1 From Words to Actions: How Ripple Gets it done

The following post examines the underlying architecture of Ripple. It is carried out by studying how Ripple realizes the different properties mentioned in the previous essay and the principles of its design and evolution. We will look into the architectural views relevant to Ripple, such as development, run time, and deployment views. Furthermore, we will also discuss the non-functional properties of the system.

1.1 Architectural View

To put it plainly, building a suitable architecture for a software project is hard, and you can imagine that it doesn't come easy for a product as large as Ripple. One look at the Rippled GitHub repository gives the idea of how big the ecosystem is. From RippleNet codebase that manages user platform to the XRP codebase that takes care of their ledger, there is a lot of software development going on. So how does one even try to begin to analyze this massive architecture? This post answers that question by leveraging the main ideas of Architectural View, as presented in Rozanski and Woods[1].

To understand the architecture of Ripple, rather than looking at it as a single overloaded model, we will partition it into different components and *"view"* them separately. Partitioning into such views helps us to focus on that particular component, and collectively, in the end, we will get a solid picture of the system as a whole. Now the question is, "What establishes a *"view"* and how can we use them?"

Rozanski and Woods[2] defines Architectural View as

- "A way to portray those aspects or elements of the architecture that are relevant to the concerns the view intends to address—and, by implication, the stakeholders for whom those concerns are important."*

The different stakeholders in considerations are already discussed in the previous post. Hence in this essay, we partition the architecture by viewing it from the perspective of stakeholders such as developers, users, and maintainers. Rather than reinventing the wheel on deciding what should go into each view, we borrow the concept of Viewpoints. Architecture Viewpoint is a collection of patterns, templates, and conventions for constructing one type of view[3]. Hence the viewpoints we will be concerned about are Development View, Run Time, and Deployment View.

1.1.1 Architecture Styles and Patterns

Rather than working on the project as a whole, Ripple implements a Component-based architecture style that divides the project into different sections and works on it individually. More on how they manage it is covered in the Development View section.

1.2 Development View

Kruchten[4] defines Development View as the view that focuses on the organization of the actual software modules in the software development environment. The software is packaged in small chunks, program

[1] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

[2] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

[3] Rozanski, Nick, and Eóin Woods. Software systems architecture: working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2012.

[4] P. B. Kruchten, "The 4+1 View Model of architecture," in IEEE Software, vol. 12, no. 6, pp. 42-50, Nov. 1995.

libraries, or subsystems that can be developed by one or more developers. The subsystems are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it.

Thanks to the excellent management of the developing team, the complexity of Ripple is divided into different segments allowing one to analyze the various components such as front-end design, configurations, software functions, auxiliary functions, testing modules, documents, etc. separately. Here we only select the software function modules in the system decomposition and list them in the form below:

Folder	Descriptions
app	Application of the ripple client
basics	Utility functions and classes. ripple/basic should contain no dependencies on other modules.[5]
beast	A HTTP and WebSocket library Beast [6]
conditions	Ripple configure requirements and runtime requirements
consensus	Implementation of a generic consensus algorithm. [7]
core	Core of the ripple program [8]
crypto	Crypto providing the crypto algorithm
json	Third-party library to do JSON operations. [9]
ledger	Data structure for ripple transaction ledger
net	network components for communications
nodestore	Providing an interface that stores, in a persistent database, a collection of node objects that rippled uses as its primary representation of ledger entries. [10]
overlay	Each connection is represented by a *Peer* object. The Overlay Manager establishes, receives, and maintains links to peers. Protocol messages are exchanged between peers and serialized using [11].
peerfinder	Maintaining and storing addresses for different endpoints and establishing and managing connections inside peers to peers network
proto	Protocol buffers source code. The protocol tool saves the output of the *.proto* files in the build directory.
resource	Working on identifying load balancing between each endpoint. Sharing load information in a cluster, warn and/or disconnect endpoints for imposing load. [12]
rpc	Allowing suspension with continuation. And a default continuation to reschedule the job on the job queue.
server	Ripple's server configurations. It configures Ripple's port and executes Internet sessions. [13]
shamap	A given SHAMap is a Merkle tree or a radix tree storing Transactions and account states.
unity	Providing links and references to other modules.

