University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Hubert Badocha
Student no. 417666

Jakub Moliński
Student no. 419502

Tomasz Domagała
Student no. 417805

Paweł Pawłowski
Student no. 418369

Drop-in Ansible replacement in Go

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz
Institute of Informatics

Warsaw, June 2022

Abstract

Go is a modern programming language that is well suited for building distributed systems and is the leading programming language for building DevOps tools — it's the main implementation language of Kubernetes, Docker, and Terraform.

Ansible is open-source software for provisioning, configuration management, and application-deployment tool enabling infrastructure as code. It includes declarative language to describe system configuration.

However, the current implementation of Ansible has many downsides. Firstly, it has performance issues during deployments involving a large number of managed nodes. Secondly, it is written in Python which makes it difficult to integrate with other DevOps tools. Moreover, it requires Python to be already installed on the server that it sets up, or it complicates the playbook in order to install the Python first.

The purpose of this thesis is to create, benchmark, and describe the drop-in replacement of Ansible implemented in Go. A subset of Ansible functionalities was implemented as part of this thesis. The subset is representative enough to run real-life playbooks.

Keywords

Ansible, Go language, deployment, Python, system configuration

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

Software and its engineering
    Software notations and tools
        Software configuration management and version control

Tytuł pracy w języku polskim

Zamiennik Ansible zaimplementowany w Go

Contents

Introduction

Server provisioning is the process of setting up IT infrastructure. As the computation demand has rapidly increased over the years and in the era of cloud computing it become ever more important for companies. Being able to quickly and easily set up a server has become one of the core concerns when developing a large-scale system.

Ansible [2] is one of the most popular tools to configure systems, deploy software, and orchestrate more advanced IT tasks such as continuous deployments or zero downtime rolling updates. It uses so-called playbooks, which are YAML files that define on which host and in what order to execute a given task. The simple format of playbooks makes Ansible configurations readable and easy to use. Ansible is also very extendable, and users can write their plugins to almost any part of the system. All of this translated to the popularity of Ansible in the world of DevOps. However, its default architecture tends to perform poorly when Ansible executes a playbook on a significant number of managed nodes. Some companies (e.g. ScyllaDB) have playbooks, the execution of which takes several to several hours. It is also difficult to integrate with other popular DevOps tools, as many of them are implemented using the Go programming language, while Ansible is written in Python.

Ansible wrappers in Go do exist, but they do not address the performance issue, and the integration with other Go code is limited by what commands Ansible provides. Also, there exists "a toy implementation" [1] of Ansible in Go from 5 years ago, but it is not compatible with the original and does not support a lot of Ansible features.

Together with ScyllaDB we concentrated on remedying Ansible's problems by introducing Gosible, the reimplementation of Ansible in Go. The reimplementation has several advantages:

- it makes integration with other DevOps tools much easier,
- our version solves the performance problem,
- Ansible's codebase is huge and complicated:
  - code often looks as if it was done as quickly as possible without ever being fixed,
  - there are often completely purposeless pieces of code that are never run,
  - documentation in the code often is not clear, not descriptive enough, or empty,
  - files and classes are often a few thousand lines long.

  We want to determine if it can be simplified by using techniques used in Go.

Our thesis is split into the following logical parts:

- **Chapter 1** is an overview of the underlaying technologies as well as introduction to Mitogen and Ansible.
- **Chapter 2** describes architecture of Ansible and the modifications that Mitogen for Ansible implements.

- **Chapter 3** describes architecture of Gosible.
- **Chapter 4** presents results and methodology of our benchmarks.

The results of the thesis are summarised in the last chapter.

Chapter 1

Ground work

Through this chapter of the paper we will go through fundamentals important for both Ansible and Gosible.

1.1. Introduction to Ansible

Ansible is a well recognized cross-platform software written in Python used to manage and provision servers. The main concepts of Ansible are playbooks which describe the actions to execute and inventory files which list the servers that Ansible will execute the actions on. Actions can be limited to only a subset of the inventory list.

Ansible allows mass management of servers, called nodes. In addition to the target nodes, there exists a control node that coordinates the execution. The software can be extended using plugins and modules, where plugins are executed on the control node and extend the core functionality, and modules are generally executed on the target nodes and either gather system information or provide task types for playbooks. The core functionality can be defined as actions such as playbook parsing, inventory management, logging, and connection to the target systems [3]. The most commonly used connection transport protocol is SSH. Modules are generally copied to the target system as Python code for every invocation of an action.

1.1.1. Playbooks

A playbook [5] is an ordered list of plays. Each play is composed of tasks. Each task makes a call to an Ansible module. Plays and tasks are executed from top to bottom. Each play specifies the groups of target nodes that it should be executed against. Here is a simple example of a playbook:

```
---
- name: Update web servers
  hosts: webservers
  remote_user: root

  tasks:
  - name: Ensure apache is at the latest version
    ansible.builtin.yum:
      name: httpd
      state: latest
  - name: Write the apache config file
```

```
    ansible.builtin.template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf

- name: Update db servers
  hosts: databases
  remote_user: root

  tasks:
  - name: Ensure postgresql is at the latest version
    ansible.builtin.yum:
      name: postgresql
      state: latest
  - name: Ensure that postgresql is started
    ansible.builtin.service:
      name: postgresql
      state: started
```

1.1.2. Inventories

An inventory [4] is a list of target nodes. By default, it is represented as a text file (most commonly in INI format). Inventory may also specify to what groups a node belongs to (e.g. webservers, dbservers).

Inventory data can also come from dynamic sources (e.g. AWS EC2). Plugins can be used to connect to nonstandard inventory sources.

Here is the same inventory represented in INI and YAML formats:

inventory.ini

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

inventory.yaml

```
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
```

```
dbservers:
  hosts:
    one.example.com:
    two.example.com:
    three.example.com:
```

1.1.3. Modules

The scripts that get pushed to target nodes and that actually execute the actions are called modules. Most modules take arguments that describe, in a declarative manner, the desired state of the system. Once the execution is finished Ansible removes the module from the target node — they are not persistent. It is possible to create custom modules and Ansible provides a library of commonly used module utilities to facilitate this.

1.1.4. Plugins

Plugins can be used to modify or extend the functionality of Ansible's core. They are executed on the control node. A plugin may specify, for example, how to connect to an inventory data source (e.g. a cloud service provider's API), how to log output, how to manage cache, and more. It is possible to create custom plugins.

1.2. gRPC basics

Remote Procedure Call (RPC) is a term used for the action of executing a procedure in a different address space than a computer program calling it, most commonly on another computer.

gRPC [16] is an open-source, highly performant, multiplatform RPC framework developed by Google. It is popular for its idiomatic libraries in many languages as well as its incredible ease of use. For visual explanation see figure 1.1.

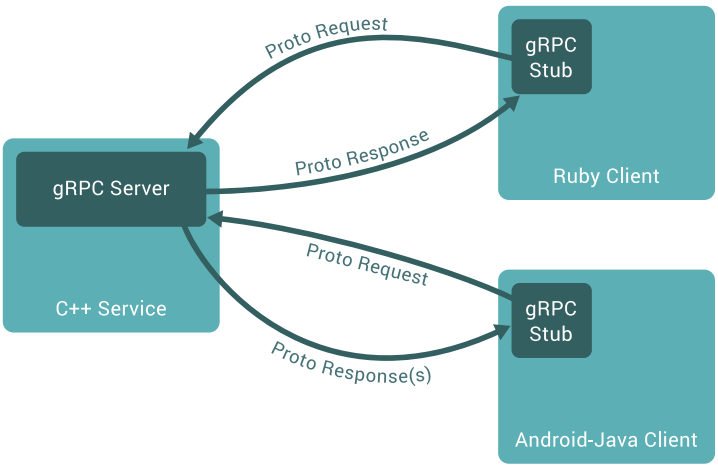

Figure 1.1: Overview of the gRPC architecture [15]

In gRPC you have to define a service that specifies its interface of methods that can be called from clients. What is especially convenient is that clients can be written in other languages than the service and still cooperate. Also multiple clients can connect at once.

gRPC uses HTTP/2 to stream data and Google's Protocol Buffers [20] to serialize data structures.

Methods take an argument and return either streams or unary messages. Streams can be priceless when we have large amount of similar queries to perform as we do not have to establish connection, send metadata etc. every time we want to perform one of these queries what significantly reduces time.

