Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains

Presentation by Ben Manley
What is a Blockchain system?

Untrusting peers holding immutable ledgers of transactions

Data: “ABC”  
Hash: f024  
Previous Hash: ----

Data: “Beans”  
Hash: 3ae7  
Previous Hash: f024

Data: “Rice”  
Hash: fb61  
Previous Hash: 3ae7
What is a Blockchain system?

Public / Permissionless

- Proof-of-work (PoW) consensus

Permissioned

- Byzantine-fault tolerant consensus
Smart Contracts

Programmable transaction logic

Cryptocurrency example:

If (sale went through):
    transfer to B
Else:
    return to A

Participant A  

Participant B
Why reinvent the wheel?
Everyone else uses Order-Execute
Order-Execute Sucks: Sequential Execution

Limits throughput

Denial-of-Service (DoS) from just a long/infinite smart contract

Ethereum solves with “gas”
(not helpful without a cryptocurrency)
Order-Execute Sucks: Non-Determinism

Operations after SMR must be deterministic.

Could require specific languages.

Can’t trust programmers w/determinism in general languages.

Repair: rollback and re-execute sequentially.
### Other Previous Limitations

<table>
<thead>
<tr>
<th>Confidentiality</th>
<th>No secret smart contract logic, etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed trust model</td>
<td>Applications stuck with BFT’s $f$ out of $&gt;3f$</td>
</tr>
<tr>
<td>Hard-coded consensus</td>
<td>Applications stuck with whatever protocol the blockchain chose</td>
</tr>
</tbody>
</table>
Fabric

Components          Transaction Flow          Evaluation
Chaincode

Programmable transaction logic (Smart Contracts) with endorsement policy

Fabric is the first to support standard programming languages (non-determinism is allowed!)

Fabric is “the first distributed operating system”
Peers

Actions
• Simulating and endorsing transactions
• Gossiping results
• Validating and committing

Components
• Docker for chaincode “simulation”
• Ledger (hash-chained block store)

Chaincode can call
- GetState(key)
- DelState(key)
- PutState(key, val)

• Key-Value Store (KVS)
  - key1: (val1, ver1)
  - key2: (val2, ver2)
  ...

...
Membership Service Provider (MSP)
- Issues credentials
- Maintains identities
- Abstracts general auth
- Can be multiple

At each node:
- Authenticates transactions
- Signs endorsements
Ordering Service

Maintains multiple *channels*
- One logical blockchain each
- Separate total order
- Reconfig and access control

Batches transactions into *blocks*
- Deterministic
  - (# transactions, # bytes, timeouts)

Made up of OSNs (Ordering Service Nodes), or *orderers*
Ordering Service

Provides atomic broadcast for ordering transactions (stateless!)

API (invoked by peer):

• `broadcast(tx)`
  Client calls to broadcast transaction after receiving endorsements

• `B ← deliver(s)`
  Client calls to retrieve block \( B \) at sequence number \( s \)
Ordering Service

Guarantees (informally):

- **Agreement**: All peers see same $B$ delivered for a given $s$
- **Hash chain integrity**: block at $s+1$ holds hash of block at $s$
- **No skipping**: If peer delivers at $s$, it has already delivered $[0, s-1]$
- **No creation**: All $tx$ in a correctly-delivered block $B$ was broadcast
- **Validity**: If a correct client calls $broadcast(tx)$, every correct peer eventually delivers a block $B$ containing $tx$
A Day in the Life of Fabric

Transaction Flow
Figure 4: Fabric high level transaction flow. (for a single channel)
Phase 1: Execution
Phase 1: Execution

Endorsement policy specified by chaincode

Example:
Send to P1-P3.
Valid if endorsed by (P1 AND P2) OR P3.
Phase 1: Execution

**Proposals**

Peer transaction manager

Docker

Endorser simulates proposal

readset = \{ (key, ver), \ldots \} 

writeset = \{ (key, val), \ldots \} 

(NO PERSISTING OR SHARING)
Phase 1: Execution

Proposals

Endorser simulates proposal

readset = \{ (key, ver), \ldots \}  
writeset = \{ (key, val), \ldots \}  
(NO PERSISTING OR SHARING)
Phase 1: Execution

Note: Application chaincodes isolated from each other and peer

New languages just require new plugins, peer agnostic to language
Phase 2: Ordering

1. Invocation
2. Chaincode execution
3/4. Endorsement collection
5. Broadcast/Delivery
5. Validation

Ordering service

client

endorsing peer 1
endorsing peer 2
endorsing peer 3

Peer (non-endorsing)

commit

broadcast(tx)
Phase 2: Ordering

Ordering Service

Transactions

Hash

Prev. hash

Org 1

Org 2

Transaction

Orderers

Consensus

Gossip

(includes endorsements)
Phase 3: Validation
Phase 3: Validation

1. Endorsement policy evaluation
   Validation system chaincode (VSCC)

2. R/W conflict check
   Sequentially for all tx’s in block, compares readset with KVS

3. Ledger update
   Append block, apply writeset to KVS for valid tx’s, store Steps 1-2

Endorsements fit chaincode policy?
Endorsements fit…?
Endorsements fit…?

