Report on Using Serf with Containerlab: Configuration and Topology Setup

Introduction

Serf is a decentralized solution designed for cluster management, failure detection, and service orchestration. Developed by HashiCorp, it uses a gossip protocol to enable lightweight discovery, monitoring, and coordination between nodes in a network without a single point of failure. Serf operates independently of a centralized server, allowing each node to communicate directly with other nodes and share state information. This functionality makes Serf ideal for large distributed systems.

In this project, we have created a network of 100 nodes using **Containerlab**, a container-based lab environment that allows for the emulation of complex network topologies. Each node runs the Serf agent, allowing the network to self-organize and maintain communication between nodes. The goal of this report is to explain how the network has been built, how Serf has been configured, and the steps to manage Serf nodes in a large topology.

Serf Overview

Serf relies on a **gossip protocol** that allows nodes to discover and communicate with each other by exchanging small messages. Each Serf agent maintains a list of all other agents in the network, and when a new node joins the network, it gossips to announce its presence to the rest of the cluster. The protocol ensures that even if a node fails or becomes unreachable, the other nodes are made aware of this change through periodic gossip updates.

Serf uses a few critical components:

- **Gossip protocol**: Each node exchanges information with a few others, which then relay that information through the network, ensuring all nodes eventually receive updates.
- **Failure detection**: Nodes detect when another node has failed or become unreachable and notify the network.
- **Event system**: Serf agents can trigger events that propagate throughout the network for orchestration or other administrative tasks.

The topology created for this project consists of 100 nodes that participate in a Serf cluster, dynamically sharing information about their state. Each node is configured to gossip with others, ensuring consistent state information across the network.

Topology Setup

The network is set up using **Containerlab** and consists of 100 nodes defined in the file 100nodes.yml. This YAML file provides the configuration for all nodes in the topology and specifies how they are connected.

100nodes.yml is the heart of this setup and includes definitions for all nodes, their interfaces, and the connectivity between them. Here's a breakdown of the file:

- **Node definitions**: Each node is represented as a container running a Linux-based image. The node definitions include the container names and configurations for interfaces.
- **Connections**: The nodes are connected to each other via virtual links. This allows the nodes to communicate as if they were connected in a physical network.
- **Network switch**: In large networks like this, a virtual switch is used to manage the connections between nodes. This switch ensures that each node can send and receive traffic to/from the rest of the network.

The actual configuration of the nodes and their links can be seen in the file 100nodes.yml. Once the file is in place, the topology is created and started by running:

containerlab deploy -t 100nodes.yml

This command brings up all 100 nodes and sets up the networking between them. Each node is assigned an IP address and connected to the network switch defined in the YAML configuration.

Serf Configuration

Each of the 100 nodes runs a **Serf agent**, which must be configured to join the network and participate in gossiping. The Serf agent's configuration is handled via command-line arguments when the agent is started. The provided script serf_agents_start.sh is used to start the Serf agents on all nodes in the topology.

The following parameters are important for configuring the Serf agent:

- **Advertise address**: Each Serf agent must be assigned an IP address on the network. The IP address allows other agents to communicate with it. This can be configured as part of the startup script using the -advertise option.
- **Join**: When a new agent starts, it must join an existing cluster. The -join option tells the agent which node it should initially contact to join the network. Once connected to the first node, the gossip protocol will propagate the new node's presence to the rest of the cluster.
- **Reconnect attempts**: The agent can be configured to retry joining the cluster if it fails initially.

In this setup, the script **serf_agents_joining.sh** is responsible for making sure each agent joins the correct cluster by identifying an existing node to connect with.

Detailed Setup Process

1. **Deploying the Topology**

The setup begins by deploying the topology described in 100nodes.yml. This YAML file contains definitions for 100 nodes, each acting as a networked container. To deploy the topology, we use the following command:

containerlab deploy -t 100nodes.yml

This command initializes all nodes and creates links between them according to the specified network configuration. Each node container is named in sequence (e.g., clab-century-serf1, clab-century-serf2, etc.), making it easy to reference each node during subsequent configuration steps.

2. **Assigning IP Addresses Using ipaddressing.sh**

After deploying the topology, each node must be assigned an IP address. This is handled by the script ipaddressing.sh, which assigns IP addresses to the eth1 interface of each container.

The script works as follows:

- It iterates through each of the 100 containers, from clab-century-serf1 to clab-century-serf100.
- For each container, it brings up the eth1 interface and assigns it an IP address from the range 10.0.1.11 to 10.0.1.110 (each node's IP address is incremented by 1).
- The default route for each container is also updated to use 10.0.1.1 as the gateway.

