Reticulum Overview

This paper will briefly describe the overall purpose and operating principles of Reticulum, a networking stack designed for reliable and secure communication over high-latency, low-bandwidth links. It should give you an overview of how the stack works, and an understanding of how to develop networked applications using Reticulum.

This document is not an exhaustive source of information on Reticulum, at least not yet. Currently, the best place to go for such information is the Python reference implementation of Reticulum. Both the reference implementation and this document may (and will) change rapidly in the current phase of development, but historical versions will always be available in the Git repositories.

After reading this document, you should be well-equipped to understand how a Reticulum network operates, what it can achieve, and how you can use it yourself. If you want to help out with the development, this is also the place to start, since it will also provide a pretty clear overview of the sentiments and the philosophy behind Reticulum.

Motivation

The primary motivation for designing and implementing Reticulum has been the current lack of reliable, functional and secure minimal-infrastructure modes of digital communication. It is my belief that it is highly desirable to create a cheap and reliable way to set up a wide-range digital communication network that can securely allow exchange of information between people and machines, with no central point of authority, control, censorship or barrier to entry.

Almost all of the various networking stacks in wide use today share a common limitation, namely that they require large amounts of coordination to work. You can't just plug in a bunch of ethernet cables to the same switch, or turn on a number of WiFi radios, and expect such a setup to provide a reliable platform for communication.

The designers of the Internet Protocol had the foresight to create a protocol that powers the modern Internet, and works brilliantly in world very different from when it was conceived. But networks using the traditional IP stack needs large amounts of coordination from the people involved, and without central actors in ultimate control of network segments, it is very easy for a single person to render the platform unusable for everyone else. These limitations are inherent to the design principles of IP, and during the design of IP, this was a very reasonable tradeoff indeed.

Reticulum aims to require as little coordination and trust as possible. In fact, the only "coordination" required is to know how to get connected to a Reticulum network. Since Reticulum is medium agnostic, this could be whatever is best suited to the situation. In some cases, this might be 1200 baud packet radio links over VHF frequencies, in other cases it might be a microwave network using off-the-shelf radios. At the time of release of this document, the recommended setup is using cheap LoRa radio modules with an open source firmware (see the chapter *Reference System Setup*), connected to a small computer like a Raspberry Pi. As an example, the default reference setup provides a channel capacity of 5.4 Kbps, and a usable direct node-to-node range of around 15 kilometers (indefinitely extendable by using multiple hops).

Goals

To be as widely usable and easy to implement as possible, the following goals have been used to guide the design of Reticulum:

- **Fully useable as open source software stack**
  Reticulum must be implemented, and be able to run using only open source software. This is critical to ensuring availability, security and transparency of the system.
- **Hardware layer agnosticism**
  Reticulum shall be fully hardware agnostic, and should be useable over a wide range physical networking layers, such as data radios, serial lines, modems, handheld transceivers, wired ethernet, wifi, or anything else that can carry a digital data stream. Hardware made for dedicated Reticulum use shall be as cheap as possible and use off-the-shelf components, so it can be easily replicated.
- **Very low bandwidth requirements**
  Reticulum should be able to function reliably over links with a data capacity as low as *1,000 bps*.
- **Encryption by default**
  Reticulum must use encryption by default where possible and applicable.
- **Unlicensed use**
  Reticulum shall be functional over physical communication mediums that do not require any form of license to use. Reticulum must be designed in a way, so it is usable over ISM radio frequency bands, and can provide functional long distance links in such conditions.
- **Supplied software**
  Apart from the core networking stack and API, that allows any developer to build applications with Reticulum, a basic communication suite using Reticulum must be implemented and released at the same time as Reticulum itself. This shall serve both as a functional communication suite, and as an example and learning resource to others wishing to build applications with Reticulum.
- **Ease of use**
  The reference implementation of Reticulum is written in Python, to make it very easy to use and understand. Any programmer with only basic experience should be able to use Reticulum in their own applications.
- **Low cost**
  It shall be as cheap as possible to deploy a communication system based on Reticulum. This should be achieved by using cheap off-the-shelf hardware that potential users might already own. The cost of setting up a functioning node should be less than $100 even if all parts needs to be purchased.

Introduction & Basic Functionality

Reticulum is a networking stack suited for high-latency, low-bandwidth links. Reticulum is at it's core *message oriented*, but can provide connection oriented sessions. It is suited for both local point-to-point or point-to-multipoint scenarios where alle nodes are within range of each other, as well as scenarios where packets need to be transported over multiple hops to reach the recipient.

