OpenCBDC: a Whirlwind Tour

Sam Stuewe

Digital Currency Initiative
Massachusetts Institute of Technology

OpenCBDC Overview

OpenCBDC is a technical research project focused on answering open questions surrounding central bank digital currencies (CBDCs).

All community members are welcome to contribute. We have representatives from central banks, private institutions, and individual contributors.

Various research themes: Privacy, Programmability, Offline Payments, Interoperability, Security, User Experience, Policy Implications, and Architecture

OpenCBDC Goals

Create an opportunity for all stakeholders to share ideas and to make technical contributions as we research potential CBDC designs

Leverage the community expertise to advance technical contribution to the existing code

digital currency initiative

Release with FRBB in February 2022

1	**Executive Summary**	https://bit.ly/34N7Wjm
2	**GitHub Repo**	https://github.com/mit-dci/opencbdc-tx
3	**Technical Whitepaper**	https://bit.ly/3GHrKSi
4	**Technical Webinar**	https://youtu.be/L7cu4J8YUcQ

Our original research question

"Could a CBDC be…?"

YES

- Secure
- Flexible
- Resilient
  - 24/7/365 availability
  - Fault tolerant (geographically distributed)
- Performant
  - Processes at least 100k transactions per second
  - Settle in 5 seconds or less

CBDCs Present Unique Design Requirements

- Prevent double spends
- Low latency finality (global consistency)
- *But,* Byzantine Fault Tolerance isn't the core problem

Approaches

- Use an off the shelf package
- Create a codebase from scratch

We chose to create a codebase - the goal is to learn!

opencbdc-tx

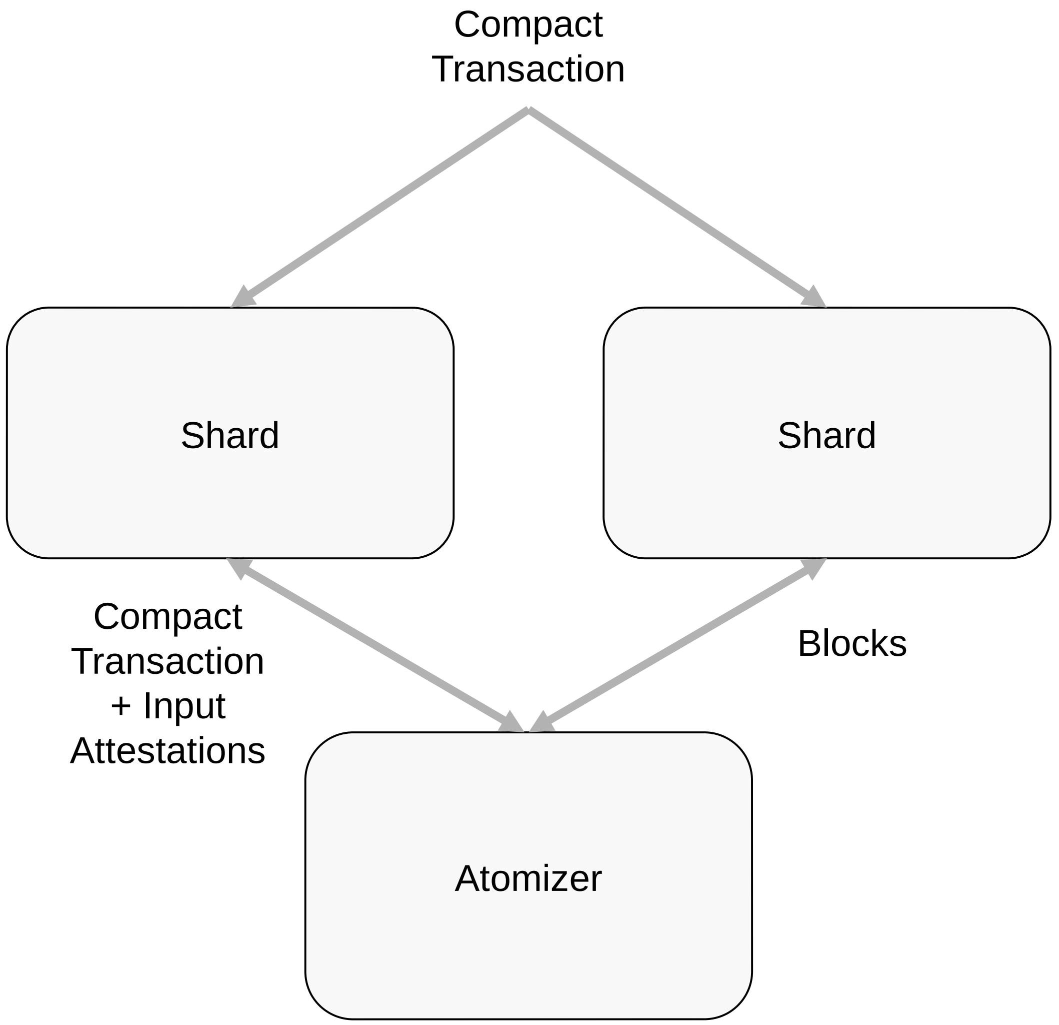
- A hypothetical CBDC codebase written in C++
- Rely on cryptography
- Store as little as possible in the core
- Parallelize as much as possible
- Present a flexible framework for experimentation
- Remember: The goal is to learn!

https://github.com/mit-dci/opencbdc-tx

  - Contributions Welcome!