At the start of the connection metadata is exchanged. It can be used to exchange needed information about the client or the server before executing the procedure, eg. what operating system is the server running.

Connection can be terminated by either side at any time. Also client can specify deadline for execution of the procedure and connection will be automatically closed when this deadline exceeds.

1.3. Comparison to Mitogen

Mitogen [19] is a custom module runtime and connection layer, that speeds up Ansible execution. Mitogen in general achieves a similar goal to our project with a similar architecture. Both our project and Mitogen use a single connection per target node and require a single network roundtrip per action invocation.

However there are also differences. Our project opted to send a single precompiled Go binary when connecting initially, whereas Mitogen instead caches the uploaded modules in RAM and reuses processes. Mitogen's approach makes sense in the Python world since precompiling is generally not viable and doing this caching allows saving time on both upload and code compilation. Uploading precompiled code also has its demerits, since a large part of the uploaded code will probably never be executed and makes supporting 3rd party modules more difficult, however it allows for persistent caching of the binary which would allow successive executions to be significantly faster. Another important difference is that our project is written in Go, which could allow some performance gains due to running natively and allow more seamless integration with a Go-based technology stack (software such as Kubernetes, Docker, etc. are all written in Go).

Chapter 2

Architectures of Ansible and Mitogen for Ansible

2.1. Ansible

Ansible supports various connection types out of the box and can be easily extended using plugins to support even more options. Although, by far the most popular connection options are **SSH** and **local**. In further sections, we will cover only how **SSH** works and only in Linux as this is the option that we implemented in Gosible.

2.1.1. Overview

When playbook is being executed a control node creates a single SSH connection to the remote node. Using this connection whenever it wants to execute a play it sends corresponding module file to the remote node and runs it. Before the module exits it prints its output to the stdout in a JSON format. Control node can then read the output and has information if the run succeeded, which files got created, deleted and changed and how, or about one of many other module specific options. This architecture is on one side brilliant as it makes creating new module very straightforward, but on the other hand it is very inefficient as for each play module needs to be prepared, sent, and executed and each action adds a considerable overhead. Moreover, sending plain JSON objects can be slow as they may be enormous.

2.1.2. File preparation

Before the module script file is send to the remote node it needs to be bundled and tailored for current play. Ansible fills the module template, bundles it with other necessary Python files such as **AnsibleModule** class that is used vastly among modules to communicate with control node or to execute commands. Then it sends it to the remote node and executes it.

2.1.3. Become

You may have realized that a problem arises when you want to execute a specific play as a different user than the one who is logged to the remote node via SSH. Fortunately, Ansible has a very easy-to-use tool to handle such a scenario, the **become** plugin. It allows you to change the user who is executing plays by specifying him in a playbook. Then you can provide a password to this user either in inventory using **Ansible Vault** or using command line argument or just password prompt as a task is executing. Become has many plugins

for various platforms. The way **su** plugin for Linux works is extremely simple. Instead of executing

```
# python <module_file>
```

on remote node Ansible executes

```
# su - <user_name> -c "python <module_file>"
```

and inserts the password whenever it is prompted for it. This approach works as each play is executed as a separate process so each play can be played with the corresponding user. Unfortunately, it also has its own issues when the provided user is not privileged [10].

2.1.4. Variables, facts and argument templates

Ansible allows you to read and set variables, often referred to as vars. Variables come from a multitude of sources and each source has a specified precedence. If two or more sources define a variable with the same name, Ansible will use the value coming from the source with the highest precedence. The sources are, namely (in order of precedence, lowest to highest) [8]:

1. command line values (for example, -u my_user, these are not variables),
2. role defaults (defined in role/defaults/main.yml),
3. inventory file or script group vars,
4. inventory group_vars/all,
5. playbook group_vars/all,
6. inventory group_vars/*,
7. playbook group_vars/*,
8. inventory file or script host vars,
9. inventory host_vars/*,
10. playbook host_vars/*,
11. host facts / cached set_facts,
12. play vars,
13. play vars_prompt,
14. play vars_files,
15. role vars (defined in role/vars/main.yml),
16. block vars (only for tasks in block),
17. task vars (only for the task),
18. include_vars,
19. set_facts / registered vars,
20. role (and include_role) params,

21. include params,

22. extra vars (for example, -e "user=my_user") (always win precedence).

Variables can be templated into task's arguments, like in this example from Ansible's documentation:

```
tasks:
  - name: Register a file content as a variable
    ansible.builtin.shell: cat /some/path/to/multidoc-file.yaml
    register: result

  - name: Print the transformed variable
    ansible.builtin.debug:
      msg: '{{ item }}'
    loop: '{{ result.stdout }}'
```

Here, the output of shell command was captured as variable named `result`. Then the next task, debug, loops over a list of items in the `result.stdout`. In each iteration the current item is available as a variable named `item`. Debug's argument `msg` is templated in each iteration, and the variable `item`'s value is pasted as the argument's value.

The template engine used by Ansible is Jinja2 [8] and it is much more powerful than shown in the simple example above.

Facts are a special type of variables. When Ansible first connects to a remote node it collects information about it — including operating system, IP addresses, attached filesystems, and more. This data is referred to as facts and it is available as a special variable. Facts are associated with hosts. In particular, it allows you to use information about one host as configuration for other hosts. Ansible also has a `set_facts` action, which allows you to set custom per-host facts.

2.1.5. Strategies

Strategies change how Ansible coordinates task execution. By default Ansible uses the linear strategy, which requires a given task to finish executing on all nodes before the next task can start executing. Ansible will run the task on several nodes in parallel (this number is called the number of **forks**), which is set to 5 by default. Alternative strategies are available and can be defined by plugins. One of the other strategies included by default is the **free** strategy, which disables waiting for other hosts before continuing to the next task and allows for a faster execution in some scenarios.

2.2. Mitogen for Ansible

Mitogen for Ansible is an alternative, redesigned connection layer and module runtime for Ansible. It features a series of improvements over Ansible that result in 1.25x to 7x speedup in playbook execution time. The main changes are described in the following subsections.

2.2.1. SSH

One SSH session is used per target host, with an additional sudo invocation for each user required. This has an advantage over SSH multiplexing of multiple sessions in that state can be maintained in RAM between individual steps.

2.2.2. Reduction of roundtrips

Thanks to the state being maintained within the SSH session, if a module already exists in the RAM of the target host executing a step only requires a single roundtrip between the control node and the target node. Eliminating the creation of a multiplexed SSH channel avoids a significant overhead.

Many popular modules use SFTP or SCP to transfer files. Mitogen replacements use the same message bus for RPC and file transfer which both lowers the number of roundtrips and decreases load on the filesystem.

Mitogen also avoids the overhead of RPCs for creating the main temporary directory for a task (as creating a temporary directory during task setup is a step required for compatibility with Ansible). Temporary directories created by Mitogen are owned privately by the target user, so operations for modifying access permissions for other users are avoided.

2.2.3. Reuse of processes

Processes are reused between playbook steps instead of being created over and over again which eliminates the overhead of Python interpreter startup and processing of imports. According to Mitogen documentation, this improvement alone saves between 300 and 800 ms for every playbook step. However, this violates the assumption that each module execution starts in a new, unmodified environment, which may not be true if a previous module for example monkey-patched a part of Ansible. Mitogen provides a list of such problematic modules and task isolation can be set on a task-by-task basis to force starting a new interpreter for a task.

2.2.4. Caching of code in RAM

Code of modules is cached in the operating memory of the target node instead of being transferred for each playbook step which allows for a significant reduction in bandwidth usage.

2.2.5. Filesystem

The number of writes to the filesystem is reduced, as Mitogen greatly reduces the number of temporary files and directories in use. In particular, Mitogen can run modules from RAM instead of saving them to temporary files first, and can transfer files directly to target users without creating intermediary files under other user ownership.