Reticulum does away with the idea of addresses and ports known from IP, TCP and UDP. Instead Reticulum uses the singular concept of *destinations*. Any application using Reticulum as it's networking stack will need to create one or more destinations to receive data, and know the destinations it needs to send data to.

Reticulum encrypts all data by default using public-key cryptography. Any message sent to a destination is encrypted with that destinations public key. Reticulum also offers symmetric key encryption for group-oriented communications, as well as unencrypted packets for broadcast purposes, or situations where you need the communication to be in plain text. The multi-hop transport, coordination, verification and reliability layers are fully autonomous and based on public key cryptography.

Reticulum can connect to a variety of interfaces such as radio modems, data radios and serial ports, and offers the possibility to easily tunnel Reticulum traffic over IP links such as the Internet or private IP networks.

Destinations

To receive and send data with the Reticulum stack, an application needs to create one or more destinations. Reticulum uses three different basic destination types, and one special:

- **Single**
  The *single* destination type defines a public-key encrypted destination. Any data sent to this destination will be encrypted with the destination's public key, and will only be readable by the creator of the destination.
- **Group**
  The *group* destination type defines a symmetrically encrypted destination. Data sent to this destination will be encrypted with a symmetric key, and will be readable by anyone in possession of the key. The *group* destination can be used just as well by only two peers, as it can by many.
- **Plain**
  A *plain* destination type is unencrypted, and suited for traffic that should be broadcast to a number of users, or should be readable by anyone.
- **Link**
  A *link* is a special destination type, that serves as an abstract channel between two *single* destinations, directly connected or over multiple hops. The *link* also offers reliability and more efficient encryption, and as such is useful even when nodes are directly connected.

Destination Naming

Destinations are created and named in an easy to understand dotted notation of *aspects*, and represented on the network as a hash of this value. The hash is a SHA-256 truncated to 80 bits. The top level aspect should always be the a unique identifier for the application using the destination. The next levels of aspects can be defined in any way by the creator of the application. For example, a destination for a messaging application could be made up of the application name and a username, and look like this:

name: simplemessenger.someuser **hash:** 2a7ddfab5213f916dea9

For the *single* destination, Reticulum will automatically append the associated public key as a destination aspect before hashing. This is done to ensure only the correct destination is reached, since anyone can listen to any destination name. Appending the public key ensures that a given packet is only directed at the destination that holds the corresponding private key to decrypt the packet. It is important to understand that anyone can use the destination name *simplemessenger.myusername*, but each person that does so will still have a different destination hash, because their public keys will differ. In actual use of *single* destination naming, it is advisable not to use any uniquely identifying features in aspect naming, though. In the simple messenger example, when using *single* destinations, we would instead use a destination naming scheme such as *simplemessenger.user* where appending the public key expands the destination into a uniquely identifying one.

To recap, the destination types should be used in the following situations:

- **Single**
  When private communication between two endpoints is needed. Supports routing.
- **Group**
  When private communication between two or more endpoints is needed. More efficient in data usage than *single* destinations. Supports routing indirectly, but must first be established through a *single* destination.
- **Plain**
  When plain-text communication is desirable, for example when broadcasting information.

To communicate with a *single* destination, you need to know it's public key. Any method for obtaining the public key is valid, but Reticulum includes a simple mechanism for making other nodes aware of your destinations public key, called the *announce*.

Note that this information could be shared and verified in many other ways, and that it is therefore not required to use the announce functionality, although it is by far the easiest, and should probably be used if you are not confident in how to verify public keys and signatures manually.

Public key announcements

An *announce* will send a special packet over any configured interfaces, containing all needed information about the destination hash and public key, and can also contain some additional, application specific data. The entire packet is signed by the sender to ensure authenticity. It is not required to use the announce functionality, but in many cases it will be the simplest way to share public keys on the network. As an example, an announce in a simple messenger application might contain the following information:

- The announcers destination hash
- The announcers public key
- Application specific data, in this case the users nickname and availability status
- A random blob, making each new announce unique
- A signature of the above information, verifying authenticity

With this information, any Reticulum node that receives it will be able to reconstruct an outgoing destination to securely communicate with that destination. You might have noticed that there is one piece of information lacking to reconstruct full knowledge of the announced destination, and that is the aspect names of the destination. These are intentionally left out to save bandwidth, since they will be implicit in almost all cases. If a destination name is not entirely implicit, information can be included in the application specific data part that will allow the receiver to infer the naming.

