Blockchain DB-unked
Chet Murthy
Independent Researcher
chetsky@gmail.com
Plan of Talk

• The Punchline
• Preliminaries and Technical Background
  – Stored-procedures, public-key crypto, consensus protocols
• (Remembering naive) Database Design
• Blockchains are *merely* Replicated Databases
  – Bitcoin
  – Ethereum
  – A better design (infraledger)
• Conclusions
The Punchline

• Blockchain design should be descended from (replicated) database system design
• …. but it wasn’t (and that’s a pity)
• Whenever you see a new blockchain, ask for a mapping back to standard database & distributed-systems concepts and design-elements, and when that isn’t forthcoming, be very, very skeptical
• The only salient difference is that in blockchains, not all replicas are trusted, and special protocols are used to deal with this
Plan of Talk

• The Punchline
• Preliminaries and Technical Background
  – Stored-procedures, public-key crypto, consensus protocols
• (Remembering naive) Database Design
• Blockchains are merely Replicated Databases
  – Bitcoin
  – Ethereum
  – A better design (infraledger)
• Conclusions
Stored Procedure DB Apps

• A collection of relational tables + client-invokable procedures (written in some restricted language – BASIC, Java)
• deployed/configured at runtime via special transactions
• ACLs restrict tables to only be invokable by stored-procs
• Authenticated clients invoke stored-procs (but NOT access tables directly)
• With stored-proc-code-based enforcement of business-logic rules
• [remember this] a tran always consists in the invocation of a single stored-proc with a list of arguments (or a list of such invocations)
  – clients **do not invoke arbitrary SQL** – only stored-procs
  – **Ideally** stored-procs are *replayable* – stateless, access no outside systems and fully-deterministic
An Example ("PAY")

<table>
<thead>
<tr>
<th>TRAN-ID (guid)</th>
<th>INDEX (int)</th>
<th>Owner (userid)</th>
<th>Balance ($$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xdeadbeef</td>
<td>0</td>
<td>chet</td>
<td>1.00</td>
</tr>
<tr>
<td>0xffff</td>
<td>1</td>
<td>Joe</td>
<td>2.00</td>
</tr>
</tbody>
</table>

• Primary key (tran-id, index)

• A stored-procedure invocation ("PAY") is a tran (identified by fresh tran-id), specifies rows to delete (by primary key) and new rows to insert (an array of (owner, balance))

• Business logic rules:
  - $\text{SUM(balance(rows-deleted))} \geq \text{SUM(balance(rows-inserted))}$
  - Authenticated user must match owner column of deleted rows

• [remember for later] every row read, is deleted; every row inserted has a fresh (never-before-seen) primary key
Public-Key Cryptography for Database Authentication

- A “public key” stored in the “owner” column of a row
- Instead of authenticating to the database, the client “signs” the stored-proc invocation with their matching “private key”
- Instead of checking that the user authenticated as the ID in the owner column, the DB checks that the invocation contains a signature for the public key in that column
  - And there might be more than one signature, so a tran could delete rows owned by Joe and Chet, and insert a row owned by Mark
Public Key .... (cont’d)

- So that PAY stored-proc invocation looks like:
  - Fresh tran-id
  - Rows to delete: (tran-id, index)
  - Array (zero-indexed) of (owner, balance) for rows to insert (primary key is (tran-id, index))
  - **Signature for each row to delete**
    - let’s not worry about precisely how signatures are implemented (just tedium)
Consensus (simplified)

- There are “replicas” and “clients”
- Replicas cooperate to maintain and extend a “log”: a consecutively numbered sequence of binary entries
- Clients send requests to replicas, “proposing” new log-entries
- Replicas reach agreement on new entries, disseminate them to all replicas and eventually inform clients of “committed” values
  - Usually involves some sort of proposal/voting/committing phases, so replicas vote on whether each new entry will be added to the log
- Correctness conditions:
  - Every entry was proposed by a client
  - If a replica “commits” a value for entry #i, then any other replica that commits a value, will commit that same value, and that replica will not later commit a different value for entry #i (Call this “no take-backsies”)
Consensus (the varieties)

• “No fault-tolerance” – one replica
  – All clients connect to one replica, that keeps track of the log (tolerates no crashes or corruptions)

• “Crash fault-tolerance (Paxos)”: \( N \geq 2f+1 \) replicas
  – Tolerates at most \( f \) crashes (but no corruptions) by having at least \( 2f+1 \) nodes, commit requires \( f+1 \) yes votes
  – “majority weighted voting”

