# Scalable, Resilient, and Configurable Permissioned Blockchain Fabric

Sajjad Rahnama, Suyash Gupta, Thamir M. Qadah[‡], Jelle Hellings, Mohammad Sadoghi
Moka Blox LLC
Exploratory Systems Lab
[‡]Purdue University
University of California, Davis

## ABSTRACT

With the advent of Bitcoin, the interest of the database community in blockchain systems has steadily grown. Many existing blockchain applications use blockchains as a platform for monetary transactions, however. We deviate from this philosophy and present ResilientDB, which can serve in a suite of non-monetary data-processing blockchain applications. Our ResilientDB uses state-of-the-art technologies and includes a novel visualization that helps in monitoring the state of the blockchain application.

## 1. INTRODUCTION

With the advent of Bitcoin [20], the interest of the distributed and database community has steadily grown towards blockchain applications. A blockchain in its simplest form is an immutable ledger. Initial blockchain applications envisioned blockchain as a platform for monetary transactions [20, 21]. The key aim of these applications is to provide a monetary unit of exchange, *a cryptocurrency*, which can be covertly exchanged between two or more parties. This covert exchange requires these applications to allow any user to participate (*permissionless*) and hide identities of the participants. As a result, these applications suffer from low throughputs (around 10 txn/s), high latencies, and face several vulnerabilities [7, 8].

Evidently, permissionless applications do not satisfy the needs of secure industry-grade applications. This led to the rise of *permissioned* blockchain applications, which require the identity of each participant to be known a priori. At the core of any permissioned blockchain application is a *Byzantine Fault-Tolerant* (Bft) consensus protocol that helps to achieve a single order of transactions among the participants [5, 10, 11, 12, 15, 16, 22, 18].

Contrary to their potential, existing permissioned blockchain systems are limited by their design choices as they (i) require exchange of a monetary transaction [2, 19]; (ii) adhere to a specific order-execute [4, 9] or execute-order design paradigm [2]; (iii) achieve very low throughput and do not scale beyond a small set of replicas replicas [1, 2, 7, 9]; (iv) are in-flexible in their choice of Bft protocol [1, 4, 19]; and (v) lack support for introducing thread parallelism and task pipelining [1, 2, 3, 4, 9].

In this demo, we introduce ResilientDB, a high-throughput permissioned blockchain fabric [13, 14]. ResilientDB[1] is a modular open-source system for deploying and operating permissioned blockchain applications. With ResilientDB, we envision data-management beyond mere cryptocurrencies via *Blockchain-as-a-Service*. ResilientDB enables developers with the power to implement and test any Bft protocol. Moreover, our fabric, which is built from scratch, employs state-of-the-art software engineering principles, supports a lean design, and eases application deployment.

ResilientDB includes multi-threaded deep pipelines that allow it to achieve high-throughput consensus among its replicas. Any developer can easily configure the number of threads and stages of the pipeline. ResilientDB also allows application developers to test their applications using both YCSB [6] transactions and Smart Contracts. Further, ResilientDB provides its users with a Graphical User Interface (GUI) to compile, deploy, and run the platform and analyze the results, all with just one click. This GUI is designed to tightly monitor the performance of each individual replica.

Despite these features of ResilientDB, our focus remains at designing a high-performance permissioned blockchain fabric. We claim that such a design is possible by adopting a *system-centric* view rather than the *protocol-centric* view employed by existing fabrics. Our system-centric view is motivated twofold: (i) no one Bft protocol fits all settings, and (ii) there is much more to a blockchain system than just its Bft protocol. To illustrate the impact of our system-centric view, we compare in Figure 1 the throughput of ResilientDB employing the PBFT [5] consensus protocol against a permissioned blockchain that uses the Zyzzyva [18] consensus protocol and adopts practices suggested in BFTSmart [3]. Note that PBFT requires three phases, of which two phases necessitate quadratic communication among the replicas, while Zyzzyva only requires a single linear phase. Despite using the costlier PBFT consensus protocol, ResilientDB outperforms the other system. We discuss reasons for the high throughput of ResilientDB in Section 2.

