, it is reasonable to give the player a way to accomplish the same outcome collaboratively, off-chain,saving the time and expense of an on-chain challenge.

3.2.3 The CONCLUDE stage

The `CONCLUDE` states are the simplest of all states in the framework, containing nothing beyond the universal properties:

Framework specification – cont'd

```
struct ConcludeState contains
    channel: Channel
    turnNum: Uint
    resolution: Outcome
end
ConcludeState::stateType: s ↦ Conclude
```

The `CONCLUDE` states are governed by the following transition rules:

Framework specification – cont'd

```
function validTransitionFromConclude(s1: State, s2: State)
    returns (Boolean)
    Require(s2.stateType == CONCLUDE)
    Require(s2.resolution == s1.resolution)
    return true
end function
```

Once the game is a `CONCLUDE` state s, players may only move to another `CONCLUDE` state s'.

The purpose of the `CONCLUDE` states are to construct a conclusion proof – a statement by all players of the game that the game is over. Conclusion proofs can be registered on-chain, as described in Section 4.4.4.

A (valid) conclusion proof is a sequence of n valid, signed `CONCLUDE` states. The validity can be checked with the `validConclusionProof` function.

Framework specification – cont'd

```
struct ConclusionProof contains
   moves: Move[]
end

function validConclusionProof(proof: ConclusionProof)
   returns (Boolean)
   moves := proof.moves
   firstMove := moves[0]
   Require(firstMove.state.stateType == CONCLUDE)
   n := firstMove.state.nParticipants
   Require(n == moves.length)
   for k in 0...n − 2 do
      Require(validMove(moves[k], moves[k+1]))
   end for
   return true
end function
```

Because the only valid transition from a `CONCLUDE` state is to another `CONCLUDE` state, the two checks taken together ensure we have a sequence of n consecutive `CONCLUDE` states and therefore a `ConclusionProof`.

In terms of the overall game, once a player has moved to a conclusion state, they should behave as though the game could conclude at any point, as they no longer have the ability to prevent a conclusion proof from being created.

3.2.4 The `PREFUNDSETUP` stage

States of type `PREFUNDSETUP` serve as agreements about how the game should begin.

Framework specification – cont'd

```
struct PreFundsetupState contains
   channel: Channel
   turnNum: Uint
   position: Position
   stateCount: Uint
   resolution: Outcome
end
PreFundsetupState::stateType: s ↦ PREFUNDSETUP
```

The `PREFUNDSETUP` states store two special attributes, which we call the *initial conditions* of the game:

1. `gameAttributes` – the proposed initial position of the game.
2. `resolution` – the proposed buy-ins, specifying how much each player should deposit in the game's adjudicator.

The transition rules governing these states are as follows:

Framework specification – cont'd

```
function validTransitionFromPreFundSetup(s1: State, s2: State)
   returns (Boolean)
   Require(s2.resolution == s1.resolution)
   if s2.stateType == CONCLUDE then
      return true
   end if

   if s1.stateCount == s1.nParticipants - 1 then
      Require(s2.stateType == Accept)
      Require(s2.stateCount == 0)
      Require(s2.gameAttributes == s1.gameAttributes)
   else
      Require(s2.stateType == PREFUNDSETUP)
      Require(s2.gameAttributes == s1.gameAttributes);
      Require(s2.stateCount == s1.stateCount + 1);
   end if
   return true
end function
```

The starting player decides on some initial game attributes a_0 and a `resolution` r_0. Each player's `PREFUNDSETUP` state must have the same game attributes a_0 and `resolution` r_0.

`PREFUNDSETUP` states also have a `stateCount` attribute, which serves as a counter to ensure that we get exactly n `PREFUNDSETUP` states, in the `PREFUNDSETUP` round. The starting state s_0 must have s_0.`stateCount` $==$ 0. A transition $s \mapsto s'$ between two `PREFUNDSETUP` states is only valid if it increments `stateCount` by 1.

For a `PREFUNDSETUP` state s, if s.`stateCount` $==$ s.`nParticipants - 1`, then it is the starting player's turn again. As they have already agreed to the initial conditions by proposing the game, they must move to a state state s' of type `POSTFUNDSETUP`. `POSTFUNDSETUP` states have the same attributes as `PREFUNDSETUP` states. The state count must be reset to 0, and the initial conditions must match s.

