Go-based Egress Monitor

JumpCloud 1

June 17, 2014

Garrison Davis
Tyler Ferguson
Jason Santilli

Table of Contents

1. Introduction

1.1 Client Description

JumpCloud provides services for hosting servers remotely, so that organizations can focus on innovation and creating content, and not on managing their server infrastructure. The JumpCloud agent, their tool for server management, consists of several standalone services that, collectively, monitor and react to events on a system and enable easy, automatic, server maintenance. When these events are detected, they are either communicated back to the JumpCloud servers, which may take actions in response to these events, or they are kept locally and logged for ease of administration.

1.2 Product Vision

The JumpCloud agent has pre existing triggers, such as a UserMonitor trigger, that fires an event whenever a user is added to, or removed from, their service. To add to the existing triggers of their agent, JumpCloud has requested an egress monitor, a trigger that monitors outbound connections from a server and notifies the agent when outgoing connections occur. The monitor must use existing JumpCloud interfaces so that it can be easily integrated into the architecture and utilize the agent's listeners to relay relevant information.

2. Requirements

2.1 Functional Requirements

- The Egress Monitor must be able to integrate with the existing JumpCloud agent
    - Implement a Service interface
        - Implement Init functionality to be called by the agent once to initialize relevant data structures
        - Implement Start functionality to be called by the agent multiple times to initialize the listeners and begin monitoring
        - Implement Stop functionality to be called by the agent multiple times to gracefully stop monitoring and clear the list of listeners
    - Implement an EventEmitter interface
        - Implement RegisterListener functionality to add a given listener to an internal list of listeners
        - Implement ResetListeners functionality to re initialize the list of listeners
    - Detect relevant event
        - Pass the necessary information back to the agent in a specific format
    - Design a format for the result information
        - Contain local IP, remote IP, local port, remote port, process name, and process ID
        - JSON format

2.2 Non-Functional Requirements

- The project must comply with the standard Go project structure to make it easily cross-platform compatible.
- The project must come with thorough unit tests.
- The project must be designed in a way that utilizes the unique features of Go, particularly concurrency.

3. System Architecture

The system consists of three main components. First is the Linux server, which allows inbound and outgoing network traffic. Next is the Egress Monitor, the focus of the project. This is a tool that monitors network traffic on the Linux server. Then is the JumpCloud Agent, which responds to events on the Linux server via triggers such as the Egress Monitor. Last, the JumpCloud Server hosts a number of services, utilities, and information. When an outgoing connection is made on the Linux server, the Egress Monitor detects this and inspects the connection to gather relevant information such as IP addresses, ports, and process IDs. This information is related the JumpCloud Agent. The Agent communicates with the JumpCloud Server, which will make a decision regarding how to handle this connection. It then informs the Agent how to proceed. See Figure 1 for a pictorial representation of this interaction.