[5] https://github.com/ripple/rippled/tree/develop/src/ripple
[6] https://github.com/vinniefalco?tab=overview&org=ripple
[7] https://github.com/ripple/rippled/tree/develop/src/ripple
[8] https://github.com/ripple/rippled/tree/develop/src/ripple

Since these modules are created independently, we divide them into the following layers according to their functions:

- Application layer
- Core
- Network layer including server, RPC, overlay, proto, beast, and net.
- Architecture layer, including nodestore, peerfinder, resources, conditions, and JSON.
- Blockchain layer, including consensus, Crypto, shamap, and ledger.

As is shown in the graph below, some of the observed dependency insights are:

- App is the entrance of Ripple's system.
- Core is the key component that schedules other layers.
- Unity is called by core; in fact, it includes all the global library inferences of other modules.
- In the Architecture layer, JSON is also globally used since it serves as a data provider for other components. Resources connect extra functions and libraries outside the software architecture. The data is saved in the nodestore.
- In the Network layer, net, server, RPC are executing net communications. Their online data should meet the protocol defined in the overlay.
- In the Blockchain layer, confidential data hashed by the Crypto is stored in shamap and finally represented by the ledger.

1.3 Run Time View

The figure below shows the Ripple's flow chart during its run time. The whole run time process can be divided into 4 phases:

- The application will pick up the users' actions and put them inside the Core.
- The architecture layer parses data, test application metrics, and stores data.
- The network layer configures the servers to ensure end-to-end remote communication.
- Finally, the blockchain layer encrypts and handles the Bitcoin business.

1.4 Deployment View

1.4.1 Where is Ripple deployed

As a commercial product, Ripple's system is deployed on its own trading network RippleNet. It's a network of banks, payment providers, and others. It can be seen as a cloud data community. Employing Ripple's solutions and a standardized ruleset allows for those connected on RippleNet to send and receive payments around the world efficiently.

1.4.2 How Ripple is deployed

The deployment of Ripple can be divided into different components. The main components deployed in RippleNet are as follows:

[9] https://github.com/ripple/rippled/tree/develop/src/ripple
[10] https://github.com/ripple/rippled/tree/develop/src/ripple
[11] https://developers.google.com/protocol-buffers/
[12] https://github.com/ripple/rippled/tree/develop/src/ripple
[13] https://github.com/ripple/rippled/tree/develop/src/ripple