3.2.5 The POSTFUNDSETUP stage

The POSTFUNDSETUP states have similar attributes and transition rules to the PREFUNDSETUP states:

Framework specification – cont'd

```
struct PreFundsetupState contains
    channel: Channel
    turnNum: Uint
    position: Position
    stateCount: Uint
    resolution: Outcome
end
PreFundsetupState::stateType: s ↦ PREFUNDSETUP
```

Framework specification – cont'd

```
function validTransitionFromPostfundsetup(s1: State, s2:
State)
    returns (Boolean)

    if s2.stateType == CONCLUDE then
        Require(s2.resolution == s1.resolution)
    else if s1.stateCount == s1.nParticipants - 1 then
        Require(s2.stateType == GAME)
        Require(GameLibrary.validTransition(s1, s2))
    else
        Require(s2.stateType == POSTFUNDSETUP)
        Require(s2.resolution == s1.resolution)
        Require(s2.stateCount == s1.stateCount + 1)
        Require(s2.gameAttributes == s1.gameAttributes)
    end if
    return true
end function
```

By moving to an POSTFUNDSETUP state, a player is stating that the funds are now present and so they are happy to proceed with the game with the specified conditions.

As per Section 3.2.4, on their second turn, the starting player may transition to an POSTFUNDSETUP state with the same initial conditions

In this case, each player follows, signing their own POSTFUNDSETUP state s' with the same initial conditions. They must also increment the state count:

Once s.`stateCount` == s.`nParticipants`, then it is the starting player's third turn. As they have already committed to beginning the game under the agreed upon conditions, the only valid transition is to a `GAME` state with `gameAttributes` set to be a_0. The `GameLibraray.validTransition` function is used to ensure that the first game move represents a valid transition from the pre-agreed starting state.

3.2.6 Backing out

To prevent players from being forced into unwanted positions, they always have the option to transition to a conclude state before the `GAME` phase begins.

From a `PREFUNDSETUP` state, a rational player would conclude the game rather than moving to a `PREFUNDSETUP` state if they do not wish to begin a game with the specified initial conditions. This may be the case, for example, if the starting game attributes a_0 would put the starting player at an unfair advantage. At this point, players could also simply ignore the game, as they haven't yet staked any funding on the game.[4]

After the `PREFUNDSETUP` phase has been completed, players are intended to fund the game. It is only safe to do so in order – this ensures that they are paid out their deposits in case of an aborted game. The starting player should move to the `POSTFUNDSETUP` phase only when they see that the game is properly funded. [5]

In the case where some players move through the `POSTFUNDSETUP` phase without having funded the game, a later player may back out by moving from `POSTFUNDSETUP` to `CONCLUDE`, rather than committing to the game.

If a player transitions from either a `PREFUNDSETUP` or `POSTFUNDSETUP` state, s, to a `CONCLUDE` state, s', we require the resolution to match.

4 Force-move games when not collaborating

At this point, we have described how the participants advance the state by exchanging signed moves off-chain. In this section, we cover what happens if this cooperative behaviour breaks down. In particular, we detail how participants can dispute on the blockchain in order to break a deadlock between them and either progress the game or terminate it fairly.

[4] No rational player would fund the game until everyone has agreed to the proposed initial conditions. To do so requires an on-chain transaction, which may be wasted in the case that a later player refuses to play the game.

[5] In a two player game, if player 1 moves to the `POSTFUNDSETUP` phase before seeing that player 2 has deposited the proper funds, player 2 can continue to play without staking any funds, with nothing to lose.

4.1 Modes

Before proceeding to explain the force-move, it is useful to step back and look at the high-level picture of the progression of a force-move game. As a force-move game progresses, it passes through several macroscopic states, which we will refer to as *modes* or *modes of play*.

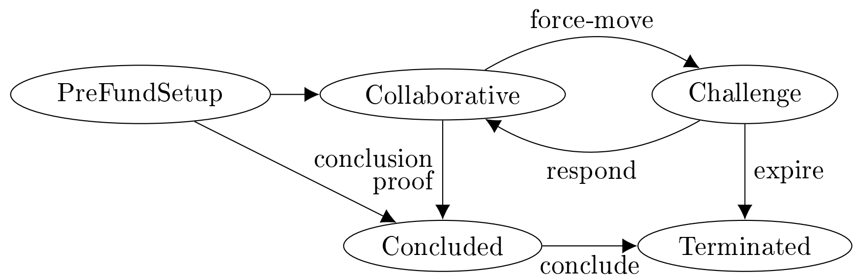


Figure 5: Modes of play.