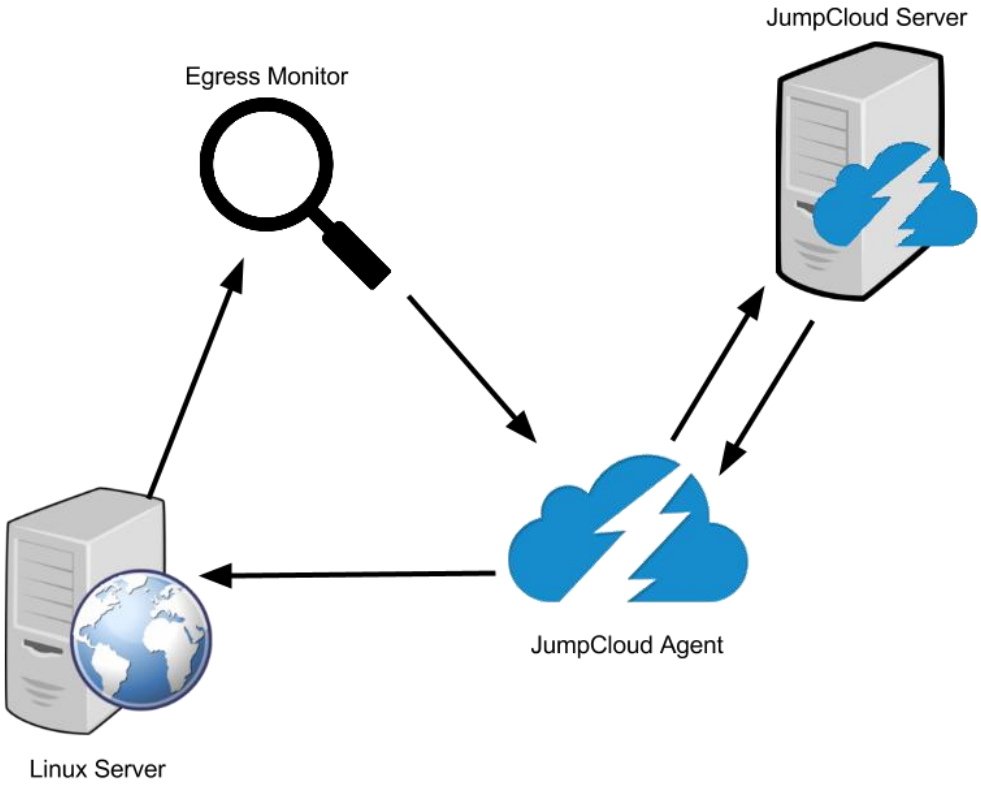

Figure 1

4. Technical Design

4.1 Process Flow

When the JumpCloud Agent launches the Egress Monitor, it first initializes a map of all seen connections. Next, it opens a pipeline that can detect all outbound network connections. As a connection is detected, the monitor collects information regarding local IP address, local port, remote IP address, remote port, process name, and process ID. This information is then compared against all other previously seen connections in the created map. Once this connection is found to be new, the various details of the connection are compiled and stored in JSON format. The monitor then triggers each of the JumpCloud Agent's listeners, passing in this JSON object.

The listeners are designed to facilitate the passing along of the information stored in the JSON object to the JumpCloud Servers. Once the JumpCloud servers have the particulars of the network connection, they are able to make a decision regarding how to handle said connection. This contact with the JumpCloud servers and decision-making process are outside the scope of this project, however. See Figure 2 for a diagram of the Egress Monitor's overall process flow.