It is important to note that announcements will be forwarded throughout the network according to a certain pattern. This will be detailed later. Seeing how *single* destinations are always tied to a private/public key pair leads us to the next topic.

Identities

In Reticulum, an *identity* does not necessarily represent a personal identity, but is an abstraction that can represent any kind of *verified entity*. This could very well be a person, but it could also be the control interface of a machine, a program, robot, computer, sensor or something else entirely. In general, any kind of agent that can act, or be acted upon, or store or manipulate information, can be represented as an identity.

As we have seen, a *single* destination will always have an *identity* tied to it, but not *plain* or *group* destinations. Destinations and identities share a multilateral connection. You can create a destination, and if it is not connected to an identity upon creation, it will just create a new one to use automatically. This may be desirable in some situations, but often you will probably want to create the identity first, and then link it to created destinations.

Building upon the simple messenger example, we could use an identity to represent the user of the application. Destinations created will then be linked to this identity to allow communication to reach the user. In such a case it is of great importance to store the user's identity securely and privately.

Getting Further

The above functions and principles form the core of Reticulum, and would suffice to create functional networked applications in local clusters, for example over radio links where all interested nodes can hear each other. But to be truly useful, we need a way to go further. In the next chapter, two concepts that allow this will be introduced, *paths* and *resources*.

Transport

I have purposefully avoided the term routing until now, and will continue to do so, because the current methods of routing used in IP based networks are fundamentally incompatible for the link types that Reticulum was designed to handle. These routing methodologies assume trust at the physical layer. Since Reticulum is designed to run over open radio spectrum, no such trust exists. Furthermore, existing routing protocols like BGP or OSPF carry too much overhead to be practically useable over bandwidth-limited, high-latency links.

To overcome such challenges, Reticulum's *Transport* system uses public-key cryptography to implement the concept of *paths* that allow discovery of how to get information to a certain destination, and *resources* that help alleviate congestion and make reliable communication more efficient and less bandwidth-hungry.

Threading a Path

In networks with changing topology and trustless connectivity, nodes need a way to establish *verified connectivity* with each other. To do this, the following process is employed:

- First, the node that wishes to establish connectivity will send out a special packet, that traverses the network and locates the desired destination. Along the way, the nodes that forward the packet will take note of this *link request*.
- Second, if the destination accepts the *link request*, it will send back a packet that proves the authenticity of it's identity (and the receipt of the link request) to the initiating node. All nodes that initially forwarded the packet will also be able to verify this proof, and thus accept the validity of the *link* throughout the network.
- When the validity of the *link* has been accepted by forwarding nodes, these nodes will remember the *link*, and it can subsequently be used by referring to a hash representing it.
- As a part of the *link request*, a key exchange takes place, that sets up an efficient symmetrically encrypted tunnel between the two nodes. As such, this mode of communication is preferred, even for situations when nodes can directly communicate, when the amount of data to be exchanged numbers in the tens of packets.
- When a *link* has been set up, it automatically provides message receipt functionality, so the sending node can obtain verified confirmation that the information reached the intended recipient.

In a moment, we will discuss the specifics of how this methodology is implemented, but let's first recap what purposes this serves. We first ensure that the node answering our request is actually the one we want to communicate with, and not a malicious actor pretending to be so. At the same time we establish an efficient encrypted channel. The setup of this is relatively cheap in terms of bandwidth, so it can be used just for a short exchange, and then recreated as needed, which will also rotate encryption keys (keys can also be rotated over an existing path), but the link can also be kept

alive for longer periods of time, if this is more suitable to the application. The amount of bandwidth used on keeping a link open is practically negligible. The procedure also inserts the *link id*, a hash calculated from the link request packet, into the memory of forwarding nodes, which means that the communicating nodes can thereafter reach each other simply by referring to this *link id*.

