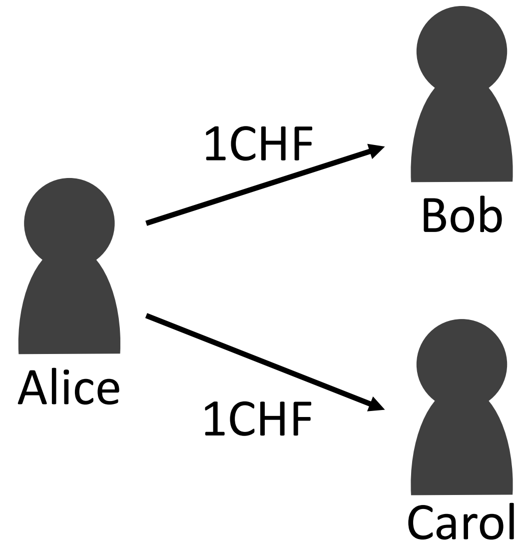
Asynchronous Consensus-Free Transaction Systems

Shoma Mori, Roland Schmid, Jakub Sliwinski, Prof. Roger Wattenhofer

Motivation

Q. Why need a consensus?

A. The systems must reach only one result



However

Solving a consensus costs much (e.g. PoW)

→ Consensus-free systems are desired

With PoW, the history of transactions can be overturned

→ Deterministic systems are also desired

Purpose

Building high <u>performance</u> systems without a consensus

- transaction/sec
- scalability

Simple Overview of ACFTS

Example: Alice $\xrightarrow{\text{1CHF}}$ Bob

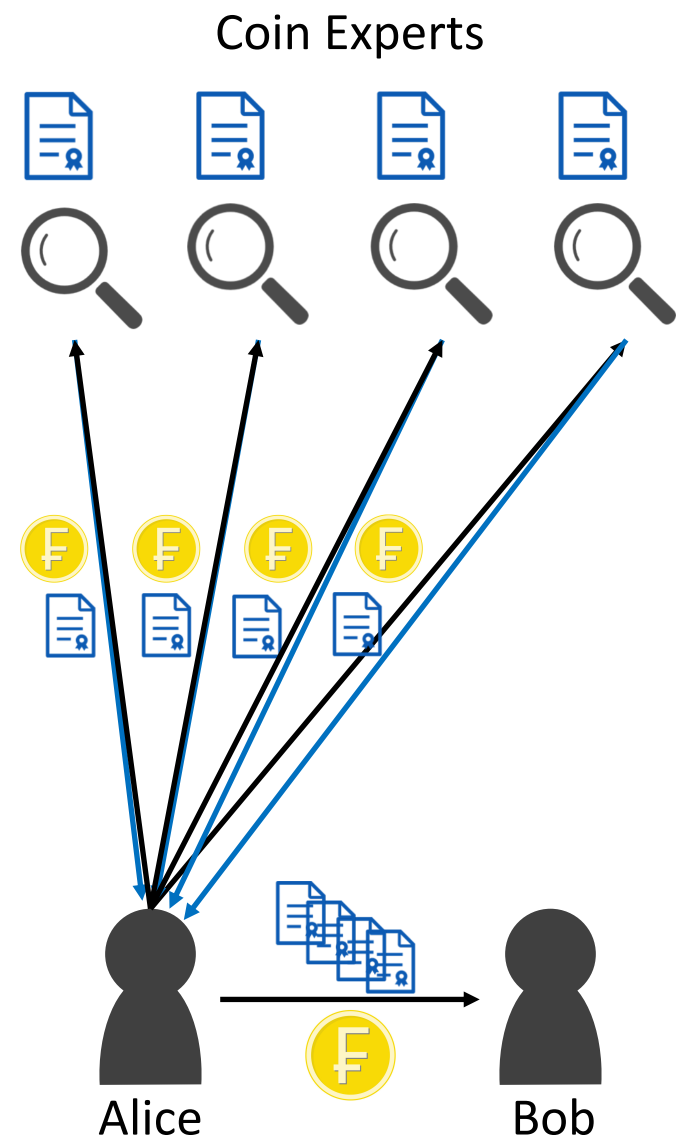
① Alice searches 1CHF coin for her wallet

