Fog Computing: Edge-ready Load Balancing

1 Documentation

The aim of this project[1] was to implement an edge-ready network communication, establishing a protocol for reliable message transfer in the application layer. We decided to demonstrate this on the use-case of a load-balancing node, which can be reached in the cloud via HTTP and would then reroute the HTTP requests to registered edge-nodes based on their available CPU and RAM availabilities.

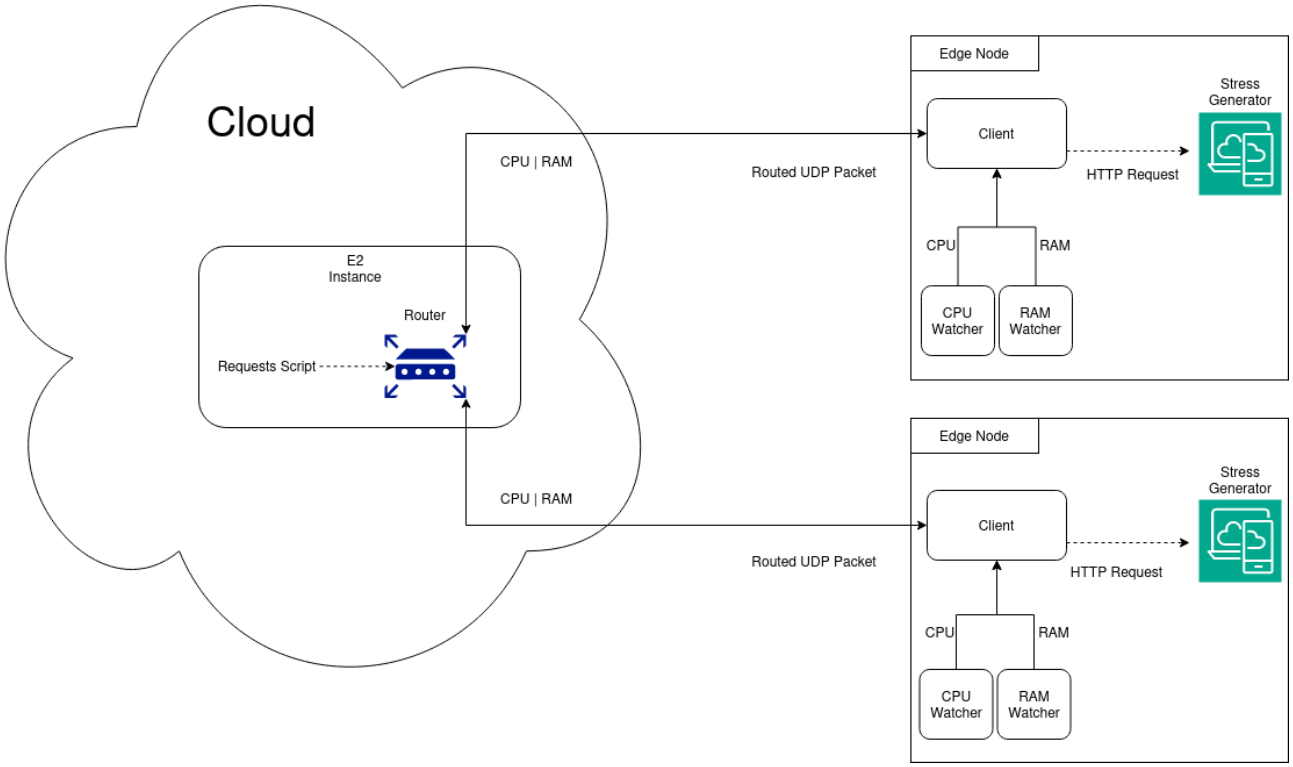

Figure 1: Structural diagram displaying the cloud setup of this load-balancing project. The router is deployed on a cloud VM and accepts connections from the edge nodes. On the edge nodes, there are the clients and watcher services deployed to monitor the edge node system and send data to the router.

