Cross-chain Deals and Adversarial Commerce

Keyi Wu, Jingyuan Li

Herlihy, Maurice, Barbara Liskov, and Liuba Shrira

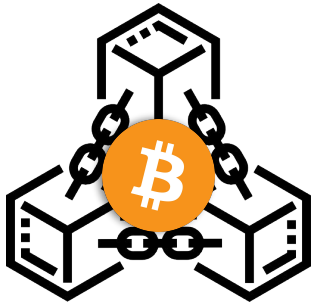
2019 May 23

RoadMap

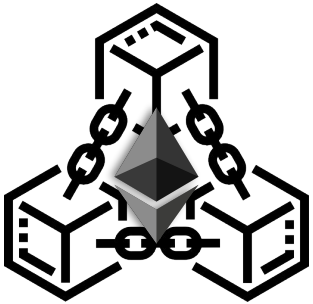
- Introduction
- How It Works
- Two Protocols
- Cost Analysis
- Discussion

Introduction

Imagine a world where everything's ownership is documented in its own blockchain...

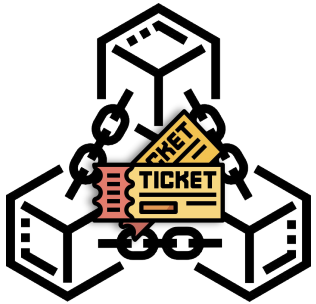


Bitcoin



Ethereum

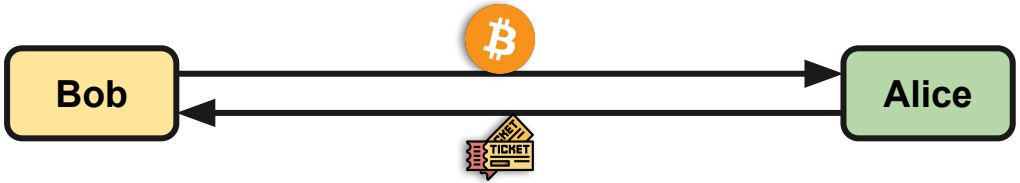
Milk



Film ticket

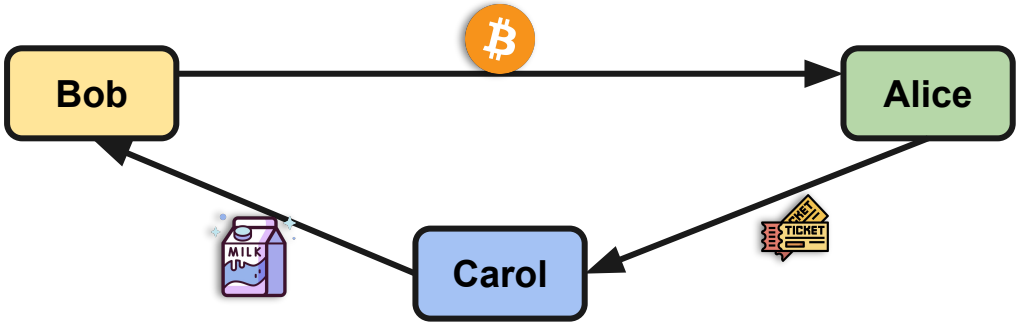
Introduction

How do we make a ***(cross-chain) transaction*** in such a world?



Introduction

How do we make a ***(cross-chain) transaction*** in such a world?



Introduction - Adversarial Commerce in A Blockchain World

- Each party wants to trade asset
- Multi-step transfers are OK
- Each asset lives on its own Blockchain
- No one trust anyone

Introduction - Cross-chain Deal

- A new computational abstraction for structuring complex distributed exchanges in an adversarial setting.
- Multi-step transfers are OK.
- Multi-blockchain transfers are OK.
- There is no size limit of faulty parties, every party can be a faulty party.

Other Approach: Cross-chain Swap

- Existing proposals for cross-chain swaps set up a collection of ***unconditional*** transfers from one party to another.
- Each party checks that it is satisfied with its own transfers in and out, and then when all approve, the transfers take place.

Incapable of expressing many kinds of standard financial transactions: broker, auction...