Example command from the script:

sudo docker exec -d clab-century-serf$i ip addr add 10.0.1.$((10+i))/24 brd 10.0.1.255 dev eth1

This results in the following IP address allocation:

- clab-century-serf1 receives IP 10.0.1.11
- clab-century-serf2 receives IP 10.0.1.12
- …
- clab-century-serf100 receives IP 10.0.1.110

3. **Setting Up Each Node Using setup_nodes.sh**

Once IP addresses are assigned, the script setup_nodes.sh is used to configure each container node for Serf operation. This setup includes copying necessary files to each node and generating a configuration file for the Serf agent.

Here's a summary of the script's actions:

- The script iterates through each container and retrieves the IP address assigned to the eth1 interface.
- For each node, it generates a JSON configuration file (node.json) that specifies the node's name, bind address, and advertised IP address.
- This JSON file, along with the Serf binary, is copied to each container's /opt/serfapp/ directory.
- The Serf binary is made executable on each node.

Example JSON configuration for a node:

```
{
  "node_name": "clab-century-serf1",
  "bind": "0.0.0.0:7946",
  "advertise": "10.0.1.11:7946"
}
```

This configuration file allows the Serf agent to know its identity and network configuration within the cluster.

4. **Starting Serf Agents Using serf_agents_start.sh**

With nodes set up and configurations in place, we use the serf_agents_start.sh script to launch the Serf agent on each node. This script performs the following steps:

- It iterates over each container and ensures that the Serf binary is executable.
- The Serf agent is then started using the configuration file (node.json) stored on each node.

Example command from the script:

```
docker exec -d "$container" /opt/serfapp/serf agent -config-file=/opt/serfapp/node.json
```

This command starts the Serf agent as a background process on each node, using the settings defined in node.json. At this stage, each node is running a Serf agent, but they are not yet fully connected in the cluster.

5. **Joining Nodes to the Cluster Using serf_agents_joining.sh**

To finalize the cluster setup, the serf_agents_joining.sh script instructs each node to join the cluster by connecting to an initial node, referred to as the "joining node" (clab-century-serf1). Here's how it works:

- The script first gathers IP addresses from nodes 2 to 100 (adjustable as needed) to use as join targets.
- For each target IP address, the joining node (clab-century-serf1) executes a serf join command to connect to that IP.
- A delay (sleep) is applied between join attempts to ensure stable connection establishment without overloading the network.

Example command from the script:

docker exec "clab-century-serf1" /opt/serfapp/serf join 10.0.1.12

By iterating over each target IP and joining them sequentially, the script builds a fully connected cluster where all nodes can communicate and maintain state via Serf's gossip protocol.

In summary, the setup described in this report enables the emulation and management of a 100-node Serf cluster using Containerlab. After deploying the topology and assigning IPs with ipaddressing.sh, each node is configured through setup_nodes.sh, which provides the necessary files and configurations for Serf. The Serf agents are started with serf_agents_start.sh, and the cluster is finalized by joining each node to the cluster via serf_agents_joining.sh. For managing the cluster, all Serf agents are able to maintain state consistency across all nodes. Nodes can be added or removed dynamically, and the cluster will adjust accordingly. Each node periodically sends heartbeat messages to the rest of the cluster to indicate that it is still alive. To monitor the cluster or trigger actions across the cluster, we can use Serf's event system. The event system allows to trigger custom events that propagate across the network. Additionally, Serf can be used to monitor the health of the cluster by checking which nodes are still reachable. The command **serf members** list all nodes in the cluster and their current status (alive, failed, and suspicion.).

Building and Configuring Serf for Optimized Gossip and Network Coordinate Systems

This document provides an explanation of customizing Serf for a 100-node cluster. Serf, a decentralized cluster management tool, is based on a gossip protocol and the Vivaldi coordinate system for network distance estimation. For this large-scale environment, extensive customization is required to avoid network congestion and to maintain optimal system health. Building and configuring the Serf binary involves several steps, including modifying the memberlist package to control gossip behaviors and fine-tuning the Vivaldi coordinate parameters within Serf.

Due to the complexity of these adjustments, each parameter must be tuned and tested to achieve stability, minimize traffic and ARP storm. This report outlines each stage in detail, highlighting considerations for a balanced, high-performance configuration.