## 2. ARCHITECTURE

In this section, we present the architecture and capabilities of our ResilientDB fabric. ResilientDB is written entirely in C++ and provides a GUI to ease user interaction with the system. Further, we also provide a *Dockerized* deployment that allows any user to experience and test the ResilientDB fabric (comprising of multiple replicas and clients)

[1]ResilientDB is open-sourced at https://resilientdb.com and code is available at https://github.com/resilientdb.

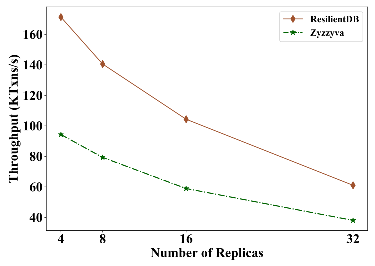


Figure 1: Two permissioned applications employing distinct BFT protocols (80 k clients per experiment).

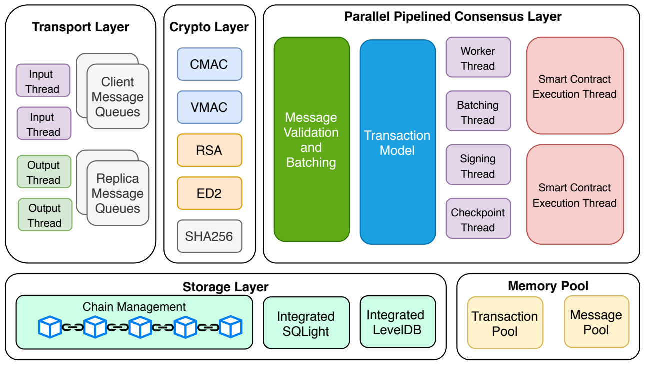


Figure 2: Architecture of the ResilientDB fabric.

on its local machine. In Figure 2, we illustrate the overall architecture, which we describe in detail next.

- **Transport Layer.** Permissioned blockchains use *communication-intensive* BFT consensus protocols. Hence, the need for an efficient transport layer to facilitate exchange of messages between replicas.[2] ResilientDB employs Nanomsg sockets to facilitate communication among replicas and clients via TCP or UDP (depending on the choice of the developer). We also provide support for fast *RDMA* communication for replicas with RDMA capabilities.

To facilitate efficient communication, ResilientDB employs multiple input/output threads with dedicated sockets. Note that the number of input/output threads can be readily adjusted based on the network requirements and buffering bottlenecks. ResilientDB also provides access to distinct message queues. Depending on the type of a message, these queues can be used by different threads to communicate with each other and to place the message on the network.

- **Crypto Layer.** Blockchains typically are designed to deal with malicious adversaries. To secure communication and prevent message tampering, ResilientDB employs NIST-recommended cryptographic constructs from the *Crypto++* library. Depending on specific needs, replicas and clients can digitally sign their messages using either (i) asymmetric-key cryptographic schemes such as ED25519 or RSA; or (ii) symmetric-key cryptographic schemes such as CMAC and AES [17]. ResilientDB also provides message digests via either SHA256 or SHA3 hashes.

- **Parallel Pipelined Consensus Layer.** At the core of any permissioned blockchain application lies a BFT consensus protocol that safely replicates client transactions among all replicas. Decades of research has brought forth several such protocols. No one protocol is the best-fit in all environments, however. For example, ZYZZYVA achieves high throughput if none of the replicas are faulty, HOTSTUFF [22] works well if latency is not critical, GeoBFT [13] scales well when replicas are geographically distant, and PBFT, although typically-considered too slow, is most robust against failures. These characteristics of existing BFT protocols permit us to conclude that any resilient permissioned blockchain fabric should *facilitate testing and implementation of different* BFT *protocols.* ResilientDB's consensus layer allows this and to support this claim we provide implementation of all of the aforementioned protocols (among many others).

Furthermore, as we argued in Section 1, there is more to a blockchain system than just its BFT protocol. In specific, we showed that a permissioned blockchain fabric adopting a system-centric design and employing a slow BFT protocol outperforms a protocol-centric fabric that uses a fast protocol. To yield such a system-centric design, ResilientDB employs transaction batching, multi-threading, pipelining, out-of-order processing, and memory pooling.

***Batching.*** Prior works have batched client transactions to reduce the cost of consensus [5, 18]. We permit batching of transactions at both clients and replicas and developers can specify any size for such batches. Batching reduces both communication costs and computation costs by reducing the number of messages that are exchanged (which also reduced the number of message signatures necessary).