New Notions of Correctness

- ***Safety***: For every protocol execution, every compliant party ends up with an acceptable payoff.
- ***Weak liveness***: No asset belonging to a compliant party is escrowed forever.
- ***Strong liveness***: If all parties are compliant and willing to accept their proposed payoffs, then all transfers happen.

How It Works - Five Phases

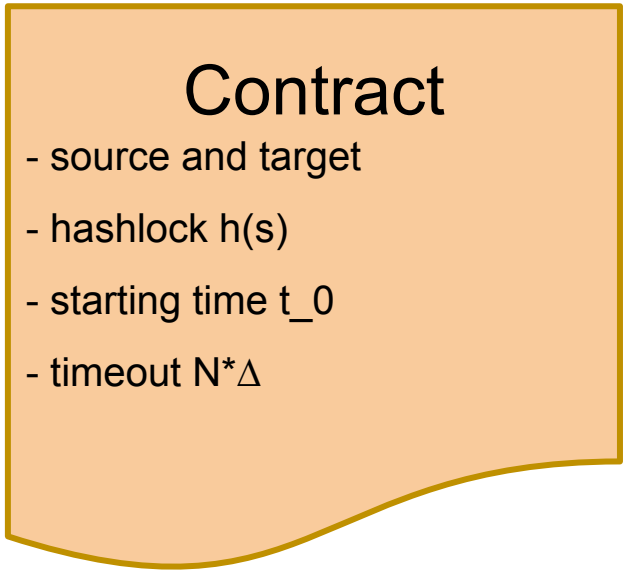
- Clearing
- Escrow
- Transfer
- Validation
- Commit

Two Protocols - Timelock Protocol

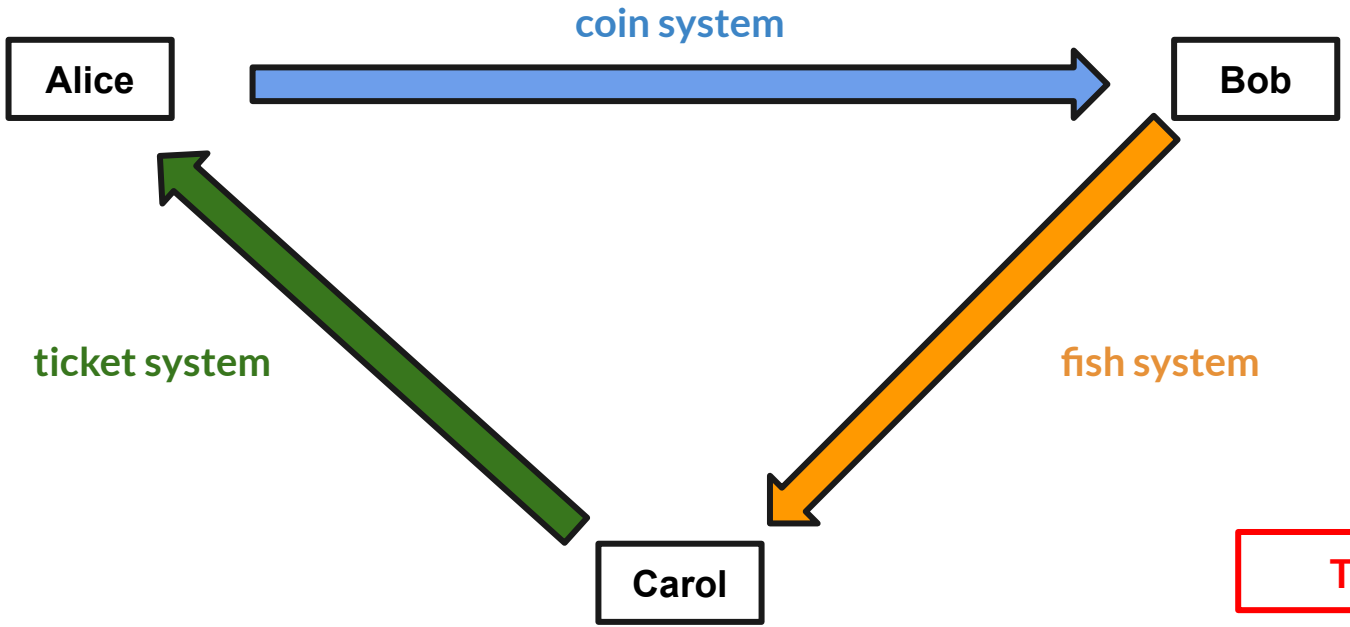
- Synchronous
- Use a timeout: determined by the commit path
- Parties do not explicitly vote to abort. Instead, timeouts are used to ensure that escrowed assets are not locked up forever if some party crashes or walks away from the deal.

