TL;DR Free and secure remote access with NAT & firewall traversal, using Git as backend

DOI 10.5281/zenodo.4673010

Made with Bash License GPLv3 Dependency Git, Bash, cURL
Tested on Ubuntu 12.04, 16.04, 20.04 and Debian 10

Repository SomajitDey/gibberish Maintained? yes maintainer Somajit Dey

What is GiBBERISh

GiBBERISh stands for ***Git and Bash Based Encrypted Remote Interactive Shell***. It is a *free*, *easy* and *portable* solution to *securely* access your *Linux* computer from another *Linux* box over the internet, when none of the machines has a public IP address. You can also transfer files to and fro the remote host, with end-to-end encryption.

GiBBERISh consists of only a short *Bash script*, to be run on both the hosts (viz. *client* and *server*), and an *online Git repository* owned and controlled by the user, such as a *free* repository at any of GitHub, GitLab, Bitbucket, SourceForge etc.

Why GiBBERISh

The standard way to remote access a Linux computer is Secure Shell (SSH). But, in order to connect to the remote host over internet, it has to have a public IP address -- something most personal and work PCs do not have. You can, however, beat this in the following ways:

1. SSH port forwarding at a server with public IP/URL (for some free services with a new URL per session see this and this)
2. Virtual Private Network or VPN

If you can afford any of the above, then GiBBERISh is not for you. However, if you are like me, who doesn't want to pay for a VPN or port-forwarding service with static URL, you are probably out of luck with SSH. Also, some corporate firewalls block non-web traffic entirely, hence SSH is not even an option there.

However, you can still manage fine with a freemium remote access app, such as TeamViewer or Anydesk. If you are happily using such Virtual Network Computing (VNC) tools, then too you won't have any use for GiBBERISh. However, VNC apps have their cons. They consume a lot of bandwidth. If you want to access just the shell of your remote host, why would you need to import the entire screen! The remote computer also need to be online all the time. Another common problem is version incompatibility. For TeamViewer, for example, every machine needs to have the same version of the app installed on it, or else it won’t work. In addition, being freemium, what they offer for free is limited and you never know how long they would keep offering those free services. To top it all of, you can do nothing but trust these service providers when they say that using their services is completely safe for your systems and your data.

Anyways, I needed a light-weight, portable, easy-to-install and completely free yet reliable platform to access my remote machine securely over the internet. I didn't mind the latency, as long as my main purpose of submitting jobs to the remote host was served. Hence, my DIY solution - GiBBERISh.

How it works

An interactive shell is nothing but the user chatting with the operating system (OS), occasionally poking the OS with interrupts or job-control signals. For GiBBERISh, the user's local machine is the **client**, and the OS in the remote host is the **server**. Correspondence between the two is managed by an automated *Git* workflow.

Git, if you don't know about it already, is a widely-used, fast, lightweight, distributed version-control system or content-tracker. Because it is distributed, widely available, and easy to use, you can use it to synchronize two machines quite easily. If you *push commits* (i.e. make some changes) to a Git *repository* (i.e. a directory with *history*) in one machine, you can synchronize or update its *clone* in another machine simply by *fetching* those new commits from the former repository (called *upstream* in Git-speak).

Git can connect two machines over the internet for push/fetch only if at least one of them has a public IP address or URL. However, one can have a free, if not private, online Git repository with a public URL hosted in any of the popular Git-servers, such as GitHub, GitLab, Bitbucket or SourceForge. Although anyone can view (and fetch) the contents of your online repository, you remain in control of who can make changes to your repository. This is because all push access is protected by your password or an access-token generated by you.

How GiBBERISh connects two hosts (client and server) over the internet without any of them having a public IP address

Whenever the user enters a command at client, the GiBBERISh script encrypts it with your Git credentials using a state-of-the-art, yet free, cryptography service called GNU Privacy Guard (GPG). It then commits this encrypted text to the user's local repository and pushes the same to the user's upstream repository (say, at GitHub). Server's script then fetches this commit from upstream, decrypts and pipes the user's command to an interactive Bash. The output emitted by Bash is again encrypted, committed and pushed to GitHub, which the client fetches, decrypts and shows to the user. (For an original implementation of a similar Git-based chat, see https://github.com/ephigabay/GIC).

Note: Because only encrypted text goes to GitHub, the public cannot see what commands the user is running, or their outputs.