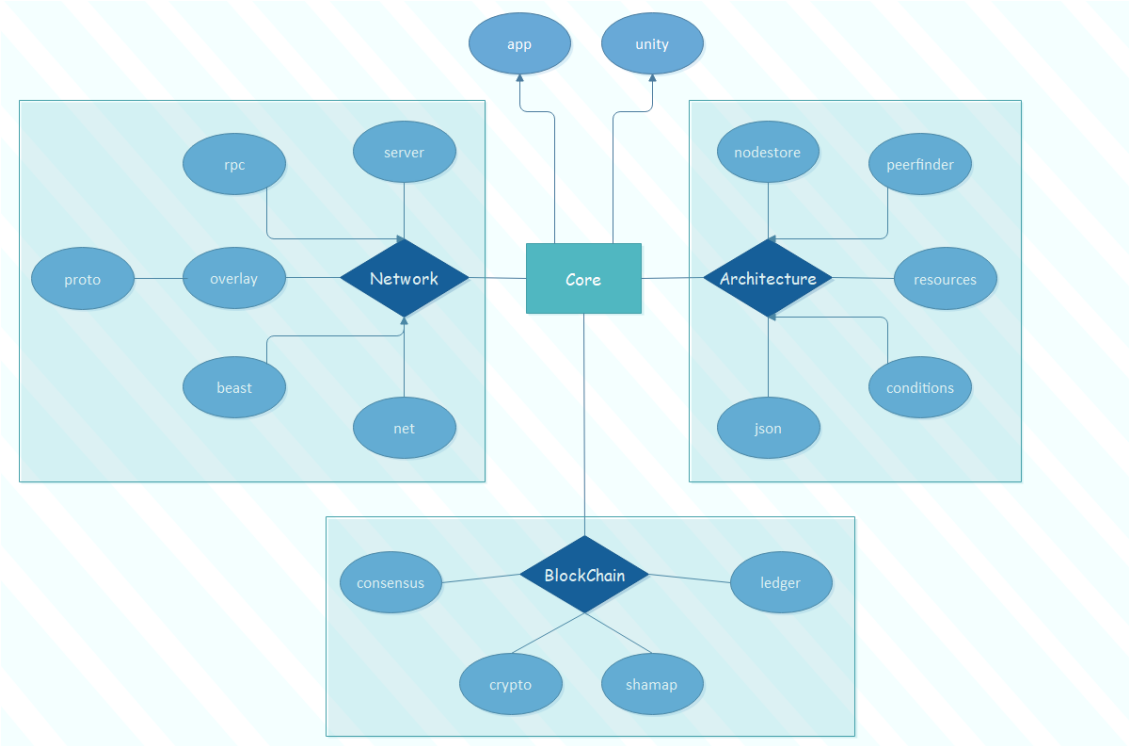

Figure 1: Module Dependensies

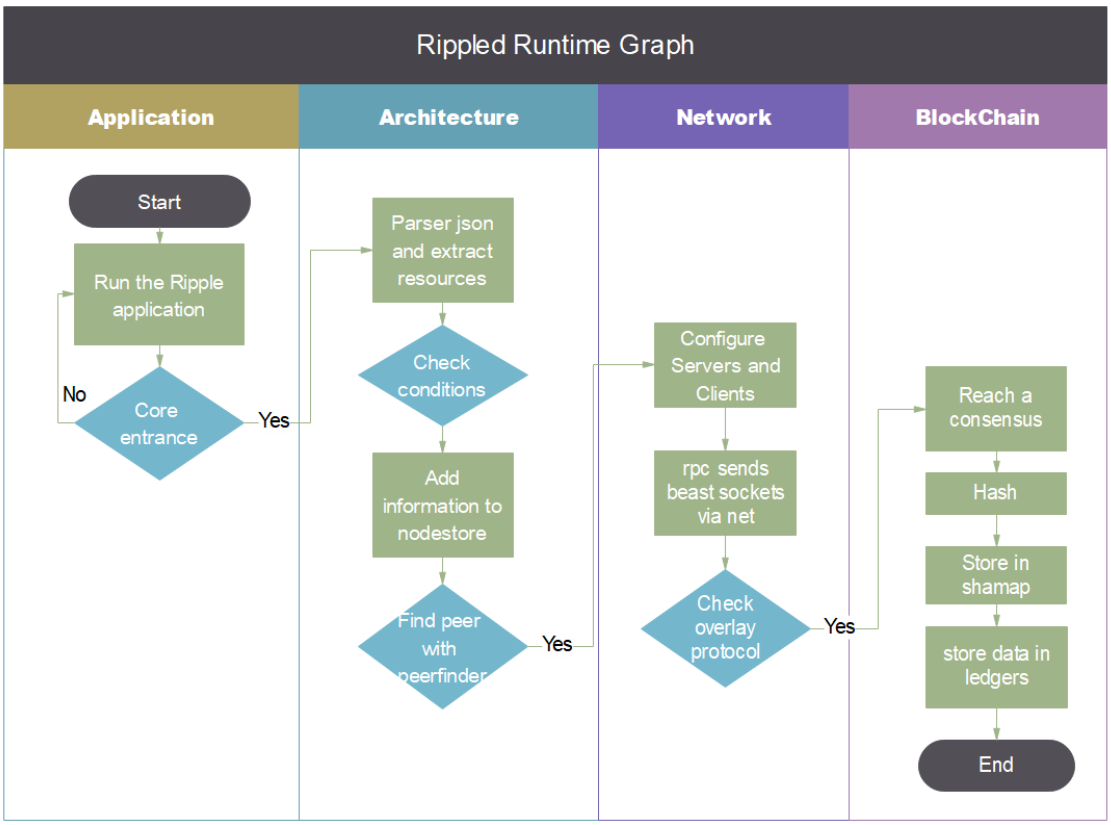

Figure 2: Ripple Run time

- Access entrances connected into a global network of institutions using standard rules, formats, and governance.
- The run time checks according to Ripple's rulebook, which is a legal framework about the rights, obligations, and business rules of network participants.[14]
- Servers who perform pathfinding capability and then ensure that payments are routed from the originator to the beneficiary in the most efficient way possible.
- Messenger API, which provides payment certainty with instant bidirectional messaging.
- The domestic digital assets converting the originating currency to XRP.