Two Protocols - Timelock Protocol

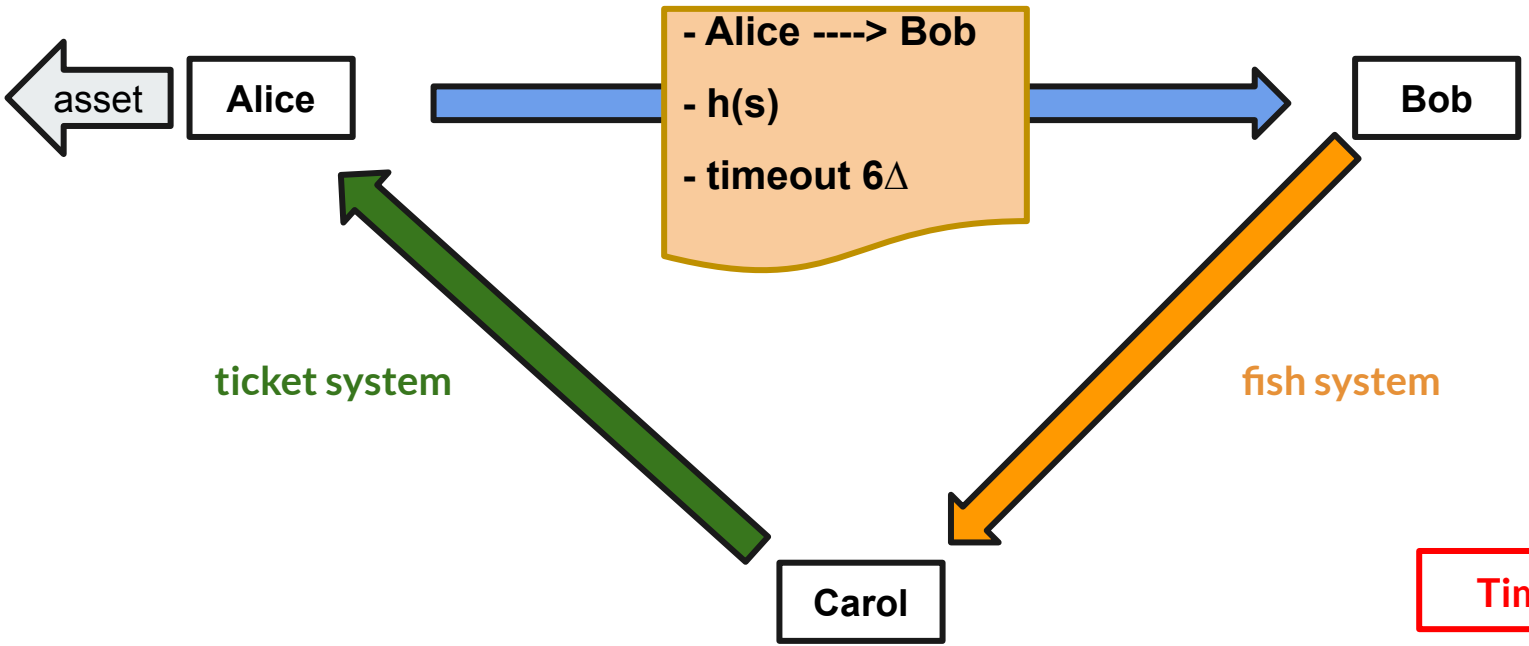
- t_0: starting time of commit phase
- Δ: time to publish and notice contract
- s: secret known by the first party of a commit path