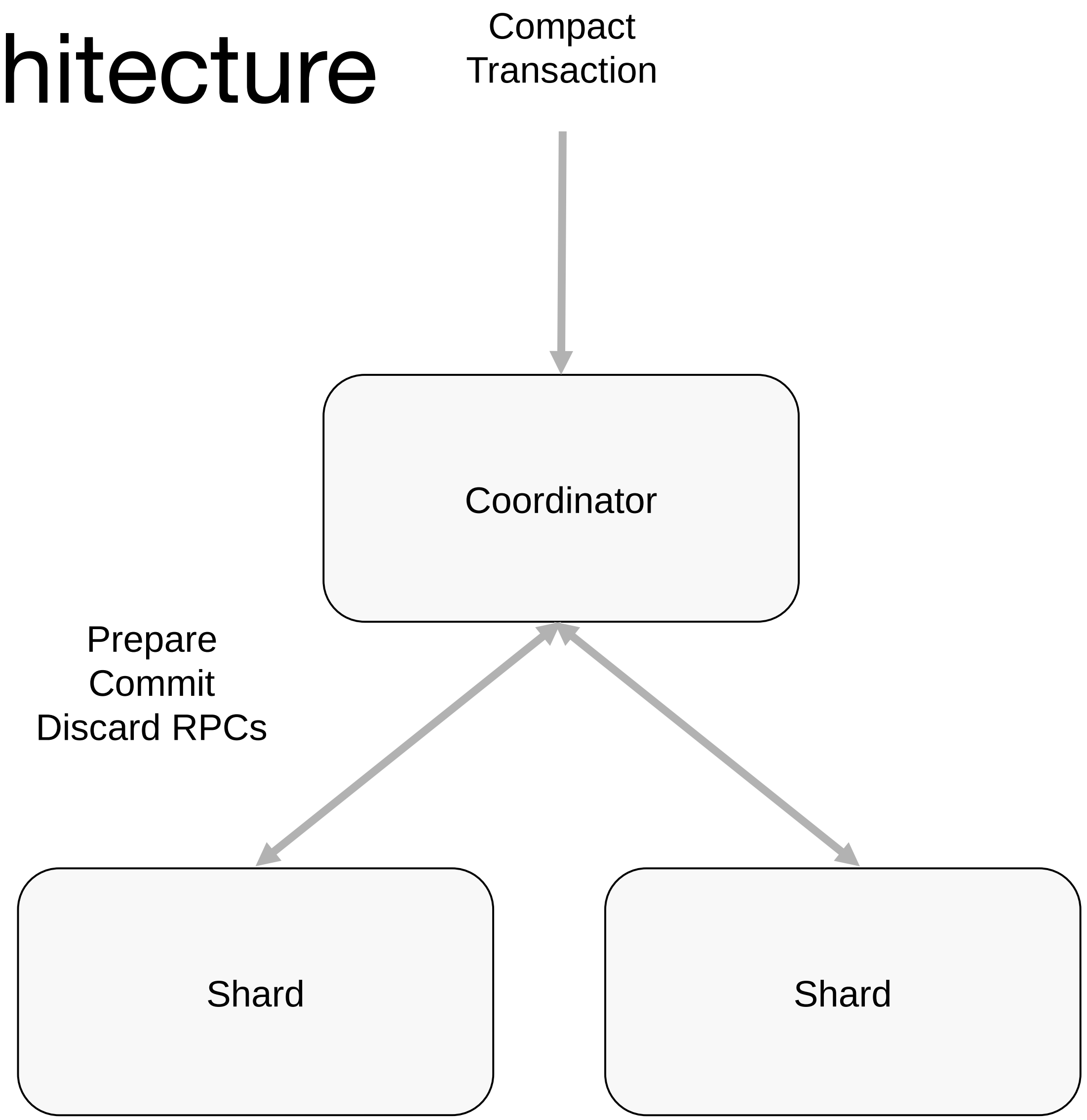
Atomizer Architecture

- Shards each store a subset of the total Unspent-funds Hash Set (UHS)
  - Receive compact transactions from sentinels
  - Check that input UHS IDs exist in the UHS
  - Forward compact transaction to the atomizer service
  - Attach “attestations” stating which input UHS IDs in the compact transaction are unspent
- Atomizer service de-duplicates and orders transactions into “blocks”
- Atomizer sends blocks to shards, which apply updates to their UHS subset