[15]

1.5 Non-functional Properties

One common goal followed by software engineers is to deliver a product that satisfies the requirements of different stakeholders. Software requirements are generally categorized into *functional* and *Non-Functional* Requirements (NFRs). [16] Broadly, functional requirements define *what* a system is supposed to do, and non-functional requirements define *how* a system is supposed to be. While NFRs may not be the main focus in developing some applications, there are systems and domains where satisfying NFRs is even critical and one of the main factors which can determine the success or failure of the delivered product. Hence, in this section, we will look into the NFRs of Ripple. One functional requirement could be *creating a new block*; meanwhile, an NFR would be *specifying the hash function and cryptographic protocols it will use* to satisfy the functional requirement.

1.5.1 Why NFR is a challenging task

Addressing NFRs in the development of a software product is a challenging task. Often NFRs are expressed informally and abstractly. they can be considered as a specification of global constraints on the software product, such as security, performance, availability, and so on, which can crosscut different parts of a system. [17]

Another problem which is mostly observed in large organizations is that different teams may have different

[14] https://ripple.com/files/ripplenet_brochure.pdf
[15] https://ripple.com/files/ripplenet_brochure.pdf
[16] Saadatmand et al. (2013). Model-Based Trade-off Analysis of Non-Functional Requirements
[17] Saadatmand et al. (2013). Model-Based Trade-off Analysis of Non-Functional Requirements

interpretations of an NFR, or refer to one NFR using different terms. Therefore, a coherent way of representing and defining NFRs can help mitigate such problems. [18]

1.5.2 Ripple NFR

In the case of Ripple, main NFRs that are mentioned in a more formal approach in the rippled repository are as follows:[19] [20]

- **Secure, Adaptable Cryptography:** The XRP Ledger relies on industry-standard digital signature systems like ECDSA (the same scheme used by Bitcoin) but also supports modern, efficient algorithms like Ed25519. The extensible nature of the XRP Ledger's software makes it possible to add and disable algorithms as state of the art in cryptography advances.
- **Responsible Software Governance:** As an entity that is obligated to hold large amounts of XRP for the long term, Ripple has a strong incentive to ensure that XRP is widely used in ways that are legal, sustainable, and constructive. Ripple provides technical support to businesses whose goals align with Ripple's ideal of an Internet of Value. Ripple also cooperates with legislators and regulators worldwide to guide the Implementation of sensible laws governing digital assets and associated businesses.
- **Censorship-Resistant Transaction Processing:** No single party decides which transactions succeed or fail, and no one can "rollback" a transaction after it completes.
- **Fast, Efficient Consensus Algorithm:** The XRP Ledger's most significant difference from most cryptocurrencies is that it uses a unique consensus algorithm that does not require the time and energy of "mining." The XRP Ledger's consensus algorithm settles transactions in 4 to 5 seconds, processing at a throughput of up to 1500 transactions per second.
- **Finite XRP Supply:** The rules of the XRP Ledger provide a simple solution to hyperinflation: the total supply of XRP is finite. Without a mechanism to create more, it becomes much less likely that XRP could suffer hyperinflation.

1.5.3 Addressing NFR and its Trade-Offs

Considering that NFRs are usually specified informally and abstractly, providing a more formal approach which enables to raise the abstraction level can help with the treatment of NFRs. Moreover, an explicit treatment of NFRs facilitates the predictability of the system in terms of the quality properties of the final product more reliably and reasonably. [21]. In satisfying NFRs, the dependencies among them should not be neglected, as meeting one NFR can affect and impair the satisfaction of other NFRs in the system. Therefore, performing a trade-off analysis to establish a balance among NFRs and identify such mutual impacts is necessary. [22]

Let's take a look at the first two NFRs mentioned above: Secure Cryptography and Responsible Software Governance. Although implementing secure cryptographic protocols that have a large key length will make the system close to tamper-proof, it has a glaring disadvantage: criminals could use the system to partake in illegal activities, and it will be harder to regulate. Of course, one might argue that the whole purpose of Distributed Ledger Systems is to free us off from these regulations. Still, in a perfect world, Ripple

[18] Saadatmand et al. (2013). Model-Based Trade-off Analysis of Non-Functional Requirements
[19] https://github.com/ripple/rippled/blob/develop/README.md
[20] https://xrpl.org/xrp-ledger-overview.html#xrp-ledger-overview
[21] Saadatmand et al. (2013). Model-Based Trade-off Analysis of Non-Functional Requirements
[22] Saadatmand et al. (2013). Model-Based Trade-off Analysis of Non-Functional Requirements

should decide on how to compromise between regulation, responsible governance, and the strength of these cryptosystems.