We will explore Kadena, Ripple, Stellar, Quorum and various other blockchains in the section.

Kadena

Kadena is a private blockchain that has successfully addressed scalability and privacy issues in blockchain systems. A new Turing incomplete language, called Pact, has also been introduced with Kadena that allows the development of smart contracts. A key innovation in Kadena is its Scalable BFT consensus algorithm, which has the potential to scale to thousands of nodes without performance degradation.

Scalable BFT is based on the original Raft algorithm and is a successor of Tangaroa and Juno. Tangaroa, which is a name given to an implementation of Raft with fault tolerance (a BFT Raft), was developed to address the availability and safety issues that arose from the behavior of Byzantine nodes in the Raft algorithm, and Juno was a fork of Tangaroa that was developed by JPMorgan. Consensus algorithms are discussed in Chapter 2, *Blockchain 101* in more detail.

Both of these proposals have a fundamental limitation—they cannot scale while maintaining a high level of high performance. As such, Juno could not gain much traction. Private blockchains have the more desirable property of maintaining high performance as the number of nodes increase, but the aforementioned proposals lack this feature. Kadena solves this issue with its proprietary Scalable BFT algorithm, which is expected to scale up to thousands of nodes without any performance degradation.

Moreover, confidentiality is another significant aspect of Kadena that enables privacy of transactions on the blockchain. This security service is achieved by using a combination of key rotation, symmetric on-chain encryption, incremental hashing, and Double Ratchet protocol.

Key rotation is used as a standard mechanism to ensure the security of the private blockchain. It is used as a best practice to thwart any attacks if the keys have been compromised, by periodically changing the encryption keys. There is native support for key rotation in Pact smart contract language.

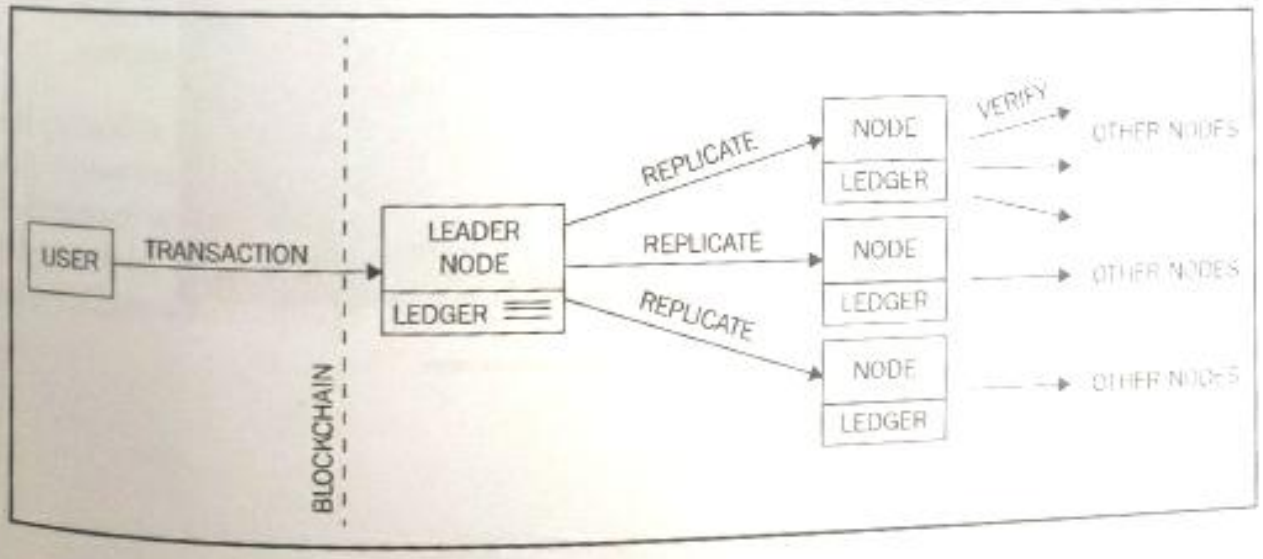
Symmetric on-chain encryption allows encryption of transaction data on the blockchain. These transactions can be automatically decrypted by the participants of a particular private transaction. Double Ratchet protocol is used to provide key management and encryption functions.