***Transactions and Smart Contracts.*** ResilientDB supports YCSB transactions and customized Smart Contracts. YCSB transactions can be used for benchmarking performance and developers can easily vary the skew (read/write percentage) of these transaction. ResilientDB also provides APIs for designing and testing Smart Contracts, which are similar to stored procedures in databases [7]. To demonstrate this, we implemented Ethereum's account-based smart contracts for banking applications [21] (see Section 3.2).

We associate each transaction with a *transaction manager* that manages the resources required for handling transactions. We provide fast lookup of transaction managers via indices on transaction identifier and batch identifier. Furthermore, transaction managers are pooled and reused to save on allocation and deallocation costs.

***Order-Execute vs. Execute-Order Paradigm.*** Traditional permissioned blockchain systems employ the order-execute paradigm, which states that a transaction needs to be ordered across all replicas prior to its execution [4, 9]. This is in contrast with the execute-order paradigm proposed by Hyperledger [2], which advocates to first execute and then order the transaction. Both of these paradigms have their pros and cons. Our ResilientDB fabric provides support for both paradigms and allows developers to select the paradigm that best fits their applications.

***Multi-Threaded Deep Pipeline.*** As stated before, permissioned blockchain systems are communication-intensive. Hence, we ensure that our ResilientDB fabric is not under-utilizing hardware and will only be limited by network capacity. To do so, we designed the consensus protocols in ResilientDB such that the critical path is as simple as possible and all other tasks are split of in their separate threads. E.g., threads to deal with message sending, with message receiving, signing messages, verifying signatures, creating transaction batches, performing checkpoints, and executing

[2]Permissionless systems are compute-intensive as they run consensus by solving a complex cryptographic puzzle.

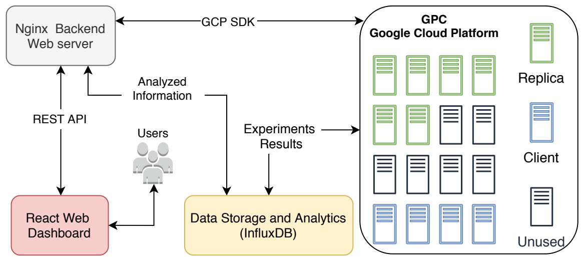


Figure 3: Demo Platform Architecture.

| Parameter | Options | |
|---|---|---|
| BFT Protocol | PBFT, Zyzzyva, GeoBFT, HotStuff | |
| Transactions | YCSB, Banking Smart Contracts | |
| Requests/Txn | 1, 5, 15, 50 | |
| Batch Size | 1, 10, 100, 500, 1000, 4000 | |
| Message Size | 0 kB, 100 kB, 200 kB, 400 kB | |
| Pipeline | Enable or Disable | |
| Threads | I/O, Worker, Execute Sign, Checkpoint | |
| Storage | In-memory, SQLite, LevelDB | |
| Signatures | Disable | |
| | Only Asymmetric | ED25519, RSA |
| | Only Symmetric | CMAC, VMAC |
| | Mix | Use both |
| Hash Schemes | SHA256, SHA3 | |

Figure 4: Parameters to mix-and-match in ResilientDB.

transactions. Users can easily adjust the number of required threads depending on the specific needs of their applications.

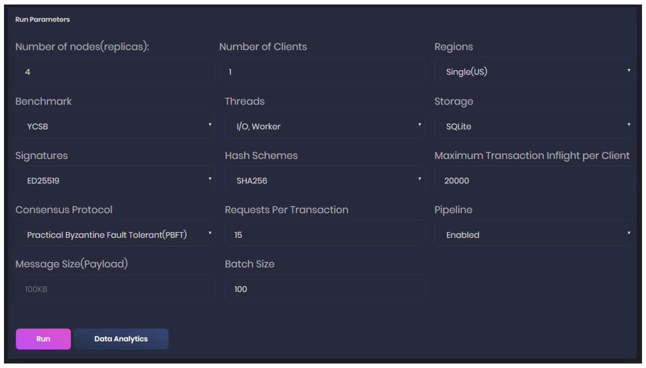
• **Memory Pool.** Blockchain systems that process thousands of transactions, smart contracts, and messages per second require high-performance management of memory resources. For ResilientDB's memory management, we employ Jemalloc. Further, we minimize memory allocation and deallocation by using distinct memory pools for messages, transaction managers, and smart contracts. Depending on the size of an allocation, each thread accesses the required pool and fetches an unallocated memory object. At the time of garbage collection, obsolete objects are marked as released and placed back in the respective memory pool for reuse.