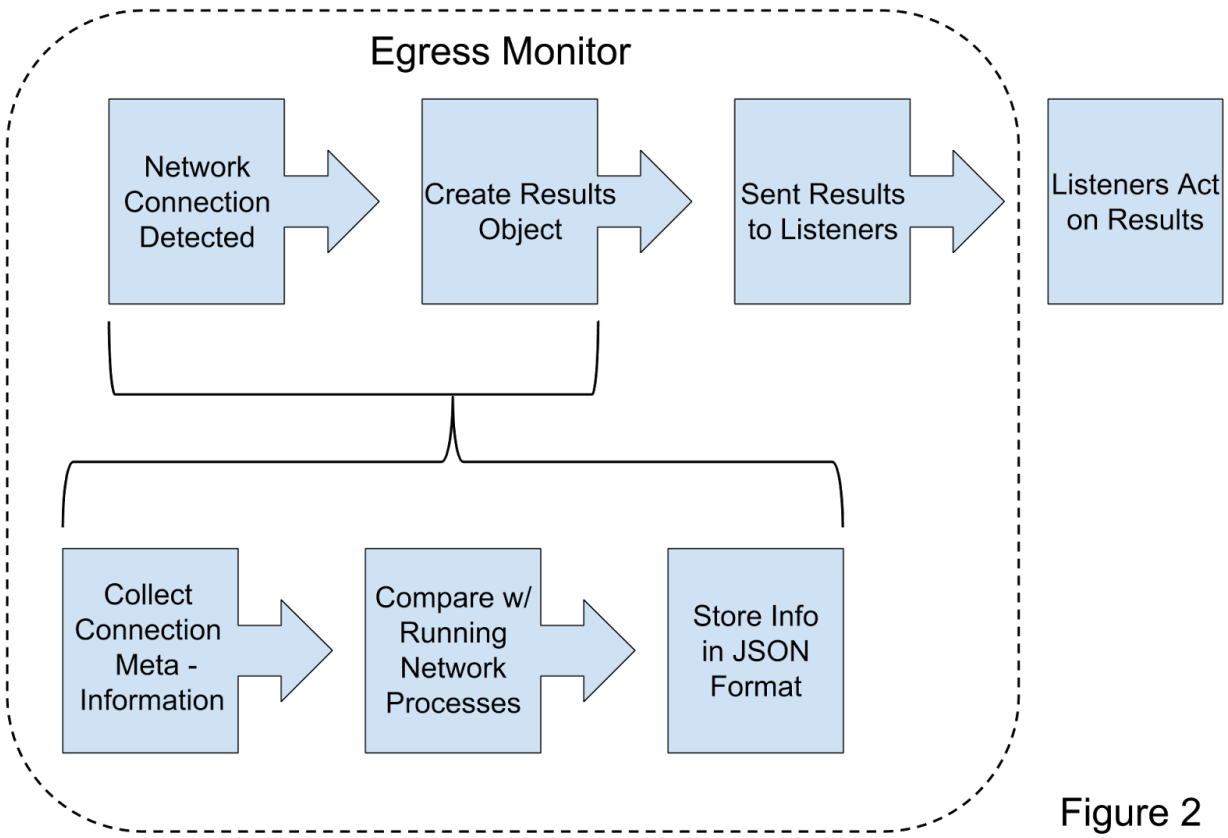

Figure 2

4.2 Reporting Results

The Egress Monitor collects and parses outgoing connection information from the Linux server using a Go pipeline of Linux utilities. this pipeline is the equivalent of running the following in the terminal:

```
ss -atunp src ${ipAddress} | grep -e “tcp” -e “udp” | sed -E ‘s/[[:space:]]+/ /g’ | cut -d\  -f 5-
```

Where ipAddress is each local IP address.

Once the Egress Monitor obtains the local IP address, local port, remote IP address, remote port, process name, and process ID from the pipeline output and converts this information into JSON format, it compares the information to a map of information from previous connections. If this connection is present in the map, it has already been reported to the JumpCloud agent and will be disregarded by the monitor. Otherwise, the connection has not been reported and the monitor calls the JumpCloud agent’s listeners to report the new connection and adds the information to the map. If adding the new information causes the map to reach maximum capacity, the map will be emptied so that older, irrelevant connection information is discarded and newer and persistent connection information is reported to the agent. See Figure 3 for a state diagram outlining the reporting process.

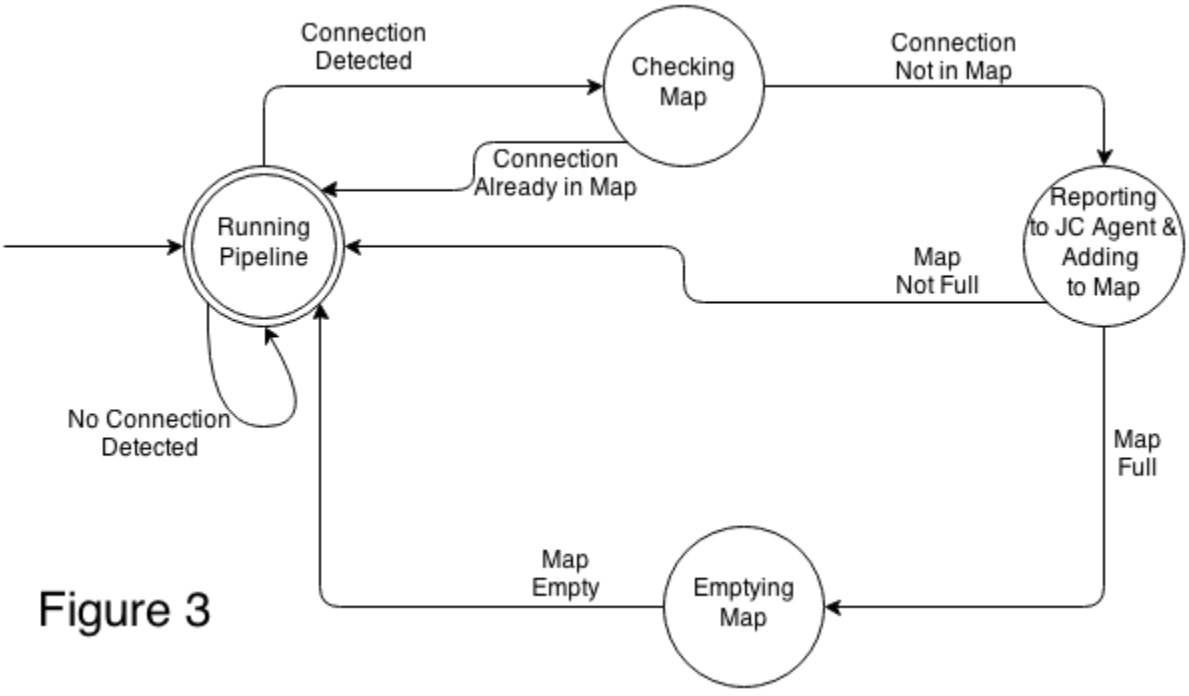

Figure 3

5. Design and Implementation Decisions

Throughout the development of the Egress Monitor numerous design options were considered. Ultimately, three major design decisions were made on the most prominent parts of the project: remaining thread-safe when using concurrency, collecting outbound connection information, and reporting back to the JumpCloud agent.