Scalable BFT consensus protocol ensures that adequate replication and consensus has been achieved before smart contract execution. The consensus is achieved by following the process described here.

This is how a transaction originates and flows in the network:

1. First, a new transaction is signed by the user and broadcasted over the blockchain network, which is picked up by a leader node that adds it to its immutable log. At this point, an incremental hash is also calculated for the log. Incremental hash is a type of hash function that allows computation of hash messages in the scenario where, if a previous original message which is already hashed is slightly changed, then the new hash message is computed from the already existing hash. This scheme is quicker and less resource intensive compared to a conventional hash function where an altogether new hash message is required to be generated even if the original message has only changed very slightly.
2. Once the transaction is written to the log by the leader node, it signs the replication and incremental hash and broadcasts it to other nodes.
3. Other nodes after receiving the transaction, verify the signature of the leader node, add the transaction into their own logs, and broadcast their own calculated incremental hashes (quorum proofs) to other nodes. Finally, the transaction is committed to the ledger permanently after an adequate number of proofs are received from other nodes.

A simplified version of this process is shown in the following diagram, where the leader node is recording the new transactions and then replicating them to the follower nodes:



Consensus mechanism in Kadena

Once the consensus is achieved, a smart contract execution can start and takes a number of steps, as follows:

1. First, the signature of the message is verified.
2. Pact smart contract layer takes over.
3. Pact code is compiled.
4. The transaction is initiated and executes any business logic embedded within the smart contract. In case of any failures, an immediate rollback is initiated that reverts that state back to what it was before the execution started.
5. Finally, the transaction completes and relevant logs are updated.

Pact has been open sourced by Kadena and is available for download at `http://kadena.io/pact/downloads.html`.

This can be downloaded as a standalone binary that provides a REPL for Pact language. An example is shown here where Pact is run by issuing the `./pact` command in Linux console:

```
drequinox@drequinox-OP7010:~/Downloads$ ./pact
pact> 1234
1234
pact> (+ 1 2)
3
pact> (if (= (+ 1 2) 3 "OK" "ERROR")
(interactive):1:31: error: unexpected
    EOF, expected: ")", ";", "{",
    Boolean false, Boolean true,
    Decimal literal, Integer literal,
    String literal, Symbol literal,
    list literal, pact, sexp, space
(if (= (+ 1 2) 3 "OK" "ERROR")
                              ^
pact> (if (= (+ 1 2) 3) "OK" "ERROR")
"OK"
pact>
```

Pact REPL, showing sample commands and error output

A smart contract in Pact language is usually composed of three sections: keysets, modules, and tables. These sections are described here:

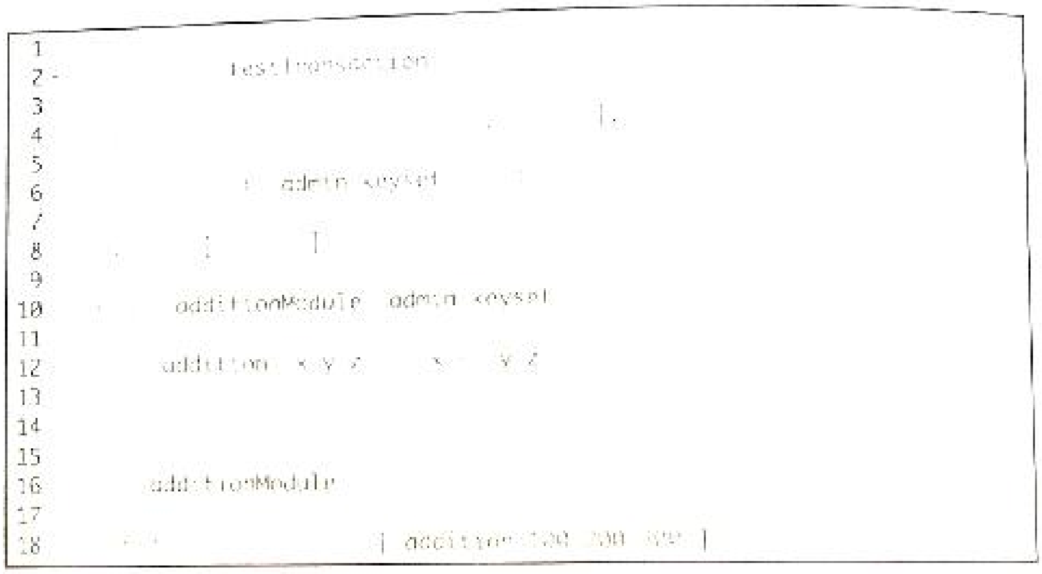
- **Keysets**: This section defines relevant authorization schemes for tables and modules.
- **Modules**: This section defines the smart contract code encompassing the business logic in the form of functions and pacts. Pacts within modules are composed of multiple steps and are executed sequentially.
- **Tables:** This section is an access-controlled construct defined within modules. Only administrators defined in the admin keyset have direct access to this table. Code within the module is granted full access, by default to the tables.

Pact also allows several execution modes. These modes include contract definition, transaction execution, and querying. These execution modes are described here:

- **Contract definition:** This mode allows a contract to be created on the blockchain via a single transaction message.
- **Transaction execution:** This mode entails the execution of modules of smart contract code that represent business logic.
- **Querying**: This mode is concerned with simply probing the contract for data and is executed locally on the nodes for performance reason. Pact uses LISP-like syntax and represents in the code exactly what will be executed on the blockchain, as it is stored on the blockchain in human-readable format. This is in contrast to Ethereum's EVM, which compiles into bytecode for execution, which makes it difficult to verify what code is in execution on the blockchain. Moreover, it is Turing incomplete, supports immutable variables, and does not allow null values, which improves the overall safety of the transaction code execution.

It is not possible to cover the complete syntax and functions of Pact in this limited length chapter; however, a small example is shown here, that shows the general structure of a smart contract written in Pact. This example shows a simple addition module that defines a function named `addition` that takes three parameters. When the code is executed it adds all three values and displays the result.

The following example has been developed using the online Pact compiler available at [illegible]



Sample Pact code

When the code is run, it produces the output shown as follows:

```
Begin Tx Just 1
testTransaction
Setting transaction data
Keyset defined
Setting transaction keys
Loaded module "additionModule"
  , hash "eaf647f843b2e88b5009253fe4ecca6f8890a646da76b4
Commit Tx Just 1
Using "additionModule"
Result : 600
```

The output of the code

As shown in the preceding example, the execution output matches exactly with the code layout and structure, which allows for greater transparency and limits the possibility of malicious code execution.

Kadena is a new class of blockchains introducing the novel concept of **pervasive determinism** where, in addition to standard public/private key-based data origin security, an additional layer of fully deterministic consensus is also provided. It provides cryptographic security at all layers of the blockchain including transactions and consensus layer.

Relevant documentation and source code for Pact can be found here `https://github.com/kadena-io/pact`.

Kadena has also introduced a public blockchain in January, 2018 which is another leap forward in building blockchains with massive throughput. The novel idea in this proposal is to build a PoW parallel chain architecture. This scheme works by combining individually mined chains on peers into a single network. The result is massive throughput capable of processing more than 10,000 transactions per second.

The original research paper is available at `http://kadena.io/docs/chainweb-v15.pdf`.

Ripple

Introduced in 2012, Ripple is a currency exchange and real-time gross settlement system. In Ripple, the payments are settled without any waiting as opposed to traditional settlement networks, where it can take days for settlement.

