Global Data Plane Router on Click

Nikhil Goyal
John Wawrzynek
John D. Kubiatowicz

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-234
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-234.html

December 14, 2015

Acknowledgement

I would like to express my gratitude to Professor John Wawrzynek and Professor John Kubiatowicz for their esteemed supervision and guidance. Their constant encouragement, support and invaluable suggestions made this work successful. I am deeply indebted to them for their time and effort in reviewing this report.

I would also like to thank the entire Global Data Plane team at Berkeley's Swarm lab for their constant support. I am grateful to Mr. Ken Lutz and Mr. Eric Allman for allowing me to be a part of this prestigious project and guiding me throughout my duration in this project. A lot of the work presented in this report overlaps with the contributions made by each team member in different domains in order to make the GDP a distributed, secure and durable storage platform.

Global Data Plane Router on Click

By Nikhil Goyal

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor John Wawrzynek
Research Advisor

(Date)

* * * * * * *

Professor John Kubiatowicz
Second Reader

(Date)

Abstract

In recent times, there has been an explosion in the number of smart devices, which when connected to heterogeneous computing platforms, offer a variety of services. With the help of the growth in Internet connectivity, these devices offer a rich interactive experience and have played a pivotal role in the emergence of the Internet of Things (IoT) paradigm. The Global Data Plane (GDP) project, developed at Berkeley's Swarm Lab, aims to provide a unified network and data storage platform for constructing IoT applications. These applications are composed of a variety of sensors and actuators running on different computing platforms. The goal of the GDP is then to provide a middleware interface that can help in the communication between these entities. The intent is for the GDP to be deployed on a large, geographically distributed collection of heterogeneous computing nodes. Sensors and actuators will communicate with nearby nodes, potentially acting as gateways, and data can in principle be placed on any storage server within the system depending on the user's needs. In order to achieve this vision, an efficient, robust, and flexible means of routing between GDP entities must be implemented. Such a routing mechanism is the focus of this work.

In this report, we introduce a new GDP programmable router design implemented completely in software. These routers have been implemented using the Click Modular Router Platform. They are responsible for making the GDP a distributed system that can make IoT applications scalable. The routers have been designed to create a hierarchical routing network, in which users are allowed to create their own local subnetworks composed of heterogeneous IoT devices and services. These local subnets may also reside behind firewalls or NAT-enabled routers for security purposes. In order to allow interaction between different private networks, our routers also form a common global network. The router nodes in this global network serve as relays to the ones residing in the local subnets created by users. Thus, using these routers we aim to create a self-organizing network, which can interconnect smart devices and storage servers.

Acknowledgment

I would like to express my gratitude to Professor John Wawrzynek and Professor John Kubiatowicz for their esteemed supervision and guidance. Their constant encouragement, support and invaluable suggestions made this work successful. I am deeply indebted to them for their time and effort in reviewing this report.

I would also like to thank the entire Global Data Plane team at Berkeley's Swarm lab for their constant support. I am grateful to Mr. Ken Lutz and Mr. Eric Allman for allowing me to be a part of this prestigious project and guiding me throughout my duration in this project. The main reason for the success of this project is the collaborative environment in the team and the outstanding ability to recognize each team member's skill set and assign work accordingly. A lot of the work presented in this report overlaps with the contributions made by each team member in different domains in order to make the GDP a distributed, secure and durable storage platform.

Finally, I would like to thank my fellow Swarm lab colleagues Nitesh Mor, Nicholas Sun and Ben Zhang for their contributions and valuable suggestions without which this project wouldn't have been successful.

Table of Content

1

Introduction

In recent times, there has been an explosion in the number of smart devices, which when connected to heterogeneous computing platforms, offer a variety of services. With the help of the growth in Internet connectivity, these devices offer a rich interactive experience and have played a pivotal role in the emergence of the Internet of Things (IoT) paradigm. In order to create large-scale IoT applications, developers tend to use existing public cloud infrastructure services. With little investment in the infrastructure, collecting sensor data and streaming it back to the cloud becomes highly trivial, even for a novice user. Several IoT cloud platforms have gone further by offering easy-to-use APIs, data processing, visualization, and sample code for various hardware platforms. Companies in this space offer complete solutions including hardware gateways, smartphone applications, web portals and cloud services. However, on closer inspection, several significant problems have been revealed when a cloud architecture is involved. These problems include issues with latency, security, locality, scalability, availability and durability [1].

This report in particular focuses on two issues: latency and locality. According to a recently released publication from the Swarm Lab at Berkeley [1], application developers view the cloud as a centralized component that interconnects smart devices. However, from a network point of view, the cloud is on the edge of the network. Even simple IoT applications, such as those that turn on an air conditioner in response to a rise of the local temperature, will experience unpredictable latencies from sensing, wireless transmission, gateway processing, Internet delivery, and cloud processing.

This gives rise to the need for a distributed system, which not only acts as a peer-to-peer storage system, but can also improve locality by allowing users to form distinct local sub-networks, within the users' environment, that interconnect IoT devices, data storage servers and applications. For example, if a given user is building multiple IoT applications using various types of sensors and actuators within her home, she should be allowed to create her very own local network in which she can interconnect these devices together along with a server that can store her data locally and securely. With this approach, the system would ensure a tight cap on the upper bound latency in the user's network application. This local network could also reside behind a firewall or a NAT device for security purposes. In such a scenario, the network must still be able to connect and interact with other local subnets via a common global network. Interaction between different local subnets becomes crucial in the scaling of IoT applications and moving forward towards the vision of a "smart city" for example [23].

Our project is primarily motivated by the ongoing Global Data Plane (GDP) project at Berkeley's Swarm Lab [2]. The GDP aims to provide a unified network and data storage platform for timeseries data. The system will be capable of dealing with sensor data (both in raw and processed forms) as well as actuation signals and will offer publish-subscribe semantics to its clients. Other important goals for the GDP include durability of data through replication and erasure coding, scaling to a large quantity, say 10^{12}, of nodes, and seamless migration of data between servers when necessary. Although this paper doesn't deal with the security aspect, we would like to mention that security is also a major concern handled in the GDP. It will be implemented using cryptographic tools like signatures.

The intent is for the GDP to be deployed on a large, geographically distributed collection of heterogeneous nodes. Sensors and actuators will communicate with nearby nodes, potentially acting as gateways, and data can in principle be placed on any storage server within the system depending on the user's needs. We expect that a variety of storage options will be available, ranging from powerful servers running in the cloud to more modest machines running in local area networks and co-located with sensors and actuators. Also, as we shall see in section 2, data can be mobile and replicated widely among these storage options to ensure durability, availability and locality. In order to achieve this vision, an efficient, robust, and flexible means of routing between GDP entities must be implemented. Such a routing mechanism is the focus of this work.

As part of this project, we have built special purpose programmable software routers that we call GDP routers. These routers have been designed using the Click Modular Router Platform and are used to build scalable IoT networks. Their main characteristic is that they perform location independent routing. In today's network, data and services can be mobile and replicated widely to ensure locality, durability and availability. Location independent routing provides a simple platform above the network, and allows for implementation of distributed applications that don't require any knowledge of the underlying dynamic network. Thus overall it provides a full virtualization of resources, which is one of the essential design philosophies of the GDP.

Since these routers are programmable in software, they can be deployed on heterogeneous platforms ranging from superior cloud servers to small-embedded hardware devices such as Beagle Bone Black, Raspberry Pi, Arduino etc. Thus, using these routers, we aim to establish a self-organizing network. We also wish for the system to be self-configuring, by allowing other GDP entities such as sensors, actuators and data servers, to automatically determine an entry point into the network, closest to them. For this we integrate our system with Zeroconf, a technology used to discover systems and services in a local area network [4]. Using Zeroconf, new GDP clients will be able to locate our GDP routers and contact them in order to connect with existing clients and make use of the services they have to offer.

Overall, this report presents our following three main accomplishments in the GDP routing domain:

1. Creation of a new router design that allows free movement of data among participating nodes, through location independent routing.
2. Development of a Routing Maintenance Protocol that makes the GDP self-organizing and self-configuring, by dynamically handling joining of new nodes, node failures and creation

of private networks behind firewalls and NATs.

3. Implementation of a programmable, configurable and event driven router architecture by deploying our new system design on Click.

We begin the report by describing the GDP Routing Problem in section 2. In this section, we first highlight the need for a location independent routing mechanism which can support replication and movement of data and services among participating nodes in the network. We also describe our previous router design and its limitations. Finally we present the goals that our new router design accomplishes in section 2.4. We next present our system overview in section 3 by classifying the type of nodes in the GDP and explaining the packet forwarding function of our new GDP routers. Sections 4 and 5 form the crux of this report. In section 4, we explain our GDP Routing Maintenance protocol (GDPRMP) that is used by the routers to handle dynamic operations such as joining of different types of GDP nodes and node failures. Section 5 next depicts our router implementation using Click. We also present our new and improved routing table architecture, used for handling router failures efficiently. In section 6, we evaluate our system by comparing it with the previous GDP router design. Section 7 highlights our future work in this project. Finally we conclude the report in section 8.

2

GDP Routing Problem

The purpose of the GDP is to provide a distributed data centric platform connecting Swarm Applications. These Swarm applications consist of sensor/actuator networks, which play a pivotal role in the notion of IoT. In this section we intend to present the GDP routing problem and goals, which our new router design must accomplish in order to make the GDP a distributed self-configuring and self-organizing network. We start by describing the need for a flexible location independent routing mechanism in the GDP in section 2.1. We then depict, in section 2.2, the format of the GDP Packet Data Units (PDU), used for communication between the GDP nodes. Section 2.3 next describes the previous GDP router design and its limitations. Finally, based on the routing requirements of the GDP and the limitations of the previous design, we present the goals for our new router design in section 2.4.

2.1 Location Independent Routing

In today's networks, data and services can be mobile and replicated widely to ensure locality, durability and availability. In such a scenario, addressing mobile services in terms of the host on which they temporarily reside may not be the best design solution. Also, in order to access a given data object for example, we require the object to be universally accessible; the ability to read a particular object should not be limited by a user's physical location [12]. This posses the need for some form of an application level routing protocol that provides location-independent naming for sources and destinations of data and services. During communication, these endpoints can be dynamically mapped to any physical device producing or consuming data. The protocol should then be able to route data requests and responses efficiently based on the location-independent names assigned to the sender and receiver.

Location independent routing provides a simple overlay indirection layer above the network and allows for implementation of distributed applications that don't require any knowledge of the underlying dynamic IP network. It decouples objects and tasks from their physical locations [25]. Messages are directed to a flat identifier instead of an IP address. Consequently, messages are delivered directly to the recipient, instead of an out of date location. As a result, objects and services could be freely moved and replicated amongst the participating physical nodes without changing the applications built on top of the network. Data can be routed in a timely fashion between the nodes while simultaneously archived for future use. On failure of a particular node, another may take over and assume the role of producing or consuming data without complex failover proto-

cols. Location independent routing also allows for a common networking platform for multiple users distributed physically across the globe, who could be connected using different types of wired/wireless interfaces. Thus overall it provides a full virtualization of resources, which is one of the essential design philosophies of the GDP [12].

In the GDP, each endpoint (such as a data log, as we shall see shortly) is defined by a unique 256-bit destination ID. This ID has nothing to do with the physical location of the endpoint. For example, we could generate the ID of a given endpoint by hashing the data or the cryptographic key associated with the endpoint using the SHA-256 algorithm. Thus, the ID is cryptographically related to some aspect of the endpoint, other than its current physical location. For the purpose of this report, we define a GDP client to be any GDP node that could host multiple endpoints represented by unique 256-bit IDs. Any node wishing to join the GDP network and utilizing its services is also considered as a GDP client. Based on this, we present four types of clients that connect to the GDP network:

1. Sensors
2. Actuators
3. Mobile Applications
4. GDP Log Servers

Figure 2.1 illustrates the different types of GDP clients. For all our subsequent diagrams, we will be representing GDP clients as purple circles. While the first 3 types of GDP clients are self-explanatory, we do need to provide some detail about GDP log servers. A log server is a server process that actually stores data and responds to commands for reading/writing data. Other GDP clients (sensors, actuators and mobile applications) read and write data from log servers in the form of logs. A log can be regarded as a queue/list structure whose each entry is called a datum. A datum consists of the real data along with some metadata, which includes sequence number (for data versioning), timestamp and other parameters for security purposes that are beyond the scope of this work. Henceforth, we will use the term Global Data Plane Channel/Log (GCL) to describe an addressable log. We would also like to mention that multiple GDP messages with different destination IDs could be routed to the same physical client. For example, each log is uniquely identified by a 256-bit ID and a given log server may store multiple logs. Hence all messages with the IDs of the logs as destination will be eventually routed to the same log server.

Figure 2.2 illustrates a typical GDP snapshot consisting of router nodes (blue circles), a few client nodes (purple circles) and the endpoints hosted by each client (green circles). The routers, belonging to different subnets, form an overlay network on top of the underlying IP layer. GDP clients connect to their nearby router nodes to preserve locality. The red lines show the path of interaction between two GDP endpoints, 1234 and 4556. These IDs are 4 digit integers simply for illustration purposes. In reality, each GDP endpoint is assigned a 256-bit ID from a common flat namespace as described earlier. It can be seen that some of the endpoints, clients and routers are behind different firewalls/NATs. In such a scenario, it is up to our routers to form an overlay such that information can be accessed universally. This will be discussed further in later sections. Although a given client could host multiple 256-bit destinations, for simplicity of our subsequent diagrams, we shall henceforth show a one-to-one mapping between clients and 256-bit IDs.

