Distributed Cloud Storage – Technical Manual

0. Table of contents

Distributed Cloud Storage – Technical Manual

1. Introduction

1.1 Overview

Distributed Cloud Storage – a set of programs that can turn your private servers into a cloud storage platform (think "Google Drive", "iCloud", or "Dropbox"). Our "node software" uses the Internet to connect your servers ("nodes") into a cloud designed for storage. Use one of our "client programs" to connect to your network and upload/download files, all as if the network was a single cloud entity.

Distributed (de-centralised), secure, intelligent.

- Leverages the nodes' underlying Operating Systems for persistent storage.
- Intelligent routing of files to the most optimal node in terms of storage load and network benchmarks.
- Reliability and privacy of storage at all times through redundancy and encrypted communications.
- Minimised single points of failure. Each node acts both as a client and as a server (a distributed system).

Portable (cross-platform), easily installable "node software" for technical/industry users requiring off-the-shelf private cloud storage solutions. Configure through a graphical or command-line interface.

"Client programs" including a mobile friendly website client and graphical desktop client for the end-users of storage. Modern file explorer UI/UX to interact with the cloud storage platform.

1.2 Glossary

Node - a server or computer system capable of participating in a storage cloud (capabilities: network stack, persistent file system, etc.)

Node software - a program that joins the computer it is running on into a storage cloud, intended to be used by technical users.

Client program - a program or interface that connects the user to the storage cloud and allows them to store and download their files, intended for end-users that may not be as technical.

Node administrator - a user that interacts with the cloud storage in a technical way, ensuring set up of "node software" and some "client software" such as the website.

End user - a user that interacts with the cloud storage non-technically, to upload and download files.

Cloud - network of computers connected via the Internet that expose some interface to the outside world.

Storage Cloud - a cloud designed to expose file storage.

Go, Golang - performant, concurrent, C like general-purpose programming language [].

Struct - a Go object-like (OOP) variable.

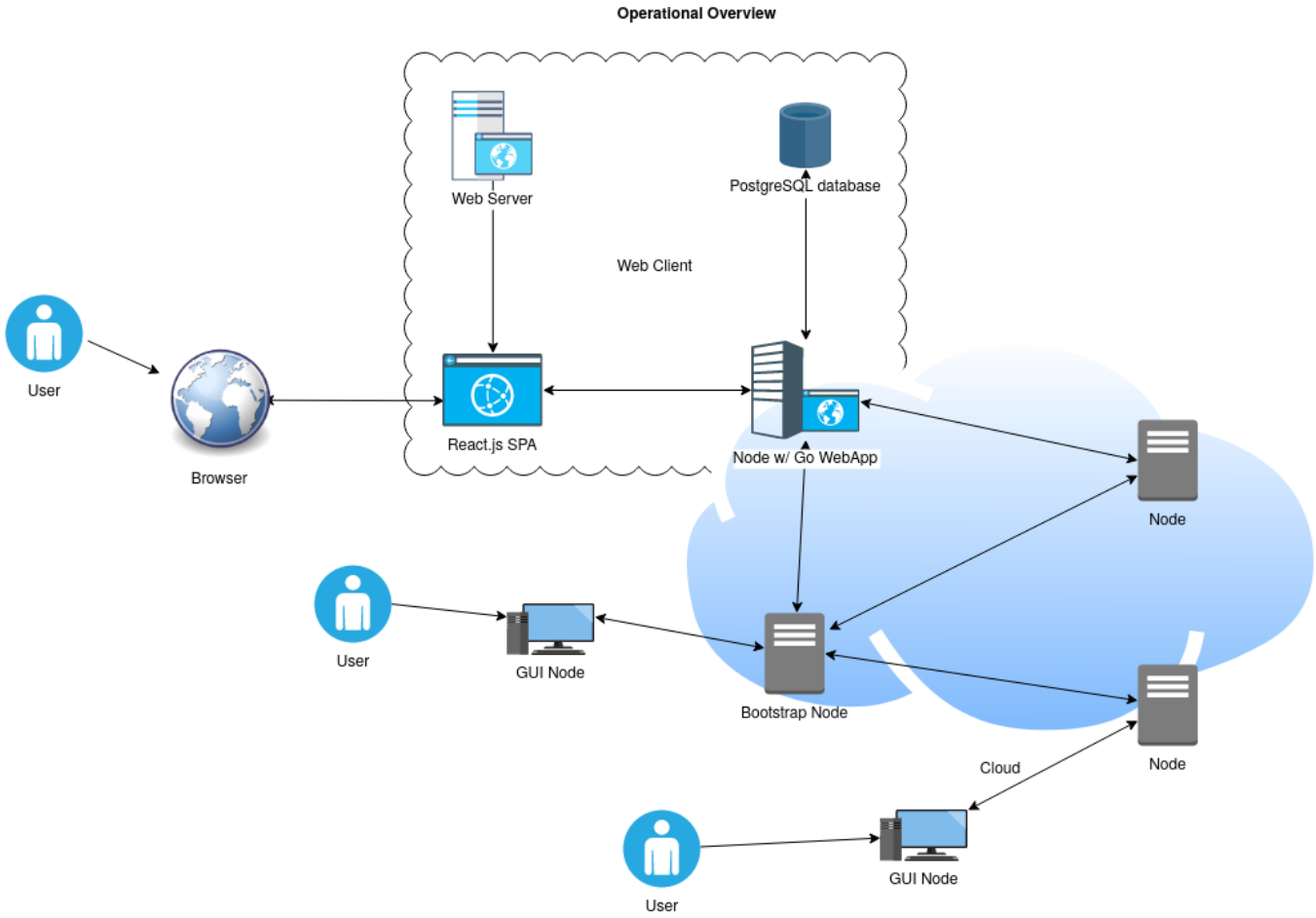
RPC (Remote Procedure Call) - executing a function on a different computer.