Step 1: Building the Serf Binary

The Serf binary, which is built from Go source files, must be recompiled after making changes to ensure that customizations are embedded in the executable. The entry point for building the binary is the cmd/serf directory.

Ensure that all dependencies are correctly installed, as any missing packages will cause the build to fail. Dependencies can be managed via **go.mod** located in the root directory, which lists required packages. Running **go mod tidy** before building will help resolve missing dependencies.

From the serf root directory, initiate the build:

cd /pathtoserf

go build -o serf ./cmd/serf

This command generates a serf executable in the main directory. During the build, it's essential to monitor for warnings or errors, as these can indicate compatibility issues or overlooked dependencies.

After building, run a basic test to verify that the binary functions correctly. Execute:

./serf agent

If the binary fails to start, revisit the build process to ensure all dependencies were correctly included, and recompile as needed.

Step 2: Customizing memberlist Gossip Parameters for 100 Nodes

The memberlist package is central to Serf's gossip mechanism, handling peer-to-peer communication. As gossip scales poorly without tuning, this customization step requires precise adjustments across several parameters in the config.go file.

Detailed Parameter Configuration

https://github.com/anjumm/memberlist/tree/master

- **GossipInterval:**
  This parameter defines the time interval between gossip messages. Reducing the frequency of gossip helps avoid overwhelming the network.
  GossipInterval = 500 * time.Millisecond // Original value can be around 200ms
  A longer interval reduces traffic but delays state updates across nodes. Testing different values may be necessary to find a stable balance for a 100-node network.

- **GossipNodes:**
  Specifies how many nodes each gossip message is sent to. A lower value means each node communicates with fewer peers, reducing message traffic.
  GossipNodes = 2 // Default could be 3 or higher
  Lower values can delay full-network convergence, especially during high-traffic periods. In larger networks, balancing GossipNodes with PushPullInterval (below) can help distribute updates more efficiently.

- **PushPullInterval:**
  Determines how often full state synchronization occurs. Infrequent synchronization minimizes network strain but may result in delayed consistency.
  PushPullInterval = 120 * time.Second // Increase for fewer full-syncs
  A lower frequency suits stable clusters but may not immediately reflect dynamic changes. Experimenting with intervals helps determine the optimal level for setup.

- **IndirectChecks:**
  Specifies the number of indirect ping checks that nodes use for failure detection. Increasing IndirectChecks can improve failure detection accuracy in larger clusters.
  IndirectChecks = 4 // Increase for better coverage
  Considerations: Higher values improve failure detection but add additional messages. Adjust carefully to balance accuracy with network load.

Each of these adjustments in **config.go** requires testing to ensure that changes achieve the desired stability and performance. Configuring each parameter individually, followed by monitoring the cluster's behavior, is essential for finding an effective balance.

Step 3: Configuring Vivaldi Coordinates for Efficient Network Mapping

The Vivaldi coordinate system allows nodes to estimate network latency by assigning virtual coordinates to each node. These coordinates are used by Serf to optimize communication paths in large clusters. By default, the Vivaldi system may use an 8-dimensional coordinate space; however, reducing this to 2 dimensions simplifies calculations, which can improve stability in large networks.

In the coordinate directory, find the parameter controlling coordinate dimensions and set it to 2 dimensional. Lower dimensions make distance estimation simpler but may impact the accuracy of latency predictions. Parameters like ErrorScale and UpdateInterval govern how coordinates adapt to network changes and react to latency variations. These are key to stabilizing the Vivaldi system in large clusters.

- ErrorScale:
  Determines the responsiveness of coordinate updates to latency discrepancies. A lower value makes adjustments more gradual, improving stability.
  ErrorScale = 0.25 // Lowered for gradual updates
  Gradual changes reduce jitter but may slow down convergence in dynamic networks.

- UpdateInterval:
  Sets how frequently nodes recalculate their positions. Lower frequencies reduce computation but may slow down response to network changes.
  UpdateInterval = 5 * time.Second // Increase if needed for stability
  Adjusting Vivaldi parameters requires an iterative approach, as each adjustment impacts coordinate accuracy, stability, and computation.

Step 4: Rebuilding and Deploying the Customized Binary

Once all modifications are complete, the final step is to rebuild the binary to include these customizations.

go build -o serf ./cmd/serf

Once built, deploy the binary to each container in the 100-node cluster.

Resource: Clone this GitHub package for Serf and Containerlab network implementation

Clone: git clone
https://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@github.com/anjumm/serf.git

Fork for Changing Parameters:

https://github.com/anjumm/memberlist/tree/master