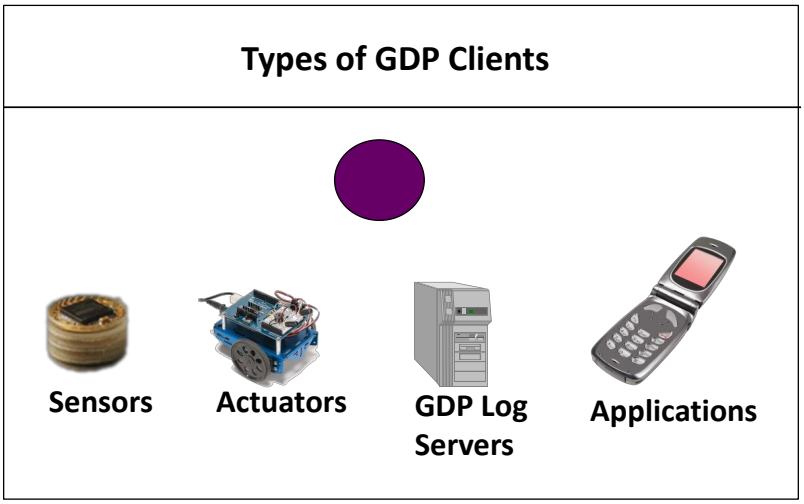

Figure 2.1: Types of GDP clients.

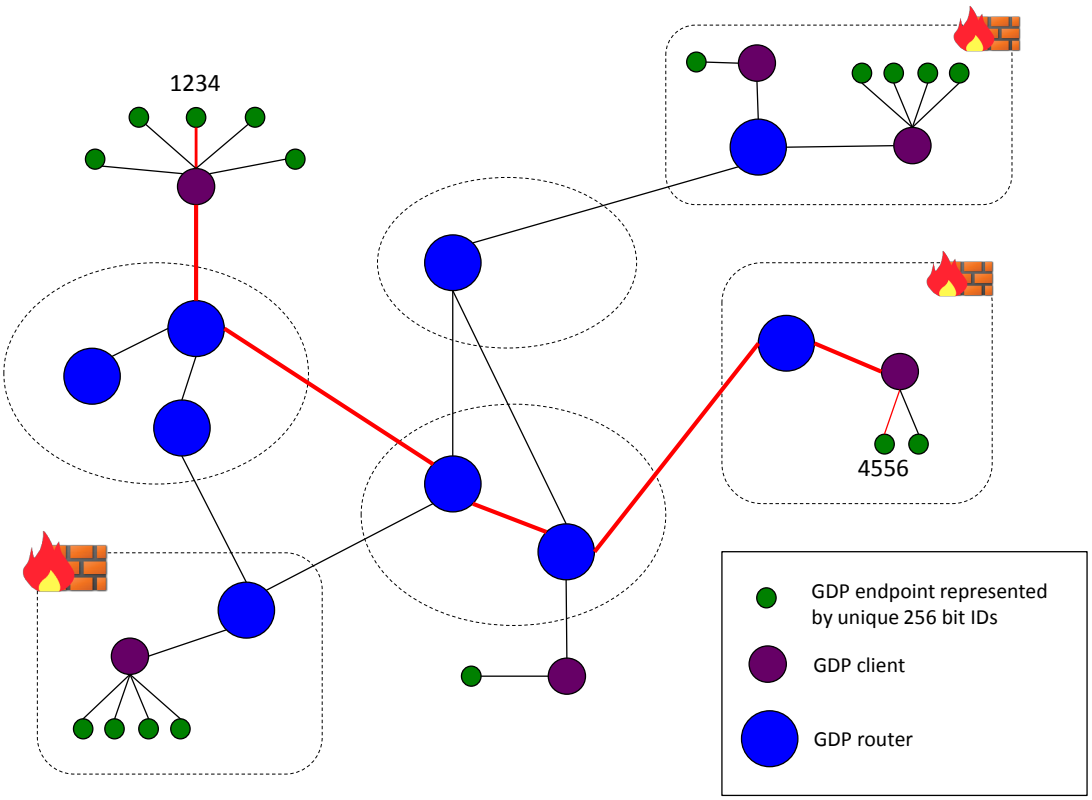

Figure 2.2: A typical snapshot of a GDP network. The routers form an overlay network on top of the underlying IP layer. Clients connect to routers. Each client hosts multiple GDP endpoints. Each endpoint is represented by a unique 256-bit ID. The routers perform location independent routing by using 256-bit IDs as source and destination. The red lines show the path taken by two endpoints with IDs, 1234 and 4556, to interact with each other.

Thus each client will have a single endpoint and so the client itself can be represented with the endpoint ID. However this shouldn't undermine the fact that a single physical client can host multiple endpoints with unique IDs.

The GDP routers have been designed to perform location independent routing by using the 256-bit IDs to forward messages between different GDP clients. Thus, each router's routing table maps 256-bit names to connections with corresponding nodes, unlike a typical router design that uses IP addresses instead. We will describe the GDP routing protocol used by our routers to forward GDP client messages in more detail in section 3. The first task for any GDP client, to join the GDP, is to find an entry point into the network. This entry point is a GDP router node close to the client. The next task for the client is to advertise all its 256-bit destination IDs to the router. This is achieved by sending a GDP PDU containing all the 256 bit IDs to the routers. The routers, in the overlay, subsequently forward all GDP PDUs, having any of these IDs as destination, to the corresponding client. When an endpoint has to move from one client to another, due to some reason, the old client sends a withdraw GDP message (the PDU has a command called `CMD_WIDTHDRAW`, as we shall in next section.) to its connected router indicating it no longer holds the endpoint. The new client re-advertises (using a PDU with a command called `CMD_ADVERTISE`) the endpoint ID to its own connected router. Both the GDP routers must convey the information about the movement of the endpoint to other router nodes in the overlay network. We call the protocol that updates and maintains each router's routing table, GDP Routing Maintenance Protocol (GDPRMP).

2.2 GDP PDU Format

GDP endpoints interact with each other by sending/receiving GDP Packet Data Units (PDUs). These packets may be of variable length. They are sent over TCP connections formed between clients and routers. Based on their purpose, GDP PDUs can be broadly categorized into two:

1. *Forwarding Packets:* These packets are sent from a source GDP client to a destination GDP client. They consist of the 256-bit IDs of the source and destination endpoints. They include commands for services (e.g. reading from a log or appending data to a log).
2. *Routing Layer Packets:* These packets are meant for the routing layer formed by the routers. They include advertisement/withdraw messages from new clients to the routers and GDPRMP (GDP Routing Maintenance Protocol) messages between the router nodes. The routers use these packets to update their states and data structures.

Figure 2.3 depicts the GDP PDU format. It can be seen that the total header size is 80 bytes. The length of the data section is calculated from the header field, *Data Length*.

The following is a description of the relevant fields in the GDP PDU:

1. *Version:* This field specifies the version of the GDP Routing Protocol used. At present this is set to 0x03, which indicates the latest version that also accounts for GDPRMP messages.
2. *Cmd/Ack:* This field indicates the purpose of the message. Tables 2.1 and 2.2 show the different types of commands/acknowledgements sent by GDP endpoints. Table 2.1 depicts the commands meant for the routing layer and Table 2.2 depicts the commands used by forwarding packets between endpoints.
3. *Time to Live:* The originator of the packet sets a number of hops that this packet is allowed to take. At each hop the routing layer must decrement this field; if it reaches zero it is assumed that the packet is looping.

4. *Reserved:* This field must be zero when the PDU is created and ignored when the PDU is read. Forwarders must transmit this field unchanged.

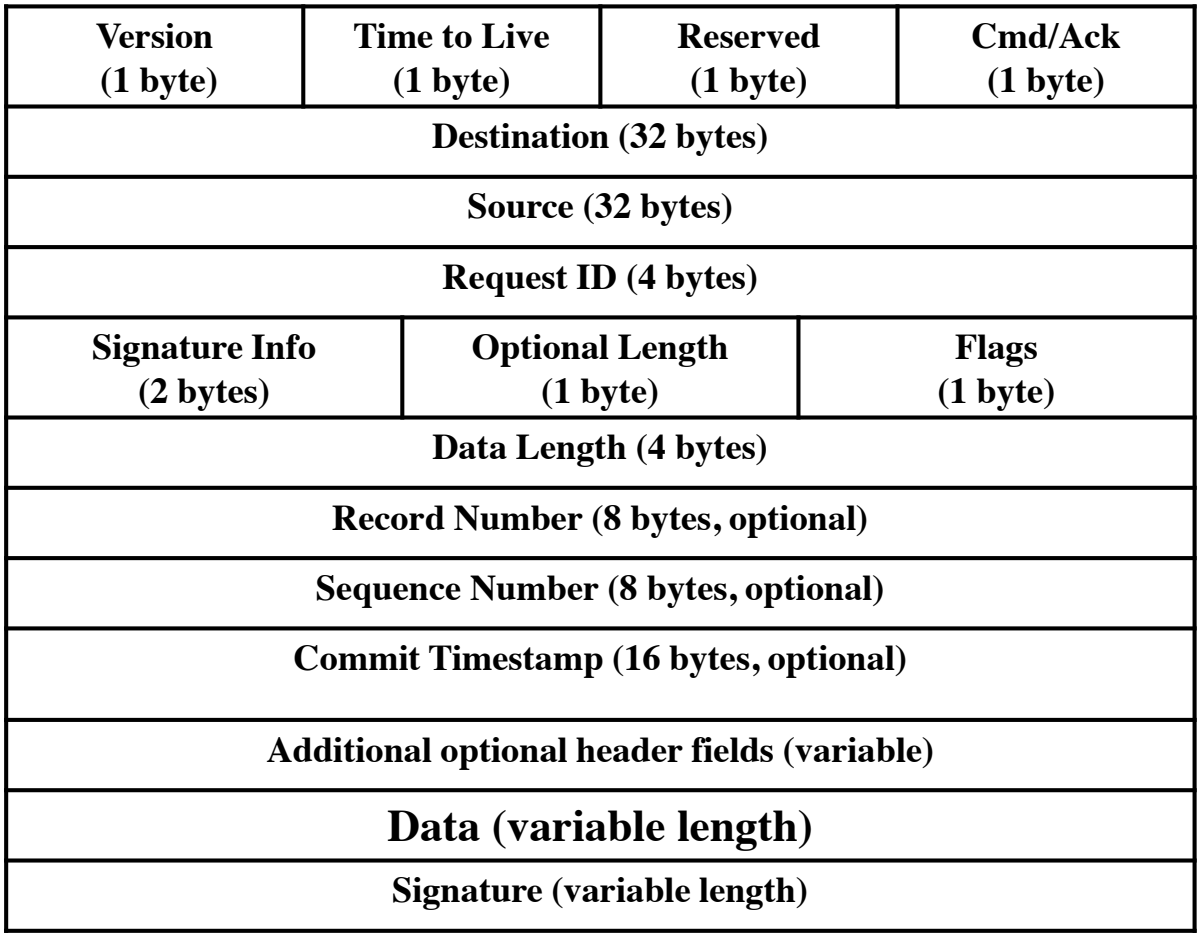

Figure 2.3: GDP PDU format.

Cmd (value)	Description
CMD_ADVERTISE (1)	Advertise willingness to respond for a given 256-bit Destination Address.
CMD_WITHDRAW (2)	Withdraw advertisement.
CMD_GDPRMP (3)	GDPRMP messages between routers (section 4)

Table 2.1: Commands used by GDP Routing layer packets.

5. *Destination:* This field represents the 256-bit destination for a given PDU. The special value consisting of alternating 255 bytes and zero bytes (”0xff00ff00...”) indicates messages to be interpreted by the routing layer.

Cmd/Ack (value)	Description
CMD_PING (64)	Test connection. Can also be used for keepAlive
CMD_CREATE (66)	Create a GCL.
CMD_OPEN_AO (67)	Open a GCL for append only.
CMD_OPEN_RO (68)	Open a GCL for read only.
CMD_CLOSE (69)	Close a GCL
CMD_READ (70)	Read a given record number
CMD_APPEND (71)	Append a record to a GCL
CMD_SUBSCRIBE (72)	Subscribe to a GCL
ACK_SUCCESS (128)	Generic operation succeeded
ACK_CREATED (129)	New resource creation succeeded
ACK_DELETED (130)	GCL deleted successfully
ACK_CHANGED (132)	Resource changed successfully
ACK_CONTENT (133)	Content returned. The payload contains the content and the signature
NAK_C_BADREQ (192)	Invalid Request
NACK_C_BADOPT (194)	A bad option value was supplied, e.g., a negative record count in a subscribe
NAK_S_INTERNAL (224)	Internal server error
NAK_S_NOTIMPL (225)	Indicates that the GDP doesn't implement the requested function; for example, Unrecognized command codes get this response.

Table 2.2: Commands/Acknowledgments used by GDP forwarding PDUs.

6. *Source:* This field represents the 256-bit source of a given PDU. For data originating from a named GCL it will be the name of that GCL; otherwise it will be the address of the program instance.
7. *Signature Info:* This field contains the message digest and signature length. If this is zero there is no signature.
8. *Request Id:* This field is used to disambiguate calls in progress. Requests (commands) are correlated with responses on the basis of the concatenation of Destination, Source, and Request Id.
9. *Record Number:* This field contains the record number to read in a particular log.

10. *Sequence Number:* When the PDU is signed, this field is included with the signature to avoid replay attacks.

2.3 Previous GDP Router

The early version of the GDP router was used for about 6 months in the Global Data Plane project. It was a stable preliminary version written in Python programming language and was used to form a fully connected mesh network between the GDP Routers. In order to start a new GDP router one needed to provide as input the locations of all other existing GDP routers in the network. The location of each router was specified using the IP address and port on which the given router was running and listening for incoming TCP connections. The new GDP router would then establish a TCP connection with each of the routers in the network. Each of the existing routers would then inform the new router about the GDP clients in the network.