Valid: 1 1 0 1 0 1 1 1 1

readset matches…?
readset matches…?
readset matches…?
readset matches current version?

Invalid tx’s included in ledger!
Execute-Order-Validate

Active replication

Passive replication
Filling In Some Details
Ledgers tolerate peer crashes.

1. Write block to persistent ledger
2. Apply writeset of valid transactions to versioned KVS
3. Compute and persist savepoint = largest successfully-applied block #

savepoint = 2
Configuration is baked into the ledger.

Channel 1

“Genesis” Config Block

Config Block (special tx)

Channel 2

“Genesis” Config Block

Config Block (special tx)

- MSP definitions
- Orderer addresses
- Ordering service / consensus config (batch size, timeouts, etc.)
- Ordering API access rules
- Config modification rules
Fabric has its own special chaincodes.

System Chaincodes (both customizable)

- Endorsement system chaincode (ESCC)
  
  \[ \text{ESCC(proposal, simulation results)} \rightarrow \text{results, endorsement} \]

- Validation system chaincode (VSCC)
  
  \[ \text{VSCC(tx)} \rightarrow \text{validity bool} \]

Run directly on peer outside of Docker
Applications have independent trust/fault models.

Application models are independent: chaincode endorsement policy.

Ordering Service

Orderers

Consensus

Single-node, CFT cluster, BFT cluster...
Evaluating Fabric is difficult.

Performance depends on...

choice of distributed application and transaction size

network parameters and topology

ordering service and consensus implementation and parameters

node hardware

number of nodes and channels

... and more config parameters
Fabcoin: Bitcoin-Inspired Fabric Coin

"coin states"

 Existence = unspent
Delete when spent

UTXO
Unspent Transaction Output

Transactions:

• MINT: request = (centralBankID, outputs, sigs)
  outputs = coin states to create

• SPEND: request = (inputs, outputs, sigs)
  inputs = list of coin states to spend (delete)
  outputs = coin states to create

(txid_#: (amt, owner))

tx0_0: ($100, Manos)
tx5_2: ($20, Manos)
tx5_3: ($50, Leslie)
Chaincode:

```java
SPEND_request(inputs, outputs, sigs):
    verify sigs;
    for (input in inputs):
        GetState(in) // add to readset
        DelState(in) // add to writeset
    for (int i = 0; i < outputs.size; ++i):
        PutState(txid_i, outputs[i]) // add to writeset
```

Verification: Check \(\text{sum(inputs)} = \text{sum(outputs)}\), etc.
No need to check double-spending!
Default Experimental Setup

- Fabric v1.1.0-preview
- IBM Cloud (SoftLayer) Data Center

Nodes:
- Dedicated VMs, 1Gbps networking
- 16-vCPU 2GHz dedicated VMs
- Ubuntu, 8GB RAM, SSD local disks
- 3 orderers (all distinct VMs)
- 5 peers (all different orgs, all endorsers)
- 256-bit ECDSA signatures
Experiment 1: Choosing Block Size

Throughput (tps)

Avg Latency (ms)

Block Size (MB)

MINT/SPEND latencies

MINT/SPEND throughput
Experiment 2: Impact of Peer CPU

SPEND only, Validation Phase only (Ordering wasn’t bottleneck)

Throughput (tps)
Validation and E2E throughput
VCSS latency (verify sigs)
Avg Latency (ms)

rwcheck and ledger latency

# vCPUs
Experiment 2: Impact of Peer CPU

Conclusion: VSCC is very parallel. Pipeline validation stages, optimize stable-storage access, parallelize dependency checks.