② Alice sends to coin experts

③ The experts identify if is real

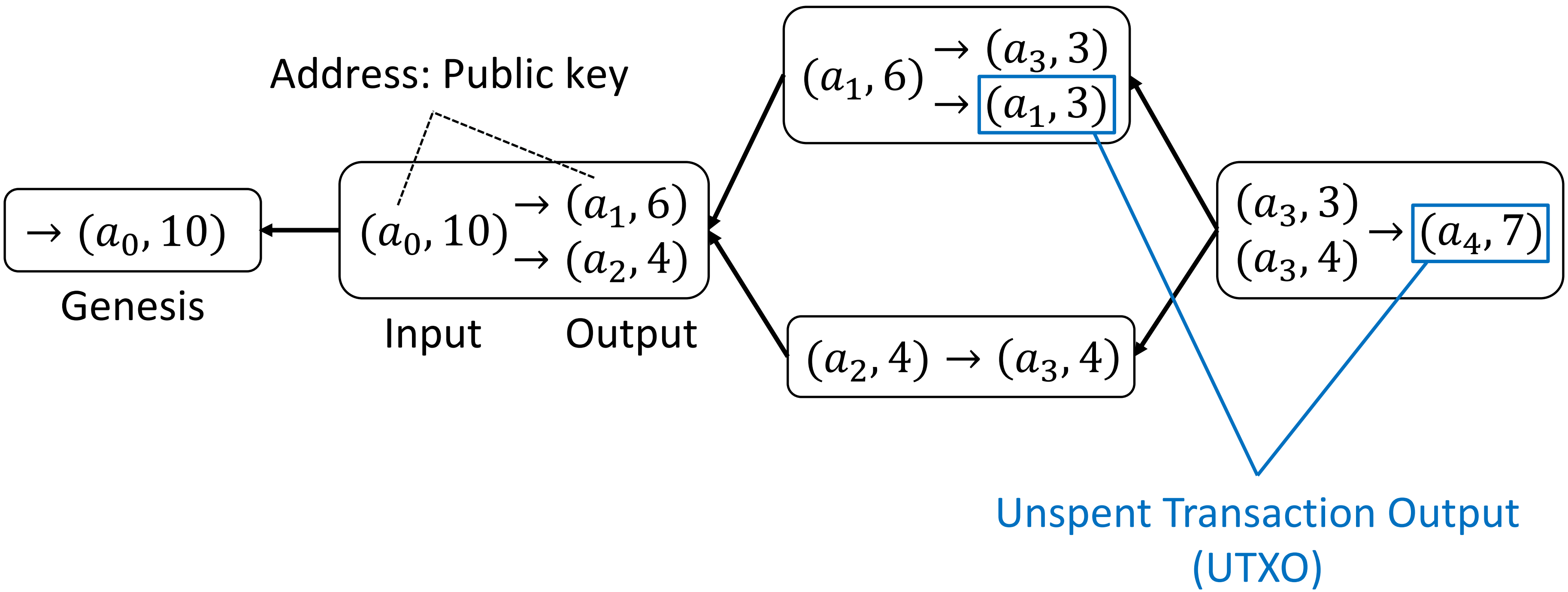
④ The experts send back to Alice

⑤ Alice sends Bob with



Transaction in ACFTS

Transaction = Transfer of cryptocurrency



How does ACFTS avoid conflicting transactions?