For a given GDP client (sensor, actuator, server) to join the network, it would need to know an entry point into the network, which was the address of any of the GDP router nodes. This had to be statically provided to the client beforehand for it to join the network. The client would establish a TCP connection with a router node and would advertise its 256-bit destination IDs to the router node. The router node would then forward the IDs to each of the other existing GDP router nodes. Each of the Router nodes had a routing table, which mapped each 256-bit ID to the specific socket connection file descriptor responsible for handling the node with the corresponding ID. For a given router node, say x, connected to a given client node, the mapping would be from the client's 256-bit IDs to the socket interface between the router and the client. For all other router nodes, the mapping would be from the client's 256-bit IDs to the socket interface, representing the connection between the router and the router node x. Thus these interconnected GDP routers performed location independent routing based on the 256-bit IDs of the endpoints hosted by GDP clients.

Figure 2.4 depicts the previous GDP Router design by showing 8 routers (blue circles) forming a fully connected mesh network. The black arrows represent the two-way TCP connections formed between the router nodes and a couple of GDP clients (purple circles). The red arrows show the path taken by a message whose source is the client with ID 4556 and destination is the client with ID 1234. These IDs are 4 digit integers simply for illustration purposes. In reality, each GDP endpoint is assigned a 256-bit ID from a common flat namespace.

Although this design was stable, it was limited in four major aspects. The first one was that a new router node had to be provided with the IP address of each of the existing routers by the programmer/administrator running the new node. In order to create an overlay network with a large number of nodes, this process is cumbersome. Thus we needed a special joining protocol, which could be used by a new router node to locate all other nodes by just providing it with the location of a single existing node in the network.

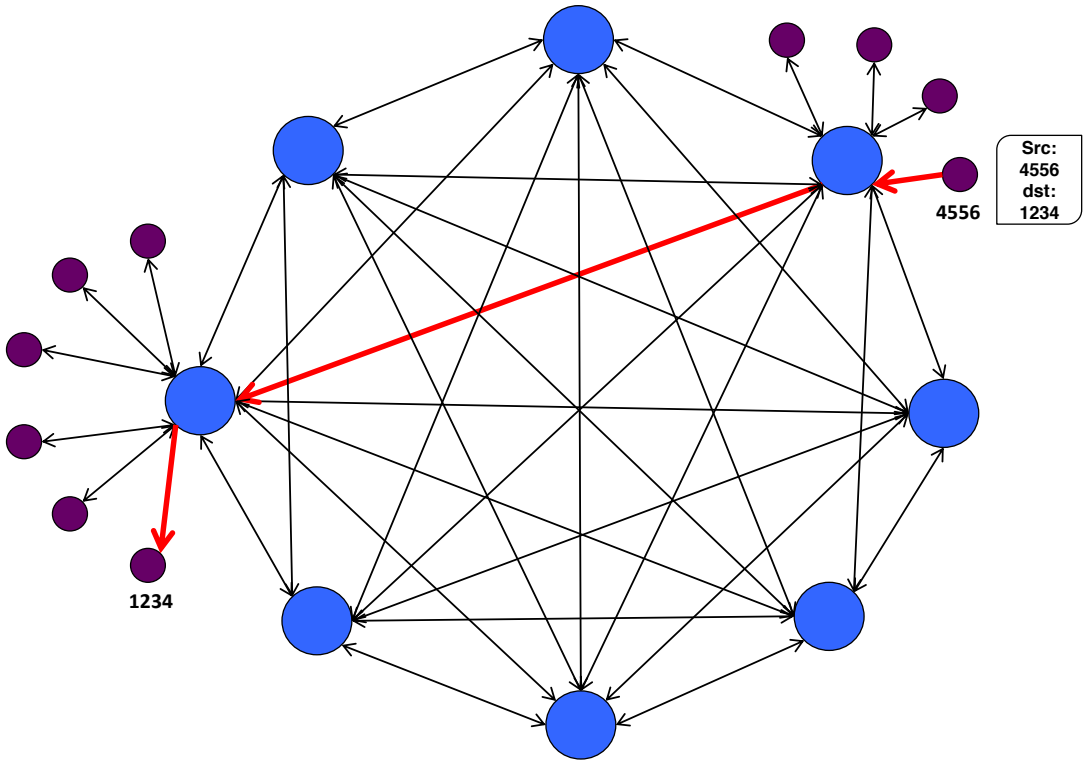

Figure 2.4: Previous Router Design. All the GDP routers, represented as blue circles, form a fully connected mesh network. The black arrows represent the two-way TCP connections formed between the router nodes and the GDP clients (purple circles). Each router maintains a routing table which maps 256-bit IDs to the appropriate connection interface.. The red arrows show the path taken by a GDP client message whose source is the client with ID 4556 and destination is the client with ID 1234.

The second limitation was that as per the design, it was the responsibility of the new router node to contact each of the other existing routers and form a TCP connection with them. This prevents the system from creating router nodes behind firewalls or NAT devices as these nodes cannot be reachable by a new node that is running outside the NAT/firewall for example. Thus this design did not allow creation of local subnets and imposed a need for a new hierarchical design, which could differentiate between NATed and non-NATed routers. For information on the working of the Network Address Translation (NAT) protocol, please refer to section A.2, Appendix A.

The third limitation of this design was based on the fact that when a new client (or an endpoint) wants to join the network, it needs to be statically provided with the IP address and port of an existing GDP router in the network. Instead, we would prefer a more dynamic solution, in which the new client could self discover an entry point into the network, once it starts running.

Finally, The last limitation was the performance degradation issue that originated from the fact that the previous GDP routers were written in Python. As we shall prove in section 6, the routers's performance in terms of end-to-end latency of GDP PDUs degraded with the increase in interaction between the GDP clients due to higher degree of network congestion. This is because Python adds an extra layer of abstraction above C in order to make code written by programmers more user-readable. This layer adds significant performance overhead and it would be beneficial to transfer

the router code to a lower level language such as C/C++.

2.4 Towards a New GDP Router Design

After studying the requirements of routing within the GDP and analyzing the limitations of the previous router design, we present the goals for our new GDP router.

1. The new router design must be able to perform location independent routing between endpoints hosted by GDP clients, using their 256-bit IDs, similar to the previous router design. In order to ensure forward compatibility, GDP endpoints must still able to interact with each other using the same GDP PDU format that was used by the previous routers.
2. Unlike the previous design, the new routing system must follow a maintenance protocol to dynamically form the overlay network on top of the IP layer. This protocol must allow new router nodes to gain sufficient information to join the overlay, using a single entry point. The protocol must also ensure that clients are able to route to endpoints on other clients even as routers fail and join the network. Thus the aim of this protocol must be to make the GDP a self-organizing system.
3. Our new design must allow creation of private networks (residing behind NATs/firewalls), by introducing a hierarchical design, which could differentiate between NATed and non-NATed routers. This design should first create a common global network consisting of public domain router nodes. This global network should connect with local subnets formed by interconnection of router nodes and clients behind a specific NAT/firewall.
4. The new design must aim towards making the GDP a self-configuring network. Clients must be able to automatically discover an entry point into the network based on their location.
5. Our new router design must perform significantly better than the previous routers, especially under scenarios where the system is under heavy workload due to large number of messages exchanged between clients.

Sections 3, 4 and 5 describe our new router design and how it meets the above goals. While section 3 addresses points 1 and 3, section 4 describes our GDP Routing Maintenance protocol used to meet goals 2 and 4. In section 5, we describe our router implementation using the Click Modular Router Platform, an event driven architecture written in C++. In section 6, we shall compare our new router design with the previous router to evaluate how far we have met requirement 5.

3

System Overview and Design

In our new router design, a GDP router node executes two main functions:

1. It serves as an entry point into the GDP by advertising itself as a GDP service. New GDP clients/endpoints discover the entry points closest to them to join the GDP network.
2. It performs location independent routing by sending messages from a given 256-bit source endpoint to a given destination.

As part of our new design, we have different types of router nodes based on whether they reside in the public domain or behind a particular firewall/NAT. Section 3.1 describes these nodes in detail, also providing the types of computing platforms on which these nodes can be deployed. We also describe the technique used by GDP clients to discover their nearby router nodes (function 1). Sections 3.2 and 3.3 next depict the routing of messages between two GDP clients by describing two essential data structures maintained by the router nodes and how they are used to route GDP messages between clients (function 2).

3.1 Types of GDP Routers and Discovery

The main purpose of the GDP router nodes is to interconnect GDP clients, by routing messages from a given source client to a given destination client. In the new GDP router design, there are 2 types of GDP routers:

1. *Primary Nodes:* A GDP router node is said to be a Primary (P) node if it doesn't reside behind a firewall or a NAT device. Thus, Primary nodes reside in a public domain space where any P node can contact all other existing P nodes. Similar to the previous GDP router design, all P nodes in the network form a fully-connected mesh network. We defer the discussion of the algorithm used to form this network to section 4. For now it is sufficient to know that every P node is aware of every other P node in the entire GDP network. Apart from connecting regular GDP clients, the P nodes also serve one more function. They connect local area networks guarded by a NAT/Firewall as well. Thus, they serve as public relays for two local area networks to interact with each other.
2. *Secondary Nodes:* A GDP router node is said to be Secondary (S) if it is residing behind a NAT or a firewall, preventing any other node outside the firewall/NAT to initiate a connection with it. Our routing maintenance protocol ensures that all S nodes in the same local subnet (behind the same NAT/firewall) form a fully connected mesh network. Thus two S nodes

behind the same NAT/firewall can directly talk to each other. Each S node also initiates a connection with a P node, which serves as its window to the rest of the GDP network. Thus, this assigned P node serves as a public relay/proxy to the given S node.

Our routers are written in software using Click. Thus these nodes can be set up on heterogeneous computing platforms. Figure 3.1 depicts the representation of the various types of GDP router nodes and the computing platforms on which the nodes can be deployed. The platforms range from powerful cloud servers all the way to embedded hardware devices like Beagle bone, Arduino, Raspberry Pi etc. The only requirement is that the computing platform should be running Linux OS as Click is not compatible with any other operating system. Since the router nodes could also run on small-embedded hardware devices, we want to ensure that each S node choses its assigned P relay node carefully. For performance and memory reasons, we would not want a P node, running on a small-embedded device with minimal memory, to serve as a proxy to an S node. Our system allows users running P nodes to specify their willingness to let their P nodes serve as public relays. Based on this, it is ensured that no S node choses a P node which doesn't wish to act as a relay. This is discussed in more detail in section 5.

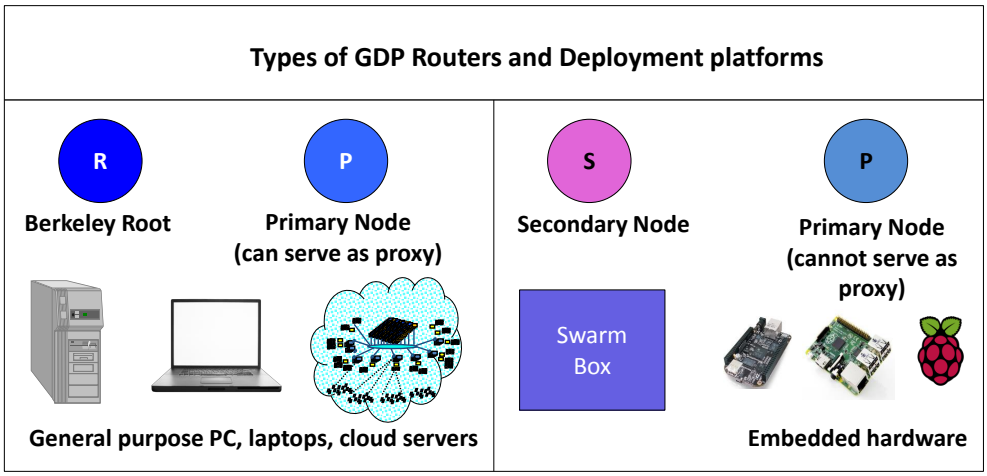

Figure 3.1: Types of GDP router nodes and the various computing platforms on which they can be deployed based on their type. All subsequent diagrams will use the appropriate circles to represent the corresponding router node type.

Each of the router nodes serves as an entry point into the GDP network. GDP clients that are non-NATed initiate a connection with a close-by P node to enter the network. A client residing behind a NAT/firewall first tries to discover an S node residing behind the same NAT/firewall and connects with it. For router discovery, we use Avahi, an open-source Zero-configuration networking (Zeroconf) implementation. It allows programs to publish and discover services and hosts running a special Avahi daemon on a local network with no specific configuration. For more information about Zeroconf/Avahi, please refer to section A.3, Appendix A. We use Avahi by enabling our GDP routers to advertise themselves as a special service, which new clients can automatically discover.

A new client will use Avahi to discover a router node in its local area network that can serve as its closest entry point into the GDP network. Avahi will locate all the GDP router nodes running in the client's local area network and send their locations (IP address and port) to the client in the

form of a list. The client will contact each node in the list until it has successfully established a TCP connection with one of the routers. It will then advertise to the router, the 256-bit destination IDs of all the endpoints that it hosts. In case there are no GDP routers running in the client's local subnet, we have installed 3 default router nodes in Berkeley with 99.99% availability. These nodes are Primary and are used as default contact points for clients who are unable to locate a router node in their LAN through Zeroconf. We call these routers Root nodes.

Figure 3.2 depicts the entire system consisting of all types of nodes and the hierarchical network they form. The Primary nodes, including the root nodes, form a fully connected mesh network. S nodes within the same private network are also fully connected. Each S node forms a 2-way TCP connection with P nodes, willing to serve as relays. Two S nodes belonging to different private networks use their assigned relays to interact with each other.