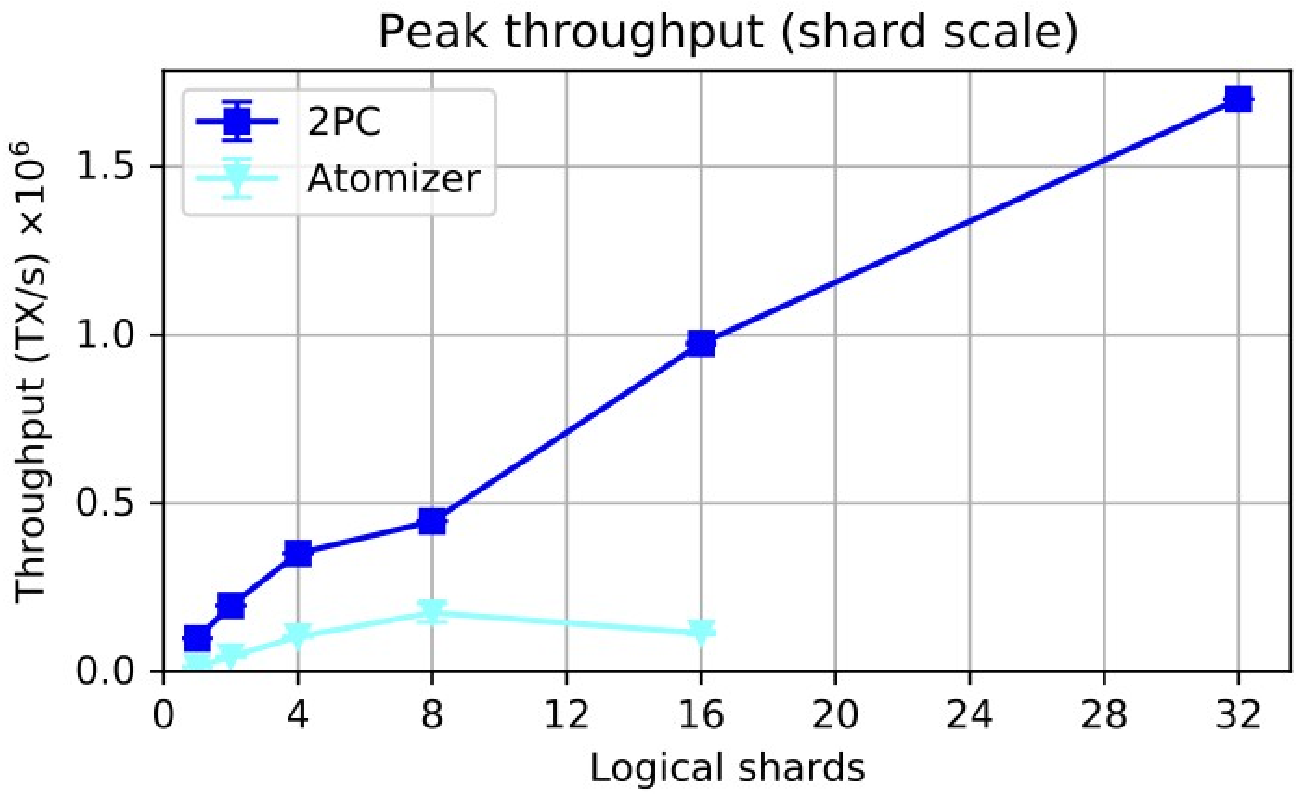
Two-Phase Commit (2PC) Architecture

- Coordinator receives compact transactions from sentinels
    - Batches compact transactions into a “distributed transaction”
    - Generates a unique distributed transaction ID for each batch
- Shards each store a subset of the total UHS
- Coordinator locks input UHS IDs on each shard for transactions in the batch (prepare)
- Collects responses from shards, decides which transaction to complete (all inputs locked), or abort (inputs don’t exist, or already locked)
- Coordinator requests commit from each shard
    - Shard unlocks inputs for aborted transactions, deletes input for completed transactions, and creates new outputs
- Coordinator finishes the distributed transaction by calling “discard” on each shard



Benchmarks

- Tested both architectures in 3 US AWS regions
- Randomized 2-in, 2-out workload
- Atomizer reaches a throughput plateau
- 2PC appears to scale linearly with additional shards



Summary Findings

- Secure transaction format and data model
  - Cryptographic authorization of funds transfer
  - Minimized data retention
  - Non-malleable
  - Non-replayable
- Two architectures
  - Atomizer, provides linear order of transactions
    - Peaks at ~170K TX/s
  - 2PC, parallel transaction processing
    - No experimentally demonstrated peak throughput

What are we working on now?

- Minimal Tamper Detection
  - Detect any inflation/deflation of the monetary supply in the face of compromise of any single component
  - Two solutions implemented (and open-sourced!), now undergoing testing
- Programmability
  - A new architecture (open-sourced!) enabling arbitrarily-complex, stateful execution

How do I get involved?

1. Have a new idea?

   Submit a proposal: https://tinyurl.com/mvr3hkc8

2. Dive in and start making technical contributions
   a. Help refactor and improve code quality (#81, #158, #173, …)
   b. Add Support for Strong Privacy Guarantees (#49, #90)
   c. Explore programmability (#72, #83, #164, …)
   d. Propose others!

3. Join the discussion!

   Join and start discussions on Zulip (https://opencbdc.zulipchat.com/register)