→ “Servers” manage spending of transactions

Overview of ACFTS

Example: $(a_{Alice}, 1) \rightarrow (a_{Bob}, 1)$

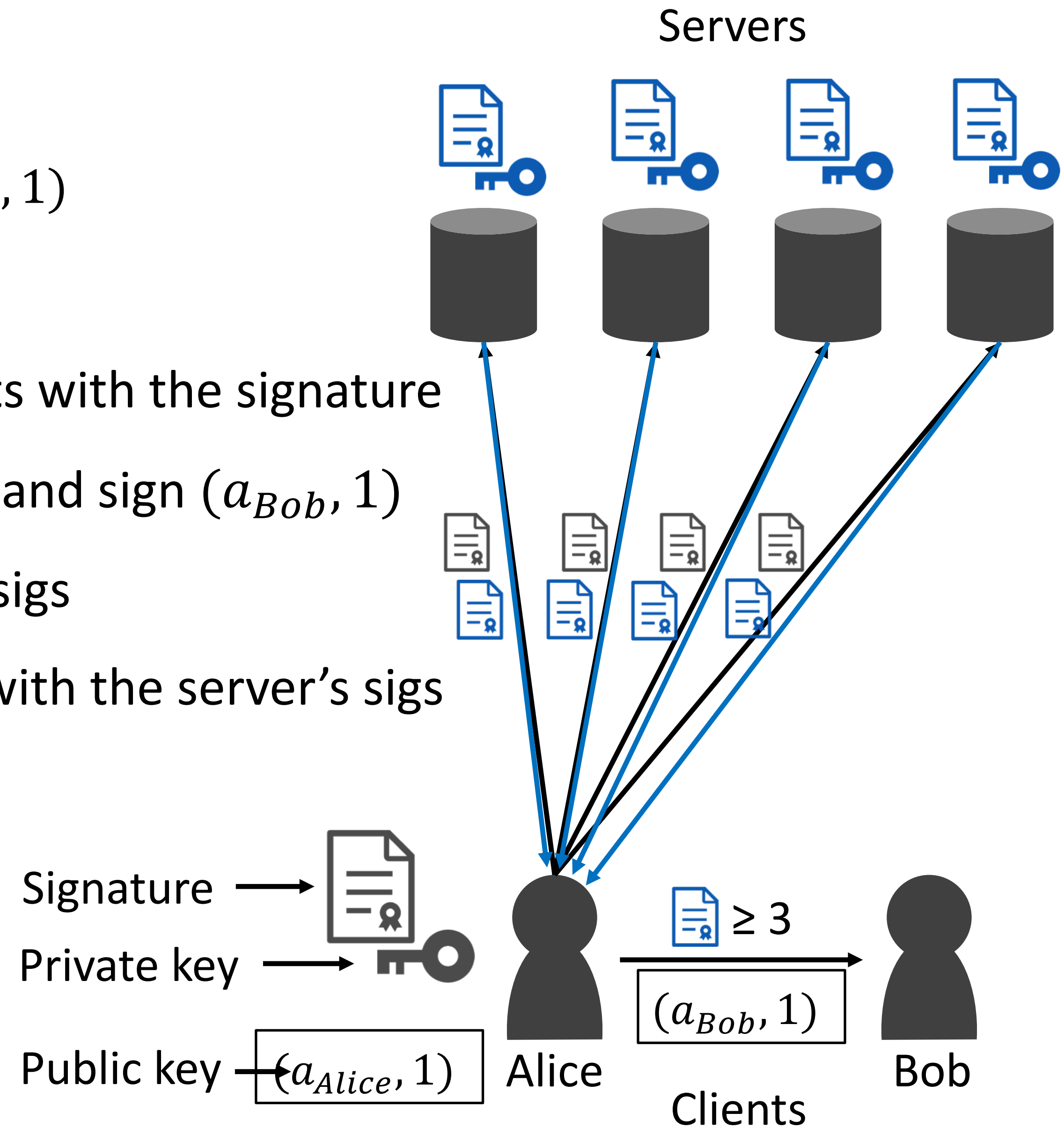
① Alice signs $(a_{Alice}, 1)$

② Alice sends servers requests with the signature

③ The servers verify the reqs and sign $(a_{Bob}, 1)$

④ The servers send back the sigs

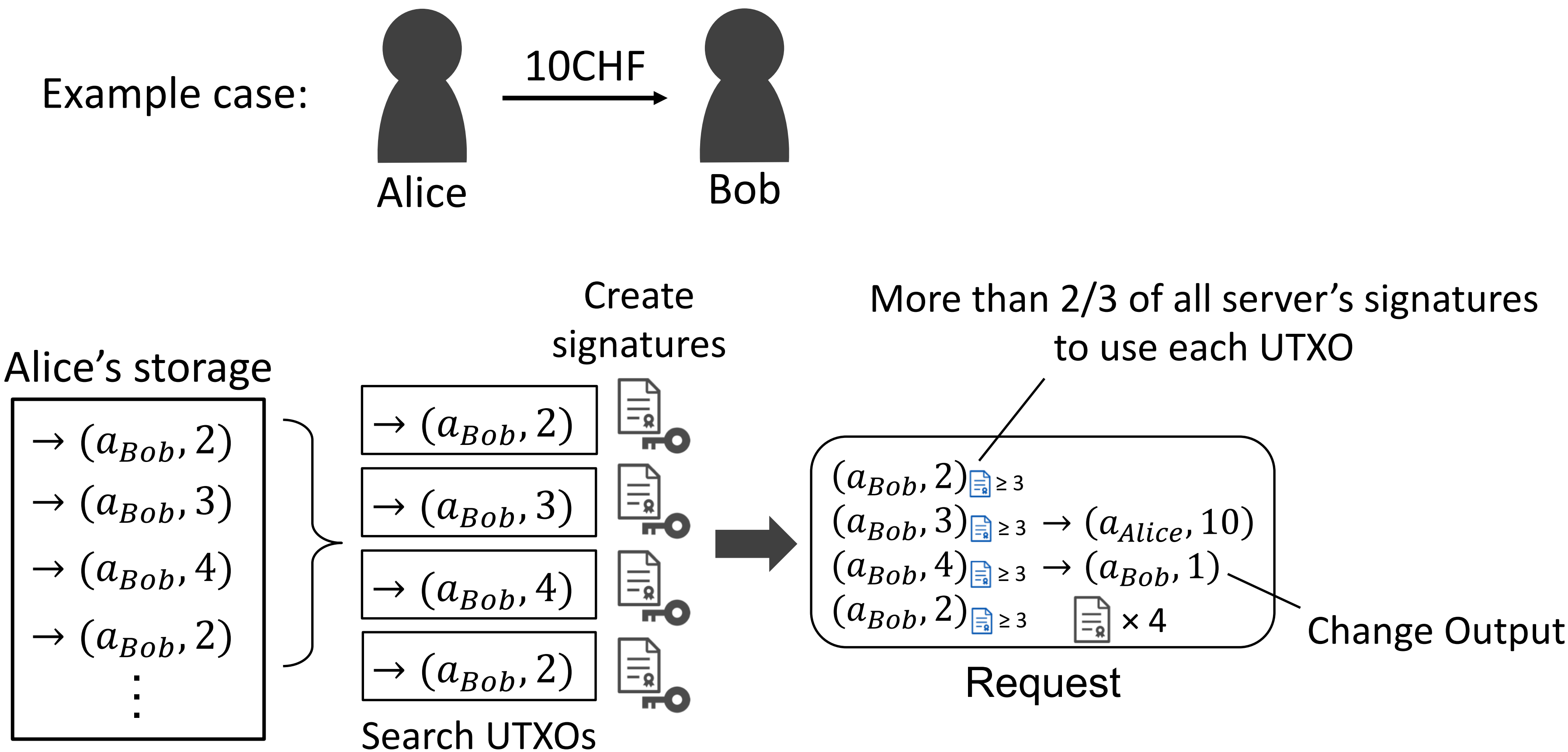
⑤ Alice sends Bob $(a_{Bob}, 1)$ with the server's sigs



The number of must be more than **2/3** of all servers[1]

[1] J. Sliwinski and R. Wattenhofer, "ABC: Asynchronous Blockchain without Consensus," arXiv e-prints, p. arXiv:1909.10926, Sep 2019.

Client



Example case:
10CHF
Alice
Bob
Alice's storage
$\rightarrow (a_{Bob}, 2)$
$\rightarrow (a_{Bob}, 3)$
$\rightarrow (a_{Bob}, 4)$
$\rightarrow (a_{Bob}, 2)$
⋮
Create signatures
$\rightarrow (a_{Bob}, 2)$
$\rightarrow (a_{Bob}, 3)$
$\rightarrow (a_{Bob}, 4)$
$\rightarrow (a_{Bob}, 2)$
Search UTXOs
More than 2/3 of all server's signatures to use each UTXO
$(a_{Bob}, 2)$ ≥ 3
$(a_{Bob}, 3)$ ≥ 3 $\rightarrow (a_{Alice}, 10)$
$(a_{Bob}, 4)$ ≥ 3 $\rightarrow (a_{Bob}, 1)$
$(a_{Bob}, 2)$ ≥ 3 × 4
Change Output
Request