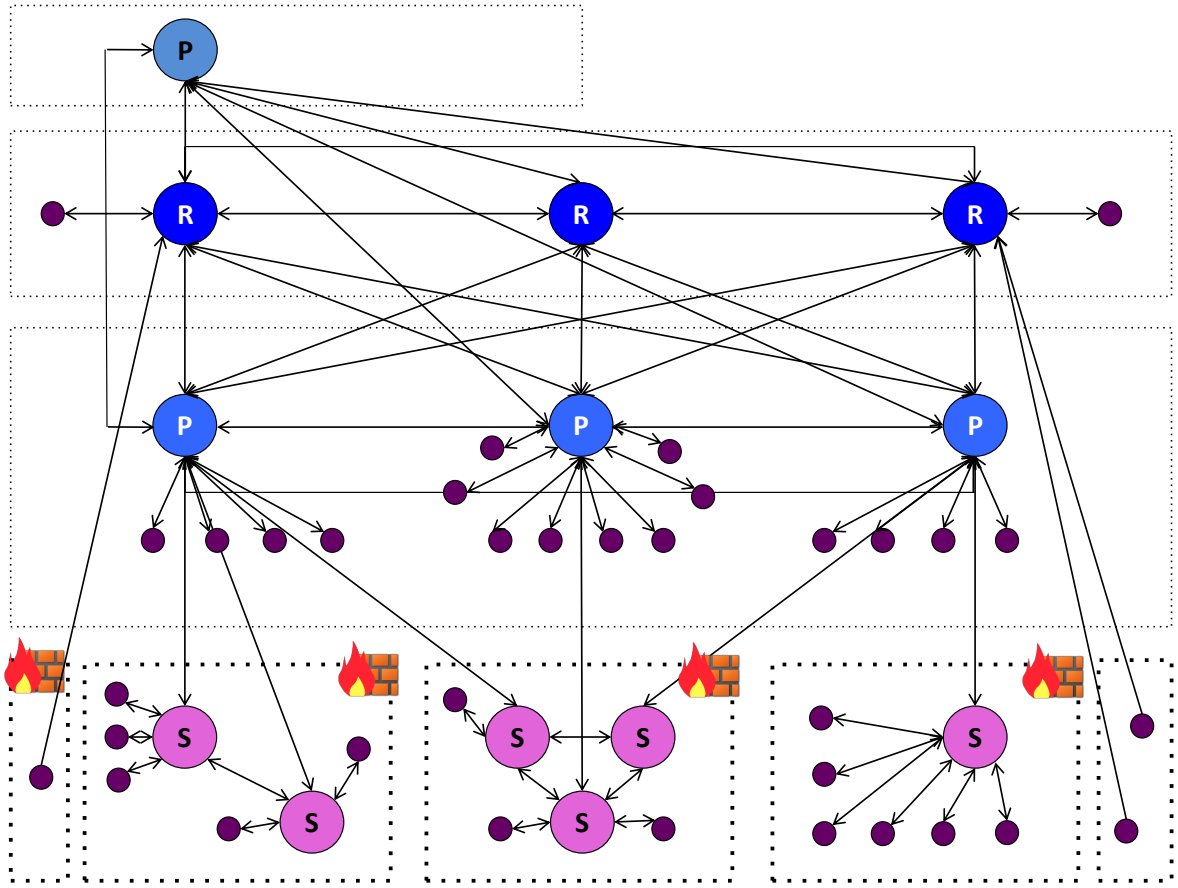

Figure 3.2: A typical GDP system snapshot. The root nodes and all other P nodes form a fully connected mesh network. S nodes within the same private network are fully connected. S nodes belonging to different private networks use P nodes as relays to interact with each other. The protocol ensures that P nodes not willing to serve as relays due to memory/performance reasons are not chosen by any S node. GDP clients either connect to their nearest router node using Zeroconf/Avahi or connect to the root nodes if no router is running in their LAN.

GDP clients either use Zeroconf to connect to router nodes close to them or connect to the root nodes in case they cannot find a router in their local network. The protocol involved in forming this system is called the GDP Routing Maintenance Protocol. We will study this protocol in Section 4.

3.2 Data Structures maintained by Routers

In order to route GDP message, the protocol uses two important data structure:

1. *Routing Table:* This table is used by router nodes to route GDP messages from one GDP client to another.
2. *Public to Private Map:* This is a hash map data structure used by P nodes serving as relays to S nodes belonging to different private networks.

As will be discussed in section 4, the GDP Routing Maintenance Protocol helps these routers to keep these structures up to date. We explain each of the data structures in the following subsections.

3.2.1 Routing Table

As mentioned in the previous sections, GDP routers route GDP client messages using location independent routing. This is performed by looking at the destination client's 256-bit ID and forwarding the message appropriately. Thus, the simplest design of a router's routing table is a single hash map table, which maps 256-bit IDs to the appropriate connection. The advantage of using a hash map is that we get a constant (O(1)) lookup time while forwarding a GDP client message.

Figure 3.3 illustrates a simple scenario where we have 3 P routers fully connected to each other. We also have a couple of GDP clients (each hosting one endpoint) with unique IDs connected to the router nodes.

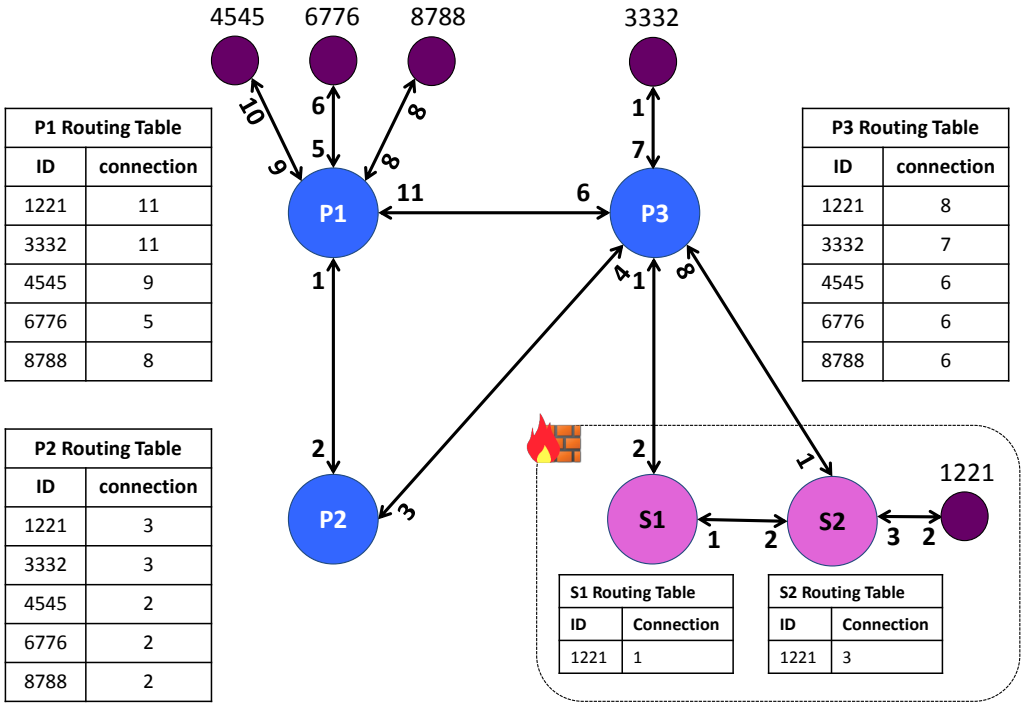

Figure 3.3: *Routing Table formed by each GDP router using GDPRMP.* Each link has two integers which depict the file descriptor representing the connection at the two end points.

On each connection link, we have two integers at both the end points. These integers are the file descriptors representing the given connection at the two end points. For the purpose of illustration each ID is again represented by a 4-digit integer in the diagram. However in reality, each endpoint ID is 256 bits as discussed in the previous sections. The routing table eventually formed by routers using the GDP Routing Maintenance Protocol (GDPRMP) is shown in the tables besides each router.

Let us look at the client entries in each router type's table.

1. For a P node, the routing table contains IDs of all GDP endpoints. These also include endpoints hosted by clients connected to all the S nodes.
2. For an S node, the routing table only contains the IDs of the GDP endpoints, present in the node's private network.

Although using a single hash map structure to form the routing table is simple and easy to maintain, we shall see in section 5 that by changing the design to use three map tables, we can reduce the amount of network bandwidth consumed during node failures by a significant margin. However for the purpose of explaining the dynamic operations performed by GDPRMP, we shall use the simple single map table design.

3.2.2 Public to Private Map

This data structure is maintained by each P node (willing to serve as a relay) to keep track of the physical locations of each S node using the P node as a relay. The physical location of an S node is represented by its own IP address, its NAT router's public IP and the port on which it is running. All three parameters are required to uniquely represent an S node in the entire GDP. The Public to Private map is a hash map structure where the key is a local subnet's public IP (public IP address of the NAT router of the local subnet). The value is a list of the physical locations of S nodes along with the file descriptors representing connections to the nodes. These S nodes have two characteristics:

1. Each S node has chosen the given P node as its relay.
2. Each S node in the list has the same public IP.

Figure 3.4 depicts a P node, which serves as a relay to S nodes belonging to two different private networks. These two private networks reside behind two different NAT-enabled routers that have public IP addresses 56.78.90.12 and 128.45.67.3. Each S node residing behind the NAT routers has its own IP address and a port on which it is running. The Table besides the P node depicts the Public to Private Map structure and its entries.

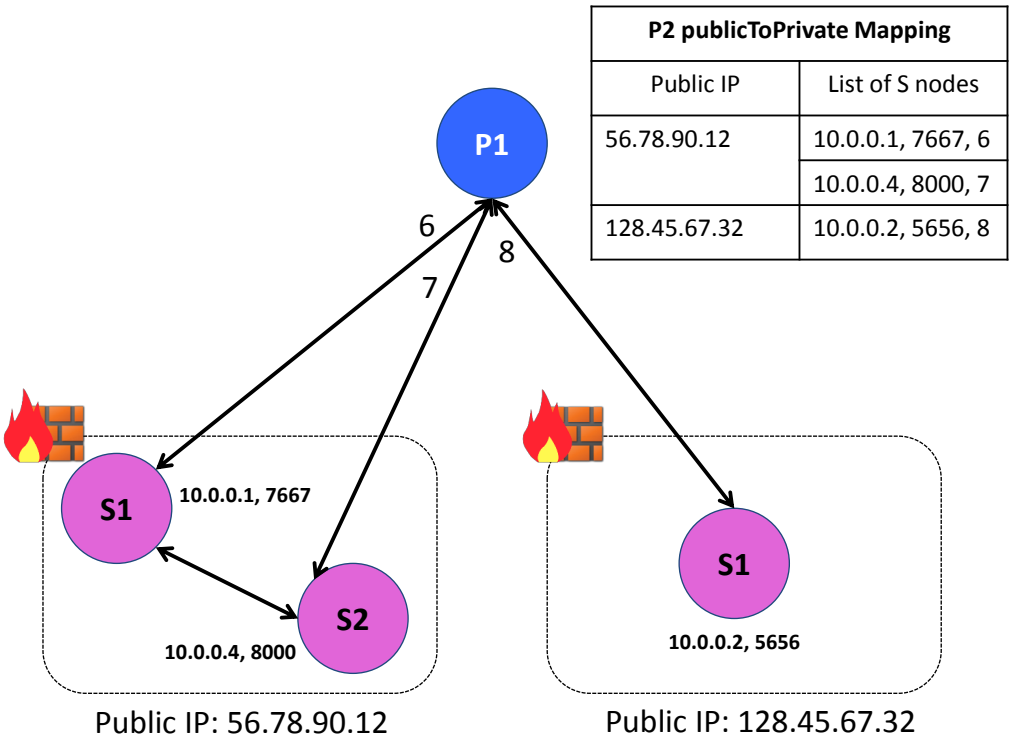

Figure 3.4: Public to Private Map maintained by a given P connected to 3 S nodes, belonging to different private networks.

3.3 Forwarding GDP Messages between Clients

In order to show how messages are routed from one GDP client to another let us take a look at figure 3.3 again. Suppose the GDP clients with IDs 1221 and 3332 wish to route messages to the GDP clients with IDs 4545 and 6776 respectively. The source clients generate GDP packets and place their IDs as well as the destination IDs inside the messages. Figure 3.5 shows the paths taken by the messages to reach the destinations and the entries in the routing table of each router used along the way. Notice that the S2 node doesn't have an entry for endpoint 4545 as the client, hosting it is not present in its private network. For all endpoints not present in an S node's routing table, the S node simply forwards the message to its relay in the hope that the relay will be able to route the message to the correct destination. If the node receives a GDP message from its relay and the destination ID in the message is not in its routing table, it sends a NAK packet back to the relay. From the figure it can be seen that the maximum number of GDP routers taken to route any GDP client message is 2.