It has a native currency called **Ripples (XRP)**. It also supports non-XRP payments. This system is considered similar to an old traditional money transfer mechanism known as *Hawala*. This system works by making use of agents who take the money and a password from the sender, then contact the payee's agent and instruct them to release funds to the person who can provide the password. The payee then contacts the local agent, tells them the password and collects the funds. An analogy to the agent is gateway in Ripple. This is just a very simple analogy; the actual protocol is rather complex but principally it is the same.

The Ripple network is composed of various nodes that can perform different functions based on their type:

- **User nodes**: These nodes use in payment transactions and can pay or receive payments.
- **Validator nodes**: These nodes participate in the consensus mechanism. Each server maintains a set of unique nodes, which it needs to query while achieving consensus. Nodes in the **Unique Node List** (**UNL**) are trusted by the server involved in the consensus mechanism and will accept votes only from this list of unique nodes.

Ripple is sometimes not considered a truly decentralized network as there are network operators and regulators involved. However, it can be considered decentralized due to the fact that anyone can become part of the network by running a validator node. Moreover, the consensus process is also decentralized because any changes proposed to make on the ledger have to be decided by following a scheme of super majority voting. However, this is a hot topic among researchers and enthusiasts and there are arguments against and in favor of each school of thought. There are some discussions online that readers can refer to for further exploration of these ideas.

You can find these online discussions at the following links:

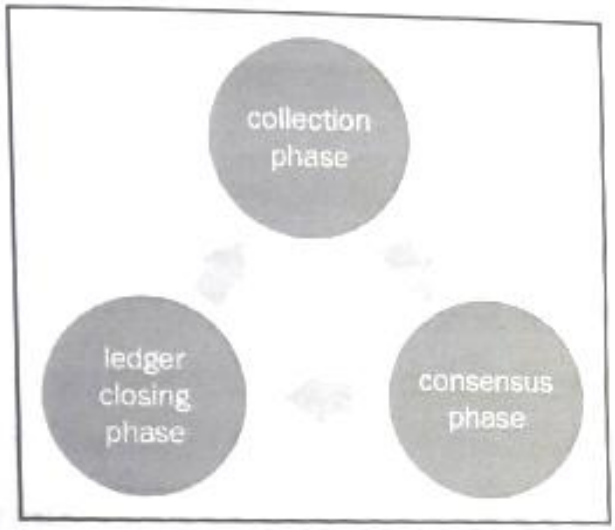
- `https://www.quora.com/Why-is-Ripple-centralized`
- `https://thenextweb.com/hardfork/2018/02/06/ripple-report-bitmex-centralized/`
- `https://www.reddit.com/r/Ripple/comments/6c8j7b/is_ripple_centralized_and_other_related_questions/?st=jewkor7bamp;sh=e39bc635`

Ripple maintains a globally distributed ledger of all transactions that are governed by a novel low-latency consensus algorithm called **Ripple Protocol Consensus Algorithm** (**RPCA**). The consensus process works by achieving an agreement on the state of an open ledger containing transactions by seeking verification and acceptance from validating servers in an iterative manner until an adequate number of votes are achieved. Once enough votes are received (a super majority, initially 50% and gradually increasing with each iteration up to at least 80%) the changes are validated and the ledger is closed. At this point, an alert is sent to the whole network indicating that the ledger is closed.

Original research paper for RPCA is available at `https://ripple.com/files/ripple_consensus_whitepaper.pdf`.

In summary, the consensus protocol is a three-phase process:

- **Collection phase**: In this phase validating nodes gather all transactions broadcasted on the network by account owners and validate them. Transactions, once accepted, are called candidate transactions and can be accepted or rejected based on the validation criteria.
- **Consensus phase**: After the collection phase the consensus process starts, and after achieving it the ledger is **closed**.
- **Ledger closing phase**: This process runs asynchronously every few seconds in rounds and, as result, the ledger is opened and closed (updated) accordingly:



Ripple consensus protocol phases

In a Ripple network, there are a number of components that work together in order to achieve consensus and form a payment network. These components are discussed individually here:

- **Server**: This component serves as a participant in the consensus protocol. Ripple server software is required in order to be able to participate in consensus protocol.
- **Ledger**: This is the main record of balances of all accounts on the network. A ledger contains various elements such as ledger number, account settings, transactions, timestamp, and a flag that indicates the validity of the ledger.