Latency:

Because of the dependence on an online Git repository, the time between entering a command and getting its output back is not insignificant. This is because `git-push` is slow. From multiple measurements with upstream at GitHub, I found the latency varies between 6 to 9 seconds. However, using GitHub's REST API, the `git-push` can be avoided and performance improved. When using the API, the latency decreases to 3-4 seconds.

Features:

1. Doesn't require a public IP address. Both machines can be behind multiple NATs.
2. Doesn't care about firewalls and blocked ports.
3. Secure (everything public is encrypted with your password / access-token).
4. Interactive. Unlike SSH, however, the raw user input is not streamed to server. The user can type, edit, erase as many times as necessary, with zero-lag, before pressing the `Enter` key.
5. Secure (encryption using AES256) file transfer from client to server and vice-versa.

6. Easy and fast switching between local and remote environments without interrupting the remote session in any way. See *brb* in the *Keywords* section below.
7. Relays user's keyboard-generated signals, such as Ctrl-c; Ctrl-z to server.
8. Monitorability and overrides: If you grant someone else access to your local machine, for remote diagnostics for example, you can see all the commands she is executing from your terminal. You can also override those executions with Ctrl-c, Ctrl-z, Ctrl-\ etc., if necessary.
9. Forever free. Given the popularity of Git in DevOps, freemium services such as GitHub are here to stay and they probably will continue hosting small public repositories for free for years to come. GiBBERISh is careful about keeping the repository size as small as possible. So, the size limit of the free-tier plans should never be an issue.
10. Lightweight: CPU usage is minimal. Polling and busy-waits are avoided wherever possible in favor of event-driven triggers.
11. Reliable: If any of your machines goes offline, it automatically gets connected once its internet connection is restored. No data is lost. Just make sure your remote host stays up even when there is power-outage.
12. Hassle-free installation, portability and flexibility: GiBBERISh only runs Git, and some basic Unix commands, all from a short, simple, stupid (KISS) Bash script. Most current Linux distributions ship with Git and Bash both. Hence, GiBBERISh should run readily on those. No superuser or admin privilege is required to install or run GiBBERISh. You also hold the perpetual right to adapt the source-code to your needs.
13. Everything is under your control. You are free to modify the single Bash script that GiBBERISh runs from. You own and manage the upstream repository. If you are connecting to your work computer from your home machine or vice versa, you control both the machines. You also choose who can access your machine, should you ever be granting someone else remote access for purposes of diagnostics, instructions etc.
14. Because Git and Bash are the only main ingredients, GiBBERISh (in an implementation that doesn't use `flock`) maybe run easily on Git-Bash from the Git for Windows package. However, that might be unnecessary, given that Windows 10 now ships with a subsystem for Linux (WSL).
15. Stores command history for the session. Use the Up, Down arrow keys or Ctrl-p and Ctrl-n to access history as usual.

How to install / run

First, create a dedicated repository at any free Git hosting service such as https://github.com/, https://gitlab.com/, https://bitbucket.org/ or https://sourceforge.net/. I recommend using GitHub, as GiBBERISh can speed itself up using their content API. The repository can be completely empty, i.e. without any commits.

Tip: Also generate a personal access-token (PAT) for your account. This makes GiBBERISh even more secure.

Tip: If you have created the repository at GitHub, then install `jq` and `base64` to experience the lowest latency (3-4s) with GiBBERISh. However, you can also run GiBBERISh without these, albeit with double latency.

At the Linux machine that you want to use as server, run the *installer* script (no superuser or admin privilege required):

```
./installer
```

Then, simply follow the on-screen instructions.

To start the server, run:

```
gibberish-server
```

At the client machine, install GiBBERISh similarly as above.

To access the remote server, simply run:

```
gibberish
```

Note: GiBBERISh allows, possibly unnecessarily, upstream repositories that are on your local disk or NFS.

Keywords or built-in commands

GiBBERISh recognizes a few keywords as listed below.

ping | **hello** | **hey** | **hi**

To test if the server is still connected. Consider the following situation. You are running a foreground process on the server, which outputs infrequently. Because it is in foreground, you do not have the command prompt and hence cannot execute a short command such as `echo` to see if the server is still responding. Just enter any of these keywords, and the server will send you a 'hello' if it can hear you, without interrupting that foreground process in any way. However, you can also do a simple `Ctrl-<spacebar>` or `Ctrl-z` to get back the prompt, at the expense of stopping the foreground process. **Note**: You can use any of these commands anytime, even if a foreground process is running at the server and you do not have a prompt.