SPA (Single Page Application) - a website that is served once and updates dynamically instead of from browser refresh.

2. System Architecture

2.1. Operational Overview

The following is an Operational Overview Diagram



The most important part of the diagram is the cloud area. This indicates the cloud network that our Go node software creates by connecting multiple computers or nodes.

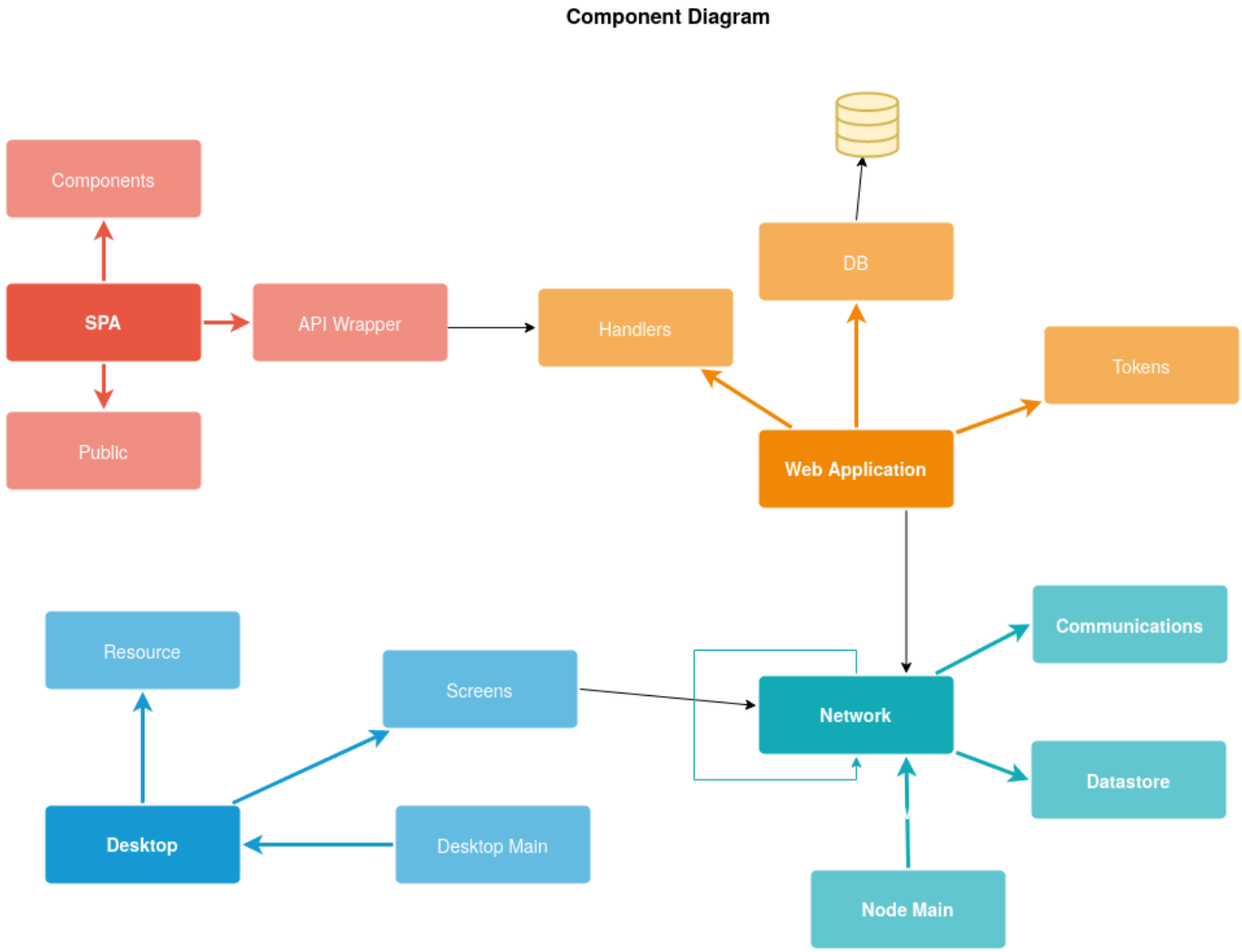
- The network starts out with a single node, the "bootstrap node".
- Other nodes connect to nodes already in the network.
- Not all nodes may be connected to all other nodes at the same time.

Outside the cloud we can see the "GUI Nodes" that connect to our cloud but do not actually store any data. They are used for desktop GUI client.

We have a "web client" area that shows all of the components set up for a functional website for the web client. That includes one of the nodes running a web application server together with a database, and a web server that serves our React SPA. The user can then access the cloud via a browser.

2.2. Component Diagram

The following is a component diagram.



Each rectangle indicates a major component or subcomponent, colour coded to show different subsystems in our project.

The arrows indicate a "uses" relationship (imports package, or initiates communication with).

Some important relationships include:

- The Network component is imported by:
    1. The Main Node sub-component, which represents the binary used for setting up the cloud, and the desktop client.
    2. The Desktop Client's screen component.
    3. The Web Application component.
- The communication between the web application and the SPA are done via the SPA's API wrapper and the Web Application's Handlers sub-components.

3. High-Level Design

In this section we will discuss the major design decisions we currently have, where relevant mentioning how they differ from the proposed design.

3.1. Go Library

As there are multiple uses of the cloud (Desktop CLI, Desktop GUI, Web app), we created a Go library to handle the cloud backbone. Making it easy to use and control outside. This library was created to be simple in use but offer as much control as possible.