The **PreFundSetup** mode corresponds to the `PREFUNDSETUP` stage described in Section 3.2.4. The reason we separate this out as its own mode is that before the `PREFUNDSETUP` round is complete, it is not possible for any player to play a force-move. Once the last `PREFUNDSETUP` state has been signed and distributed, the game enters the **Collaborative** mode.

In the **Collaborative** mode, the game progresses off-chain with the exchange of signed moves between the participants (see Section 3.1.3). This mode includes all the other stages (`POSTFUNDSETUP`, `GAME`, `CONCLUDE`) covered previously. The **Concluded** mode is reached when enough `CONCLUDE` moves exist to construct a conclusion proof (Section 3.2.3).

The game enters the **Challenge** mode when a player triggers the force-move operation on-chain (Section 4.3). When this happens, a game timeout is set for an on-chain response to the force-move. If no response is received within the timeout period, the game transitions into the **Terminated** mode. In this mode, no further play is possible, but the players can reclaim their funds according to the outcome of the game.

In order to prevent the game from being terminated, some party must respond to the force-move on-chain before the game timeout passes. If a valid response is received, the game returns to the **Collaborative** mode and progress can continue off-chain. We introduce the force-move operation and the ways to respond to it in section 4.4.

4.2 The adjudicator

The purpose of the adjudicator is to both hold the players' funds in escrow throughout the game and to manage the dispute process. The adjudicator will typically be a smart contract stored on a blockchain, though we do not make this a requirement.[6]

In addition to holding funds in escrow and allowing players to collaboratively conclude a game, the adjudicator must implement Interface 3.

Interface 3 Framework specification – noncollaborative interface

```
function forceMove(moves: Move[])
function refute(m: Move)
function respondWithMove(m: Move)
function alternativeRespondWithMove(moves: Move[])
function withdraw(channelId: Byte[], player: Address)

struct Challenge contains
   endState: State
   endTime: Uint
end
Challenge::inProgress: c ↦ c.endTime > now()
Challenge::terminated: c ↦ c.endTime > 0 && !c.inProgress

// The following functions are implementation dependent and not
// specified by the framework
function setChallenge(state: State, endTime: Uint)
function getChallenge(channelId: Byte[]) returns (Challenge)
function getCurrentOutcome(channelId: Byte[]) returns (State)
function cancelChallenge(channelId: Byte[])
function challengeInProgress(channelId: Byte[]) returns
(Boolean)
function isTerminated(channelId: Byte[]) returns (Boolean)
```

The next few sections will explain the methods in this interface.

4.3 Playing the Force-Move

The force-move is a mechanism to handle an unresponsive opponent. From the blockchain's perspective, the force-move operation will either result in an advance to the state of the game or in the termination of the game. In the latter case, each participant is allowed to reclaim the "fair" proportion of their funds.

[6] For instance, an adjudicator may be counterfactually instantiated in the case of a dispute.

In what follows we will refer to the players in the context of the force-move: the *challenger* is the participant who submits the force-move, the *challengee* is the participant whose turn it is next, according to the state stored on the challenge. Note that there's nothing to prevent the challenger and challengee from being the same player, though it would not ordinarily be in the player's interest to do this.

Informally, the force-move operation represents the challenger laying down a challenge to the challengee. For a 2-player force-move game this would be along the lines of:

Challenger: You moved m_t for turn t; I moved to m_{t+1} for turn $t+1$; now you need to provide your move for turn $t+2$.

The challenger triggers the force-move by calling the `forceMove` method on the adjudicator.

Framework specification – cont'd

```
function Adjudicator.forceMove(moves: Move[])
    firstMove := moves[0]
    channelId := firstMove.channelId
    n := moves.length
    Require(n == firstMove.state.nParticipants)
    Require(!Adjudicator.challengeInProgress(channelId))
    Require(!Adjudicator.isTerminated(channelId))
    for k in 0...n − 2 do
        Require(validMove(moves[k], moves[k + 1]))
    end for

    c := moves[n − 1]

    Adjudicator.setChallenge(c, now() + defaultExpirationTime)
end function
```

If all of the checks pass, the adjudicator will have a challenge registered with the challenge state and the end time set. The game will have transitioned to the **Challenge** mode.