Chapter 3

Architecture of Gosible

3.1. Gosible

3.1.1. Basic architecture

Unlike Ansible which uses Python, Gosible is written in Go. This alone requires a significant number of architecture changes not driven by performance factors alone. Go is a compiled language, which makes sending scripts infeasible. In our architecture, similarly to Mitogen for Ansible, we start a process on the target node which we then communicate with. The following key points characterize our approach:

- The process run on the target node is called a *remote runner*.
- The remote runner starts a gRPC server, which is exposed to the control node using stdin/stdout (which are relayed over SSH). This slightly differs from standard use of gRPC, in which the gRPC server is exposed as a TCP service. Since we wanted to avoid requiring additional access to network resources, which may require additional permissions in some environments, as well as introduce potential security issues, we opted to make this change.
- The remote runner is statically compiled with all the modules that are supposed to be available to the user.

We want to minimize the number of processes running on the target node due to performance concerns as launching a new process has a non-negligible overhead. However it can't be fully avoided — for example Ansible supports running different commands as different SSH users, which requires establishing additional connections. The `become` functionality [9] uses `sudo` in order to launch the script as a different user, which also can not be avoided for obvious reasons (we are not willing to introduce new mandatory suid binaries). Therefore to support these cases, we are willing to accept having multiple processes simultaneously running on a single target node, however, we also aim to reduce the number of these when possible, by avoiding closing connections that can be used later.

3.1.2. RPC library choice

We opted to use gRPC for our project, since it's popular, well-supported, we have experience using it and deemed it good enough for this application. Other RPC libraries could also be considered, however since the majority of the overhead in Ansible is caused mainly by the number of network round trips and data needed to establish a new connection and upload

the script, we perceived the performance of the RPC library itself a secondary factor. While gRPC might not be the fastest library out there, we believe it to be a good choice. The library is commonly used in the Go ecosystem, is stable and well-tested. No unnecessary network round trips are introduced in RPC invocations which is critical as users often run scripts on remote servers with high connection latency. While other libraries that have lower overheads either in serialization or communication do exist, we expect to make a relatively low number of RPC calls per second, with the amount of data transmitted per call being a few KiB, which makes these overheads rather low. Some other libraries also offer a feature called promise pipelining, which allows the usage of a not-yet-received result of a RPC call as a parameter to another call, which in some cases allows a significant time saving, however this project is not able to make use of it in order to improve performance.

3.1.3. Task execution flow

Playbook execution begins on the control node, which parses command-line arguments and checks the host list. After the playbook is initially parsed, Gosible starts collecting host information (Ansible facts) and proceeds to execute the tasks defined in the playbook.

Since tasks can contain templates, they can't be fully parsed when the playbook is loaded. When Gosible begins executing a task with a loop, it does so by first unrolling the loop. Then the task description is run through the template engine, which replaces template variables and expressions with the evaluated values. After parsing the task, Gosible gathers the list of hosts to execute the task on and either reuse or establishes a new connection to them if it's necessary (for example when using `become` to execute the commands as root). When establishing a new connection, Gosible connects to the node, uploads the platform-specific remote runner binary to the remote node, and executes it on a new SSH channel, which allows establishing a gRPC connection with standard IO as the transport. When executing a task on a remote node, a remote procedure call is used to request the execution of the task, passing information such as the target task name and arguments. After requesting the execution, based on the execution strategy Gosible might proceed to execute the task on another host or execute another task. When the task finishes executing, a response will be sent to the control node which includes information such as the results of the executions, errors if any occurred, and list of updated Ansible facts. This information is received and logged by Gosible and is used to inform the user about the execution status and update variables that can be used by later parts of the script. If the task was executed on all nodes and the executing strategy required it, the next task is executed.

3.2. Gosible functionalities

Ansible is a huge and complex project consisting of 144 thousand lines of Python code. This fact forced us to implement only a subset of its functionalities, which we considered the most important. The parts that we implemented are:

- Usage as a command line application.
- Inventory parsing.
- Playbook parsing.
- Support for templates in playbooks.
- Variables and precedence rules.

- Establishing a connection between control node and managed nodes.
- Implementation of a few core modules.
- Implementation of common action, lookup and become plugins.
- Execution of modules on managed nodes.
- Support for linear and free playbook execution strategies.

The listed functionalities are described in more detail in the following subsections.

3.2.1. Usage as a command line application

Gosible supports play command, analogous to ansible-playbook command, and the following options

```
$ gosible play --help
Execute a playbook

Usage:
  gosible play [flags]

Examples:
gosible play \
    -i inventory.txt \
    -e ansible_ssh_private_key_file=key.pem \
    -e custom_config=42 \
    playbook.yml

Flags:
  -K, --ask-become-pass                          ask for privilege escalation
                                                 password
  -k, --ask-pass                                 ask for connection password
  -b, --become                                   run operations with become (does
                                                 not imply password prompting)
      --become-method ansible-doc -t become -l   privilege escalation method to use
                                                 (default=sudo), use ansible-doc
                                                 -t become -l to list valid choices.
                                                 (default "sudo")
      --become-user string                       run operations as this user
                                                 (default=root)
  -e, --extra-vars strings                       set additional variables as
                                                 key=value or YAML/JSON, if filename
                                                 prepend with @
  -h, --help                                     help for play
  -i, --inventory string                         specify inventory host path or
                                                 comma separated host list

Global Flags:
  -c, --config string   Configuration file
  -f, --format string   Format type of output (default "json")
```

Code responsible for commands and arguments handling is located in the `/command` directory.

3.2.2. Inventory parsing

Gosible supports inventories as files or comma separated host list argument. Inventory files can have `YAML` [23] or `INI` [17] format. Gosible's inventory supports hosts, groups and variables. Code responsible for parsing inventories is located in `/inventory` directory.

3.2.3. Playbook parsing

Gosible parses playbooks [5], which are `YAML` [23] files. Gosible supports specifying plays, strategy for a play, hosts for a play, tasks for a play, naming tasks and arguments for a task, variables for plays and tasks, task loops, and module arguments. Playbook parsing is located in `/playbook` directory.

3.2.4. Support for templates in playbooks

Gosible supports templating in playbooks, which enables access to variables and facts in play and task arguments, names, and loops. Gosible supports loops to execute a task multiple times. Code responsible for templates is located in `/template` directory.

3.2.5. Variables and precedence rules

Gosible supports variables [6] and conforms to Ansible's variable sources precedence [7]. Code responsible for variables is located in `/vars` directory.

3.2.6. Establishing a connection between control node and managed nodes

Gosible supports establishing a connection between control node and managed nodes. Code responsible for establishing a connection is located in `/connection` directory.

3.2.7. Implementation of common plugins

Gosible supports following plugins:

- action plugins: `debug`, `set_fact`, `wait_for_connection`. Located in `/plugins/action` directory.
- lookup plugins: `env`, `indexed_items`, `items`, `list`, `random_choice`, `url`, `sequence`, `varnames`, `vars`. Located in `/plugins/lookup` directory.
- become plugins: `su`, `sudo`. Located in `/plugins/become` directory.
- `meta` plugin. Located in `/plugins/meta` directory.

3.2.8. Implementation of common modules

Gosible supports following modules: `get_url, group, shell, command, hostname, ping, pip, service, wait_for`. Code responsible for modules is located in `/modules` directory.

3.2.9. Execution of modules on managed nodes

Gosible supports the execution of modules on managed nodes. Code responsible for modules execution is located in `/executor` directory.

3.2.10. Support for linear and free playbook execution strategies

Gosible supports linear and free playbook execution strategies. Code responsible for execution strategies is located in the executor in `/executor/playbook_executor.go`.

3.3. Implementation correctness

It is crucial to ensure that Ansible and Gosible will modify managed nodes the same way given identical inventory and playbooks. We developed an End-To-End(e2e) test framework to check for the correctness of Gosible implementation. Implementation of the e2e framework heavily relies on containerization using Docker [13].

The developer defines the test case by creating directory with playbook and inventory files. During testing, the e2e framework will create the following containers: Gosible control node, Ansible control node, and two managed nodes, one for each control node. Each control node will share a network [12] with one of the managed nodes. Control nodes will then execute the given playbook with a provided inventory.

After successful execution of playbooks on managed nodes, both modified containers will be saved as a new image, and their file systems will be compared using the container-diff [11] tool. Not every difference in files should be treated as an error. For example, `/var/log/lastlog` file will never be the same in the two filesystems, and the difference does not indicate a failure, so we ignore it. If all not-ignored files are the same, the test case successfully passed.

3.3.1. Example test

Below is an example of a simple shell module test. It creates a file with some text using shell commands. When the execution finishes on both Gosible and Ansible managed nodes, the generated filesystems are compared, in particular, the existence and content of newly created file. If the Gosible's shell module is working correctly, no difference will be found.