Connecting to an existing network is as easy as:

```
cloud, err := network.BootstrapToNetwork("networkip:port", network.Node{
                        Name:      "Node Name",
                        IP:        ":9002",
                        PublicKey: key.PublicKey,
                }, privateKey, network.CloudConfig{FileStorageDir:
"/path/to/dir/to/store})
```

Allowing for easy interfacing with the cloud:

```
// Syncs local folder to a folder on the cloud.
c.SyncFolder("/folder/on/cloud", "/local/folder");

// Adds a local file to the cloud.
c.AddFile(fileMetadata, "/path/on/cloud", "/local/file")
```

3.2. File Chunking

We have decided to split each file into a number of chunks. This way we distribute parts of the file to the cloud instead of the entire file.

This could easily allow for updating only the part of a file that changed, reducing message sizes, etc.

3.3. Web and Mobile Clients

We have decided to implement the web application as part of the node software (as opposed to a separate program, communicating with the cloud via our own protocol). The web application could be easily enabled, turning the node software into a web server as well.

The reason for this is because the communications between the web application and the cloud will be simplified and speeded up. Since the web application is running on the same program (same address space) as the node software, web responses can query the cloud immediatelly. We also do not need to write an additional communications layer between the web application and the cloud.

In contrast to the initial design, we have decided that the web client will supersede the mobile client (mobile application). Thus, the website is designed to be mobile friendly. We made this decision due to time constraints. However, because the web SPA is written in `React.js` (https://reactjs.org), it could be argued that making a `React Native` mobile app will not take as much time if more development time was available.

3.4. Compression Layer

As part of the initial design, we have indicated that we will have a compression layer that will compress files to reduce file sizes and increase the potential use of space. We have decided against having this layer for performance and extensibility reasons. We have looked into compressing the files to reduce the file sizes and potentially increase the space available on the nodes. We have looked at two options:

- Compressing the entire file before chunking.
- Compressing each chunk invidually.

We found that compressing the entire file before chunking would yield better compression results than compressing each chunk invidually. However, this would make file chunking pretty redundant, as any small changes to a file can result in most, if not all, of the chunks changing.

Compressing each chunk invidually would not yield as good as results as compressing an entire file, as there is less data to work with.

Storing the files compressed would also increase the latency by quite a large margin, depending on the compression level. Any meaningful space saved by compression would suffer by increased latency. We found that this is not worth the trade off.

As part of future extensibility, with compression we will not be able to implement a possible real-time streaming feature, as the entire file will have to be transfered or decompressed before initiating transfer.

Additionally, Windows and Linux both support compression on their file system. We believe it's better to leave compression handling to the operating system and the user.

4. Problems and Solutions

We discussed how we have solved major challenges in various sub-systems of the project.

4.1. Network Communications

4.1.1. Choice of Communication Method

One of the major design considerations is the communication layer between nodes. As the nodes are in constant communication, it plays a key part in the cloud.

The main options we considered for communication are:

1. Existing RPC libraries, such as gRPC

The first consideration was using an existing RPC library, mainly considering gRPC as this is the most known/widely used library.

One of the main benefits of gRPC is cross-language communication. gRPC is supported by most modern programming languages that are used, such as C++, Java, Golang, Python, etc... However this would not benefit us, as we had no plans to implement the cloud in other languages.

Another benefit of gRPC is it's speed. As gRPC files requires to be compiled seperately, it allows for much faster encoding and decoding of variables than using gob (Go standard library package for serialisation), as the structure is known before. Unfortunately, it does need to be compiled seperately which is less than ideal.

The drawback of gRPC is it's built for client-to-server communication. As our cloud is decentralized, every node is a client and a server. With this, twice as many sockets would have to be opened. Additionally, it would be impossible for nodes to participate in the network if their ports are closed. As they would only be able to send requests, and not receive. We decided it was not worth the trade off for a tiny performance improvement with encoding and decoding.

2. Acting as HTTP server and communicating using REST API

The second consideration was to create a HTTP server and expose it using rest API and passing the data with json. This had the same drawback of being client-to-server communication like gRPC. Additionally, it would not have the performance improvements as gRPC would, as the encoding and decoding has to be flexible on the data structures passed.