- **Last closed ledger**: A ledger is closed once consensus is achieved by validating nodes.
- **Open ledger**: This is a ledger that has not been validated yet and no consensus has been reached about its state. Each node has its own open ledger, which contains proposed transactions.
- **Unique Node List:** This is a list of unique trusted nodes that a validating server uses in order to seek votes and subsequent consensus.
- **Proposer**: As the name suggests, this component proposes new transactions to be included in the consensus process. It is usually a subset of nodes (UNL defined in the previous point) that can propose transactions to the validating server.

Storj

Existing models for cloud-based storage are all centralized solutions, which may or may not be as secure as users expect them to be. There is a need to have a cloud storage system that is secure, highly available, and above all decentralized. Storj aims to provide blockchain based, decentralized, and distributed storage. It is a cloud shared by the community instead of a central organization. It allows execution of storage contracts between nodes that act as autonomous agents. These agents (nodes) execute various functions such as data transfer, validation, and perform data integrity checks.

The core concept is based on **Distributed Hash Tables (DHTs)** called **Kademlia**, however this protocol has been enhanced by adding new message types and functionalities in Storj. It also implements a peer to peer **publish/subscribe (pub/sub)** mechanism known as **Quasar**, which ensures that messages successfully reach the nodes that are interested in storage contracts. This is achieved via a bloom filter-based storage contract parameters selection mechanism called **topics**.

Storj stores files in an encrypted format spread across the network. Before the file is stored on the network, it is encrypted using AES-256-CTR symmetric encryption and is then stored piece by piece in a distributed manner on the network. This process of dissecting the file into pieces is called **sharding** and results in increased availability, security, performance, and privacy of the network. Also, if a node fails the shard is still available because by default a single shard is stored at three different locations on the network.

It maintains a blockchain, which serves as a shared ledger and implements standard security features such as public/private key cryptography and hash functions similar to any other blockchain. As the system is based on hard drive sharing between peers, anyone can contribute by sharing their extra space on the drive and get paid with Storj's own cryptocurrency called **Storjcoin X (SJCX)**. SJCX was developed as a *Counterparty* asset and makes use of Counterparty (Bitcoin blockchain based) for transactions. This has been migrated to Ethereum now.

A detailed discussion is available at `https://blog.storj.io/post/158740607128/migration-from-counterparty-to-ethereum`.
Storj code is available at `https://github.com/Storj/`.

BigchainDB

This is a scalable blockchain database. It is not strictly a blockchain itself but complements blockchain technology by providing a decentralized database. At its core it's a distributed database but with the added attributes of a blockchain such as decentralization, immutability, and handling of digital assets. It also allows usage of NoSQL for querying the database.

It is intended to provide a database in a decentralized ecosystem where not only processing is decentralized (blockchain) or the filesystem is decentralized (for example, IPFS) but the database is also decentralized. This makes the whole application ecosystem decentralized.

This is available at `https://www.bigchaindb.com/`.

Platforms and frameworks

This section covers various platforms that have been developed to enhance the experience of existing blockchain solutions.

Eris

Eris is not a single blockchain, it is an open modular platform developed by Monax for development of blockchain-based ecosystem applications. It offers various frameworks, SDKs, and tools that allow accelerated development and deployment of blockchain applications.

The core idea behind the Eris application platform is to enable development and management of ecosystem applications with a blockchain backend. It allows integration with multiple blockchains and enables various third-party systems to interact with various other systems.

This platform makes use of smart contracts written in Solidity language. It can interact with blockchains such as Ethereum or Bitcoin. The interaction can include connectivity commands, start, stop, disconnection, and creation of new blockchains. Complexity related to setup and interaction with blockchains have been abstracted away in Eris. All commands are standardized for different blockchains, and the same commands can be used across the platform regardless of the blockchain type being targeted.