At peak throughput (3560+ tps SPEND) with 32-vCPU, 2MB blocks:

<table>
<thead>
<tr>
<th>Step</th>
<th>avg</th>
<th>st.dev</th>
<th>99%</th>
<th>99.9%</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) endorsement</td>
<td>5.6 / 7.5</td>
<td>2.4 / 4.2</td>
<td>15 / 21</td>
<td>19 / 26</td>
</tr>
<tr>
<td>(2) ordering</td>
<td>248 / 365</td>
<td>60.0 / 192</td>
<td>324 / 624</td>
<td>523 / 636</td>
</tr>
<tr>
<td>(3) VSCC val.</td>
<td>31.0 / 35.3</td>
<td>10.2 / 37.0</td>
<td>64.0 / 624</td>
<td>113 / 108.4</td>
</tr>
<tr>
<td>(4) R/W check</td>
<td>34.8 / 61.5</td>
<td>3.9 / 9.3</td>
<td>70.0 / 88.5</td>
<td>59.0 / 93.3</td>
</tr>
<tr>
<td>(5) ledger</td>
<td>50.6 / 72.2</td>
<td>6.2 / 8.8</td>
<td>70.1 / 97.5</td>
<td>72.5 / 105</td>
</tr>
<tr>
<td>(6) validation (3+4+5)</td>
<td>116 / 169</td>
<td>12.8 / 17.8</td>
<td>156 / 216</td>
<td>199 / 230</td>
</tr>
<tr>
<td>(7) end-to-end (1+2+6)</td>
<td>371 / 542</td>
<td>63 / 94</td>
<td>612 / 805</td>
<td>646 / 813</td>
</tr>
</tbody>
</table>

MINT/SPEND (in ms)

Ordering dominates time

Sub-second tail E2E (tails from initial load / first blocks)
Experiment 3: SSD vs. RAM Disk

RAM disk (tmpfs) on all peers instead of SSD

(only helps ledger phase of validation)

32-vCPU peer sustained ~3870 SPEND tps (+9% vs. SSD)
Experiment 4: Scalability on LAN

20-100 16-vCPU peers in one data center. 10 endorsers, no gossip

Experiment 5: Scalability Over 2 Data Centers

20-90 16-vCPU peers in 2 data centers (Hong Kong & Tokyo)
Ordering service, all 10 endorsers, and clients in Tokyo.
Non-endorsers in HK
Experiments 4/5: Scalability

Non-endorsing peer throughput (tps)

Past 30 peers, orderers’ network saturated

Expected LAN drop from orderer network saturation, but IBM Cloud had provisioned higher bandwidth

LAN

2190 SPEND tps @ 90 peers over 2DC

# peers

2DC
Experiment 6: Multiple Data Centers

5 data centers (Tokyo, HK, Melbourne, Sydney, Oslo)
20 peers each. Ordering service, 10 endorsers, and clients in Tokyo

Without gossip: 1 peer/org
With gossip: 10 orgs of 10 peers, 2 orgs per data center
**Experiment 6: Multiple Data Centers**

Gossiping helps recover some of the tps lost in transition to more peers / data centers!

<table>
<thead>
<tr>
<th>netperf to TK [Mbps]</th>
<th>HK</th>
<th>ML</th>
<th>SD</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>peak MINT / SPEND</td>
<td>240</td>
<td>98</td>
<td>108</td>
<td>54</td>
</tr>
<tr>
<td>throughput [tps]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(without gossip)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1914 / 2048</td>
<td>1914 / 2048</td>
<td>1914 / 2048</td>
<td>1839 / 1838</td>
</tr>
<tr>
<td></td>
<td>2553 / 2762</td>
<td>2558 / 2763</td>
<td>2271 / 2409</td>
<td>284 / 20</td>
</tr>
</tbody>
</table>

Sydney had CPU limitations
Heeere’s Mallory!

Transactions committed per second

This graph is Fabric v0.6

Execute-Order-Validate in v1 fixes this ("performs well and... immune")

Infinite-loop chaincodes: 2 tps

Time (s)

## Applications and Use Cases

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreign exchange netting</td>
<td>Private Fabric channel for each pair of institutions; blockchain resolves non-settling trades, data available in ledger</td>
</tr>
<tr>
<td>Enterprise asset management</td>
<td>Track hardware asset life-cycle (mfg., shipping, receiving, customers)</td>
</tr>
<tr>
<td>Global cross-currency payment</td>
<td>Process int’l transactions; blockchain records payments + conditions endorsed by participants. Fabric decides settlement method</td>
</tr>
</tbody>
</table>
Conclusion

Fabric is a distributed operating system for permissioned blockchains.

Key features:

Execute-Order-Verify
Transaction execution separated from consensus
Policy-based endorsement
Thank You!