Having to work with the data as json would also add to the work required when implementing functions. As instead of receiving fixed data structures, we would end up with working on a json object.

3. Using websockets

Websockets was another consideration. It would allow bi-directional communication (client-to-client). It would not provide much more benefits over than using pure TCP sockets, and as such we decided not to go with it.

4. Creating our own communication library

We decided to create our own library to facilitate communication between nodes. The library is built upon TCP sockets to ensure reliability. It allows bi-directional communication (client-to-client) and gives us full control over the communication layer. By encoding and decoding to gob, it allows for minimal overhead.

All communication is encrypted, using a system based of TLS. Public keys are exchanged. A symmetric key is generated, encrypted using the public key, and sent over. The symmetric key is used for encrypting and decrypting the data that is sent as opposed to the public key, as symmetrical encryption is much faster than asymmetrical.

4.1.2. RPC (Communication between Nodes)

Nodes are in constant communication between each other, making it important to get the communications right. All communications between nodes is done using a TCP socket, ensuring the communication between them is reliable. The data sent is encoded and decoded using Gob on the fly. This allows for highely performant communication while maintaining ease of use. This allows passing Go structs as parameters and the data will be decoded/encoded in the communication layer.

The function that is responsible for sending RPC communications is `SendMessage(functionName string, args...interface{})`. In that function, the arguements passed are serialized into gob data. The call is blocking until a response is received. The variables received are returned as an array. Additionally, the `SendMessage` returns an error. If the function on the receiving end returns an error, the error is seperated from outputs and placed into the `err` variable. Additionally, the error can be caused by failure to send the request, or the request timing out.

An example demonstrating the ease of writing communication functions between nodes. Node 1 calls `GetNetwork()`, which calls `OnGetNetworkRequest()` on Node 2, returning any information to node 1.

```
type Network struct {
        Name string
        Nodes []Node
}

func GetNetwork() (Network, error) {
        network, err := client.SendMessage(GetNetworkMsg)
  return network[0].(Network), err
}

func OnGetNetworkRequest() (Network, error) {
        return Network{...}, nil
}
```

4.1.3. Message Structure

The message structure for socket communication starts off with 9 bytes of headers. The first byte, determines if the message is a response to another message. The next 4 bytes hold the message ID. The message ID is an unsigned int and is incremental. This is used to identify messages and send responses back to specific messages. As the communication is done over a single TCP socket connection, this allows multiple messages to be sent and received at the same time. It is safe for the message ID to overflow. The next 4 bytes, the last 4 bytes, hold the message length. As it is not guaranteed for the packet to arrive in it's entirety, this is used to combine multiple packets into one request. Allowing for much larger requests than packet size.

The remaining data, of which length is determined by the last 4 bytes in the header, contains the function name to call and the corresponding data. In the case of the message being a response, function name is omitted.

The function name is escaped by a NULL (\0) character. Anything after that is encrypted gob data of variables that are passed. The function name determines which function on the receiving end to call for the request.

Each function name will have a handler, which points towards a function in the code to call.

4.1.4. Authentication

Every node connecting to the cloud network has to authenticate before participating in the network. Each node has a unique identifier, which is sha256 sum of a public key. Any node in the network can add a node ID to be able to join the network. Upon establishing a socket connection, public keys are exchanged. The node on on the network generates a symmetric key that will be used for encrypting and decrypting all communication between those two nodes, and encrypts it with the connecting node's public key and sends it to them. This ensures that the connecting node owns the private key corresponding to the public key and cannot fake another node's identity.

4.2. Cloud and Network data structures

We have decided to represent the "cloud" using two Go structs. The first struct `Cloud` is the representation of the "cloud" only for the current node. It contains node specific information such as the node's configuration and connections to other nodes.

The second struct `Network` is the shared representation of the "cloud" that includes the nodes in the cloud and a reference of the data stored on the cloud. Each node has the same `Network` struct.