Playbook file:

```
# shell_minimal.yml
- hosts: all
  tasks:
    - name: create foo file
      shell: |
        echo "foo text" > /home/sshtest/foo
      args:
        creates: /home/sshtest/foo
```

Inventory file:

```
# inventory.yml
all:
  hosts:
    managed:
      ansible_user: sshtest
      ansible_private_key_file: /root/.ssh/id_rsa
      ansible_host : managed
```

Of course, the fact that the shell module was correctly executed means that other, less visible parts of Gosible work too, including but not limited to: inventory and playbook file parsing, connection establishing and Gosible's binary sending, executing modules via SSH.

The rest of the test cases and source code of the e2e framework can be found in the `/e2e` directory in the Gosible's repository. All currently written tests pass correctly.

3.4. Operation latency

Since one of the most important parts of Gosible is improving Ansible's performance, we needed to optimize the amount of round trips at most important parts of the execution. Plain Ansible takes several round trips just to execute a single task invocation, however Mitogen's Ansible integration and Gosible both are able to perform a single invocation within a single round trip if the network conditions allow it.

A potentially expensive operation is establishing a new SSH connection to a host. This operation doesn't need to be as optimized as the execution of a single task, as a connection will usually be only established once to a given remote node. However it is still important to understand how Gosible fares in this regard. Understanding the full latency of establishing a connection will require going through the entire network stack: TCP, SSH, and lastly gRPC used by our remote executor binary. We will assume an optimistic case where there is no packet loss at all. In the case that a packet is dropped, the number of round trips will obviously increase due to TCP's workings.

Information about the initial SSH handshake can be found online [21] and we have summed it up in figure 3.1. Since SSH uses TCP as the transport of choice, a TCP connection has to be established first which takes a single round trip. Afterwards both the client and server immediately send their protocol versions and each party after receiving the other party's protocol version then follows up with a `SSH_MSG_KEXINIT` message containing the supported key exchange algorithms. SSH messages are simply specifically encoded data frames that are sent over the TCP connection. We will focus on the Elliptic Curve Diffie-Hellman (ECDH) key agreement algorithm, since we expect this algorithm to be the most relevant within the next few years. The exchange is started by the client sending a `SSH_MSG_KEX_ECDH_INIT` message, after learning that the server supports this algorithm by receiving the server's `SSH_MSG_KEXINIT` message. The server then computes some values, and returns them to the client in a `SSH_MSG_KEX_ECDH_REPLY` message. Aftewards a `SSH_MSG_NEWKEYS` message sent by both parties (immediately sent by the server along with the `SSH_MSG_KEX_ECDH_REPLY` message, and by the client once it receives this message) completes the exchange. The actual cryptographic content of this exchange is not relevant to this work and as such is not explained, however it is explained in the original blog post.

After the SSH handshake is completed, we are now logged into the target remote server and can start to execute commands, upload files, etc. We will need to understand the latency of each of those operations. We have shown the messages exchanged in figure 3.2. The channel creation starts by the client sending a `SSH_MSG_CHANNEL_OPEN` message, which doesn't specify any details about what the channel will be used for, which then has to be confirmed by the server with a `SSH_MSG_CHANNEL_OPEN_CONFIRMATION` message. Then after receiving the confirmation, the client requests a particular SSH feature using a `SSH_MSG_CHANNEL_REQUEST` message and awaits a `SSH_MSG_CHANNEL_SUCCESS` message. The `SSH_MSG_CHANNEL_REQUEST` message specifies the target subsystem. In our cases the relevant ones are `shell` subsystem, which is used for executing programs and shell commands on the remote node, and `SFTP` subsystem which is used to transmit files. Both of these are established after the server

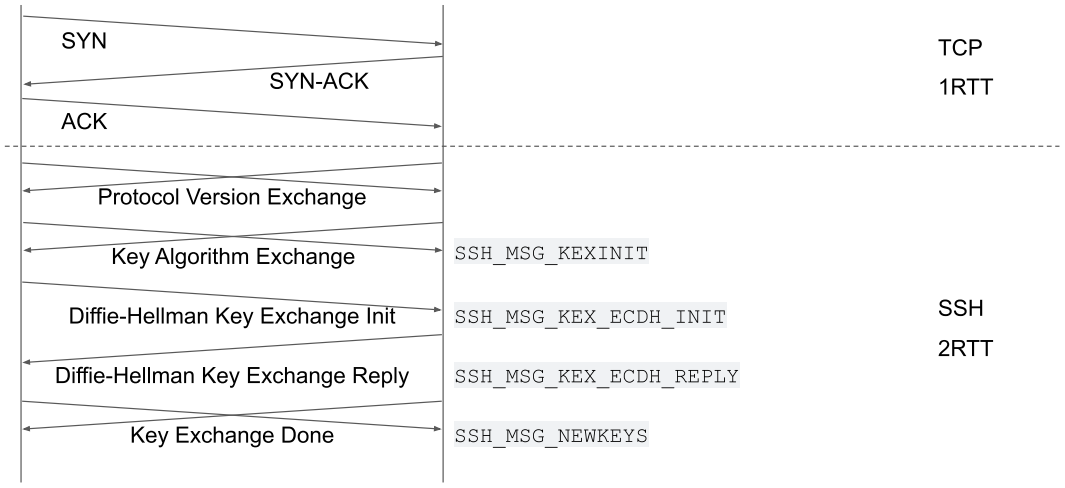

Figure 3.1: SSH connection latency graph. Based on information from a goteleport.com blog post [21].

replies with the `SSH_MSG_CHANNEL_SUCCESS` message, which means 2 network round trips. Later the relevant data is transferred using `SSH_MSG_CHANNEL_DATA` messages (eg. standard output from a command or file data).

When connecting to a remote host Gosible current performs the following operations:

- checks the system architecture and presence of a cached Gosible binary (it is assumed the host has a POSIX compatible shell) — a shell execution — 2 network round trips to establish the channel, 0 to get the reply.
- if the binary is not present, it is uploaded the the host to the directory holding cached versions of Gosible — 2 network round trips to establish the channel, at lease one to send the binary to the server.
- the binary is launched — 2 network round trips to establish the channel.

gRPC uses HTTP/2 protocol behind the scenes, which means that the first message sent when the channel is established will be a H2C `HELLO` message in order to make both sides aware that the HTTP/2 protocol is used. Since the packet is sent immediately by both sides, the SSH client will receive the `SSH_MSG_CHANNEL_DATA` message with this H2C message shortly after `SSH_MSG_CHANNEL_SUCCESS` and as such no additional round trips are required to establish the HTTP/2 connection.

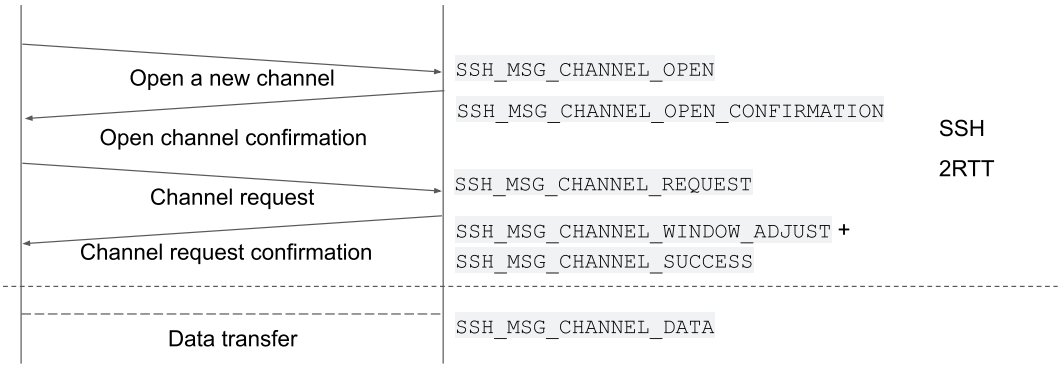

Figure 3.2: SSH channel creation latency graph. Based on the SSH RFC [22].