4.4 Responding to a Force-Move

In order to prevent a game from terminating, the opponent must respond to the force-move operation before the game times out. There are four ways to respond to a force-move:

- Refute the force-move.

- Respond with a move.
- Respond with a move from an alternative state.
- Register a conclusion proof.

We call the party who responds to the force-move the *responder*. This will typically be the challengee, but it does not have to be.

In what follows, is important to note that each of these responses lead to an increase in the `turnNum` from the challenge state stored in the adjudicator. This ensures that the force-move will always advance the game from the blockchain's perspective. In combination with extensions discussed in Section 4.5, this prevents the game from stalling indefinitely.

4.4.1 Refuting a Force-Move

Refuting is the action a responder takes when the challenger launched a force-move from an outdated state. To refute a force-move the responder must demonstrate that the state the challenger provided was not the challenger's latest state – in other words, they must present a state with a higher state nonce that was signed by the challenger. Informally:

Challenger: You moved m_t for turn t; I moved to m_{t+1} for turn $t+1$; now you need to provide your move for turn $t+2$.

Responder: That was a long time ago though. You've since moved to $m_{t'}$ for turn $t' > t$.

The responder refutes a force-move, rendering it cancelled, by calling the `refute` method on the adjudicator.

Framework specification – cont'd

```
function Adjudicator.refute(refutation: Move)
    channelId := refutation.channelId
    Require(Adjudicator.challengeInProgress(channelId))

    challengeState := Adjudicator.getChallenge(channelId).state
    Require(refutation.stateNonce > challengeState.stateNonce)
    Require(refutation.signedBy(challengeState.mover))

    Adjudicator.cancelChallenge(channelId)
end function
```

This case is interesting because, as explained in more detail in Section 4.5, it is the one case where it is potentially possible to identify that the challenger was acting in bad faith and punish them for their behaviour.

4.4.2 Responding with a Move

Responding with a move is arguably the action that the challenger was hoping for when they played the force-move – their opponent was unresponsive, and by playing the force-move, they have spurred them into action. To respond with a move, the responder must provide a signed state that represents a valid transition from the challenge state. Informally:

Challenger: You moved m_t for turn t; I moved to m_{t+1} for turn $t+1$; now you need to provide your move for turn $t+2$.

Responder: Ok. I'm going to move m_{t+2} for turn $t+2$.

In order to perform this action, the responder must make an on-chain transaction to call the `respondWithMove` method on the adjudicator.

Framework specification – cont'd

```
function Adjudicator.respondWithMove(response: Move)
    channelId := response.channelId
    Require(Adjudicator.challengeInProgress(channelId))

    challengeState := Adjudicator.getChallenge(channelId).state
    Require(response.signedBy(response.mover))
    Require(validTransition(challengeState, response.state))

    Adjudicator.cancelChallenge(channelId)
end function
```

In responding with a move, the responder provides the exact signed state required to progress the game. This allows the participants to continue the rest of the game cooperatively off-chain.

4.4.3 Responding with an Alternative Move

In this response the responder provides an alternative valid sequence of n moves, where the penultimate move has the same `turnNum` as the `challengeMove`. Informally:

Challenger: You moved m_t for turn t; I moved to m_{t+1} for turn $t+1$; now you need to provide your move for turn $t+2$.

Responder: But you also moved to m'_{t+1} at turn $t+1$. I'm choosing to move on from there instead.

In order to perform this action, the responder must make an on-chain transaction to call the `alternativeRespondWithMove` method on the adjudicator.

Framework specification – cont'd

```
function Adjudicator.alternativeRespondWithMove(moves: Move[])
    channelId := moves[0].channelId
    Require(Adjudicator.challengeInProgress(channelId))

    challengeState := Adjudicator.getChallenge(channelId).state
    n := moves.length

    Require(n == challengeState.nParticipants)
    for k in 0...n − 2 do
        Require(validMove(moves[k], moves[k + 1]))
    end for
    Require(moves[n − 2].turnNum == challengeState.turnNum)

    Adjudicator.cancelChallenge(channelId)
end function
```

Note that allowing the `alternativeRespondWithMove` action is an unavoidable consequence of the rule that a player has the freedom to choose their preferred move if there are multiple moves available to them.