Step 1, pathfinding

The pathfinding method builds on the *announce* functionality discussed earlier. When an announce is sent out by a node, it will be forwarded by anyone node receiving it, but according to some specific rules:

- If this announce has already been received before, ignore it.
- Record into a table which node the announce was received from, and how many times in total it has been retransmitted to get here.
- If the announce has been retransmitted $m+1$ times, it will not be forwarded. By default, m is set to 18.
- The announce will be assigned a delay $d = c^h$ seconds, where c is a decay constant, by default 2, and h is the amount of times this packet has already been forwarded.
- The packet will be given a priority $p = 1/d$.
- If at least d seconds has passed since the announce was received, and no other packets with a priority higher than p are waiting in the queue (see Packet Prioritisation), and the channel is not utilized by other traffic, the announce will be forwarded.
- If no other nodes are heard retransmitting the announce with a greater hop count than when it left this node, transmitting it will be retried r times. By default, r is set to 2. Retries follow same rules as above, with the exception that it must wait for at least $d = c^{h+1} + t$ seconds, ie., the amount of time it would take the next node to retransmit the packet. By default, t is set to 1.
- If a newer announce from the same destination arrives, while an identical one is already in the queue, the newest announce is discarded. If the newest announce contains different application specific data, it will replace the old announce, but will use d and p of the old announce.

Once an announce has reached a node in the network, any other node in direct contact with that node will be able to reach the destination the announce originated from, simply by sending a packet adressed to that destination. Any node with knowledge of the announce will be able to direct the packet towards the destination by looking up the next node with the shortest amount of hops to the destination. The specifics of this process is detailed in *Path Calculation*.

According to these rules and default constants, an announce will propagate throughout the network in a predictable way. In an example network utilising the default constants, and with an average link distance of L_{avg} = 15 kilometers, an announce will be able to propagate outwards to a radius of 180

kilometers in 34 minutes, and a *maximum announce radius* of 270 kilometers in approximately 3 days. Methods for overcoming the distance limitation of $m * L_{avg}$ will be introduced later in this chapter.

Step 2, link establishment

After seeing how the conditions for finding a path through the network are created, we will now explore how two nodes can establish reliable communications over multiple hops. The *link* in Reticulum terminology should not be viewed as a direct node-to-node link on the physical layer, but as an abstract channel, that can be open for any amount of time, and can span an arbitrary number of hops, where information will be exchanged between two nodes.

- When a node in the network wants to establish verified connectivity with another node, it will create a *link request* packet, and broadcast it.
- The *link request* packet contains the destination hash *Hd*, and an asymmetrically encrypted part containing the following data: The source hash *Hs*, a symmetric key *Lk*, a truncated hash of a random number *Hr*, and a signature *S* of the plaintext values of *Hd*, *Hs*, *Lk* and *Hr*.
- The broadcasted packet will be directed through the network according to the rules laid out previously.
- Any node that forwards the link request will store a *link id* in it's *link table*, along with the amount of hops the packet had taken when received. The link id is a hash of the entire path request packet. If the path is not *proven* within some set amount of time, the entry will be dropped from the table again.
- When the destination receives the link request packet, it will decide whether to accept the request. If it is accepted, it will create a special packet called a *proof*. A *proof* is a simple construct, consisting of a truncated hash of the message that needs to be proven, and a signature (made by the destination's private key) of this hash. This *proof* effectively verifies that the intended recipient got the packet, and also serves to verify the discovered path through the network. Since the *proof* hash matches the *path id* in the intermediary nodes' *path tables*, the intermediary nodes can forward the proof all the way back to the source.
- When the source receives the *proof*, it will know unequivocally that a verified path has been established to the destination, and that information can now be exchanged reliably and securely.

It's important to note that this methodology ensures that the source of the request does not need to reveal any identifying information. Only the intended destination will know "who called", so to speak. This is a huge improvement to protocols like IP, where by design, you have to reveal your own address to communicate with anyone, unless you jump through a lot of hoops to hide it. Reticulum offers initiator anonymity by design.

When using *links*, Reticulum will automatically verify anything sent over the link, and also automates retransmissions if parts of a message was lost along the way. Due to the caching features

of Reticulum, such a retransmission does not need to travel the entire length of an established path. If a packet is lost on the 8th hop of a 12 hop path, it can be fetched from the last hop that received it reliably.

Crossing Continents

When a packet needs to travel farther than local network topology knowledge stretches, a system of geographical or topological hinting is used to direct the packet towards a network segment with direct knowledge of the intended destination. This functionality is currently left out of the protocol for simplicity of testing other parts, but will be activated in a near future release. For more information on when, refer to the roadmap on the website.

Resourceful Memory

In traditional networks, large amounts of data is rapidly exchanged with very low latency. Links of several thousand kilometers will often only have round-trip latency in the tens of milliseconds, and as such, traditional protocols are often designed to not store any transmitted data at intermediary hops. If a transmission error occurs, the sending node will simply notice the lack of a packet acknowledgement, and retransmit the packet all the way, until it hears back from the receiver that it got the intended data.