3.5. Python executor

Gosible, in addition to the built-in Go modules, is also able to execute Python modules if Python is installed on the remote node. This allows easy porting of Ansible modules with minimal changes. A portion of the Ansible source code tree can be downloaded and extracted (module_utils and modules) in order to enable this feature (a small code patch is also applied). When building the remote executor binary, a zip file with the modified Ansible sources and additional Gosible-specific interoperability code is created. This file is called the Python executor runtime.

Since Go-based modules allow for lower overhead and cleaner integration, we have decided to make the Python executor part on-demand. This means that when execution of a Python-based module is requested, the remote node checks if it has the Python runtime file cached or not, and if it doesn't it requests the runtime to be uploaded from the control node. While this approach does introduce an additional round trip, it is important to note that we expect most playbooks to use Go-only modules and as such not uploading the Python runtime allows us to save a little bit more on the executor binary size. The performance penalty is that it however that it introduces an additional round trip, and requires a ∼0.5MiB file to be uploaded in order to be used.

Similar to Mitogen, the Python executor reuses a single Python process for all tasks to be executed, which the Go executor then communicates with. In addition, relaunching the Python interpreter from modules is not supported. These limitations might cause issues with some modules, however, this allows for a significant performance benefit.

Chapter 4

Benchmarks comparison

In ScyllaDB, it is common to run the same playbook on a significant number of nodes, so we want good performance in such scenarios. To assess the effect of Gosible's improvements on the performance of playbook execution we created a benchmarking framework and compared the execution of the same playbooks among Gosible, Ansible, and Ansible with Mitogen. In this chapter, we describe how it worked and what results were achieved.

4.1. Measurement methods and benchmarks environment

We had the following systems available to run the benchmarks on:

- the `students` system, hosted in Warsaw, Poland, running Debian 11.
- 32 virtual machines hosted on AWS in the `us-east-1` region (ping ∼109ms, bandwidth ∼37Mbit/s from the `students` system), running Ubuntu 22.04.
- 8 virtual machines hosted on AWS in the `eu-west-2` region (ping ∼25ms, bandwidth ∼91Mbit/s from the `students` system), running Ubuntu 22.04.

The `students` system is shared and has 64 CPU cores and 328GiB of RAM available. The AWS virtual machines used the `t3a.small` instance type with 2 vCPU and 2GiB of RAM.

We have developed a simple Python utility that launched tests in four configurations:

- **Gosible (cached)** — in this configuration, the Gosible executor binary is cached before each run.
- **Gosible (clean)** — in this configuration, the cached Gosible executor binary is deleted before each run, so that the executor binary needs to be resent during the run.
- **Ansible** — in this configuration, the Ansible's linear execution strategy is used.
- **Ansible with Mitogen** — in this configuration, Ansible is configured to use the Mitogen's linear execution strategy.

We have decided to run the tests on some of the following three node configurations:

- `us-east-1` VMs only, with the `students` system as the control node, the possible number of nodes being 1, 4, 8, 16 and 32.

- `us-east-1` VMs only, with one of the VMs as the control node, same number of nodes as above. In addition, one of the VMs is also used as a executor node in the case of the 32 node test.
- `us-east-1` and `eu-west-2` VMs, with the `students` system as the control node, the possible number of nodes being 2, 4, 8 and 16 (half in one region and half in another).

We have opted for these node configurations for the following reasons:

- The first one represents a typical scenario where a company might launch the Ansible scripts from the office, or from a third-party CI service that is hosted by a different provider, in a different world region.
- The second one represents a typical scenario where a company will launch the tests from the cloud in the same cloud provider and in the same region.
- The third one represents a typical scenario where the cloud regions might be mixed.

We have built a tool that allowed us to automatically execute the benchmarks on the servers. Each of the tests was run three times for each of the configurations, except for the non-Mitogen Ansible configurations which were only run twice due to the time constraints. The times were then averaged.

In addition, due to the fact that some of the tests mutate the system state in a way that's not trivially reversible, for some of the tests we have set up a single-use Docker container before each run. The base image for the Docker environment was also Ubuntu 22.04. In addition, for some of the tests, we have run a validation pass that the playbook was indeed correctly executed in addition to the output from the tools themselves.

4.2. Results

The following graphs show the dependence of the playbook execution time (in seconds) on the number of managed nodes. All playbooks executed during benchmarks can be found in appendix A.

4.2.1. Task execution request benchmark

This benchmark is designed to show the impact of different connection methods used in Ansible, Ansible with Mitogen, and Gosible, which are explained in chapter 2. The playbook executes many tasks that are fast to perform, in this case, small file creation. Because this operation is very fast, most of the time measured comes from requesting task execution over the network.

As figures 4.1, 4.2 and 4.3 show, the Gosible's overhead of a task execution request is very low compared to Ansible.

4.2.2. Real-world example

This benchmark is designed to show the difference in the execution of a small, but realistic playbook. The `realworld_1.yml` playbook downloads Go binary ($\sim$ 135 MB), extracts with tar, installs by linking in `/usr/bin` directory and saves the Go version to a temp file. Downloading files or packages and running some shell scripts are typical actions performed in real-world Ansible playbooks.

Figure 4.5 shows graph of `realworld_1.yml` playbook execution on control node in Poland and managed nodes in the USA, AWS region us-east-1 (average ping 109ms). Figure 4.7 shows results of execution on control node and managed nodes in the same AWS region us-east-1 (ping below 1ms). Figure 4.6 shows result of execution on control node in Poland and managed nodes in the two AWS regions - USA and Frankfurt.

4.2.3. Summary

The charts illustrate how the performance of Ansible and Gosible changes depending on the number of managed nodes. Comparing the benchmarks results between different regions, it can be seen that the network conditions (such as ping and bandwidth) have a significant impact on execution times. The obvious observation is that Gosible is faster than Ansible in every case, and the difference in times increases as the number of managed nodes increases. The benchmark results show that Gosible is more suitable for managing large numbers of nodes.