4.4.4 Registering a Conclusion Proof

If the game has not been abandoned, and has ended collaboratively (i.e. a conclusion proof exists), any player may use a conclusion proof to counteract a force-move.

Registering a conclusion proof is done by calling the framework's `conclude` method, which marks the channel's game as concluded.

Framework specification – cont'd

```
function conclude(proof: ConclusionProof)
    endState := proof.moves[0].state
    Require(!Adjudicator.isTerminated(endState.channelId))
    Require(validConclusionProof(proof))

    Adjudicator.setChallenge(endState, now())
end function
```

Note that registering the conclusion proof causes an already-expired challenge to be stored in the adjudicator – after the fact, a game concluded with a conclusion proof behaves exactly the same as one that was terminated by an expired challenge.

4.4.5 Failing to notice a Force-Move

The playing of a force-move and immediate transition to **Challenge** mode occurs externally to the channel where the players are exchanging states. It is therefore quite possible that, for example, Alice does not immediately realise that the game is in **Challenge** mode and continues to play moves, as if they were in **Collaborative** mode.

Thankfully, as long as Alice notices Bob's challenge within the timeout period, she cannot hurt her position by playing on as though still in **Collaborative** mode. By playing on, by definition, she has provided at least one move that can cancel the force move via `respondWithMove` or `alternativeRespondWithMove` operations. If Bob also continues to play, Alice would then have the means to cancel the force move via the `refute` operation (and potentially punish Bob, depending on the protocol.) When the force-move is cancelled, all the moves made in **Collaborative** still stand.

At worst, failing to notice a force-move represents a missed opportunity to move on from a state that Alice prefers, in the case that Bob offers multiple moves. This is no different from the off-chain cases, where Alice commits to her move just before an alternative move arrives.

4.5 Extensions to the Force-Move protocol

As described so far, the force-move operation has a number of weaknesses:

- It is possible to grief your opponent, with a ~1:1 griefing factor, by repeatedly playing the same force-move.
- Even without replaying the same force-move, it is possible to grief your opponent, with a ~1:1 griefing factor, by playing the force-move with a sequence of old moves.
- It is possible for anyone who holds n consecutive states (e.g. a witness) to grief the players by playing the force-move operation.

These weaknesses can be mitigated with the following two extensions:

- At the time of response, store the `turnNum` of the response state in the adjudicator. Only allow new force-moves if their response will increase this stored `turnNum`.
- Add a `certificate` argument to the force-move operation, to identify the challenger and prove that they intended to play that force-move. Require that the challenger is one of the participants.

In practice, the `certificate` could be the challenger's signature of something like

$$\texttt{keccak256("forceMove:",moves)}. \tag{4}$$

A possible alternative here would be to ensure that the caller of the `forceMove` method is the challenger, but this rules out the likely-common scenario where the players use an ephemeral set of keys (without funds for gas costs) for signing state updates.

These extensions should probably be part of the default protocol – we only chose to introduce them separately as they are somewhat orthogonal to the main idea.

It is worth noting that these extensions do not completely eliminate the risk of griefing from the force-move. In fact, this is impossible, as the following three situations are indistinguishable from the perspective of the blockchain:

- A sends m_A to B, B stalls, A calls the force-move on B to advance the game [B at fault]
- A sends m_A to B, B sends m_B to A, A fails to acknowledge m_B and calls the force-move on B to grief them [A at fault]
- A sends m_A to B, B sends m_B but A does not receive it (e.g. due to network issues), A calls the force-move on B to advance the game [no-one at fault]

Adding the `certificate` does allow us to control the griefing factor in the case that the force-move is refuted, by requiring a forfeitable deposit at the time a force-move is made. For example:

- In order to play the force-move, the challenger must provide a deposit
- If the force-move is refuted, the challenger loses the deposit (and/or some of it is transferred to the challengee)
- Otherwise, the deposit is returned to the challenger

This works as the refute case is the one case where it is clear to all that the challenger was acting in bad faith: they either did not submit their latest state, or they continued to play and sign updates as though the game was in **Collaborative** mode.

5 Funding force-move games

So far we have discussed the mechanics of force-move games, including how to split the assets in the case where the game is uncooperatively terminated. In this section we look at how the assets are deposited and assigned to the game in the first place. We call this action *funding* the game.