Server's verification

$(a_{Bob}, 2)$ ≥ 3
$(a_{Bob}, 3)$ ≥ 3 $\rightarrow (a_{Alice}, 10)$
$(a_{Bob}, 4)$ ≥ 3 $\rightarrow (a_{Bob}, 1)$
$(a_{Bob}, 2)$ ≥ 3 × 4

Request

The status of all outputs

Output	Used
$(a_{Bob}, 2)$	false
$(a_{Bob}, 3)$	false
$(a_{Bob}, 4)$	false
$(a_{Bob}, 2)$	false
⋮	⋮

<u>Verification process</u>

- ☑ Does each input have the owner's sigs?
- ☑ Does each input have enough server's sigs?
- ☑ Are the UTXOs unused?

Server's signing

If the request is valid,

- sign a hash of **all outputs**

$\rightarrow (a_{Alice}, 10)$
$\rightarrow (a_{Bob}, 1)$

Not each output
→ For efficiency

Therefore

to use $(a_{Alice}, 10)$,
Alice needs to send  and $(a_{Bob}, 1)$

- update the status table

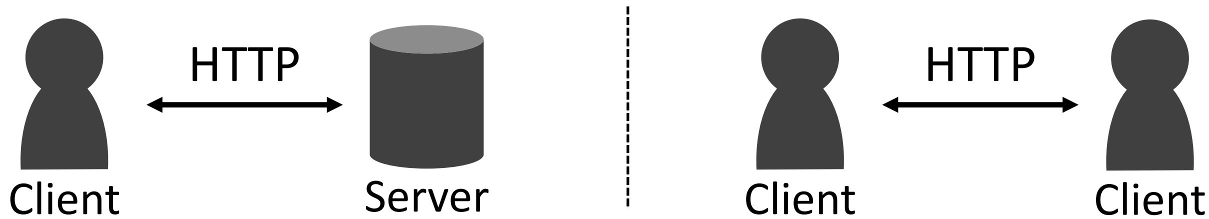
- send the signature to the client

The status of all outputs

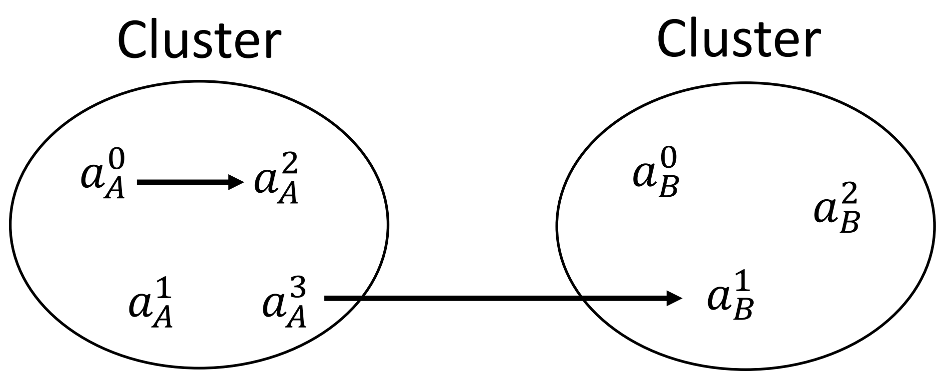
Output	Used
$(a_{Bob}, 2)$	true
$(a_{Bob}, 3)$	true
$(a_{Bob}, 4)$	true
$(a_{Bob}, 2)$	true
$(a_{Alice}, 10)$	false
$(a_{Bob}, 1)$	false
⋮	⋮

Implementation

- Communication protocol: **HTTP**



- Clients and servers manage **relational databases**
- Cluster
  - One cluster can own multiple addresses