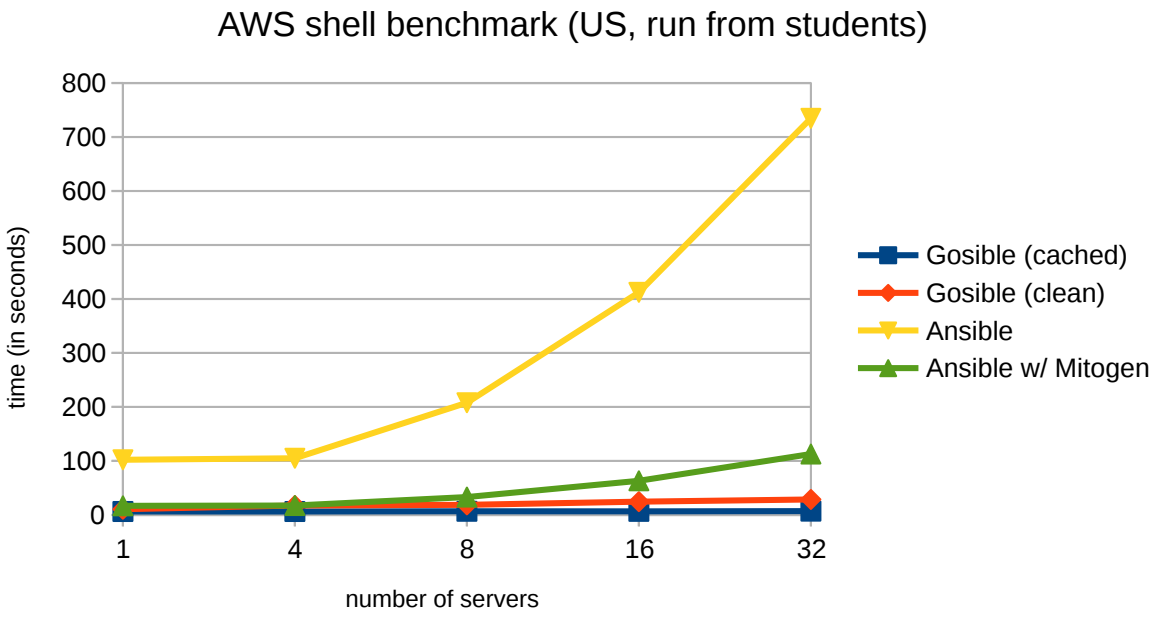

Figure 4.1: Benchmark of `shell_test_1.yml` playbook

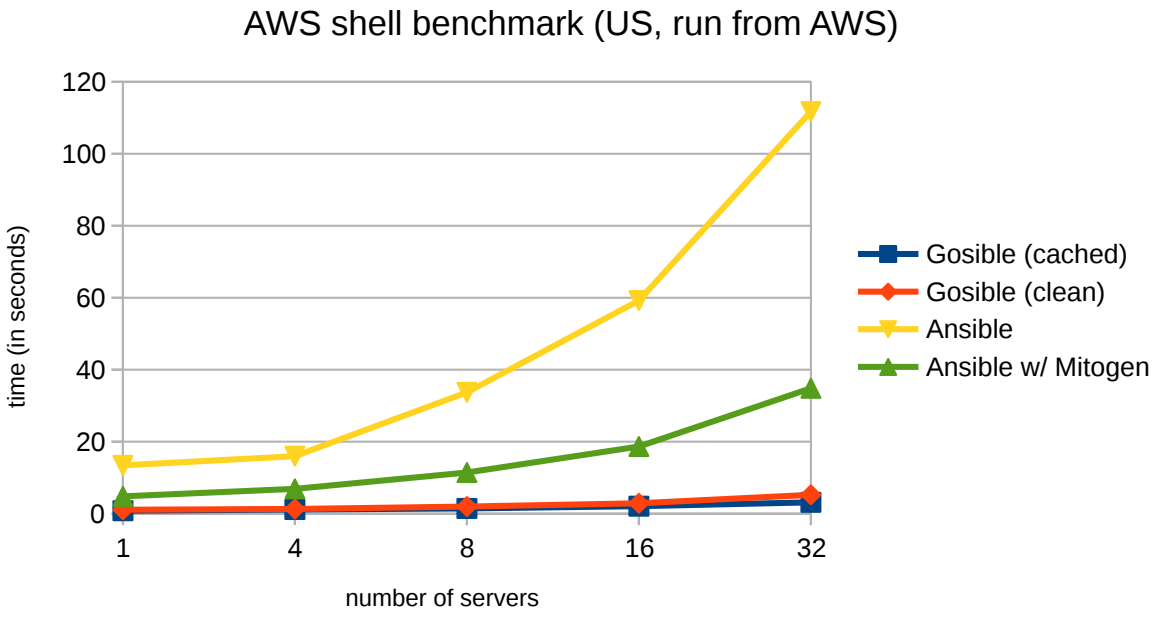

Figure 4.2: Benchmark of `shell_test_1.yml` playbook

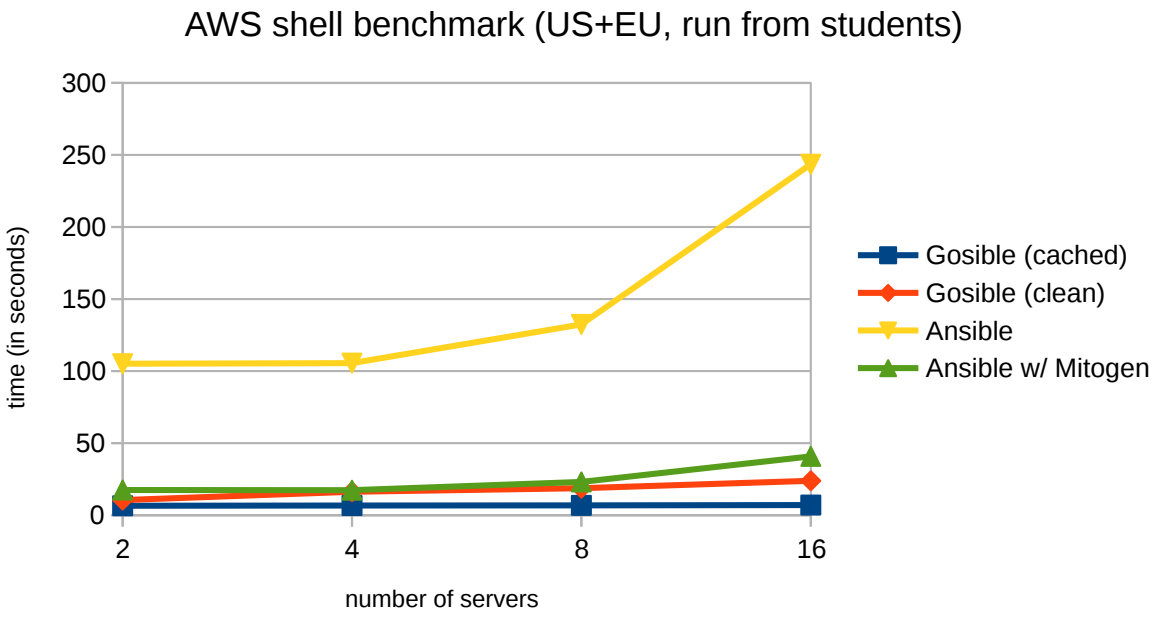

Figure 4.3: Benchmark of `shell_test_1.yml` playbook

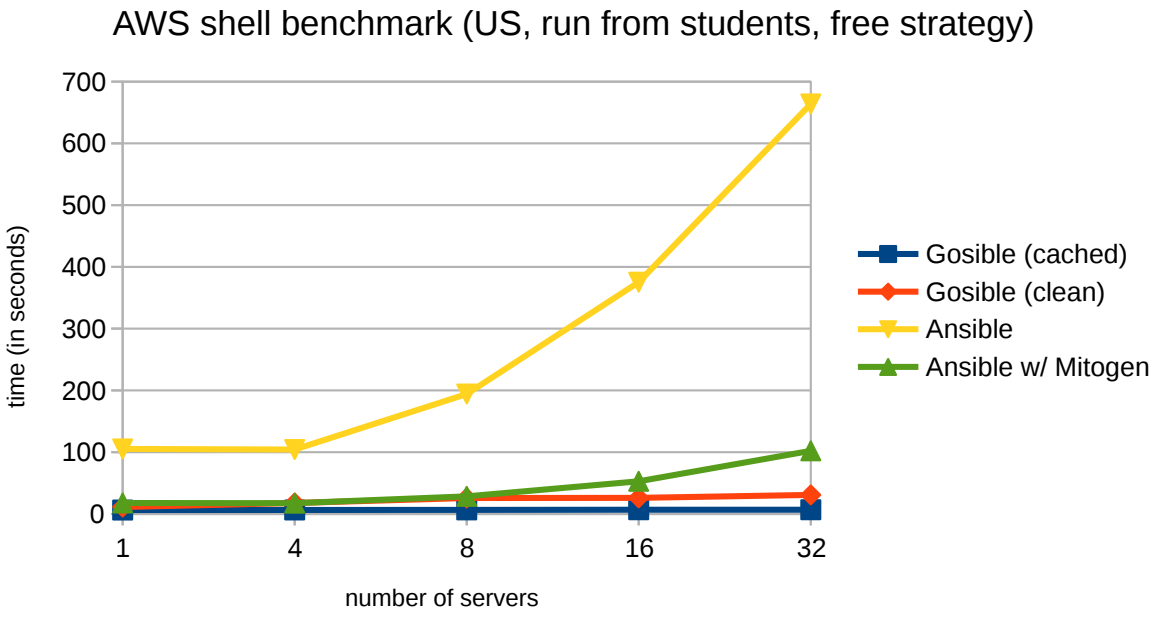

Figure 4.4: Benchmark of `shell_test_free.yml` playbook

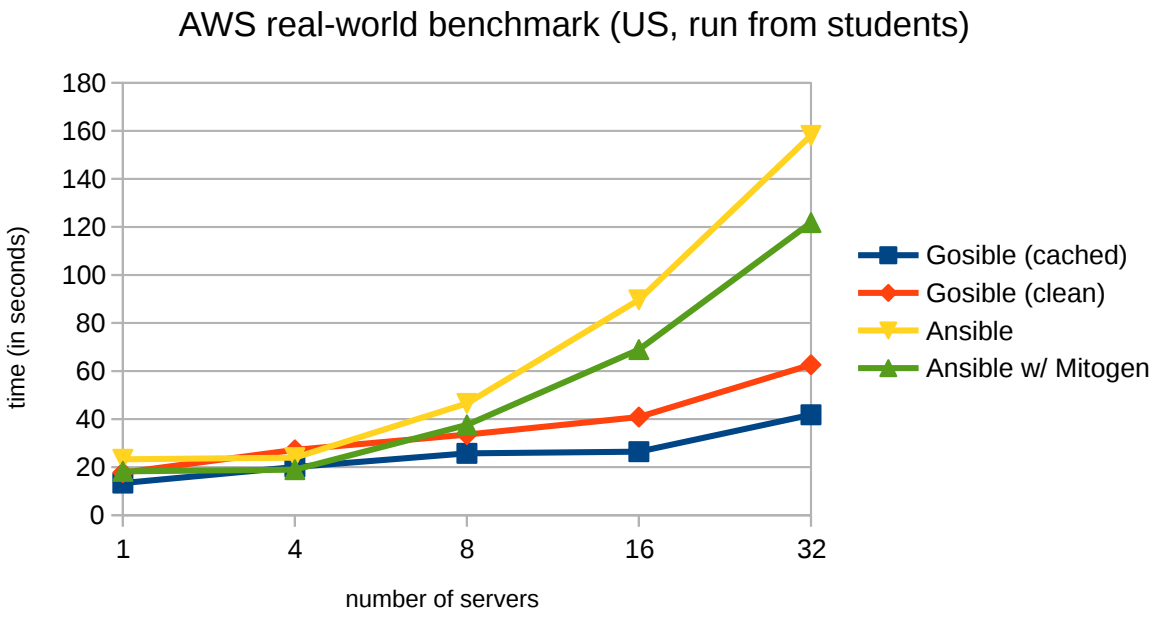

Figure 4.5: Benchmark of `realworld_1.yml` playbook

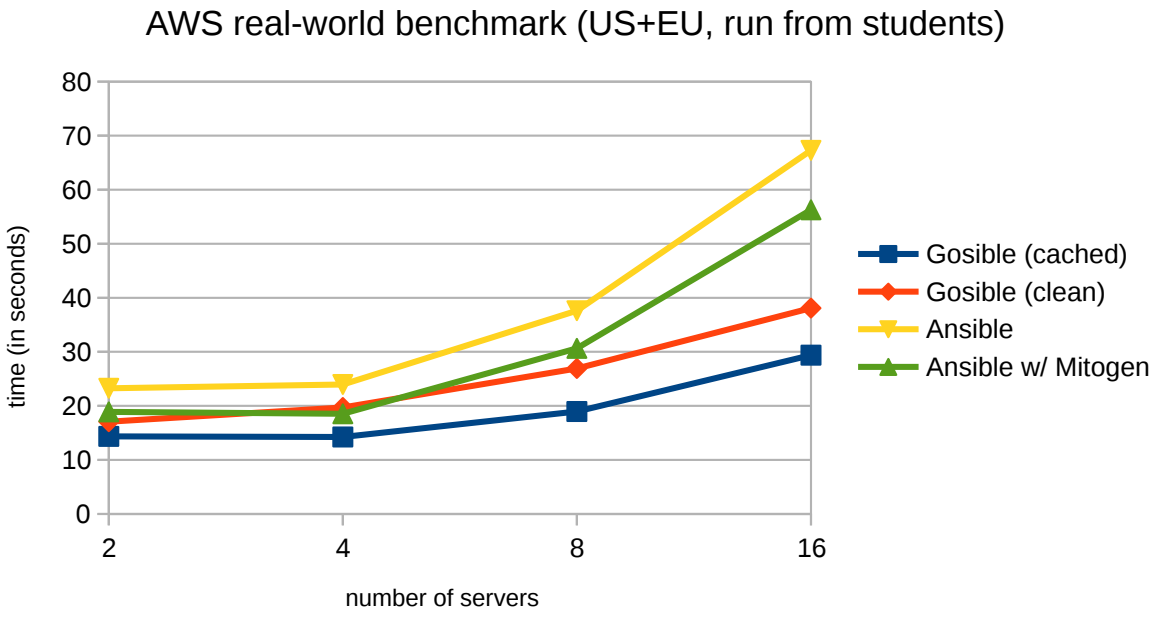

Figure 4.6: Benchmark of `realworld_1.yml` playbook

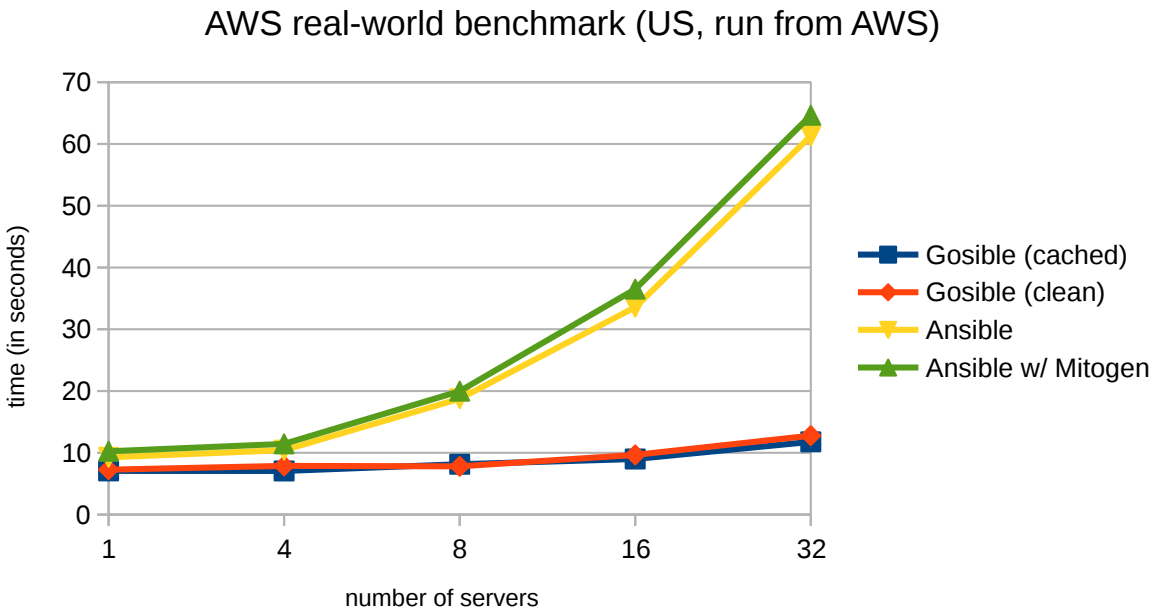

Figure 4.7: Benchmark of `realworld_1.yml` playbook

Chapter 5

Summary

5.1. Stated goals and results

1. Re-implement subset of functionalities:
    - We selected the most important Ansible functionalities and re-implemented them in Gosible.
    - With selection of the most important Ansible features, Gosible can be used to configure hosts with real-world playbooks.
    - Details of selected features can be found in the Gosible functionalities section.
2. Improve performance:
    - Thanks to modified architecture, our implementation significantly reduces overhead of communication between nodes, which in our benchmarks resulted in improved playbook execution times.
    - Details of time comparison can be found in the benchmark results section.
3. Test correctness of the solution:
    - We developed end-to-end testing framework to ensure correctness of our implementation.
    - Modules, action plugins, playbook and inventory parsing and other implemented features are all tested.
    - Details of the end-to-end framework can be found in the implementation correctness section.
4. Improved compatibility with other DevOps tools
    - Gosible can be imported to any Go project as a package, and used by simply calling functions.

5.2. Accomplishments

Our main accomplishment is that we developed a proof of concept of improved architecture, that can be used in the future by ScyllaDB Inc. or other entities to improve their Ansible workflows.

5.3. What could be done next

Allow third-party modules

Ansible makes it possible for its users to develop custom modules in Python. We opted to exclude this functionality at this point of time, however we designed Gosible in a way that would allow integration of third-party modules. The implementation should look as follows:

- the remote runner base would be extracted into a common library,
- a public API would be developed for it that would allow implementing custom modules,
- the user would build their own executable with their own modules for all platforms they want to run it on,
- a way to discover which modules are provided by each module without a significant runtime overhead would have to be implement (either as a separate file for each module or by inspecting the binary).

Replace the template engine

The template engine used in Gosible, Gonja [14], is not fully compatible with Ansible's template engine jinja2 [18]. To enable Gosible's use as a drop-in replacement in more applications, jinja2 should be ported to Go and used as a replacement for Gonja. This change wouldn't require any major changes to Gosible's source code.