• **Chain Storage.** ResilientDB provides support for secure ledger (blockchain) management. To enable efficient execution of client transactions, we also support efficient read and write accesses to client records. In specific, each replica can use popular databases such as SQLite and LevelDB to achieve data persistence for the ledger and client records under failure. ResilientDB provides several simple APIs that allow developers to read and write to these databases and modify their schema if necessary.

## 3. DEMONSTRATION SCENARIO

During our demonstration, each user will get access to a graphical web-based interface of ResilientDB. Figure 3 illustrates the architecture of our demonstration environment. We provide a web-based UI for specifying experiment parameters, for monitoring the real-time throughput and latency of the system, and for the analysis of collected data.

In specific, users can specify their choice of parameters on our React Web Dashboard, which uses REST APIs to forward these parameters to our Nginx back end. The back end compiles the code and deploys the executables on the Google Cloud Platform (GCP). Once the executables start


Run Parameters

Number of nodes(replicas): 4
Number of Clients: 1
Regions: Single(US)
Benchmark: YCSB
Threads: I/O, Worker
Storage: SQLite
Signatures: ED25519
Hash Schemes: SHA256
Maximum Transaction Inflight per Client: 20000
Consensus Protocol: Practical Byzantine Fault Tolerant(PBFT)
Requests Per Transaction: 15
Pipeline: Enabled
Message Size(Payload): 100KB
Batch Size: 100

Figure 5: The interactive WebUI dashboard.

running, any emitted result is continuously stored in InfluxDB. Throughout this process, our dashboard shows the user the current state of the system. If the user wants to visualize the ongoing results, our dashboard asks the back end to fetch the data from InfluxDB and plot the required graphs. This allows us to show the user real-time system throughput and latency metrics. We employ React (open-source), Nginx (performance), and InfluxDB (eases management of time-series data) for their associated advantages.

We provide our users access to *two* demonstration scenarios. The first demonstration scenario focuses on making users understand the different parameters that affect the performance of a blockchain fabric. The second demonstration scenario allows interested users to create and deploy their own smart contracts on-the-fly. We explain these next.

### 3.1 Mix-and-Match

The key ***takeaway*** of the mix-and-match demonstration is to make users experiment and observe the different parameters that affect throughput (transactions per second) and latency (time from the client request to the response) of a permissioned blockchain application. We give users a GUI (see Figure 5) and ask them to *mix-and-match* the parameters listed in Figure 4.

We first require the user to ***Sign-up/Sign-in*** to our ResilientDB portal. Next, the user can ***Configure*** experiments of its choice. To do so, the user first selects a ***BFT*** protocol. At present, we have already implemented four state-of-the-art protocols. Next, the user decides whether it wants clients to send YCSB transactions or to run Banking Smart Contracts. The user can set the number of *requests* each client includes in its transactions and the *size of the batch* (if batching is employed by replicas). To test the limit of the network, we also provide capability to add a *predefined load* to messages.

We allow our users to select whether they want to *enable or disable pipelining*. Enabling pipelining in ResilientDB allows the various phases of a BFT protocol to be executed in parallel. For example, PBFT is a three-phase *preprepare-prepare-commit* protocol. If we pipeline PBFT, then one transaction is *prepared* while previous transactions are *committed* and *executed*. To ensure safety, the ordering is delayed until execution [5, 13]. In similar ways, the phases of other protocols can be pipelined. As stated earlier, ResilientDB also divides tasks across threads. We allow users to *choose the number of threads* needed to create batches, to sign messages, to fetch data from the network and to place

```
/* return: 1 for commit, 0 for abort */

int TransferMoney::execute()
{
  int source_bal = db->Get(this->source);
  int dest_bal = db->Get(this->dest);
  if (this->amount <= source) {
    db->Set(this->source, source_bal - amount);
    db->Set(this->dest, dest_bal + amount);
    return 1;
  }
  return 0;
}
```

Figure 6: Transfer Smart Contract in ResilientDB.

output on the network. Further, we allow users to select the *type of storage* for their blockchain ledger and client records. At present, we support the in-memory databases SQLite and LevelDB for storing the ledger and client records. Finally, users can decide the type of *cryptographic constructs*, signatures and digests, they want to employ. Note that a user need not specify all parameters. In such a case, the system will proceed with the *default parameters*.