4.3. File Storage data structures

Files are represented by the `File` struct that includes fixed information such as the file's filename or path on the cloud, the size of the file, etc.

In order to split a file into chunks, we have created a `Chunk` struct that represents a chunk in a file. `Chunk` includes chunk ID (hash of its contents), sequence number (which chunk is it), size, etc. We split files based on the configured size of each chunk in bytes. For example, if our chunk size is 5 bytes, then a 10 byte file is split into two 2 chunks (5 bytes each).

In terms of the actual chunk contents, we don't store contents in-memory anywhere. Instead we pass around `reader` variables that can read from a stream and write the reader's contents into persistent storage.

4.4. Distribution Algorithm

We have to make decisions about which node receives which file chunk. For that we have implemented a "distribution algorithm". The algorithm was designed with two objectives in mind:

1. Efficiency
    - We query node benchmarks such as storage space and network quality.
    - Using the benchmarks, storage load and network usage are optimised.
2. Reliability
    - We implement storage redundancy by replicating (making copies of) chunks and checking nodes for "anti-affinity" (same node does not store a chunk multiple times).

We create a "distribution scheme" that the network layer uses to actually send out the chunks, decoupling distribution decisions from the distribution itself.

4.5. Desktop GUI Client

Creating a GUI in Golang is not the smoothest experience. There is no official library to do this, and many libraries are still under-developed. When choosing a GUI library for the desktop client, we mainly looked at those factors:

1. No runtime requirements - The compiled executable should be enough to run the program. There should be no third-party programs that are required to be installed in order to run the program.
2. Cross compatibility - One of the main benefits of Golang is it's cross compatibility. As such, it was very important to keep this.
3. Lightweight - The library should be lightweight and performant to run. This eliminated a lot of libraries that depend on HTML/CSS/JS combo to run.
4. Decent design - Having a decent working design that can be used is really beneficial. It will save a lot of time designing our own from scratch.
5. Flexible - Having full control over the GUI was an important factor.

With all of those factors considered, we decided to settle on fyne.io library. It ticked all of the boxes above.

4.6. Web Client

Creating the web client has been a lot of work since it involves developing both the frontend and a web backend connected to the cloud.

4.6.1. Frontend

We have decided to write a `React.js` SPA that allows for quick prototyping. One of the team members had small experience with the library. Additionally with access to npm (node package manager), we could easily use third party libraries for certain components.

We have also used `Bootstrap` for CSS and styling to achieve mobile friendly layouts and accessible visuals.

4.6.2. Backend

We have written a Go web application and a Go HTTP server to serve that web application. We have aimed to expose a RESTful HTTP web API to the frontend.

We have used the excellent Go standard library `net/http` to start a HTTP server. We use `Gorilla Mux` go library for routing. This allows us to specify advanced routes and specify a handler function that should serve those routes.

To store user accounts, we have decided to use `PostgreSQL`, a relational database and a program. We decided that a relation based database will suit for user accounts and future storage needs. We use a third party ORM (object-relational mapper) library in Go to access the database. We have a Go wrapper for querying the database for an account's username and password.

4.6.3. Authentication

For any website worth using outside the development / playground environment we require good security and authentication practices.

We have implemented a username/password login page that persists logins for a certain amount of time using cookies. Upon logout the cookies are cleared.

The authentication type used is token based authentication. The client sends their credentials (username and password) to a public login endpoint on the web application. If the credentials are correct, the web application issues an access token with an expirty time to the client. Further client requests to protected routes include the access token. We do not use OAuth as we have decided that authentication through third parties is out of scope of the project.

The token is a `JWT` (JSON Web Token). The token is generated and verified using a private key known to the web application only. The simplicity of JWT and the ability to include "claims" in the token is the reason why we have chosen it.

We have written our own authentication middleware to verify the token for all protected routes.

4.6.4. Downloads

In order to send a file to the client and reliability trigger a browser download, we have decided that the best way is to create an unprotected temporary download link. We use a token as a parameter of the link as a form of authentication before initiating the download. We reuse JWT functions that we have written.