Commit phase

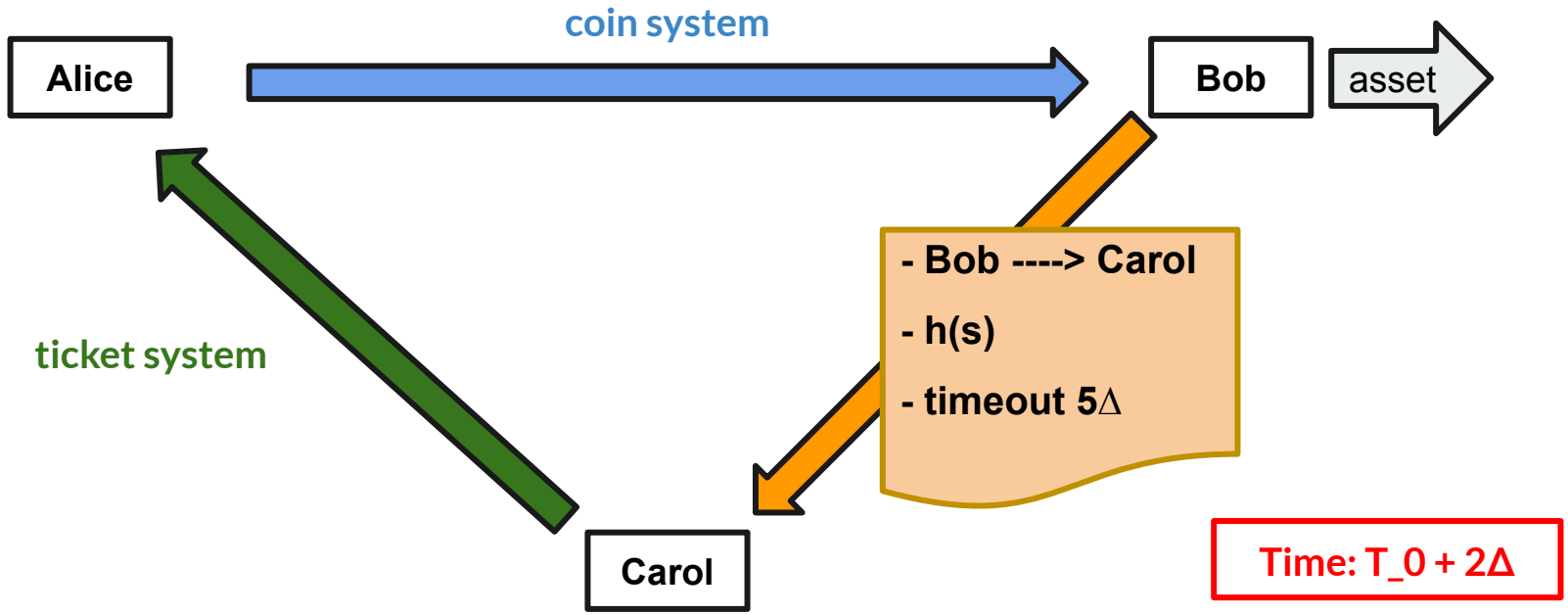


Commit phase - 1

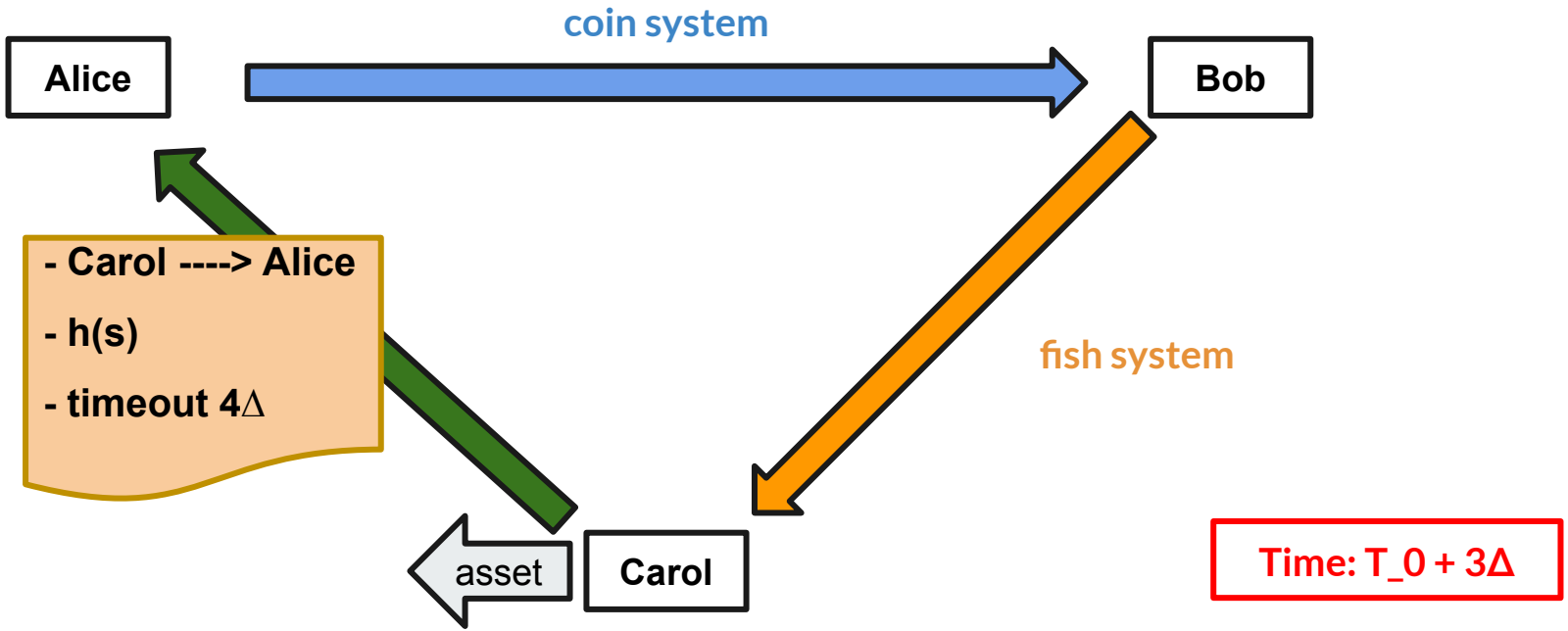


Time: T_0 + Δ