Finally, the user can deploy the experiment via the ***Run*** button, which initiates the script that will compile, deploy, and run the experiment. The user is presented with a *webpage* that tracks the experiment progress. The user also has an option of *monitoring* the results in real-time. Once the experiment completes, the user can query the InfluxDB database holding all results.

### 3.2 Deploying Smart-Contract

The key takeaway of the *Deploying Smart-Contract* demonstration is to show how users can design their own applications in ResilientDB. We believe that demonstrating users the ease with which they can use ResilientDB to create new applications illustrates ResilientDB's general applicability.

Say we want to design a banking application. The *transfer* transaction is a key utility, as it allows movement of money from one account to another. To support transfers, we create a smart contract that allows a user *Bob* to transfer an amount $X$ from his account to the account of *Alice* (see Figure 6). Prior to transferring $X$, the smart contract also needs to check if *Bob* (`source`) has at least amount $X$ (`source_bal`) in his account. The smart contract needs access to the database with client records, for which we use `GET` and `PUT` APIs.

We provide a *base class* (`SmartContract`) that developers can inherit to define their functionalities. Further, the client needs to provide the required parameters for the new smart contract. For example, the client specifies the source, destination and the amount. Note that these changes do not affect the process of compiling, deploying, and running the code, which can still be done through our GUI. Hence, with simple changes, users can build their own applications using our ResilientDB fabric.

## 4. CONCLUSIONS

In this demonstration, we present ResilientDB, a high-throughput permissioned blockchain fabric. The key aim of this demonstration is to make users understand the different parameters that affect the throughput of a blockchain fabric. We allow users to mix-and-match different parameters and illustrate how easy it is for users to make their applications around ResilientDB.

**Acknowledgments.** This work is partially supported by the U.S. Department of Energy, Office of Science, Office of Small Business Innovation Research, under Award Number DE-SC0020455.

## 5. REFERENCES

[1] M. J. Amiri, D. Agrawal, and A. E. Abbadi. CAPER: A cross-application permissioned blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, 2019.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 30:1–30:15. ACM, 2018.

[3] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[4] E. Buchman, J. Kwon, and Z. Milosevic. Revisiting fast practical byzantine fault tolerance, 2018.

[5] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154. ACM, 2010.

[7] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Trans. Knowl. Data Eng.*, 30(7):1366–1385, 2018.

[8] Eyal, A. E. Gencer, E. G. Sirer, and R. V. Renesse. Bitcoin-NG: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation*, pages 45–59, Santa Clara, CA, 2016. USENIX Association.

[9] G. Greenspan. Multichain private blockchain, 2015.

[10] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. abs/1911.00838, 2019.

[11] S. Gupta, J. Hellings, and M. Sadoghi. Brief announcement: Revisiting consensus protocols through wait-free parallelization. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146, pages 44:1–44:3, 2019.

[12] S. Gupta, J. Hellings, and M. Sadoghi. Scaling blockchain databases through parallel resilient consensus paradigm. abs/1911.00837, 2019.

[13] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. ResilientDB: Global scale resilient blockchain fabric. *Proceedings of the VLDB Endowment*, 13(6):868–883, 2020.

[14] S. Gupta, S. Rahnama, and M. Sadoghi. Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. In *40th International Conference on Distributed Computing Systems*. IEEE, 2020.

[15] J. Hellings and M. Sadoghi. Brief announcement: The fault-tolerant cluster-sending problem. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146, pages 45:1–45:3, 2019.

[16] J. Hellings and M. Sadoghi. Coordination-free byzantine replication with minimal communication costs. In *Proceedings of the 23rd International Conference on Database Theory*, volume 155, 2020.

[17] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2nd edition, 2014.

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39, 2009.

[19] Libra Association Members. An introduction to libra, 2019.

[20] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[21] G. Wood. Ethereum: a secure decentralised generalised transaction ledger, 2016. EIP-150 revision.

[22] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019.