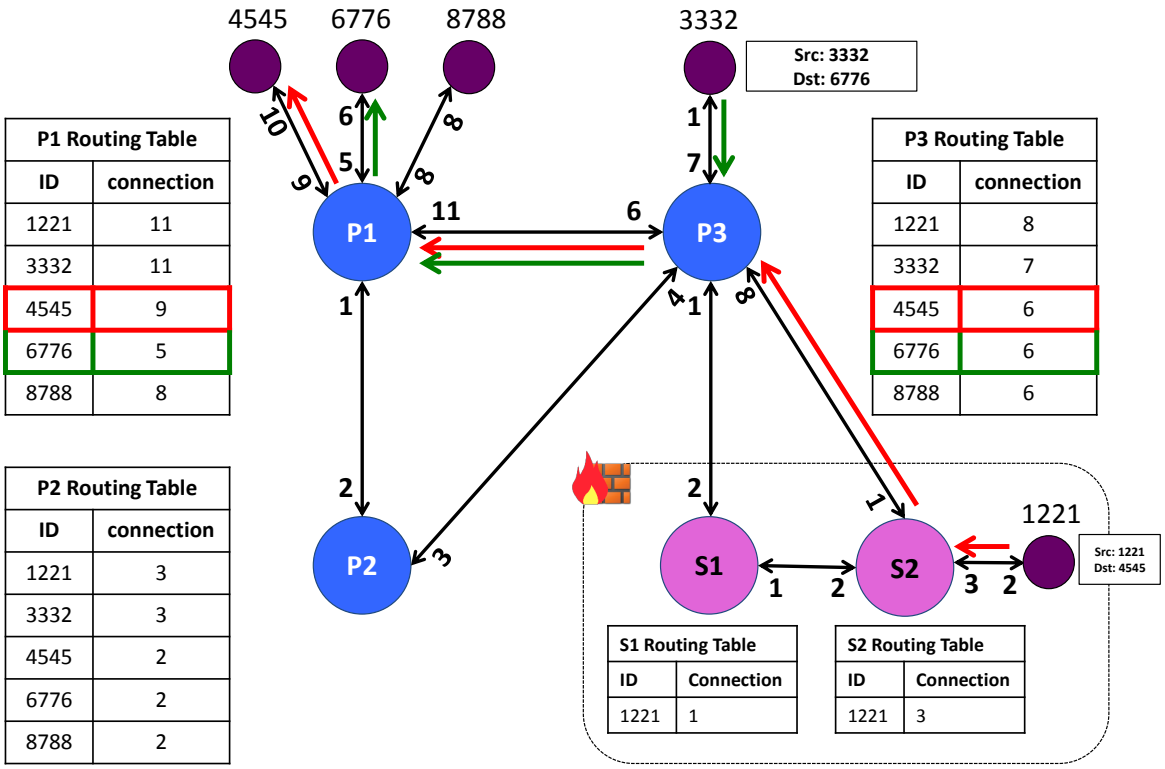

Figure 3.5: Routes taken by two GDP client messages.

4

GDP Routing Maintenance Protocol

This section describes the protocol used by GDP routers to form a distributed hierarchical network. The GDP Routing Maintenance Protocol (GDPRMP) is an application layer protocol that runs on top of TCP. GDP routers use this protocol to pass messages meant for the routing layer. The router nodes together form the routing layer and any message updating the data structures and states maintained by the routers, is a message for the routing layer. These messages mainly include joining/deletion of a new router node or a GDP client and pinging the router nodes. Overall, the protocol addresses the following issues:

1. *Decentralization:* The GDP Routing Maintenance Protocol forms a distributed network of routers. This improves robustness and makes the protocol appropriate for loosely organized GDP swarm applications.

2. *Locality:* The protocol supports joining of non-NATed nodes as well as nodes that reside in a private network behind a NAT or a firewall. This is a crucial property as it allows us to create local subnets behind a NAT-enabled router, in which data from the GDP clients residing in the subnet can be preserved locally.

3. *Latency:* The maximum number of GDP router hops to reach from one GDP client to another is 2. Although this makes the design less scalable, its performance based on network latency will be better than a Distributed Hash Table (DHT) design.

4. *Resilience:* The protocol ensures that the routers automatically adjust their internal states to reflect newly joined router nodes as well as node failures, making this system resilient to network churns as well as dynamic GDP clients frequently leaving and joining the network.

5. *Location Independent Routing:* Our protocol performs location independent routing by forwarding messages between GDP endpoints based on their 256-bit IDs as opposed to their physical locations.

When a given GDP router wants to send a message meant for the routing layer, it encapsulates its message using the GDP Router Maintenance protocol. The message is then encapsulated in a GDP PDU and sent over the connection between the routers. We begin the discussion of the protocol by first specifying the GDPRMP message format in section 4.1. Section 4.2 then depicts the dynamic operations and failure handling performed by the protocol, which includes joining and deletion of GDP clients and routers.

4.1 GDPRMP Message Format

A GDPRMP message consists of a header of 38 bytes length, followed by the data section. The length of data section can be calculated from the header parameters. Figure 4.1 depicts the GDPRMP message format.

<table>
<tr><td colspan="3">Router Command (1 byte)</td></tr>
<tr><td>Source Public IP
(4 bytes)</td><td>Source Private IP
(4 bytes)</td><td>Source Port
(2 bytes)</td></tr>
<tr><td>Destination Public
IP (4 bytes)</td><td>Destination Private
IP (4 bytes)</td><td>Destination Port
(2 bytes)</td></tr>
<tr><td colspan="3">NumClientAdvertisments
(4 bytes)</td></tr>
<tr><td colspan="3">NumSecondaryAdvertisments
(4 bytes)</td></tr>
<tr><td colspan="3">NumSecondary
(4 bytes)</td></tr>
<tr><td colspan="3">NumPrimary
(4 bytes)</td></tr>
<tr><td colspan="3">TypeAssigned
(1 byte)</td></tr>
<tr><td colspan="3">Data</td></tr>
</table>

Figure 4.1: GDPRMP message format

The following is a description of each parameter in the GDPRMP header. Some of these parameters may be optional depending on the type of message sent.

1. *Router Command:* This parameter specifies the message command using 1 byte. Table 4.1 shows the different types of commands exchanged between the router nodes. Significance of each command will be discussed in more detail in the subsequent sections.

2. *Source/Destination public IP:* This parameter is the IP address of the source/destination router that is perceived by the recipient of the message. For a Secondary (S) node, it is the public IP address of the NAT-enabled router behind which, the S node resides. For a Primary (P) node, it is the IP address of the link interface on which the P node sends messages.

3. *Source/Destination private IP:* This parameter is the real IP address of the source /destination router. For an S node it will be its LAN IP address. For a P node, the private IP is the same as the public IP.

4. *Source/Destination port:* This parameter specifies the port number on which a given router node is running. It is this port, along with the node's private IP, that is used to create the TCP server socket in the router in order to listen for new incoming connections and accepting them. For information on how to create a TCP server socket, please refer to section A.1, Appendix A.

5. *numClientAdvertisments:* This parameter is used to specify the number of directly connected endpoints advertised by the source router to the destination router.

Command (value)	**Description**
JOIN (0x8)	Sent by a new router node, willing to joining the GDP network, to an existing P node
JOIN_ACK (0x9)	Acknowledgement, for a JOIN request, sent by an existing P node to a new router
ADD_PRIMARY (0xA)	Sent by a new P node to all other existing P nodes, indicating its presence
ADD_PRIMARY_ACK (0xB)	Acknowledgment received by a new P node from all other existing P nodes
NEW_PRIMARY (0xC)	Sent by a relay node to all directly connected S nodes indicating presence of a new P node
WITHDRAW_PRIMARY (0xD)	Sent by a relay node to all directly connected S nodes indicating a P node left the network
JOIN_SECONDARY (0xE)	Sent by an S node to a P node requesting the P node to serve as the S node's relay
JOIN_SECONDARY_ACK (0xF)	Acknowledgment, for a JOIN_SECONDARY request, sent by a P node to an S node
NEW_SECONDARY (0x11)	Sent by a P node indicating presence of a new directly connected S node
WITHDRAW_SECONDARY (0x12)	Sent by a P node indicating deletion of an existing directly connected S node
ADD_SECONDARY (0x13)	Sent by existing S nodes to a new S node behind the same NAT/firewall
ADD_SECONDARY_ACK (0x14)	Acknowledgments sent by a new S node to all S nodes behind the same NAT/firewall
NEW_CLIENT_PRIMARY (0x15)	Sent by a P node indicating presence of a new directly connected GDP client
NEW_CLIENT_SECONDARY (0x16)	Sent by an S node indicating presence of a new directly connected GDP client
WITHDRAW_CLIENT_PRIMARY (0x17)	Sent by a P node indicating deletion of a directly connected GDP client
WITHDRAW_CLIENT_SECONDARY (0x18)	Sent by an S node indicating deletion of a directly connected GDP client
UPDATE_SECONDARY (0x1A)	Sent by an S node to another P node, when its assigned P relay leaves the network.
UPDATE_SECONDARY_ACK (0x1B)	Acknowledgment, for an UPDATE_SECONDARY request
PING (0x1D)	Used by a router node to keep its connections with other router nodes alive

Table 4.1: GDPRMP commands and descriptions

6. *numSecondaryAdvertisments:* This parameter is used by an S node when it wishes to advertise

its directly connected endpoints to its relay, which is a P node. The P node in turn uses this parameter to inform all other P nodes about the clients connected to the S node to which it serves as a relay.

7. *numPrimary:* This parameter is used by a P node to specify to a new joining router node, the total number of P nodes that exist in the GDP.

8. *TypeAssigned:* When a new router node joins the network, it contacts an existing P node in the network. The existing node determines the source router's type (P or S) based on the source public IP and Private IP parameters and conveys it to the new node using this parameter.

An endpoints ID is 256 bits, which is equivalent to 32 bytes. Also, a router's location is represented by its Public IP, Private IP and port. Thus, the total number of bytes it takes to specify a router's location is 4+4+2 = 10 bytes. Knowing this, the total length of the data section of a given GDPRMP Router Protocol message is calculated as follows:

$$\textit{Length of data section} = \textit{numClientAdvertisments} * 32\textit{bytes} + \textit{numSecondaryAdvertisments} * 32\textit{bytes} + \textit{numPrimary} * 10\textit{bytes}$$

4.2 Dynamic Operations and Failures

The GDP Routing Maintenance Protocol deals with routers as well as client nodes joining the system and nodes those fail or leave voluntarily. This section describes how the protocol handles these situations. Before we begin, we would again mention that each GDP router's physical location is specified by a public IP, a private IP and a port on which the router listens for incoming connections. The private IP is the IP address of the link interface over which the node sends/receives messages. For the case of an S node, the public IP is the address of the NAT router behind which the node resides. For a P node, the private IP and the public IP are the same.

4.2.1 Router Node Join

When a new router node starts, its type (P or S) is unknown. The router is only aware of its Private IP and port on which it is running and listening for incoming connections. In order to join the network, the new node is specified with the address of an existing P node in the network. This node serves as the bootstrap node for the new node. The first question is how does a new node find an existing P node in the network. There are two approaches:

1. The first approach is that the new router can use Avahi/Zeroconf to find the closest P node existing in the network.

2. The second approach is that the new router can use one of the default Berkeley root nodes that we have set up. These root nodes are P nodes as well that have been deployed on powerful servers to ensure 99.99% availability.

The first approach poses an important constraint that Zeroconf should return an existing P node. For GDP clients, the only constraint was that Zeroconf returns any node (P or S) residing within the client's LAN. We thus need to add a filtering mechanism inside Avahi, which returns the appropriate list of entry points based on the category of the new node (router or client). This filtering mechanism has been included as part of the future work for this project. For now, a new router

uses the second approach to join the network.

The new router begins by initiating a TCP connection with an existing P node in the network (Berkeley root nodes in this case, since we are using approach 2). This existing node serves as a bootstrap for the new node. The new node sends a GDPRMP message to its bootstrap with the `JOIN` command. The source public and private IP are made equal to the private IP known by the new node.

It is the responsibility of the bootstrap node to determine the type of the new router node and inform it. The bootstrap obtains the public IP address of the new node from the IP datagram packet in which the message was encapsulated and sent. It next obtains the private IP address of the new node from the `JOIN` GDPRMP message. It then compares these two IP addresses. If the IP addresses are equal, the bootstrap concludes the new node is a P node. If they are not equal, the new node is an S node. The subsequent steps in the join protocol differ based on the type of the new node. Once the new node has successfully joined the network, it must advertise itself upon request to clients running Avahi daemon in its LAN. By doing this, it can be successfully discovered by a new GDP client wanting to join the network.

4.2.1.1 Primary Router Node Join

After the bootstrap node identifies that the new node is a P node, the GDPRMP performs the following steps:

Step 1 – Bootstrap Sends `JOIN_ACK` to new P node: Each P node in the network maintains a list consisting of the physical locations of all P nodes in the network, After the Bootstrap identifies the new node is a P node, it updates this list and sends the new P node a `JOIN_ACK` message. The message contains the following:

1. Type assigned to the new node (P). This type is assigned in *typeAssigned* parameter.
2. A list of the physical locations of all the P nodes existing in the network (public IP, private IP and port). The size of the list is specified in the *numPrimary* parameter.
3. 256-bit IDs of all the GDP endpoints hosted by clients directly connect to the bootstrap. The number of IDs is specified in the *numClients* parameter
4. 256-bit IDs of GDP endpoints hosted by clients connected to the S nodes using the bootstrap as their relay. The number of IDs is specified in the *numSecondaryClients* parameter.

Step 2 – new P node contacts every other P node in the network: The new node on receiving the `JOIN_ACK` realizes that it is a P node. It then obtains the list of all other P nodes in the network and initiates a connection to each of them using the nodes' public IP and port. It also stores the location of all these P nodes in a list.

Over each TCP connections, it sends a GDPRMP message with the `ADD_PRIMARY` command. The message content is similar to the `JOIN` sent earlier. This message is sent to all the P nodes existing in the network in order to make them aware of the new node. The new node next obtains the list of the endpoints IDs directly connected to the bootstrap and the IDs connected to S nodes

using the bootstrap as their relay. It stores these IDs in its routing table, which maps the IDs to the file descriptor representing the connection to the bootstrap node.

Step 3 – Each existing P node sends an ADD_PRIMARY_ACK to the new P node: Upon receiving an ADD_PRIMARY message, each existing P node updates its list of all existing P nodes in the network, by adding the location of the new P node. It then sends the new P node an ADD_PRIMARY_ACK message, which contains the following:

1. A list of the physical locations of all the P nodes existing in the network (public IP, private IP and port). The size of the list is specified in the *numPrimary* parameter
2. 256-bit IDs of all the GDP endpoints hosted by clients directly connect to the P node. The number of IDs is specified in the *numClients* parameter
3. 256-bit IDs of GDP endpoints hosted by clients connected to the S nodes using the P node as their relay. The number of IDs is specified in the *numSecondaryClients* parameter

One might think that sending the list of all P nodes in the network is redundant, as the new P node has already received this list from the bootstrap earlier. However, it must be noted that if two or more P nodes are joining the network at the same time, they might contact different root P nodes in the network. This could result in a race condition, which might lead to a flaw that the new P nodes might remain unaware of each other's presence. This race condition can be avoided if each P node sends the new P node its list of existing P nodes in the network.