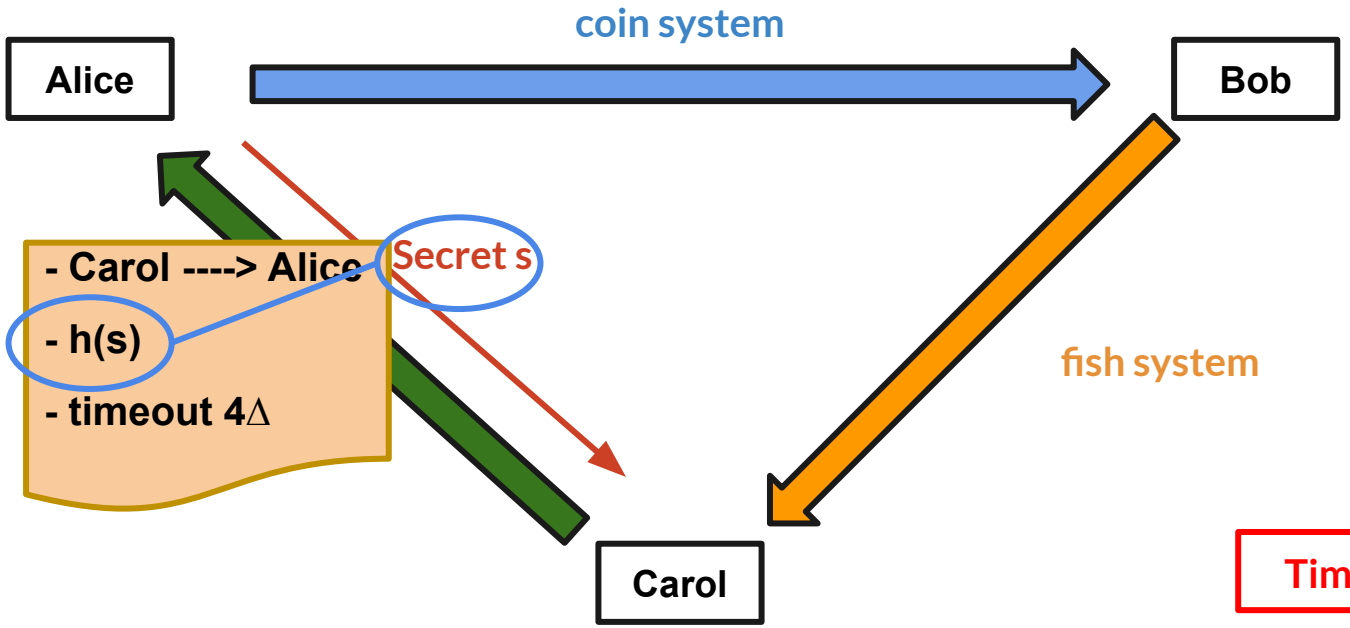
Commit phase - 1



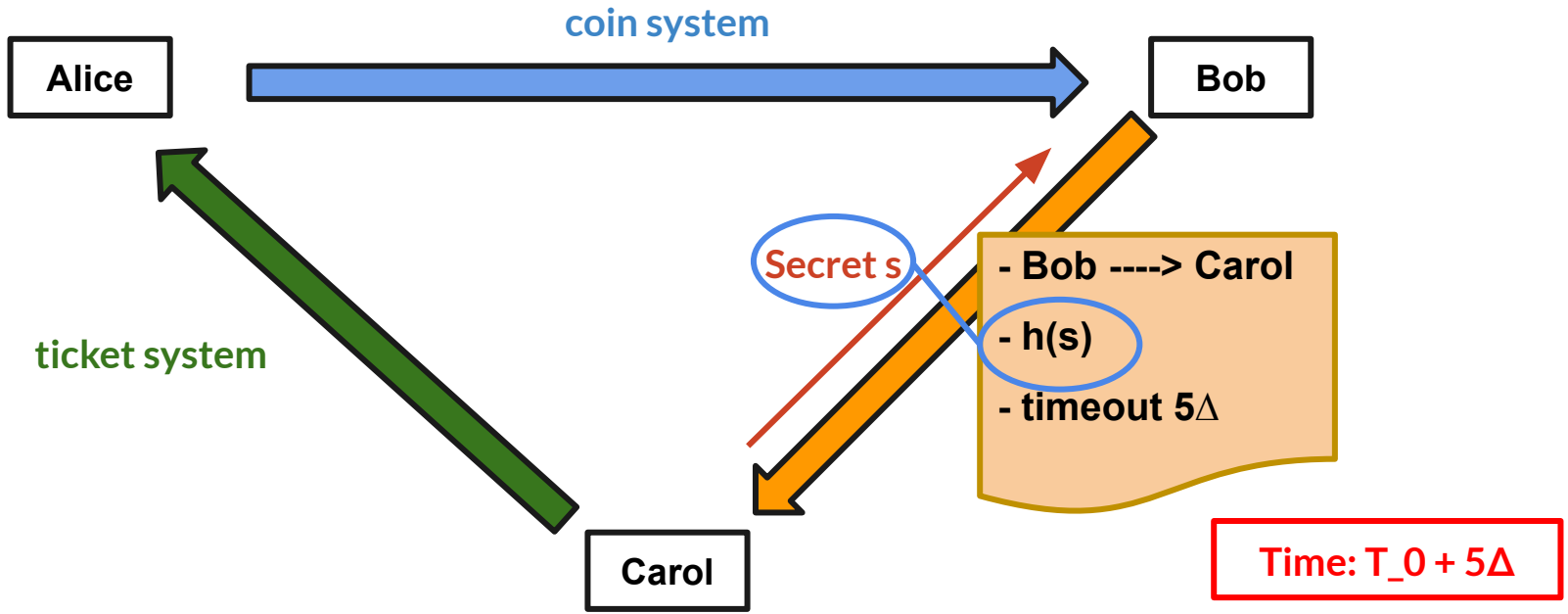
Commit phase - 1