In bandwidth-limited and high-latency conditions, such behaviour quickly causes congestion on the network, and communications that span many hops become exceedingly expensive in terms of bandwidth usage, due to the higher risk of some packets failing.

Reticulum alleviates this in part with it's *path* discovery methodology, and in part by implementing *resource* caching at all nodes that can support it. Network operation can be made much more efficient by caching everything for a period of time, and given the availability of cheap memory and storage, this is a very welcome tradeoff. A gigabyte of memory can store millions of Reticulum packets, and since everything is encrypted by default, the storing poses very little privacy risk.

In a Reticulum network, any node that is able to do so, should cache as many packets as it's memory will allow for. When a packet is received, a timestamp and a hash of the packet is stored along with the full packet itself, and it will be kept in storage until the allocated cache storage is full, whereupon the packet that was last accessed in the cache will be deleted. If a packet is accessed from the cache, it's timestamp will be updated to the current time, to ensure that packets that are used stay in the cache, and packets that are not used are dropped from memory.

Some packet types are stored in separate caching tables, that allow easier lookup for other nodes. For example, an announce is stored in a way, that allows other nodes to request the public key for a certain destination, and as such the network as a whole operates as a distributed key ledger.

For more details on how the caching works and is used, see the reference implementation source code.

Reference System Setup

This section will detail the recommended *Reference System Setup* for Reticulum. It is important to note that Reticulum is designed to be usable over more or less any medium that allows you to send and receive data in a digital form, and satisfies some very low minimum requirements. The communication channel must support at least half-duplex operation, and provide an average throughput of around 1000 bits per second, and supports a physical layer MTU of 500 bytes. The Reticulum software should be able to run on more or less any hardware that can provide a Python runtime environment.

That being said, the reference setup has been outlined to provide a common platform for anyone who wants to help in the development of Reticulum, and for everyone who wants to know a recommended setup to get started. A reference system consists of three parts:

- **A channel access device**
  Or *CAD*, in short, provides access to the physical medium whereupon the communication takes place, for example a radio with an integrated modem. A setup with a separate modem connected to a radio would also be termed a “channel access device”.
- **A host device**
  Some sort of computing device that can run the necessary software, communicates with the channel access device, and provides user interaction.
- **A software stack**
  The software implementing the Reticulum protocol and applications using it.

The reference setup can be considered a relatively stable platform to develop on, and also to start building networks on. While details of the implementation might change at the current stage of development, it is the goal to maintain hardware compatibility for as long as entirely possible, and the current reference setup has been determined to provide a functional platform for many years into the future. The current Reference System Setup is as follows:

- **Channel Access Device**
  A data radio consisting of a LoRa radio module, and a microcontroller with open source firmware, that can connect to host devices via USB. It operates in either the 430, 868 or 900 MHz frequency bands. More details on the exact parts and how to get/make one can be found on the website.
- **Host device**
  Any computer device running Linux and Python. A Raspberry Pi with Raspbian is recommended.
- **Software stack**
  The current Reference Implementation Release of Reticulum, running on a Debian based operating system.

It is very important to note, that the reference channel access device **does not** use the LoRaWAN standard, but uses a custom MAC layer on top of the plain LoRa modulation! As such, you will need a plain LoRa radio module connected to an MCU with the correct Reticulum firmware. Full details on how to get or make such a device is available on the website.

With the current reference setup, it should be possible to get on a Reticulum network for around 70$ even if you have none of the hardware already.

Protocol Specifics

This chapter will detail protocol specific information that is essential to the implementation of Reticulum, but non critical in understanding how the protocol works on a general level. It should be treated more as a reference than as essential reading.

Node Types

Currently Reticulum defines two node types, the *Station* and the *Peer*. A node is a *station* if it fixed in one place, and if it is intended to be kept online at all times. Otherwise the node is a *peer*. This distinction is made by the user configuring the node, and is used to determine what nodes on the network will help forward traffic, and what nodes rely on other nodes for connectivity.

Packet Prioritisation

The packet prioritisation algorithms are subject to rapid change at the moment, and for now, they are not documented here. See the reference implementation for more info on how this functionality works.

Path Calculation

The path calculation algorithms are subject to rapid change at the moment, and for now, they are not documented here. See the reference implementation for more info on how this functionality works.

Binary Packet Format

The binary packet format is subject to rapid change at the moment, and for now, it is not documented here. See the reference implementation for the specific details on this topic.