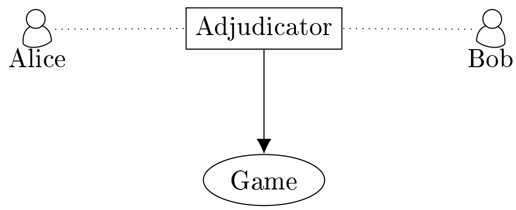


Figure 6: Simple adjudicator.
A simple adjudicator only supports a single instance of a force-move game, between a fixed set of participants.

The force-move game framework has been designed to decouple a game's funding from its execution. Funding happens externally to the channel and there is no way of telling from the properties of the channel how (or even whether) the game is funded. This allows force-move games to run within many different frameworks that are capable of guaranteeing their funds. In this section we will look at some different possibilities for funding force-move games. Their are many other possible approaches beyond those discussed here.

5.1 Non-funded games

The first thing to note is that nothing in the description of force-move games *requires* them to be funded. All the game mechanics work perfectly well even if no assets are allocated to the game. Of course, if the game is not funded, participants will not receive any funds when the game terminates: they are playing for nothing. We also cannot reason about the incentives in the same way.

It is always possible to avoid starting a non-funded force-move game by backing out at the `POSTFUNDSETUP` stage (see Section 3.2.6). We will, therefore, typically assume that any games that get to the `GAME` stage are funded in some way.

5.2 Simple funding

The simplest way to fund a force-move game is to use the *Simple Adjudicator* – an on-chain adjudicator that supports exactly one force-move game. Funds are deposited into an on-chain contract, which stores a single, hard-coded `channelId`, and implements the adjudicator functionality for handling on-chain challenges. Operations fail for all states whose `channelId` differs from that in the adjudicator.

This setup is highly inefficient in terms of minimising on-chain actions and storage, as each new game requires a new on-chain contract to be deployed and

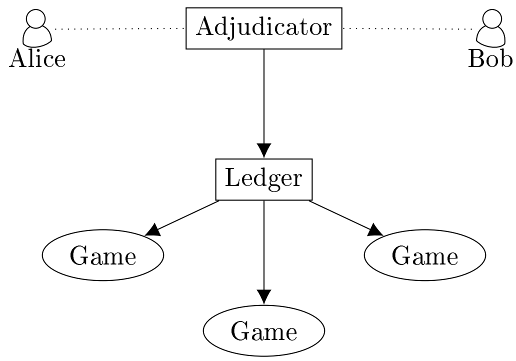


Figure 7: Ledger channel setup
An adjudicator that supports a ledger channel allows players to fund multiple games with a single on-chain deposit. State updates in the ledger channel dictate what proportion of the deposit funds which games.

the funds deposited can only be used for a single game, with a predefined set of participants.

Section 6 details the design for the simple adjudicator.

5.3 Ledger channels

Ideally, a user would prefer to use a single on-chain deposit to fund multiple state channels. A *ledger channel* is a state channel that serves to securely allocate funding from single adjudicator to different games through off-chain agreements.

In order to support funding through ledger channels, the adjudicator must be adapted to interpret the ledger agreements and allow funds to be withdrawn accordingly. This is a subject of current research.

5.4 Virtual channels

Virtual channels are state channels that can be constructed off-chain via existing state channels. This approach is described comprehensively in [11].

In order to support virtual channels in a force-move game setting, the adjudicator would need to be updated so that it understands the virtual channel agreements and allows funds to be withdrawn accordingly. This is another subject of ongoing research.