5.4. Division of work

Our effort wasn't divided into completely independent parts. Rather than that, we developed the project iteratively, splitting the work into small, manageable, separate tasks. Over the course of 8 months over 150 Merge Requests have been merged. The majority of the code has been worked on by more than one person — either through design, implementation, code review, modification to meet the requirements of a later iteration, bug fixing, or additional testing. We believe that each of the authors contributed a fair share of the combined effort.

Below we list for each or the authors the parts of the project where they had a significant contribution:

Hubert Badocha

- inventory parser,
- utilities for modules implementation,
- modules implementation,
- the executor,
- an adapter for the template engine,
- connection management,
- lookup plugins,
- become support.

Tomasz Domagała

- E2E testing framework,
- CI integrations,
- variables manager,
- benchmarks.

Paweł Pawłowski

- the executor,
- connection management,
- gRPC,
- Python modules executor,
- facts management,
- benchmarks.

Jakub Moliński

- playbook parser,
- configuration management,
- logging support,
- the executor,
- fixing bugs in template engine dependency,
- variables and templating support,
- lookup plugins,
- loops and conditionals support.

Appendix A

Playbooks used in benchmarks

A.1. Shell test playbook

This playbook creates 32 small files.

```
# shell_test_1.yml
- hosts: all
  gather_facts: no
  tasks:
    - name: delete test files
      shell: cd; rm -rf testfiles && mkdir testfiles
    - name: create files
      shell: cd; echo test > testfiles/{{item}}.txt
      with_sequence: start=1 end=32 stride=1
    - name: create file www1
      shell: cd; curl https://example.com/ > testfiles/www1.txt
    - name: create file www2
      shell: cd; curl https://example.com/ > testfiles/www2.txt
    - name: create file www3
      shell: cd; curl https://example.com/ > testfiles/www3.txt
```

A.2. Real-world playbook

This playbook downloads Go binary, extracts, installs, and performs a small task with it.

```
# realworld_1.yml
- hosts: all
  gather_facts: no
  tasks:
    - name: Download go binary
      get_url:
        url: https://go.dev/dl/go1.18.2.linux-amd64.tar.gz
        dest: /tmp/go.tar.gz
        mode: '0755'
    - name: Remove previous go binary
      shell: rm -rf /usr/local/go
    - name: Extract go binary
      become: yes
      become_user: root
      become_method: sudo
      shell: |
        tar -C /usr/local -xzf /tmp/go.tar.gz
    - name: Install go binary
      become: yes
      become_user: root
      become_method: sudo
      shell: |
        ln -s /usr/local/go/bin/go /usr/bin/go
    - name: Save go version in temp file
      shell: go version > /tmp/go_version.txt
```

Appendix B

Result data

The following data shows playbook execution times (in seconds) depending on the number of managed nodes (from 1 up to 32) and the control node's Ansible implementation (Ansible, Ansible with Mitogen, Gosible). A further explanation of the data can be found in chapter 4.

B.1. AWS real-world benchmark (US, run from students)

	1	4	8	16	32
Gosible (cached)	13.36	20.04	25.73	26.41	41.77
Gosible (clean)	17.69	27.16	33.54	40.83	62.58
Ansible	23.24	23.93	46.51	89.68	158.1
Ansible w/ Mitogen	18.23	18.92	37.57	68.91	121.8

B.2. AWS real-world benchmark (US, run from AWS))

	1	4	8	16	32
Gosible (cached)	7.07	7.06	8.14	8.99	11.77
Gosible (clean)	7.26	7.88	7.80	9.65	12.77
Ansible	9.26	10.42	18.75	33.57	61.30
Ansible w/ Mitogen	10.24	11.43	19.97	36.49	64.64

B.3. AWS real-world benchmark (US+EU, run from students)

	1	4	8	16
Gosible (cached)	14.34	14.23	18.95	29.377
Gosible (clean)	17.05	19.68	26.91	38.097
Ansible	23.24	23.94	37.60	67.277
Ansible w/ Mitogen	18.88	18.53	30.63	56.287

B.4. AWS shell benchmark (US, run from students)

	1	4	8	16	32
Gosible (cached)	6.41	6.42	6.77	6.68	7.01
Gosible (clean)	10.63	17.42	18.69	24.60	28.60
Ansible	102.14	104.96	207.70	412.37	734.48
Ansible w/ Mitogen	16.89	17.47	33.04	63.22	112.80

B.5. AWS shell benchmark (US, run from AWS)

	1	4	8	16	32
Gosible (cached)	0.83	1.16	1.44	2.08	3.16
Gosible (clean)	1.10	1.30	1.99	2.89	5.23
Ansible	13.43	15.98	33.69	59.22	111.71
Ansible w/ Mitogen	4.82	6.90	11.43	18.65	34.82

B.6. AWS shell benchmark (US+EU, run from students)

	1	4	8	16
Gosible (cached)	6.55	6.71	6.81	6.99
Gosible (clean)	10.48	16.24	18.68	23.87
Ansible	105.07	105.51	132.38	243.54
Ansible w/ Mitogen	17.43	17.35	23.07	40.80

B.7. AWS get_url benchmark (US, run from students)

	1	4	8	16	32
Gosible (clean)	8.12	10.94	17.31	19.07	26.22
Gosible (Python, cached)	2.18	2.37	2.51	2.67	4.81
Gosible (Python, clean)	6.34	8.70	15.71	18.96	30.58
Ansible	5.32	5.86	9.95	19.52	33.26
Ansible w/ Mitogen	11.53	11.74	22.14	42.63	75.93

B.8. AWS shell benchmark (US, run from students, free strategy)

	1	4	8	16	32
Gosible (cached)	0.83	1.16	1.44	2.08	3.16
Gosible (clean)	1.10	1.30	1.99	2.89	5.23
Ansible	13.43	15.98	33.69	59.22	111.71
Ansible w/ Mitogen	4.82	6.90	11.43	18.65	34.82

Bibliography

[1] *A toy implementation of Ansible in Go*. URL: https://github.com/pdbogen/gosible.

[2] *Ansible*. URL: https://docs.ansible.com/.

[3] *Ansible architecture - Ansible Documentation*. URL: https://docs.ansible.com/ansible/latest/dev_guide/overview_architecture.html.

[4] *Ansible inventory*. URL: https://docs.ansible.com/ansible/2.9/user_guide/intro_inventory.html#inventory-basics-formats-hosts-and-groups.

[5] *Ansible playbook overview*. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html.

[6] *Ansible variables*. URL: https://docs.ansible.com/ansible/2.9/user_guide/playbooks_variables.html.

[7] *Ansible variables precedence*. URL: https://docs.ansible.com/ansible/2.9/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable.

[8] *Ansible variables precedence table*. URL: https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#understanding-variable-precedence.

[9] *Ansible: Understanding privilege escalation: become*. URL: https://docs.ansible.com/ansible/2.9/user_guide/become.html.

[10] *Ansible: Understanding privilege escalation: become Limitations*. URL: https://docs.ansible.com/ansible/2.9/user_guide/become.html#risks-and-limitations-of-become.

[11] *Container-diff tool*. URL: https://github.com/GoogleContainerTools/container-diff.

[12] *Docker networking*. URL: https://docs.docker.com/network/.

[13] *Docker overview*. URL: https://docs.docker.com/get-started/overview/.

[14] *Gonja template engine*. URL: https://github.com/noirbizarre/gonja.

[15] *gRPC documentation*. URL: https://grpc.io/docs/.

[16] *gRPC overview*. URL: https://grpc.io/about/.

[17] *INI file format*. URL: https://en.wikipedia.org/wiki/INI_file.

[18] *Jinja2*. URL: https://palletsprojects.com/p/jinja/.

[19] *Mitogen overview*. URL: https://mitogen.networkgenomics.com/ansible_detailed.html.

[20] *Protocol Buffers overview*. URL: https://developers.google.com/protocol-buffers.

[21] *SSH Handshake Explained*. URL: https://goteleport.com/blog/ssh-handshake-explained/.

[22] *The Secure Shell (SSH) Connection Protocol RFC*. URL: https://datatracker.ietf.org/doc/html/rfc4254.

[23] *YAML file format*. URL: https://en.wikipedia.org/wiki/YAML.

All sites were visited on 5 June 2022.