• “Byzantine fault-tolerance (BFT)”: \( N \geq 3f+1 \) replicas
  – Tolerates at most \( f \) crashes or corruptions (called “Byzantine failures”) by having at least \( 3f+1 \) nodes, and commit requires \( 2f+1 \) yes votes
  – “\( 2/3+1 \) votes to commit”

• In all of the above, the set of replicas is “managed” (== “permissioned”) – replicas cannot arbitrarily join the system without being granted permission by the current replica-set
Proof-of-Work (not-BFT-Consensus)

- Consensus is just a way for N replicas to agree on the contents of a log so that no *take-backsies* occur
- **Proof-of-work (PoW, aka “Nakamoto consensus”)** is a protocol that is *not consensus*: *take-backsies* can occur, but it’s **very, very, very unlikely**
  - Uses crypto-currency as intrinsic part of consensus (defer to after talk if anybody’s interested), using computationally hard problem to implement leader-election
- Why would you want this?
  - Because no global permissioned replica-list is needed
  - In PoW, replicas can join and leave **arbitrarily** with no identification or permissioning whatsoever
Plan of Talk

- The Punchline
- Preliminaries and Technical Background
  - Stored procedures, public-key crypto, consensus protocols
- (Remembering naive) Database Design
- Blockchains are *merely* Replicated Databases
  - Bitcoin
  - Ethereum
  - A better design (infraledger)
- Conclusions
Recap of Database Design

Lock Manager

Acquire locks

Append tran Log-entry

Write-Ahead Log

Apply committed trans

Running Transaction

Dirty (modified) Rows/Pages

Read rows
(recap) Replicated Database

Consensus
(optimistic #1) tran lifecycle

• Typically how things are done (viz. Spanner)

  • Trans are run at “submitting replica” and track versions/values of rows they read&write (using single-instance/sharded lock-manager)

  • At commit-time, lock-manager ensures that no other tran has modified those rows (and if not, tran aborts)
    – If two trans try to modify the same row, one of them will be aborted/not permitted to commit

• If all is well, writes tran (list of “postimages” of rows) to log (via consensus)

• Only after tran is (durably) committed to log (“post-commit time”) (at all replicas), update tables with postimage of rows

• Key observations:
  – lock-manager may delay/abort trans
  – Replicas trust submitting client/replica, lock-manager, consensus
  – Thus, intrinsically not BFT
(optimistic #2) tran lifecycle

- Running tran at submitting replica tracks versions/values of rows read&written (using per-replica lock-manager)
- At commit-time, lock-manager constructs a list of these versions ("locks") for inclusion in the transaction record ("MVCC information")
- Submitting replica submits `proposal=MVCC+postimage` information to log via consensus
- At post-commit time (at all replicas), check that locked rows have not changed, and only then update tables with postimage of rows (otherwise, abort)
  - If two trans try to modify the same row, and both are submitted concurrently, one will abort (again, during post-commit time)
- Key observations
  - lock-manager does **not** delay/abort trans
  - [remember this for later] replicas trust submitting replica & consensus (**must trust that MVCC+postimage is correct and not corrupt**)
    - E.g. that a PAY tran deducts from payer, and credits payee honestly (business-logic rule, enforced by stored-proc code, not database)
  - The difference from #1 is **no lock-manager**, instead, lock-information goes in the tran and is checked at post-commit time
(“vacuous”) tran lifecycle

• Since a tran invokes a single stored-proc, just put that *invocation* into the tran-record and commit that to the log via consensus
  - No “run the tran, accumulate locks+changed rows”

• At post-commit time *(at all replicas)*, run the tran and apply *postimage changes*
  - Since *only one tran at a time runs* *(at post-commit time)*, no lock-mgmt is needed, trans never abort

• Key observations:
  - Every tran is run at every replica, and **in order with no concurrency**
  - Replicas trust consensus, but nothing else
OK, we’re finally ready ....