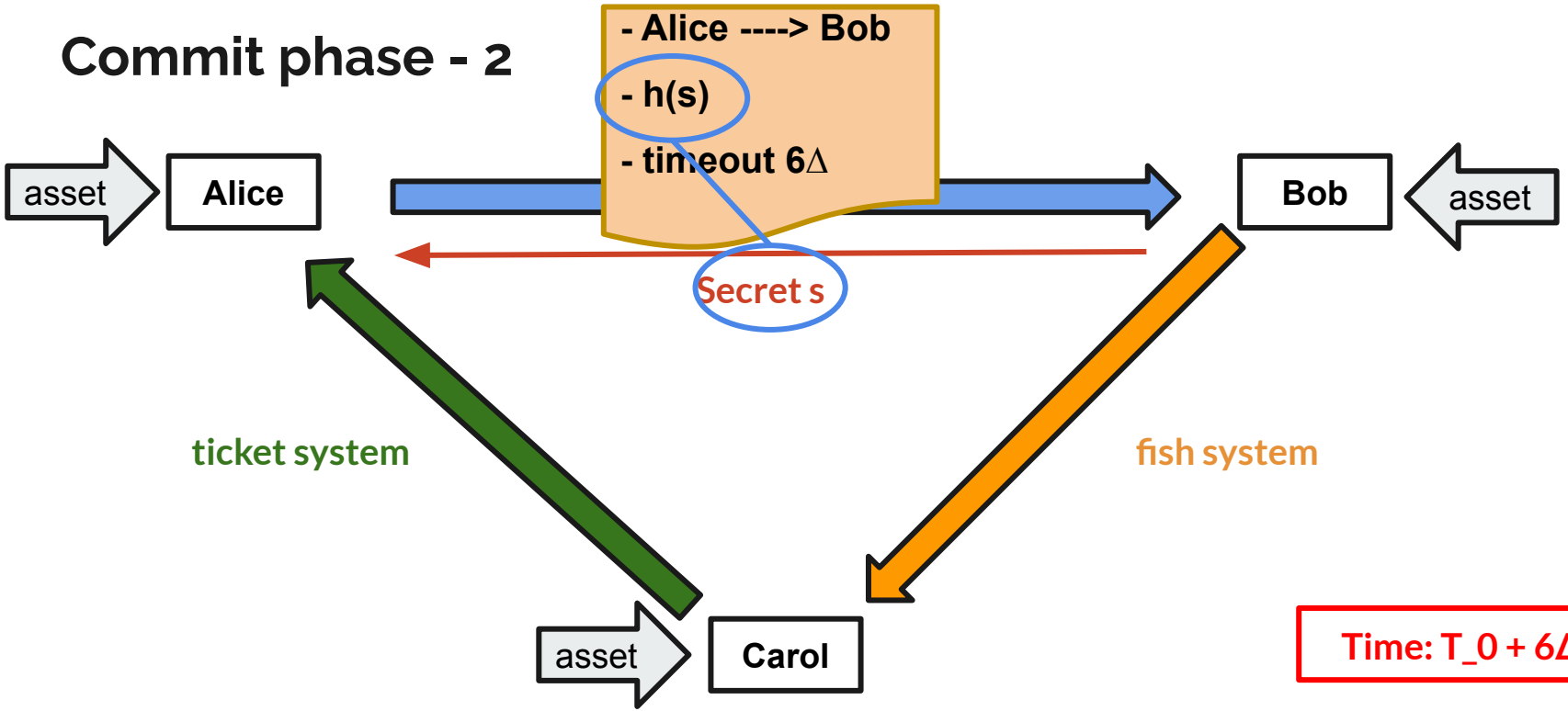
Commit phase - 2



Commit phase - 2



Commit phase - 2



CBC Protocol

- Semi-synchronous
- Certified blockchain, CBC
- Votes on the entire deal
- Extract a proof from the CBC, presents to the contract
- A proof of commit vs A proof of abort