An ecosystem application can consist the Eris platform, enabling the API gateway to allow legacy applications to connect to key management systems, consensus engines, and application engines. The Eris platform provides various toolkits that are used to provide various services to the developers. These modules are described as follows:

- **Chains**: This allows the creation of and interaction with blockchains.
- **Packages**: This allows the development of smart contracts.
- **Keys**: This is used for key management and signing operations.
- **Files**: This allows working with distributed data management systems. It can be used to interact with filesystems such as IPFS and data lakes.
- **Services**: This exposes a set of services that allows the management and integration of ecosystem applications.

Several SDKs has also been developed by Eris that allow the development and management of ecosystem applications. These SDKs contain smart contracts that have been fully tested and address specific needs and requirements of business. For example, a finance SDK, insurance SDK, and logistics SDK. There is also a base SDK that serves as a basic development kit to manage the life cycle of an ecosystem application.

Monax has developed its own permissioned blockchain client called `eris:db`. It is a PoS-based blockchain system that allows integration with a number of different blockchain networks. The `eris:db` client consists of four components:

- **Consensus**: This is based on the Tendermint consensus mechanism, discussed before
- **Virtual machine**: Eris uses EVM, as such it supports Solidity compiled contracts
- **Permissions layer**: Being a permissioned ledger, Eris provides an access control mechanism that can be used to assign specific roles to different entities on the network

- **Interface**: This provides various command-line tools and RPC interfaces to enable interaction with the backend blockchain network

The key difference between Ethereum blockchain and `eris:db` is that `eris:db` makes use of a **Practical Byzantine Fault-Tolerance** (**PBFT**) algorithm, which is implemented as a deposit-based Proof of Stake (DPOS system) whereas Ethereum uses PoW. Moreover, `eris:db` uses the ECDSA `ed22519` curve scheme whereas Ethereum uses the `secp256k1` algorithm. Finally, it is permissioned with an access control layer on top whereas Ethereum is a public blockchain.

Eris is a feature-rich application platform that offers a large selection of toolkits and services to develop blockchain-based applications.

It is available at `https://monax.io/`.

Notable projects

Following is a list of notable projects in the blockchain space that is currently in progress. In addition to these projects, there is also a myriad of start-ups and companies working in the blockchain space and offering blockchain-related products.

Zcash on Ethereum

A recent project by the Ethereum R&D team is the implementation of Zcash on Ethereum. This is an exciting project whereby developers are trying to create a privacy layer for Ethereum using ZK-SNARKs already used in Zcash project. With Zcash implementation on Ethereum, the aim is to create a platform that allows applications such as voting where privacy is of paramount importance.

It will also allow the creation of anonymous tokens on Ethereum that can be used in a number of applications.

Bitcoin-NG

This is another proposal for addressing scalability, throughput, and speed issues in the Bitcoin blockchain. **Next Generation** (**NG**) protocol is based on a mechanism of leader election which verifies transactions as soon as they occur, as compared to Bitcoin's protocols, where the time between blocks and block size are the key limitations concerning scalability.

Bletchley

This project has been introduced by Microsoft, indicating the commitment by Microsoft to blockchain technology. Bletchley allows the use of Azure cloud services to build blockchains in a user-friendly manner. A major concept introduced by Bletchley is called cryptlets, which can be thought of like an advanced version of Oracles that reside outside the blockchain and can be called by smart contracts using secure channels. These can be written in any language and execute within a secure container.

There are two types of cryptlets: **utility cryptlets** and **contract cryptlets**. The first type is used to provide basic services such as encryption and basic data fetching from external sources, whereas the latter is a more intelligent version that is created automatically when a smart contract is created on the chain and resides off the chain but still linked to the on-chain contract. Due to this off-chain existence, there is no need to execute the contract cryptlets on all nodes of the blockchain network. Therefore, this approach results in increased performance of the blockchain.

The whitepaper is available at `https://github.com/Azure/azure-blockchain-projects/blob/master/bletchley/bletchley-whitepaper.md`.