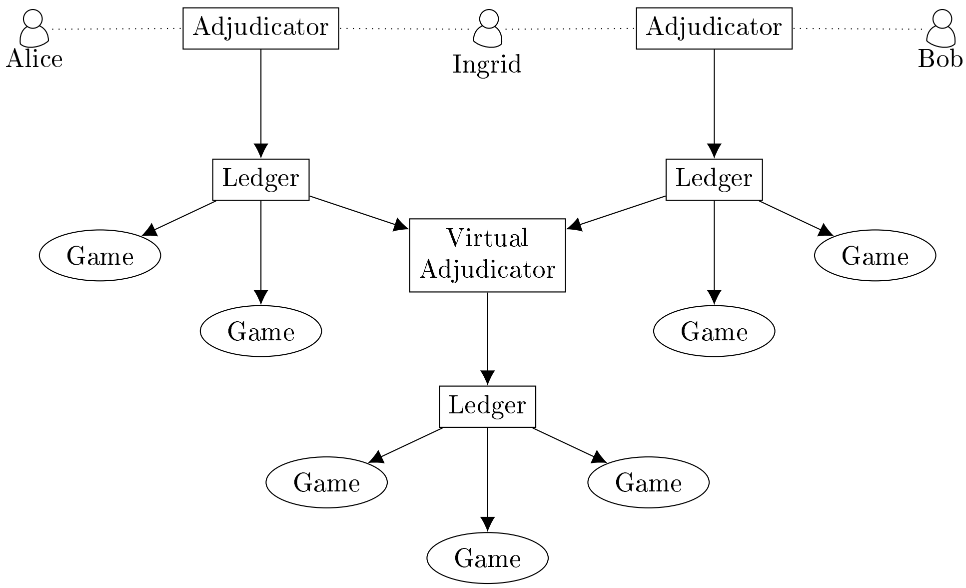


Figure 8: Virtual channels
A virtual state channel between Alice and Bob can be created through state channels backed by an on-chain agreement between Alice and an intermediary Ingrid, and another between Ingrid and Bob.

5.5 Withdrawal

Once the game has ended – either collaboratively via `conclude`, or non-collaboratively via an ignored force-move – a game's adjudicator should release funds for withdrawal.

We leave the implementation of the `withdraw` method up to the framework developer.

Framework specification – cont'd

```
function Adjudicator.withdraw(channelId, playerAddress)
    Require(Adjudicator.isTerminated(channelId))
    // Release mechanism unspecified
end function
```

6 The simple adjudicator

We limit our description of the simple adjudicator to methods and implementation details not covered by Sections 3 and 4. This means looking at the format of the internal storage, the deployment of the adjudicator and depositing of funds, and the withdrawal of funds from the adjudicator.

6.1 Internal storage

The SimpleAdjudicator has the following fields:

- `channelId`
- `currentChallenge`

The `channelId` stores the identity of the channel that the adjudicator supports. The `currentChallenge` variable stores a `Challenge` object, specified in Section 4.2, which is used to determine whether the game is in the **Challenge** or **Terminated** mode.

Specification 4 Simple adjudicator

```
function Adjudicator.getChallenge(channelId: Byte[])
  returns (Challenge)
  Require(channelId == Adjudicator.channelId)
  return Adjudicator.Challenge.state
end function

function Adjudicator.setChallenge(s: State, t: Uint)
  Require(s.channelId == Adjudicator.channelId)
  Adjudicator.currentChallenge := Challenge(s, t)
end function

function Adjudicator.cancelChallenge(channelId: Byte[])
  Require(state.channelId == Adjudicator.channelId)
  c := Adjudicator.currentChallenge
  c.endTime := 0
  Adjudicator.currentChallenge := c
end function

function challengeInProgress(channelId: Byte[])
  returns (Boolean)
  Require(channelId == Adjudicator.channelId)
  return Adjudicator.currentChallenge.inProgress
end function
```

cont'd below

Simple adjudicator – cont'd

```
function isTerminated(channelId: Byte[])
   returns (Boolean)
   Require(state.channelId == Adjudicator.channelId)
   return Adjudicator.currentChallenge.terminated
end function
```

6.2 Deployment and depositing funds

In order to deploy the simple adjudicator and safely deposit their funds the players follow the following protocol:

1. Alice signs $\texttt{PREFUNDSETUP}_0$ and sends it to Bob.
2. Bob signs $\texttt{PREFUNDSETUP}_1$ and sends it to Alice.
3. Alice deploys the simple adjudicator, passing in the `channelId` according to channel specified in $\texttt{PREFUNDSETUP}_0$. She sends the address of the deployed adjudicator to Bob.
4. Alice deposits `aResolution`, as specified by $\texttt{PREFUNDSETUP}_0$.
5. Bob waits until he can verify to an acceptable level of confidence that the adjudicator contains `aResolution`. It is then safe for him to deposit `bResolution`.
6. Alice waits until she can verify to an acceptable level of confidence that the adjudicator contains `aResolution + bResolution`. She then signs $\texttt{POSTFUNDSETUP}_0$.
7. On receiving $\texttt{POSTFUNDSETUP}_0$, Bob replies with $\texttt{POSTFUNDSETUP}_1$

As an optimization, Alice could combine steps 3 and 4, doing her deposit alongside the deployment, and thus saving one on-chain transaction.