Phases of a CBC protocol

- Clearing Phase: startDeal(D, plist)
- Escrow Phase: escrow(D, plist, h, a, . . .)
- Transfer Phase: transfer (D, a, Q).
- Validation Phase
- Commit Phase: commit(D, h, X) or abort(D, h, X)

Byzantine Fault-Tolerant Consensus

- 3f + 1 validators, at most f malicious
- Each block contains the next block's group of validators
- A certificate containing at least f + 1 validator signatures
- Identify correct validators when putting assets in escrow
- Check validators' credentials before voting to commit

Proof-of-work (Nakamoto) Consensus

- Lack finality
- A proof includes some confirmation blocks
- The number of confirmation blocks depends on the value of the deal

Cost Analysis

Two kinds of operations: writing to long-lived storage, and each signature verification

Gas costs for a deal with n parties, m assets, and t ≥ n transfers.

Protocol	Escrow	Transfer and Validation	Commit or Abort
Timelock	$O(m)$ writes	$O(t)$ writes	$O(mn^2)$ sig. ver. + $O(m)$ writes
CBC	$O(m)$ writes	$O(t)$ writes	$O(m(f+1))$ sig. ver. + $O(m)$ writes

```
1  contract EscrowManager {
2      ERC20Interface asset;                // contract holding assets
3      mapping(address => uint) escrow; // escrowed assets
4      mapping(address => uint) onCommit; // result of tentative transfers
5       ...
6      // transfer into escrow account
7      function escrow (uint amount) public {
8          require (asset.transferFrom(msg.sender, this, amount));
9          escrow[msg.sender] = escrow[msg.sender] + amount;
10         onCommit[msg.sender] = onCommit[msg.sender] + amount;
11     }
12     // tentative transfer
13     function transfer (address to, uint amount) public {
14         require (onCommit[msg.sender] >= amount);
15         onCommit[msg.sender] = onCommit[msg.sender] - amount;
16         onCommit[to] = onCommit[to] + amount;
17     } ... }
```

O(m)

O(t)

```
1  contract TimelockManager is EscrowManager{
2      address[] parties;              // participating parties
3      address[] voted;                // which parties have voted
4      ...
5      function commit(address voter, address[] signers, bytes32[] sigs) public {
6          require (now < start + (path.length() * DELTA)); // not timed out
7          require (parties.contains(voter));               // legit voters only
8          require (!voted.contains(voter));                // no duplicate votes
9          require (checkUnique(signers));                  // no duplicate signers
10         for (int i = 0; i < signers.length; i++) {
11             require (checkSig(voter, signers[i], sigs[i])); // expensive
12         }
13         voted.push(voter);                               // remember who voted
14     }}
```

O(mn^2) + O(m)

```
1  contract CBCManager is EscrowManager{
2      address[] validators;          // CBC validators
3      ...
4      // check commit proof is valid
5      function commit(address[] signers, bytes32[] sigs) public {
6          require(checkUnique(signers));              // no duplicate signers
7          require(validators.contains(signers));      // only validators voted
8          require(signers.length >= f+1);             // enough validators voted
9          for (int i = 0; i < f+1; i++) {
10             require(checkSig(signers[i], sigs[i])); // expensive
11         }
12         outcome = COMMITTED;                        // remember we committed
13     } ... }
```

O(m(f+1)) + O(m)

Time cost for synchronous communication

Protocol	Escrow	Transfer and Validation	Commit	Abort
Timelock	Δ	$k\Delta$ or Δ	$O(n)\Delta$	$O(n)\Delta$
CBC	Δ	$k\Delta$ or Δ	$O(1)\Delta$	per-party timeout

Discussion

- Provide incentives for good behavior
- Denial-of-service attack
- CBC: censorship
- n parties VS f + 1 validators

Questions?

Traditional Notions of Correctness

- ***Atomicity***: Either all steps happen, or none do.
- ***Isolation***: No transaction observes another's intermediate state.
- ***Consistency***: ensures that a transaction can only bring the database from one valid state to another, maintaining database invariants
- ***Durability***: once a transaction has been committed, it will remain committed even in the case of a system failure