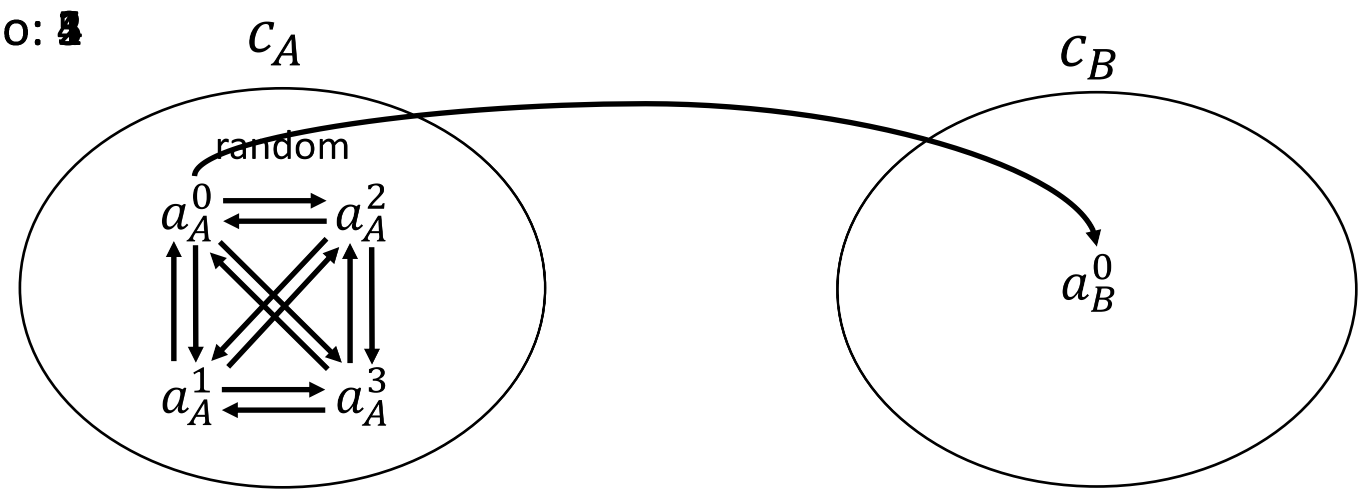


Demo

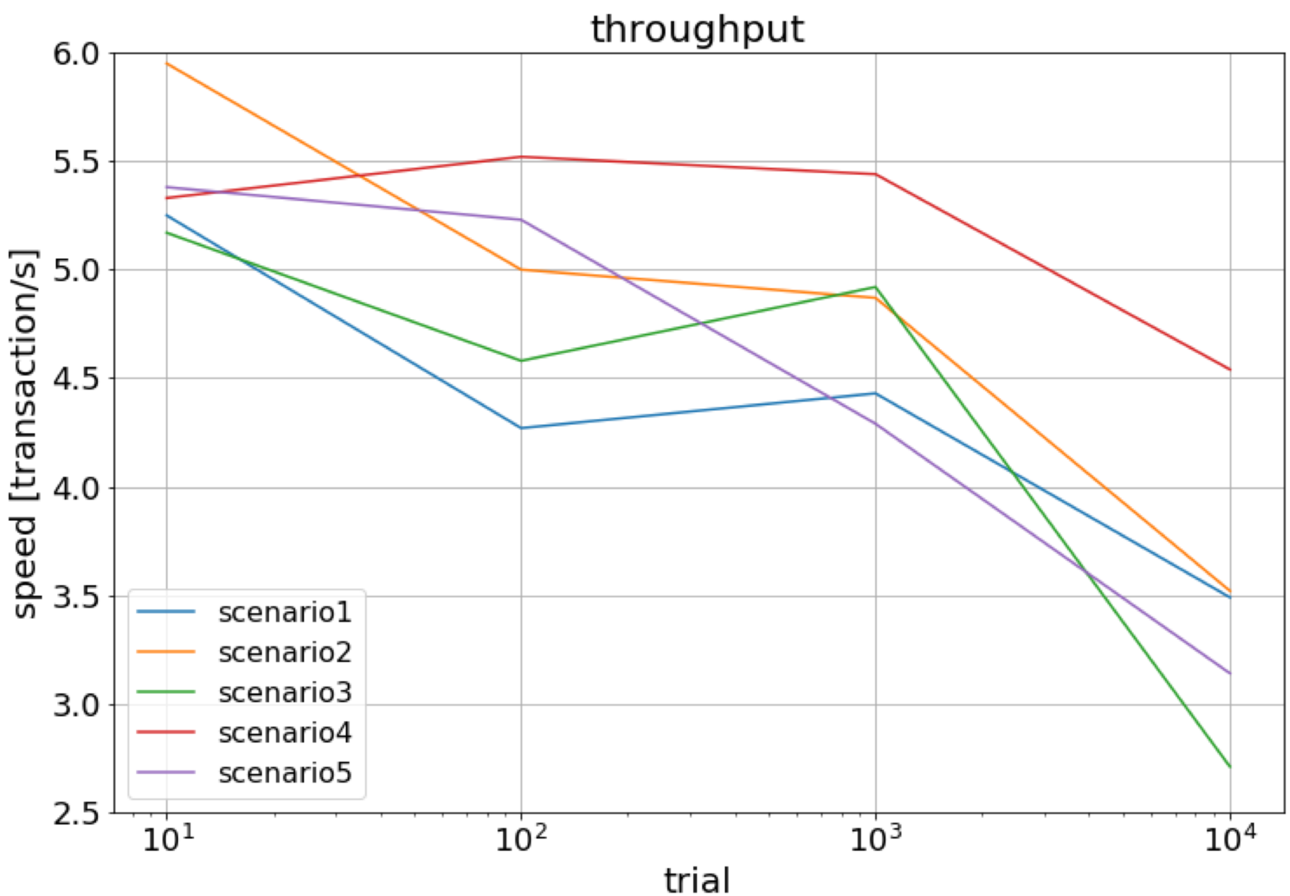
Experiment

- Purposes
  - Measuring the throughputs (transaction/sec)
  - Finding bottlenecks
- Environment
  - 4 servers
  - 2 clients (clusters)
- Scenarios [“→”: 1 amount transfer, genesis: $(a_A^0, 100000)$]

Scenario: 4



Results



- Approximately 2.7–6.0 transaction/sec
- Generally, the throughputs go down as increasing trial

➡ What are the bottlenecks preventing scalability?

Bottlenecks

Profile [scenario1, trials = 10000]

CPU usage in the client-side

name	first 60s [%]	last 60s [%]
getClientSig	43.4	40.4
updateOutputs	41.0	39.9
getServerSigs	6.13	10.3
findUTXOs	3.06	4.23

→ Sign a UTXO
→ Add new outputs & server's sigs to DB
→ **Search the DB** for server's sigs
→ **Search the DB** for available UTXOs

Client: Searching the DB does not seem to scale well

CPU usage in the server-side

name	first 60s [%]	last 60s [%]
verifyUTXO	65.1	64.1
unlockUTXO	22.7	23.1
createSignature	10.9	10.5

→ Verify server's sigs on inputs
→ Verify client's sigs on inputs
→ Sign an output

Server: All functions seem to scale