It is safe for Alice to deposit at step 4 since at this point she holds $\texttt{PREFUNDSETUP}_0$ and $\texttt{PREFUNDSETUP}_1$. This would allow her to recover her funds in the case where Bob does not deposit and stalls. The resolution at this point would attempt to give `aResolution` to Alice and `bResolution` to Bob, paying out Alice first, allowing her to recover her funds even if Bob does not deposit.

It is safe for Bob to deposit in step 5, as he is holding $\texttt{PREFUNDSETUP}_0$ and $\texttt{PREFUNDSETUP}_1$, which allows him to force-move Alice to move to either a `POSTFUNDSETUP` or a `CONCLUDE` state. If she responds, he could then move to `CONCLUDE`; if not, he can terminate the game at $\texttt{PREFUNDSETUP}_1$. In either of these cases, he can recover the `bResolution` he deposited.

6.3 Withdrawing funds

In the main force-move game specification, we touched on the transition from the **Challenge** mode to **Terminated** but did not talk about how to recover the assets from this states. This is because the method of recovery will depend on the funding mechanism used.

For the simple adjudicator, the `withdraw` method is specified as follows. Its correctness relies on the specification of `Adjudicator.setChallenge` in 4.

Simple adjudicator – cont'd

```
function Adjudicator.withdraw(channelId: Byte[], k: Uint)
   Require(channelId == Adjudicator.channelId)
   Require(Adjudicator.isTerminated(channelId))
   Require( 0 ≤ k && k < n )

   s := Adjudicator.currentChallenge.endState
   w := Adjudicator.withdrawnAmounts
   r := s.resolution
   pending := sum(r[i] - w[i]; i := 0; i < k; i++)
   owed := r[k] - w[k]
   Require(Adjudicator.balance > pending)

   amount := min(owed - w[k], Adjudicator.balance - pending)
   Adjudicator.withdrawnAmounts[k] += amount
   participant = s.participants[k]
   Adjudicator.transfer(participant, amount)
end function
```

Acknowledgements

We would like to thank Erik Bryn, Jeff Coleman, Liam Horne, and Li Xuanji for introducing us to the topic and for numerous formative discussions, and Patrick McCorry and Kristina Hostáková for their discussions and feedback on the paper. We are also very grateful to James Fickel for supporting the project through a personal grant and to L4 for providing follow-on funding.

References

[1] A. Hertig, "How will ethereum scale?." https://www.coindesk.com/information/will-ethereum-scale/.

[2] https://bitinfocharts.com/comparison/bitcoin-transactionfees.html.

[3] https://blockchain.info/charts/avg-confirmation-time.

[4] 2015. https://usa.visa.com/dam/VCOM/download/corporate/media/visa-fact-sheet-Jun2015.pdf.

[5] J. Coleman, L. Horne, and L. Xuanji, "Counterfactual: Generalized state channels." http://l4.ventures/papers/statechannels.pdf, 2018.

[6] T. D. Joseph Poon, "The bitcoin lightning network: Scalable off-chain instant payments." https://lightning.network/lightning-network-paper.pdf, 2016 January.

[7] https://en.bitcoin.it/wiki/Payment_channels.

[8] 2015. https://raiden.network/.

[9] A. Miller, I. Bentov, R. Kumaresan, and P. McCorry, "Sprites: Payment channels that go faster than lightning," *CoRR*, vol. abs/1702.05812, 2017.

[10] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptographic currencies." Cryptology ePrint Archive, Report 2017/635, 2017. https://eprint.iacr.org/2017/635.

[11] S. Dziembowski, S. Faust, and K. Hostakova, "Foundations of state channel networks." Cryptology ePrint Archive, Report 2018/320, 2018. https://eprint.iacr.org/2018/320.

[12] Wikipedia contributors, "Rock-paper-scissors — wikipedia, the free encyclopedia," 2018. [Online; accessed 28-March-2018].

[13] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger byzantium version 71dcbdc," *https://ethereum.github.io/yellowpaper/paper.pdf*, 2018.