Step 4 – The new P node contacts P nodes that weren't received from the bootstrap: The new node on receiving an ADD_PRIMARY_ACK from an existing P node first obtains the list of all other P nodes in the network. It initiates a connection to all the P nodes that were not already present in its list of P nodes and sends them an ADD_PRIMARY message. The new node next obtains the list of the endpoints IDs directly connected to the source P node and the IDs connected to S nodes using the source P node as their relay. It stores these IDs in its routing table, which maps the IDs to the file descriptor representing the connection to the source P node.

Figures 4.2 - 4.6 depict the flow of messages between a new P node and all other P nodes existing in the network, in a typical GDP network snapshot. In each diagram, we also show the routing table entries of the relevant P nodes and the list of other P nodes they maintain. We have set up 3 fully connected P root nodes (represented as R nodes in the diagram). For simplicity of the diagram only, we have not shown the data structures for nodes S3, R1 and R3. Although a new P node initially contacts only one of the root nodes, it eventually connects to all the three root nodes. From the diagrams, it can be seen that the P nodes together with the Berkeley root nodes, form a fully connected mesh network where every P node is aware of every other P node.

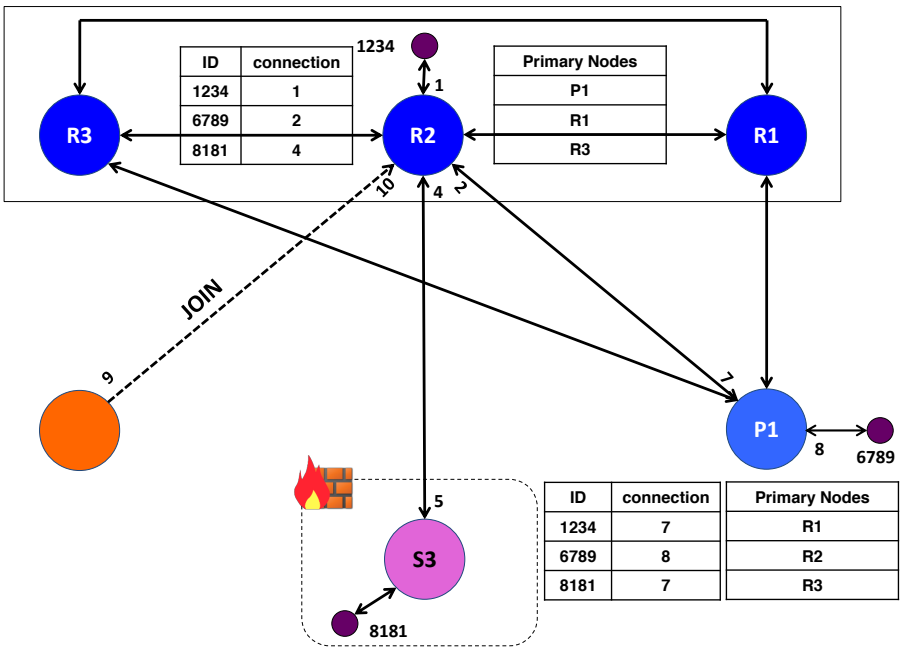

Figure 4.2: *Primary Node Join.* A new node initiates a TCP connection with a root node (R2) and sends the node a `JOIN` message

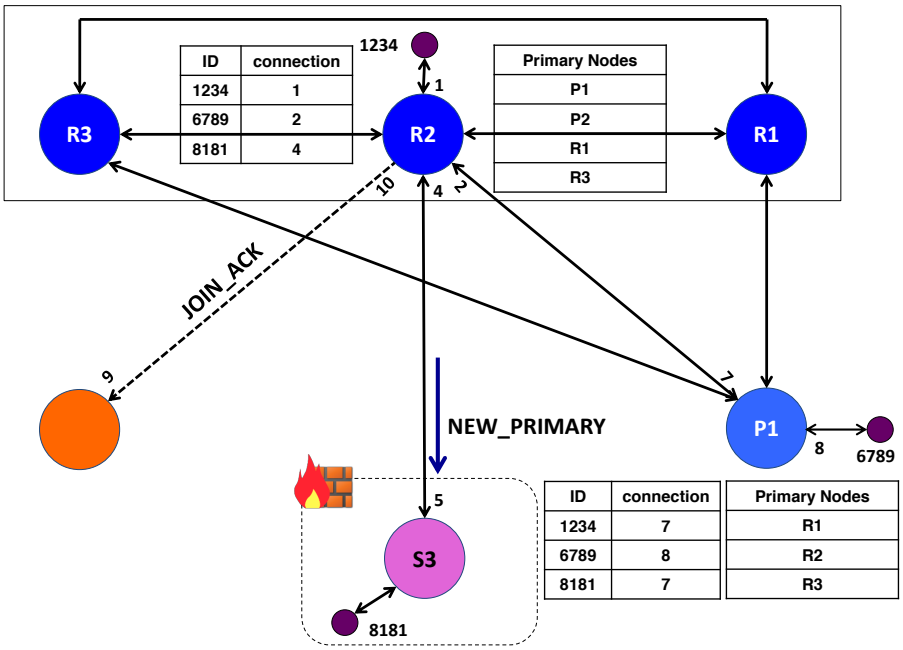

Figure 4.3: *Primary Node Join.* The root node R2 determines the new node is a P node. It places the node in its list of P nodes and sends a `JOIN_ACK` to the new node containing the endpoint IDs of all directly connected clients, clients connected to S3 and the list of all P nodes (step 1). In order to make sure S3, has the most up to date list of locations of all P nodes, R2 sends S3 a `NEW_PRIMARY` message containing location of new node.

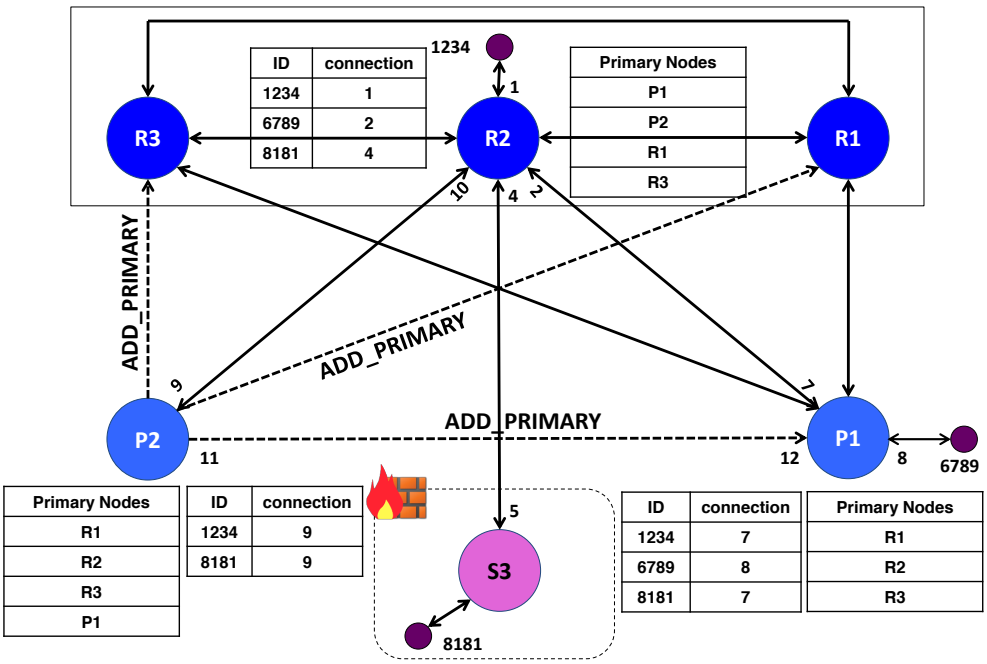

Figure 4.4: *Primary Node Join.* The new P node updates its list of P nodes and routing table. It then contacts each P node by sending an `ADD_PRIMARY` message (step 2).

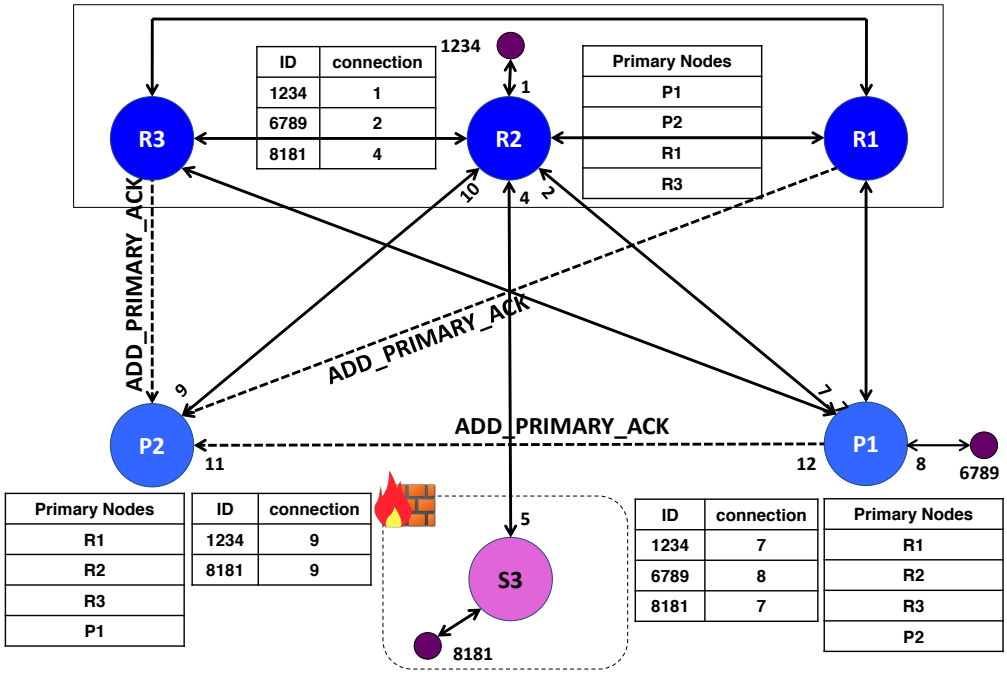

Figure 4.5: *Primary Node Join.* Each P node updates its list of P nodes and sends the new node an `ADD_PRIMARY_ACK` containing a list of directly connected endpoints and its list of P nodes (step 3)

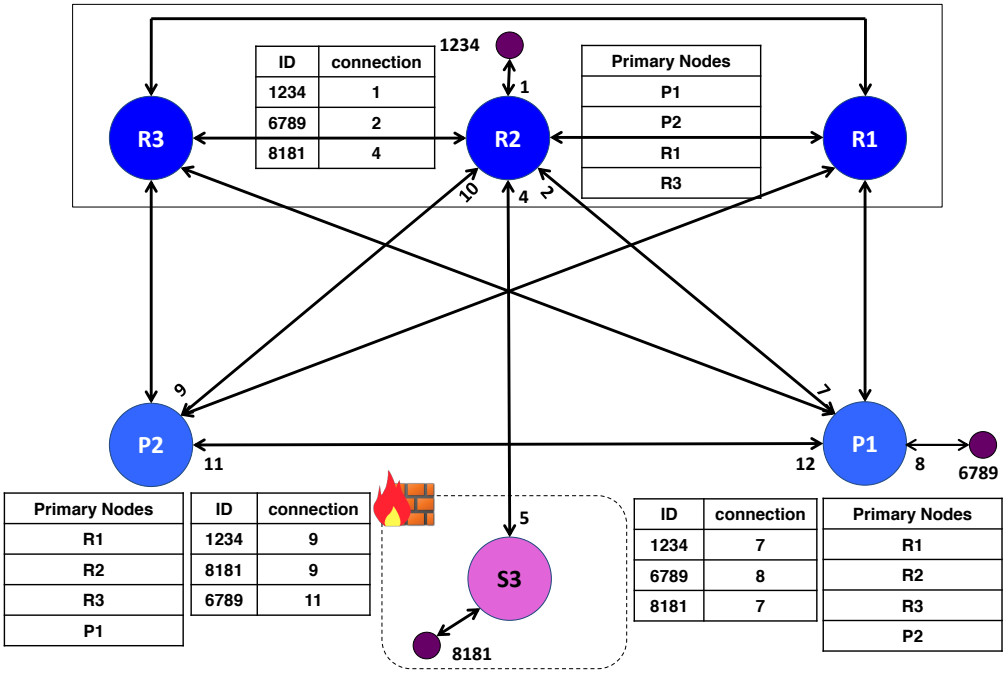

Figure 4.6: *Primary Node Join.* All the P nodes including the root nodes form a fully connected network

4.2.1.2 Secondary Router Node Join

After the bootstrap node identifies that the new node is an S node, the GDPRMP performs the following steps.

Step 1 - Bootstrap Sends `JOIN_ACK` to new S node: The bootstrap sends a `JOIN_ACK` message back to the S node. The contents of the message are as follows:

1. Type assigned to the new node (P). This type is assigned in *typeAssigned* parameter.
2. In the destination public IP, the bootstrap places the public IP address of the S node that it found earlier.
3. A list of the physical locations of all the P nodes existing in the network (public IP, private IP and port). The size of the list is specified in the *numPrimary* parameter.

After sending the `JOIN_ACK`, the bootstrap closes its side of the connection.