- Things to look for:
  - How are stored-proc apps deployed (if at all)
  - Are stored-procs computationally (non-)trivial?
  - What sort of transaction lifecycle
  - The trustworthiness of which components is assumed? (submitting replica? Consensus? (BFT?) Lock-manager?)
  - Are stored-proc invocations run everywhere? Only at a bounded # of replicas?
  - What is the rate-limiting factor for tran throughput?
    - What computation happens at every replica, for every tran?
Plan of Talk

• The Punchline
• Preliminaries and Technical Background
  – Stored procedures, public-key crypto, consensus protocols
• (Remembering naive) Database Design
• Blockchains are *merely* Replicated Databases
  – Bitcoin
  – Ethereum
  – A better design (infraledger)
• Conclusions
Bitcoin (BTC) as a simple replicated database

- The only application is (remember from earlier?) “PAY” stored-proc (with public-key auth)
  - So: one table, no range-queries
  - I’m lying a little, but not much (discuss “smart contracts” after the talk)

- Lifecycle strategy “optimistic #2” since tran-invocation lists rows deleted/inserted explicitly (so trans contain MVCC information!)

- Consensus is proof-of-work but at “consensus proposal” time, MVCC information is evaluated, and only commitable trans are passed thru consensus

- Key properties
  - (similar to “vacuous”) each tran must commit before next tran can enter proposal phase – throughput bottleneck
  - Every tran is run at every replica (at both proposal and post-commit time), but since “PAY” is a computationally trivial stored-proc, this is not problematic
  - Because trans are run at proposal-time, no need to trust submitter, only consensus
  - Size of actual tables is limited (b/c “take-backsies” requires rolling back trans, and this is not intelligently implemented)
    - 65GB “blockchain” (== “full log”), but 660MB table-size (!)
Ethereum as a replicated database

- DB allows deploying arbitrary stored-procedures at runtime
  - Database tables have only two columns (“key”, “value”), primary-key “key” and no range queries
- Lifecycle strategy “vacuous”
- proof-of-work (or new proof-of-stake)
- Every tran is run at every replica and this is problematic b/c stored-procs can and do perform near-arbitrary computation
What do we really want?

• Like “optimistic #2”:
  − Near-arbitrary stored-procs (viz Ethereum)
  − Trans are run at only a bounded subset of replicas (so no throughput bottleneck)
  − Only MVCC/postimage information is evaluated at all replicas (viz BTC; this is cheap)

• **But**: remember that we cannot trust submitter’s computation of MVCC/postimage (!) (whereas “optimistic #2” trusts submitter)

• **So**: in addition to “run at submitter”, run at f+1 validators and cross-check (to detect corruption of at most f replicas)
(optimistic #2 MODIFIED)

- Running tran at submitting replica tracks versions of rows read&written (using per-replica lock-manager)
- At commit-time, lock-manager constructs a list of these locks for inclusion in the transaction record (“MVCC information”)
- Submitting replica sends “proposal” (== tran-invocation+MVCC+postimage) to F+1 other replicas (validators) who replay tran, checking that MVCC+postimage is not corrupt, and returning signature on proposal
- Submitting replica submits proposal+validation signatures information to log via consensus
- At post-commit time (at all replicas), check that validation signatures are legitimate, locked rows have not changed, and only then commit postimage (otherwise, abort)
  - If two trans try to modify the same row, and both are submitted concurrently, one will abort (again, during post-commit time)
- Key properties
  - lock-manager does not delay/abort trans
  - Trans are run at submitting replica, re-run (for validation) at f+1 validators, and replicas must trust that at most f validators will be corrupt (as well as trusting consensus)
This proposed design exists (infraledger)

- Arbitrary stored-proc applications, deployed at runtime (currently ocaml, but Golang is coming soon)
- Per-app database is collection of tables with typed columns, multicolumn primary&secondary keys in customary manner
- Trans run as just described
- # of replicas can be >> f (# of failures tolerated)
- Effective throughput is limited by ability of replicas to process MVCC+postimage, not rate of executing stored-proc invocations
- Data-set size of tables is limited only by time to transfer full snapshot or enough log to catch-up out-of-date replicas
- Compatible with CFT, BFT, and (with UNDO/REDO records) PoW
  - So: “history-of-database-design-based” support for “take-backsies”
Conclusions

- The “blockchain” in “Blockchains” is **the log of a database**. Those who cannot explain precisely what the database is (its schema, trans, etc) are probably lost.

- They’re **just** replicated databases, and their design should follow from database & distributed-systems considerations.
Questions?
Things not Discussed?

● Business Model
  - Hype? Reality? F(ear)O(f)M(issing)O(ut)?
● Decentralization/Consortia/Application Governance
● Confidentiality/Data-Privacy
● Anonymization/Unlinkability
● Sharding
● Low-latency