To achieve this goal, we decided to implement 4 different services: the router, the client, and two watcher services for CPU and RAM monitoring. Additionally, a system-stress-generating service[2] to receive the forwarded HTTP requests and cause system stress for demonstrating routing decisions was needed, as well as a script to generate those initial HTTP requests, that would be forwarded via the router. The projects were written in Go[3] to allow usage of Go's concurrent capabilities using Go routines. The structural setup of how these components work together is depicted in fig. 1.

```go
type PacketUDP struct {
        Id   string
        Data []byte
}

type SafeAcks struct {
        Mu   sync.Mutex
        Acks []string
}

type SafeBuffer struct {
        Mu   sync.Mutex
        Data []PacketUDP
}
```

Figure 2: Struct definitions for UDP communication

The entire edge communication is built using UDP sockets and the 3 struct definitions depicted in fig. 2. The `PacketUDP` struct is the packet that is sent via UDP as a byte array, containing either CPU or RAM data when coming from the client or an HTTP Request body when coming from the router. These structs are created every time an HTTP request reaches either the client or the router, and both store those requests in a `SafeBuffer`, which allows thread-safe storage, as another Go routine will be handling the sending process. As UDP does not implement reliable message transfer on the transfer layer, each sent UDP packet containing a `PacketUDP` struct is expecting an acknowledgment for the sent packet. So, a short time after sending the packet, it is checked whether or not the packet was acknowledged by the other side using the `SafeAcks` struct, which stores the UUIDs of received acknowledgments. Otherwise, the packet is resent until this process results in a valid acknowledgment. In case the UDP connection needs to be re-established, there are certain safeguards that will reestablish the UDP connection when required for sending the packet, even though UDP, in theory, is a connectionless protocol. The `SafeAcks` struct also allows thread-safe access, as sending and listening on the UDP socket is done by separate Go routines.

```go
func Send(conn *net.UDPConn, data []byte, id string, acks *SafeAcks) error {
	for i := 0; i < 10; i++ {
		err := SendMessage(conn, data)
		if err != nil {
			log.Println("Retrying... ", err)
			time.Sleep(100 * time.Millisecond)
			continue
		}
		for i := 0; i < 10; i++ {
			time.Sleep(100 * time.Millisecond)
			acks.Mu.Lock()
			data := acks.Acks
			acks.Mu.Unlock()
			for _, ack := range data {
				if ack == id {
					return nil
				}
			}
		}
	}
	return &net.OpError{}
}
```

Figure 3: Function that wraps the Socket-sending-function and demonstrates retries during packet send attempt

The function implementation depicted in fig. 3 shows the attempts for each request to ensure the successful transfer of data. `SendMessage` is called up to 10 times until no error is reported from sending the UDP packet. Afterward, the function searches for a valid acknowledgment of its sent message, by searching for the UUID of its sent packet. If, after 10 tries, no acknowledgment is found to contain the UUID, it returns an error, indicating to the calling Go routine to handle the not acknowledged message.

```go
func handleBufferPacketSend(socket *net.UDPConn, reqBuffer *RequestBuffer, acks *SafeAcks) {
	var sleepFactor = 1
	var baseSleep = 1000
	for {
		if len(reqBuffer.Buffer) <= 0 {
			time.Sleep(1000 * time.Millisecond)
			continue
		}
		reqBuffer.Mu.Lock()
		req := (reqBuffer.Buffer)[0]
		reqBuffer.Mu.Unlock()

		err := Send2Router(socket, req, acks)

		if err != nil {
			var sleepDuration = baseSleep * sleepFactor
			log.Printf("Retrying after %d seconds", sleepDuration/1000)
			time.Sleep(time.Duration(sleepDuration) * time.Millisecond)
			if sleepFactor <= 16 {
				sleepFactor = sleepFactor * 2
				continue
			}
			break
		}

		sleepFactor = 1
		reqBuffer.Mu.Lock()
		reqBuffer.Buffer = removePacketUDP(reqBuffer.Buffer, req)
		reqBuffer.Mu.Unlock()
	}
}
```

Figure 4: Function that handles the send routing for the client service

The up-propagated error causes timeouts that scale up as depicted in **??** and retry the sending of the packet after the scaled timeout. A packet is only removed from the `SafeBuffer` struct if and when the packet was sent successfully and acknowledged to have been received successfully. This mechanism is implemented for communication in both directions, thereby guaranteeing a reliable message transfer. The implementation using the buffers also guarantees that if one side is unavailable or crashes, the messages meant for it are kept in the buffer of the other component and will be sent out as soon as the components can establish or re-establish a connection.

References

[1] "Numyalai/fog-computing-assignment: Fog computing prototyping project. the project includes services like router, and a watcher, all implemented in golang." (2024), [Online]. Available: `https://github.com/numyalai/fog-computing-assignment` (visited on 07/09/2024).

[2] "Numyalai/go-stress." (2024), [Online]. Available: `https://github.com/numyalai/go-stress` (visited on 07/09/2024).

[3] "The go programming language." (2024), [Online]. Available: `https://go.dev/` (visited on 07/09/2024).