Step 2 – The new S node updates its public IP, upon receiving the `JOIN_ACK` message: The new node checks the *typeAssigned* parameter and realizes it is an S node. It then closes its TCP connection with its bootstrap (by calling `close()` system call). It obtains the public IP of its NAT-enabled router using the destination public IP parameter in the `JOIN_ACK` message. It uses this parameter as its own public IP for all subsequent communications with the other routers.

Step 3 – The new S node choses its relay: The new S node tries to find a suitable relay which can serve as its window to the other local subnets and P nodes. It obtains the list of P nodes from

the JOIN_ACK message. It then issues an ICMP ping message to each P node in the list, using the P node's public IP. It obtains the ICMP ping response from each P node and records the total round trip time taken to ping the node. The S node also has a certain timeout interval before which it should obtain the ping responses of each P node. P nodes, whose ping responses arrive after the timeout are ignored from the relay selection criteria. Based on all the round trip latencies, it choses the P node having the smallest latency as its relay. Thus, this technique preserves some degree of locality, by ensuring that an S node choses a P node, which is located closest to it amongst all the other P nodes, as its relay.

All the other P nodes locations are kept in a backup list in case the chosen P node relay fails and the S node has to choose a new proxy. It is important to mention that in order to keep this backup list up to date, whenever a new P node joins the network and contacts all the P nodes, each existing P node in the network sends a NEW_PRIMARY command to all its directly connected S nodes containing the physical location of the new P node. The S nodes then update their backup list by adding the new P node.

Step 4 – The new S node contacts its chosen relay: The new S node initiates a TCP connection with its chosen P relay. If the connection is established, it sends the relay a message with the JOIN_SECONDARY command. This command is used to inform the P node that a new S node wants the P node to serve as a relay to the new node. The most important parameters in this message are the source public IP, private IP and port.

Step 5 – The relay P node informs all other P nodes about the new S node: When a P node receives a JOIN_SECONDARY message, its first task is to inform all other P nodes about the new S node. Its sends a GDPRMP message with the NEW_SECONDARY command to each of the other P nodes in the network. The message mainly contains the public IP, private IP and port of the new S node.

Step 6 – The chosen P relay sends the new node a JOIN_SECONDARY_ACK: The chosen P node relay sends the new S node a message with the JOIN_SECONDARY_ACK command. This is to acknowledge the new S node's request for a relay. The message also contains the list of all the P nodes in the network. This might seem redundant since the S node already received this list from its bootstrap. However during the time period from the point when the S node received the list from the bootstrap to the point when it sent a JOIN_SECONDARY command to its chosen relay, the number of P nodes in the network might have changed. For example a new P node could have joined during this time frame or an existing P node could have failed. During this time, the new S node is not connected to any P node that can inform it about the present state of the global network of P nodes. Thus it is the responsibility of the chosen proxy to update the S node of all the changes in the number of P nodes. The simplest way of doing this is to send the new S node its list of P nodes and the new S node upon receiving the list can update its own backup list of P nodes.

Step 7 – Each P node informs about new S node to all directly connected S nodes, having the same public IP as new S node: When the chosen P relay receives a JOIN_SECONDARY message from the new S node, it searches its Public to Private map to determine if it has S nodes, belonging to the same subnet as the new S node, directly connected to it. It uses the new S node's public IP (present inside the JOIN_SECONDARY source public IP parameter) as the key for the hash map. If

it does have directly connected S nodes, having the same public IP as the new S node, the P node sends each of the S nodes a NEW_SECONDARY message containing the public IP, private IP and port of the new S node. The relay then updates its Public to Private map by adding the location of the new S node.

All other P nodes perform the above task when they receive a NEW_SECONDARY message from the chosen P relay node. They forward the NEW_SECONDARY message to all directly connected S nodes, having the same public IP as the new S node.

Step 8 – Existing S nodes having the same public IP as new S node, contact the new node: When an S node receives a NEW_SECONDARY message from its P relay node, it initiates a connection with the new S node using the new S node's private IP and port. This is possible as the GDPRMP protocol has ensured that the S node receiving the NEW_SECONDARY message and the new S node are both part of the same private network and hence behind the same NAT/firewall. Once the connection is successfully established, the S node sends the new node an ADD_SECONDARY message. This message contains the following:

1. Physical location of the S node (source public IP, source Private IP, source port)
2. 256-bit IDs of all the GDP endpoints hosted by clients directly connected to the S node. The number of IDs is specified in the *numClients* parameter

Thus, this step along with the last two previous steps ensures that all S nodes behind the same NAT/firewall form a fully connected mesh network. The GDPRMP protocol also allows that every S node in this mesh network can chose a different P relay node.

Step 9 – The new S node updates its routing table , upon receiving an ADD_SECONDARY message: The new S node obtains the list of the endpoints IDs directly connected to the source S node. It stores these IDs in its routing table, which maps the IDs to the file descriptor representing the connection to the source S node.

Figures 4.7 – 4.15 depict the above steps for a new S node (S2) to join, in a typical GDP network snapshot. It should be noted that in the end, the new S node's routing table only has endpoints IDs hosted by clients within the same private network as the node. Also different S nodes within the same private subnet may be assigned different P relays.

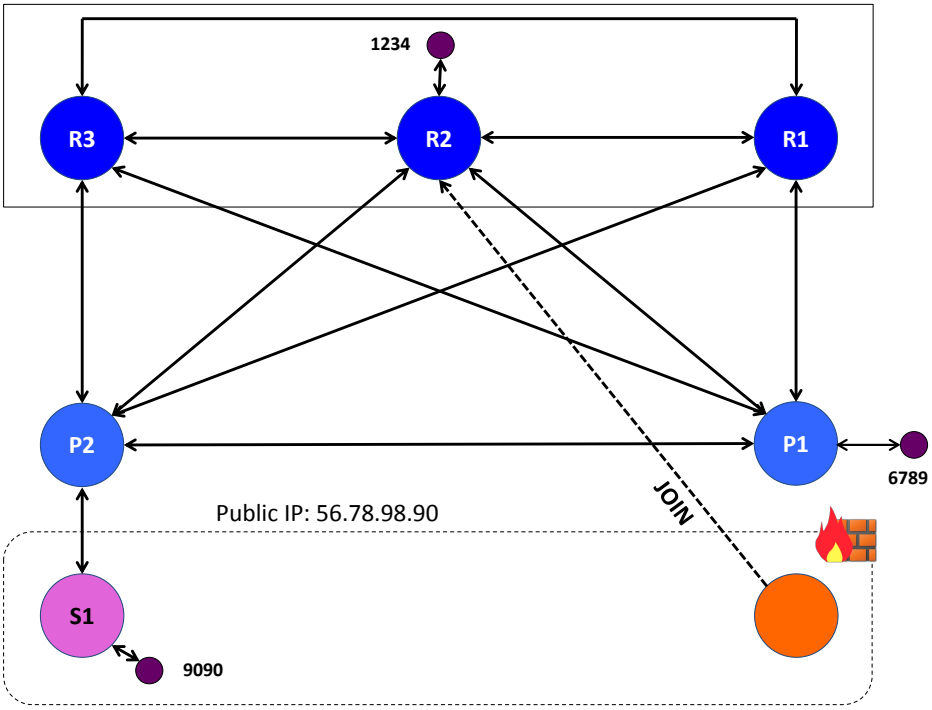

Figure 4.7: *Secondary Node Join.* A new node from a private network contacts a root node

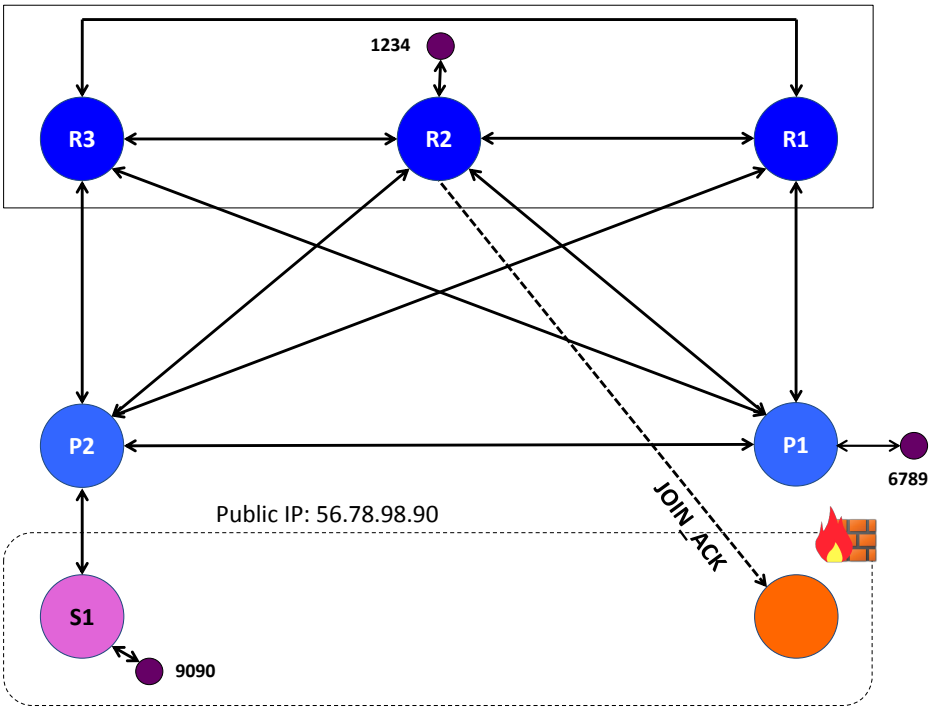

Figure 4.8: *Secondary Node Join.* The root node determines that the new node is an S node. It sends a `JOIN_ACK` message back to the new node containing list of other P nodes (Step 1)

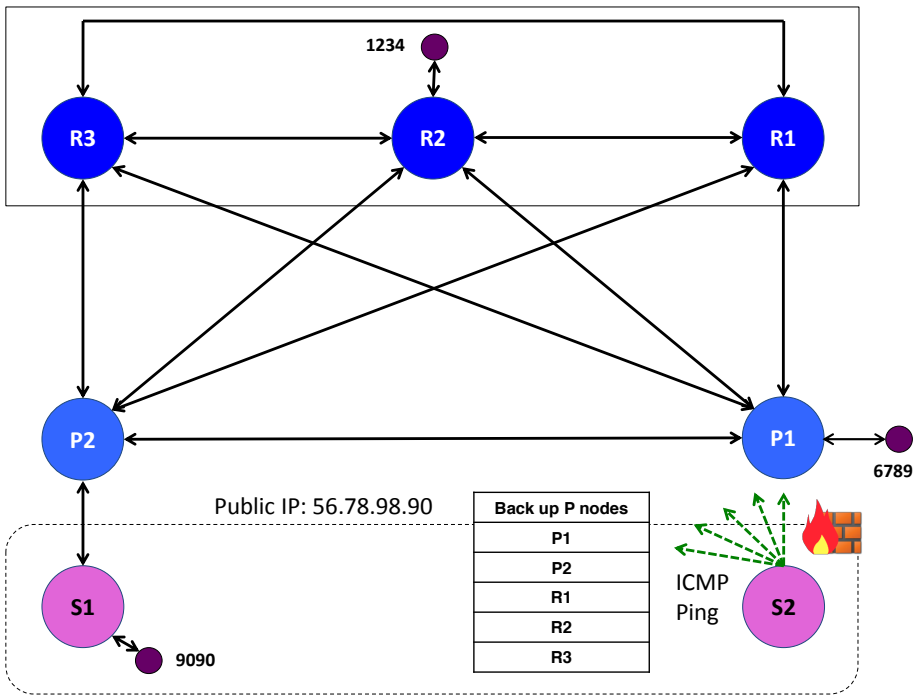

Figure 4.9: *Secondary Node Join.* The new S node updates its public IP and creates a list of all P nodes in the network. It then choses a relay P node by pinging all the P nodes in the network and choosing the node with the smallest round trip time of ping echo and response. In this case, the node choses P1 as its relay (Step 3)

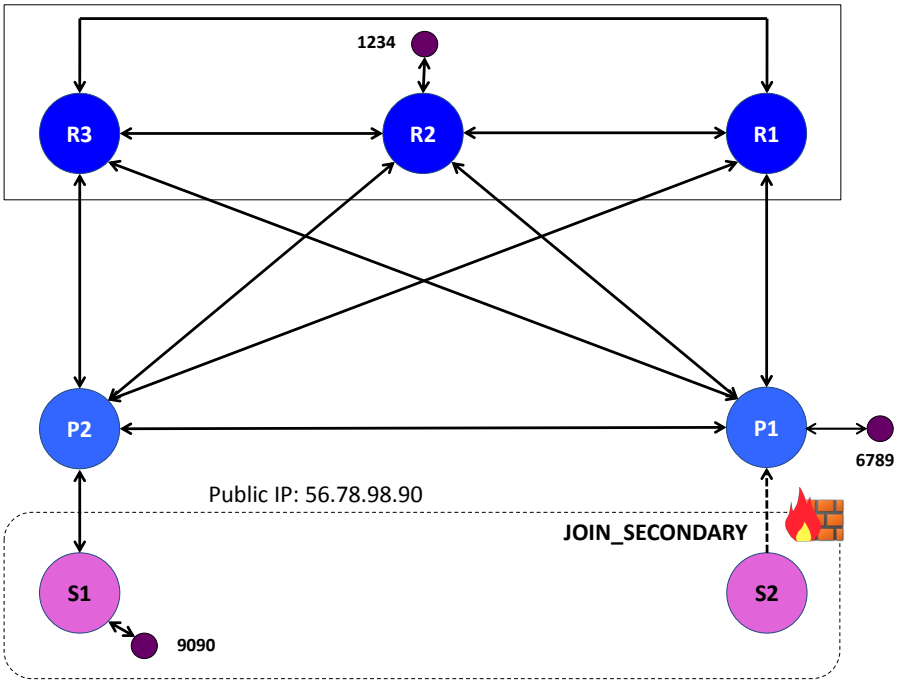

Figure 4.10: *Secondary Node Join.* The new S node sends P1 a `JOIN_SECONDARY` command requesting the P node to serve as relay (Step 4)