exit | **quit** | **logout** | **hup** | **bye**

To end the session, and disconnect or hangup. When you do this, the current interactive shell in the server is closed and a new, fresh shell is launched ready for the next session. You therefore, would lose any environment variable you had set or function definitions you had sourced during the last session. Beware that this should also close any process running on the server that the exiting shell sends SIGHUP to, unless the process has a handler installed for HUP. Start processes in background with `nohup` if you intend to keep them running even after you logout using these keywords.

brb

Be right back, viz. to quickly switch to your local environment for a short while, without ending or interrupting the remote session. Any foreground process running in the server, keeps running uninterrupted. With this keyword, you simply get back your local command prompt, whenever you need to run commands locally during a remote session. To return to the remote session, just enter `gibberish` again. You will be shown all the server output since the time you did `brb`, so you miss nothing.

local

Run commands locally, i.e. at the client, inside a sub-shell. The syntax is:

```
local <commands>

# To run codes from a file
local . <path to script>

# To see current working directory at client
local # equivalent to: local pwd
```

rc

Run commands. This is akin to the `.` or `source` built-in of Bash. Whereas `source` reads commands from a local file and executes them in the current shell, `rc` takes commands from a client-side file and executes them in the server-side shell that the user is currently interacting with. The syntax is:

```
rc <local path to script>
```

All the commands in the given script are passed to upstream in one Git-push. Hence, use of this keyword helps with latency issues.

take | **push**

See next section

bring | **pull**

See next section

help | **tutorial**

Gives the link to this section

latency |**rtt**

Gives the latency or round-trip-time for the current connection in seconds. **Note**: You can use any of these commands anytime, even if a foreground process is running at the server and you do not have a prompt.

Tip: If you need to run a command that matches any of the keywords described above, use the `command` built-in of Bash.

File and directory transfer

Copying file from client to server:

```
take <local path> <remote path>
#### or
push <local path> <remote path>
```

Copying file from server to client:

```
bring <remote path> <local path>
#### or
pull <remote path> <local path>
```

File transfer is atomic, which guarantees you never end up with a corrupt file, even if the transfer operation gets interrupted or terminated prematurely. If the destination file is existing, it will be overwritten after backup. If the destination path is a directory, the file would be put inside it. The paths can be absolute or relative. As should be intuitive, any relative path would be interpreted with PWD at the corresponding host as its base, i.e. relative local (remote) path would be relative to the client-side (server-side) working directory. Similarly, tilde and shell-variable expansion in the path specifications, are done with respect to the corresponding host.

Tip: To transfer directories or a collection of files, archive them first, with `tar` for example, and then use the above commands to exchange that single archive file.

Tip: To transfer a file from the Windows filesystem in WSL, first change its path to Unix path using the command: `wslpath`.

Note: File transfer is end-to-end encrypted with your Git credentials. To keep your Git repository size small, the files are transferred using free, public file-hosting servers.

Keyboard-generated job-control signals

All the familiar Ctrl key generated signals are supported except `Ctrl-\`, which has been replaced by `Ctrl-e` (mnemonic: 'e' for exit). Because the user doesn't have the liberty to open a second terminal to control a runaway foreground process that ignores SIGTSTP (as generated with `Ctrl-z`), GiBBERISh provides `Ctrl-<spacebar>` key-binding to force pause a foreground process with SIGSTOP - which cannot be ignored or handled.

Try GiBBERISh at localhost

If you are a first-time user, you may want to get familiar with GiBBERISh at your local machine. To do that, install GiBBERISh twice, once as server and then as client. Now, open a terminal and run:

```
export GIBBERISH=server
gibberish-server
```

Then, open another terminal and run:

```
export GIBBERISH=client
gibberish
```

Granting remote-access while you monitor

There might be desperate situations where you need to grant access to your local machine to someone else. To do that safely, do the following.

1. If your machine is not already configured as server, install GiBBERISh to run as server.
2. Create a temporary access-token at your upstream account.
3. Send the access-token to the person you are granting remote access to and ask her to install GiBBERISh as client with the given token as password/PAT. [If she already has GiBBERISh client configured with a different password, ask her to backup her `~/.gibberish` directory before reinstallation. After the session, she can simply restore the backup.]
4. Run: `gibberish-server <access-token>`
5. Ask the other person to run: `gibberish`

6. Monitor everything on-screen. Whenever you feel you need to interrupt any command the remote client is running on your machine, use `Ctrl-c`, `Ctrl-z` or `Ctrl-\` as usual. In the worst case, simply close the terminal. It would kill all processes spawned in that session.
7. After the session is over, close your terminal to kill the server.
8. Revoke the access-token from your upstream account.