4.6.5. Security

Our web communications use HTTPS. We require HTTPS TLS certificates to be passed both for the web server serving the React SPA and the web application responding to API calls from the SPA.

Account passwords are stored in PostgreSQL as hashed and salted using bcrypt.

4.7. Automation Tools

4.7.1. Compilation

We have used the `Make` Unix tool with appropriate `Makefile`'s to compile and test parts of our project with commands as simple as `make` or `make test`.

4.7.2. Deployment

To simplify deployment of our software onto actual nodes we have taken the following steps:

- All deployment specific variables such as keys, certificates, database addresses, etc. go into `.env` environment variable files.
- We have created an `Ansible` (automation tool that configures machiness via SSH) playbook in the `/deploy` directory for automatically distributing our node software binary to a set servers.

4.7.3. Dependency Management

For all `Go` related subsystems, we use "go modules", which produce a `go.mod` file with all required Go dependencies, by automatically scanning the source code.

For the React.js SPA, we use the famous `npm` tool with `package.json` for all dependencies.

5. Installation Guide

All our software is cross-platform and compatible with most modern operating systems, including Windows, Linux, Mac OS X. Some clients such as the web client work anywhere where there is a web browser. We do not require special hardware. The software can manage both powerful and less powerful machines.

For executable binaries we provide precompiled releases on our GitLab (see our releases). Another option is to compile from source.

See the User Manual (Node Administrator section) for more details on set up of the software.

Where there are command-line examples, it is assumed that the environment is Unix (corresponding commands can be found for Windows).

5.1. Node Software

Obtain a binary distribution of our node software (named "cloud").

It is recommended for the node machine to have enhanced storage hardware (in storage space, RAID, etc.) and good or excellent network connectivity.

5.1.1. Compile from source.

Clone the project's GitLab repository: `git clone https://gitlab.computing.dcu.ie/baltrut2/2020-ca326-tbaltrunas-cloudstorage.git`

Change directory into the node software: `cd 2020-ca326-tbaltrunas-cloudstorage/code/cloud`

Compile the software into a binary: `go build cloud`

Or optionally with the `Make` tool: `make`.

Find the node software executable binary under the name `cloud`.

5.2. Desktop GUI Client

Obtain a binary of the desktop GUI client.

The client requires the machine to have a graphical monitor and a graphics driver.

5.2.1. Compile from source.

Clone and change into the repository as in 5.1.1.

Change directory into the desktop client's directory: `cd 2020-ca326-tbaltrunas-cloudstorage/code/cloud/desktop`

Compile the software into a binary with `go build` or `make` (with the `Make` tool).

Find the desktop binary.

5.3. Web Client

The web client exposes a website to the end user. See the user guide for in depth details of how to set up the web client from a node administrator's point of view.

6. Testing

6.1. GitLab CI

In order to enforce quality of code we have configured the available GitLab Continuous Integration (CI) runner to automatically build and test our code upon each commit. We only merge requests if all if the CI pipeline does not fail.

6.2. Unit & Integration Tests

For our code we have written unit and integration (major component) tests. In the case of Go we have `*_test.go` files together with the source. We use the `testing` library from the Go standard library for assertions and a Go test runner. These tests are also ran by the GitLab CI runner.

While we do not have an official coverage reporting tool, we do make sure to test every package for expected behaviour. Go has an excellent compiler and type checking that covers many error cases.

In the case of the web client we have a test that the application runs without crashing.

6.3. System Tests

In order to test our project on actual servers, we have used GCP (Google Cloud Platform), to set up multiple virtual machines at different locations and deploy our node software to.

We have verified that the servers can become nodes and connect into a cloud network.

6.4. User Tests

In the case of the web client, we have performed verbal user testing. The user testing included checking the colour schemes, layout, expected functionality, etc.

We have kept the testing verbal and without record to comply with ethics requirements.