Both: **Cryptographic processes** (verification & signing) are bottlenecks

Discussion

Use a hash table for the server's signatures for scalability

Relational database

id	output
1	$(a_{Alice}, 2)$
2	$(a_{Alice}, 4)$
3	$(a_{Alice}, 3)$

output_id	signature
1	3eqr42
1	a7eq6i
1	3qu8iu
2	1qer5i
2	4qit6u

$O(n)$

Hash Table

key	data
$(a_{Alice}, 2)$	[3eqr42, a7eq6i, 3qu8iu]
$(a_{Alice}, 4)$	[1qer5i, 4qit6u]
$(a_{Alice}, 3)$	[]

$O(1)$

Conclusion

- ACFTS can avoid conflicting transactions without a consensus
- Verification & signing are bottlenecks
- In the client-side, searching the DB does not scale well

Future Work

- Change how to store outputs for scalability
- Reduce time for cryptographic processes
- Run in a real environment
- Adapt to the replacement of servers

Hash table for findUTXOs

A hash table is not that effective for finding available UTXOs

Need to store sibling outputs

Relational database

id	output
1	$(a_{Alice}, 2)$
2	$(a_{Bob}, 1)$
3	$(a_{Alice}, 3)$
4	$(a_{Bob}, 2)$
5	$(a_{Alice}, 4)$

$O(n)$

Hash Table

key	data
a_{Alice}	$(a_{Alice}, 2), (a_{Alice}, 3), (a_{Alice}, 4)$
a_{Bob}	$(a_{Bob}, 1), (a_{Bob}, 2)$

$O(m)$

Why more than 2/3?