Library, not executable

The Bash script for GiBBERISh, viz. *gibberish_rc.bash*, is a library rather than an executable. On startup, your interactive Bash sources this script. The *installer* puts the corresponding run-command in .bashrc.

The *installer*, on the other hand, is an executable script. Hence, do a *chmod +x* on it, if necessary.

Internals (pointers to understand the code)

The client-side and server-side codes are almost the same structurally. This is why a single shell-script suffices for both. There are two types of events:

1) Internal events: User@client or Bash@server creates new data to push

2) External events: Git-fetch brings new data from outside to display or process

Internal events are mostly waited for, while external ones need to be polled for continuously. Everything else is mostly triggered by these events. For example, upon creation, the data to be pushed is first siloed and an encrypt-commit-push pipeline is triggered.

The Git repository for GiBBERISh has two linear branches, viz. server-branch and client-branch. Server fetches from server-branch, but pushes to client-branch. Client does the converse. Every user-command@client or Bash-output@server, is committed to the proper outgoing branch as a single text file. As new commands or outputs become available, this file is simply revised to hold the same. The Git-worktree, therefore, always contains only a single file, and all the commands/outputs can always be tracked from its revision history. Every time a commit happens, it triggers a post-commit hook that pushes the commit automatically.

Every iteration of Git-fetch triggers a checkout sequence that tracks all the new commits chronologically, restoring the corresponding version of the abovementioned text file for decryption. Upon successful decryption, the checkout function pushes the data down a pipeline to user (@client) or Bash (@server), before moving on to the next commit in the git-revision-list.

Note that if every user-command was passed using that single text file alone, then all the commands would end up in the queue for Bash@server. To relay a user-generated signal (SIGINT, SIGTSTP, SIGQUIT) to a foreground process on demand, however, we need to route `kill` commands to a second, parallel shell in the server. We, therefore, need a way to commit commands in the repository other than the single text file mentioned before. GiBBERISh exploits Git's commit-message field for this purpose. Only those commits that carry the user-generated control signals, or any other command that is to be executed by the parallel shell only, would have commit-messages which hold the commands themselves. All other commits have empty commit-messages. Whether the commit is meant for the main shell or the parallel, can therefore be ascertained simply by checking whether the commit-message is empty or not.

Such commit-message commands, meant for the parallel shell, are mostly kill(@server) and file-download(@client) commands. Hence, they do not contain any sensitive data and are consequently not encrypted to save on time. These commands are called **hook**s. Both server and client share this hook-execution architecture. Hooks are the only way server can make client do some work, such as required during file transfer from server to client.

In contrast to SSH, GiBBERISh does not stream every keystroke made by the user to the server in real-time. Rather, it first lets the user enter the complete command, then reads it, checks for keywords, and only then decides what to do. If the command-line is not a keyword, it is pushed to the server as is for execution. To generate signals at server from particular key-sequences entered at the client, key-binding is done at the client-terminal such that it commits and pushes the appropriate hook to be executed by the parallel shell at server, whenever those keys are pressed by the user.

The current prompt at the server-side terminal is appended to the text file in the repository below the PGP message block containing the output of Bash@server. When a foreground process is running in that terminal, only a newline is appended instead. Client extracts this appendix to remain informed about the current state of the dynamic server-side prompt, and displays the same at the client-side terminal as required. This feature is strictly designed for adding to the user experience and is in no way a necessary part of the GiBBERISh engine.

Strategy to decrease latency

`git-push` is the most obvious way to transfer a file to upstream. However, it is slow (~3s). Using a file-update API, if provided by the upstream host, can make file-transfer faster (~1s). GiBBERISh uses the REST API of GitHub for this purpose. The speed up is significant:

Using API: 3-4s | Using git-push: 6-9s

Bug-reports, Feature-requests, Comments

Create issue(s) and comment at https://github.com/SomajitDey/gibberish

You can also contact me directly at dey.somajit@gmail.com

Legal

GiBBERISh-- Git and Bash Based Encrypted Remote Interactive Shell

GiBBERISh comes with ABSOLUTELY NO WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. Use it at your own risk. The developer or copyright holder won't be liable for any damage done by or done using this software.

Acknowledgements

Soumalya Bhowmik contributed by testing this software thoroughly.

Thanks to Patrick Bedat and Seth Kenlon for encouragement and discussions.

Future directions

https://github.com/SomajitDey/gibberish/projects/1