The first significant decision was the use of mutexes to manage the Egress Monitor's internal data in a way that is thread-safe. Mutexes were used rather than channels, the built in concurrency primitives in Go, because the internal data is stored in slices, which are not thread-safe. While a channel could be used to maintain thread-safety when managing slices, a mutex would be required regardless, to ensure that the program does not attempt to read from an empty channel and crash. By only using a mutex, the code remains simple and understandable as unlocking a mutex merely requires use of Go's `defer` command.

The second major design decision was the use of a Go pipeline that utilizes Linux utilities to collect the relevant connection information. Initially, the Egress Monitor was designed to use packet capture with Google's Go libraries for pcap. Because pcap is installed by default on the vast majority of Linux distributions and is written in C, allowing for direct integration with Go, it seemed like the best option, but upon further testing two issues arose. The first problem was the inability of pcap to gather the name and process ID of the process responsible for the connection. Secondly, when subjected to tests with extremely high numbers of connections, pcap proved to be fairly heavy on memory usage. To fix these problems, a Go pipeline using linux utilities ss, grep, sed, and cut was implemented. When subjected to the same tests, the pipeline used a fraction of the memory that pcap required and was able to return the process ID and name.

The final important decision was the use of JSON to format the information gathered by the Egress Monitor for reporting to the JumpCloud agent. JSON was chosen for a number of reasons. Firstly, it is a format that is easy to read and understand for testing purposes as well as compare for determining what information to report to the JumpCloud agent. In addition, Go includes tools to generate and format JSON simply and quickly. Finally JumpCloud believes that receiving the information from the Egress Monitor in JSON format will be the best for their agent and has agreed to ensure that it will be dealt with correctly.

6. Results

6.1 Benchmarks

Running the loop responsible for listening to incoming connections 100 times, yielded an average of 18.7 ms/cycle, on a machine running Ubuntu 14.04 with 5.8 GiB of RAM, and an Intel i7 920 @2.67 GHz. Additionally, while running idly, the project consumes an average of 5 MiB memory.

The project checks the `/proc` tables (through ss) which do not clear that quickly, so missing connections is not a point of concern.

6.2 Future Work

Currently, the project lacks Windows support and does not completely have IPv6 support.

During the testing and design phases, the Windows functionality that was implemented was found to be too error-prone and unreliable, as well as having massive memory usage. It is currently a design problem that warrants further investigation.

IPv6 support is nearly implemented; however one of the helper functions is currently unable to parse IPv4 and IPv6 addresses, which it would need to do to enable full IPv6 support. This is a problem that could likely be fixed within a few hours, but was outside the scope of the project.

7. Appendices

Appendix A. Example Implementation

The following is an example use case (similar to the demo prepared for the presentation) of the egress monitor.

```
// Inside a file, for example, main.go
package main

import (
    "time"
    "github.com/TheJumpCloud/egress-monitoring/trigger"
    "github.com/TheJumpCloud/egress-monitoring/util"
)

func main() {
    // Initialize a new EgressMonitor
    egressMonitor := new(trigger.EgressMonitor)

    // Call the run function, which currently only calls both the
    // Init and Start methods.
    run(egressMonitor)

    // Block for 10 minutes here (the listening happens unabated in a
    // goroutine) to simulate the passage of time.
    select {
    case <-time.After(10 * time.Minute):
        // After 10 minutes, stop listening.
        stop(egressMonitor)
    }
}

func run(service util.Service) {
    service.Init()
    // Here, listeners can be added.
    // Additionally, while running, adding additional
    // listeners after Start is called is supported.
    service.Start()
}

func stop(service util.Service) {
    service.Stop()
}
```

This use case assumes that whatever listener is being used is able to print or verify the results itself.