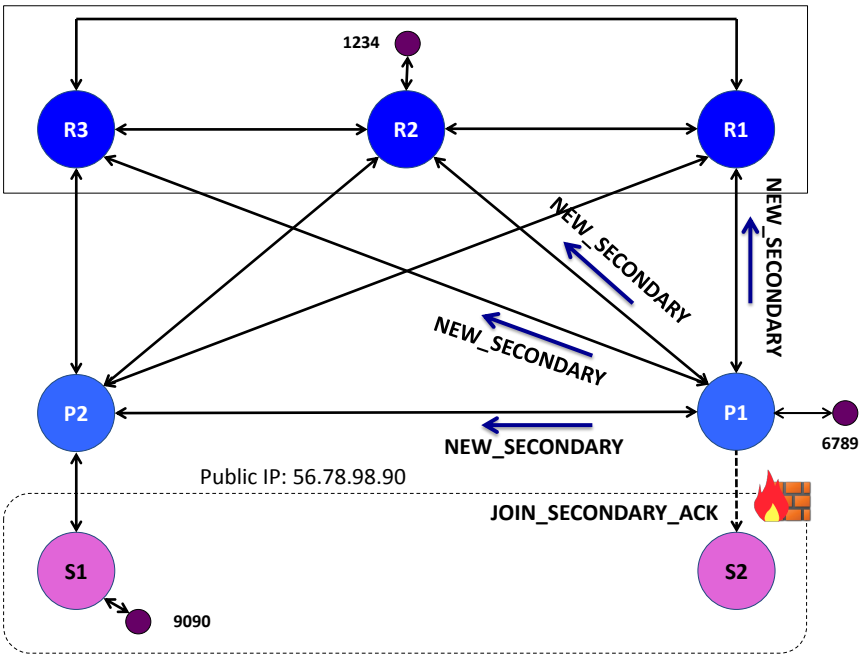

Figure 4.11: *Secondary Node Join.* The chosen relay informs all other P nodes of the new S node's presence by sending a `NEW_SECONDARY` message. It also acknowledges the new S node by sending a `JOIN_SECONDARY_ACK` message containing the most recent version of the list of all P nodes in the network (in this case the list doesn't change from before). (Steps 5 and 6)

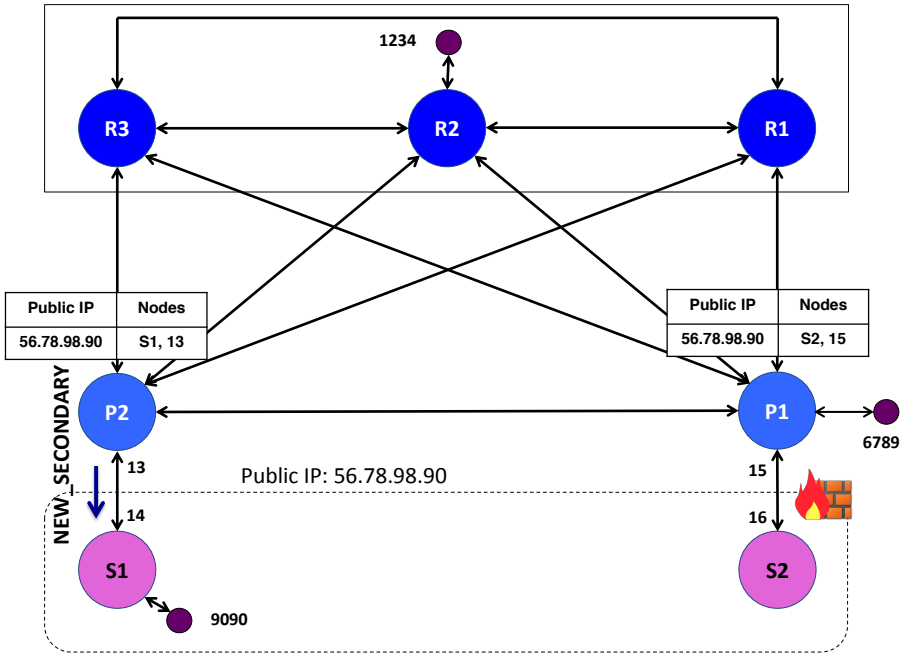

Figure 4.12: *Secondary Node Join.* The chosen P node updates its Public to Private map by storing the location of the new S node and the file descriptor representing the connection to the new S node. Meanwhile P2 node determines that it is a relay to another node in the same private network as the new S node. It sends the node a `NEW_SECONDARY` command (Step 7)

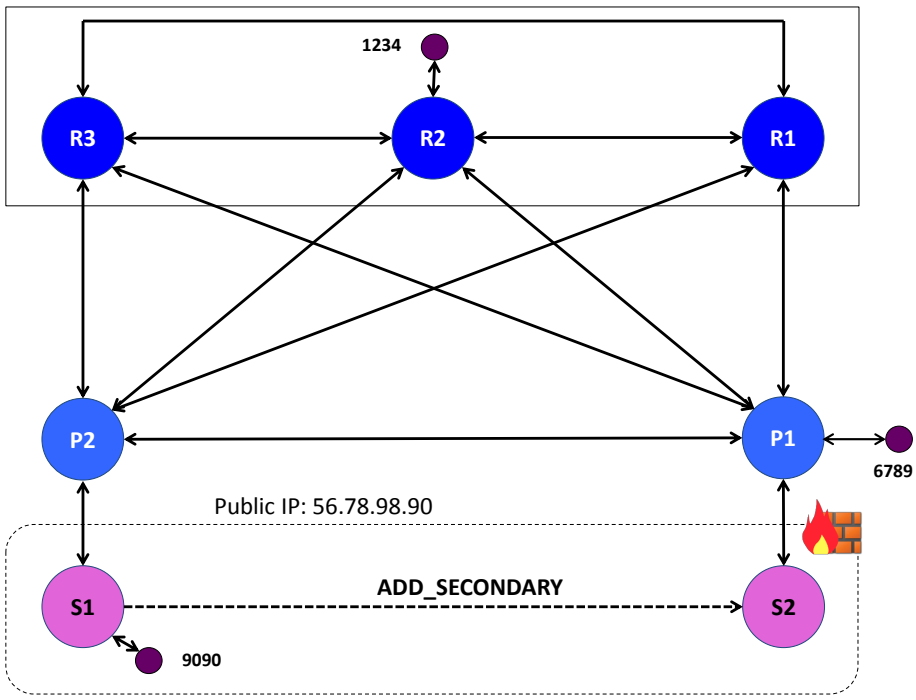

Figure 4.13: *Secondary Node Join.* Upon receiving a `NEW_SECONDARY` command, S1 initiates a TCP connection to the new S node and sends an `ADD_SECONDARY` command to the new S node containing its list of directly connected endpoints. S2 stores the ID 9090 in its routing table mapped to the file descriptor, representing the connection to S1 (step 8).

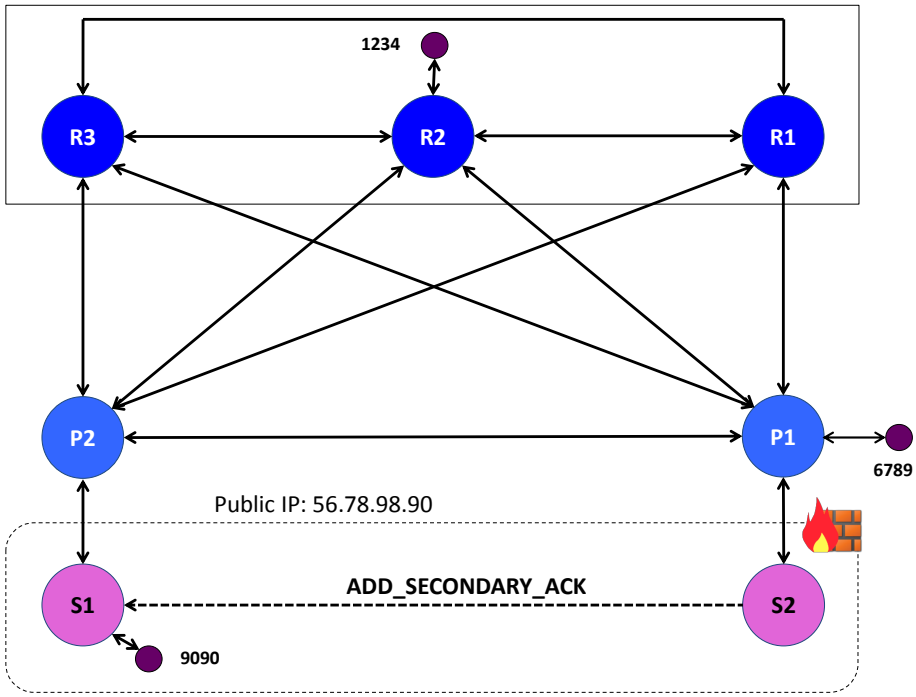

Figure 4.14: *Secondary Node Join.* The new S node acknowledges the S1 node's `ADD_SECONDARY` request (Step 9)

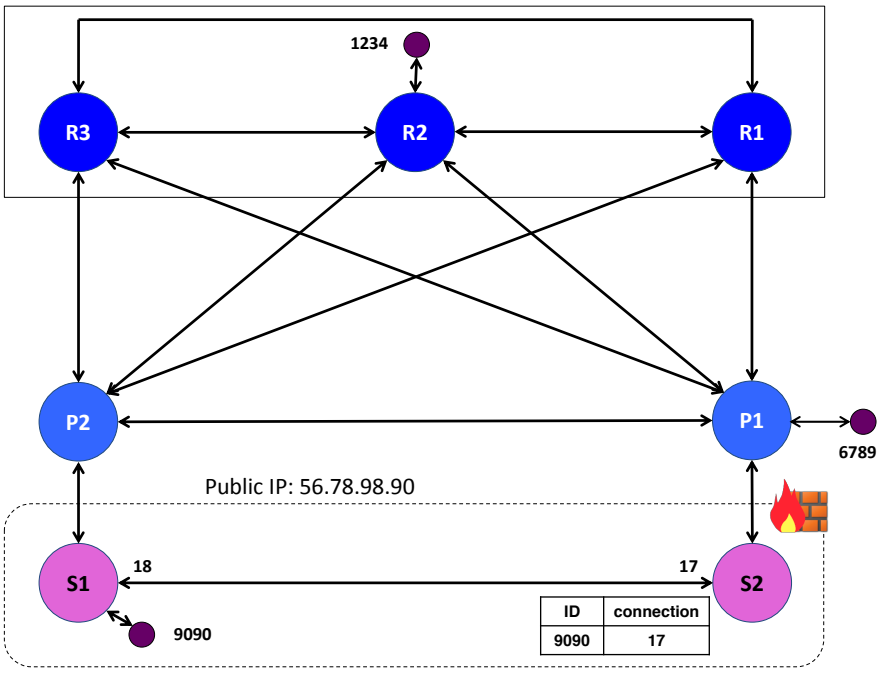

Figure 4.15: *Secondary Node Join.* S nodes within the same private subnet form a fully connected network. The new S node's routing table only has the endpoint IDs of clients within the same private network.

4.2.2 GDP Client Node Join

This section provides the steps taken by GDPRMP when a new client node establishes a connection with a router node and advertises its 256-bit hosted endpoint IDs. Before we proceed, let us first take a look at how a given client locates a router node. The following are the steps taken by a new GDP client to enter the GDP network and subsequently interact with other existing clients.

1. The new client discovers a GDP router node using Avahi
2. The client then initiates a TCP connection to the router node
3. After the connection has been established, the client sends an advertisement to the router specifying its 256-bit hosted endpoint names. This is done by sending a GDP PDU with the `CMD_ADVERTISE` command. The PDU will contain the the IDs in the data section

GDP routers advertise themselves as a special service, which new clients can automatically discover. As part of Step 1, a new client contacts Avahi to discover a router node (P or S) in its local area network. Avahi will locate all the GDP router nodes running in the client's local area network (using an mDNS multicast query) and send their locations (IP address and port) to the client in the form of a list. The client will contact each node in the list until it has successfully established a TCP connection with one of the routers. As mentioned earlier, we have also set up three Primary router nodes in Berkeley. These nodes serve as default root nodes, which a client can contact when there is no GDP router node running in the client's local area network. The IP address and port on which these nodes are running are fixed and specified to a client when the client installs the GDP library. Once a client choses a router node, it advertises its 256-bit endpoint IDs to the router. We now describe the protocol the contacted router node follows to convey the information about the

new client to other GDP routers based on its type (P or S).

4.2.2.1 New GDP client connects to a P node

Step 1 – the P node informs all other P nodes of the new GDP client: When a new client initiates a connection to a P node and advertises its 256-bit endpoint IDs, the P node updates its routing table by adding an entry for each ID. After that, the main responsibility of the router node is to forward the IDs to all other P nodes in the network. The P node also conveys to the other P nodes that any GDP protocol message with destination ID equal to any of the endpoint IDs should be forwarded to the P node connected directly to the client. This is done by sending a GDPRMP message with the `NEW_CLIENT_PRIMARY` command. The message contains all the endpoint IDs and sets the *numClients* parameter accordingly. Upon receiving a `NEW_CLIENT_PRIMARY` command, each P node updates its routing table by mapping the IDs to the file descriptor representing the connection to the source P node.

Figure 4.16 – 4.18 depict the steps performed by GDPRMP when a new client connects to a P node. The new client will have a corresponding entry in the routing tables of all P nodes. The S nodes won't be aware about this new client.

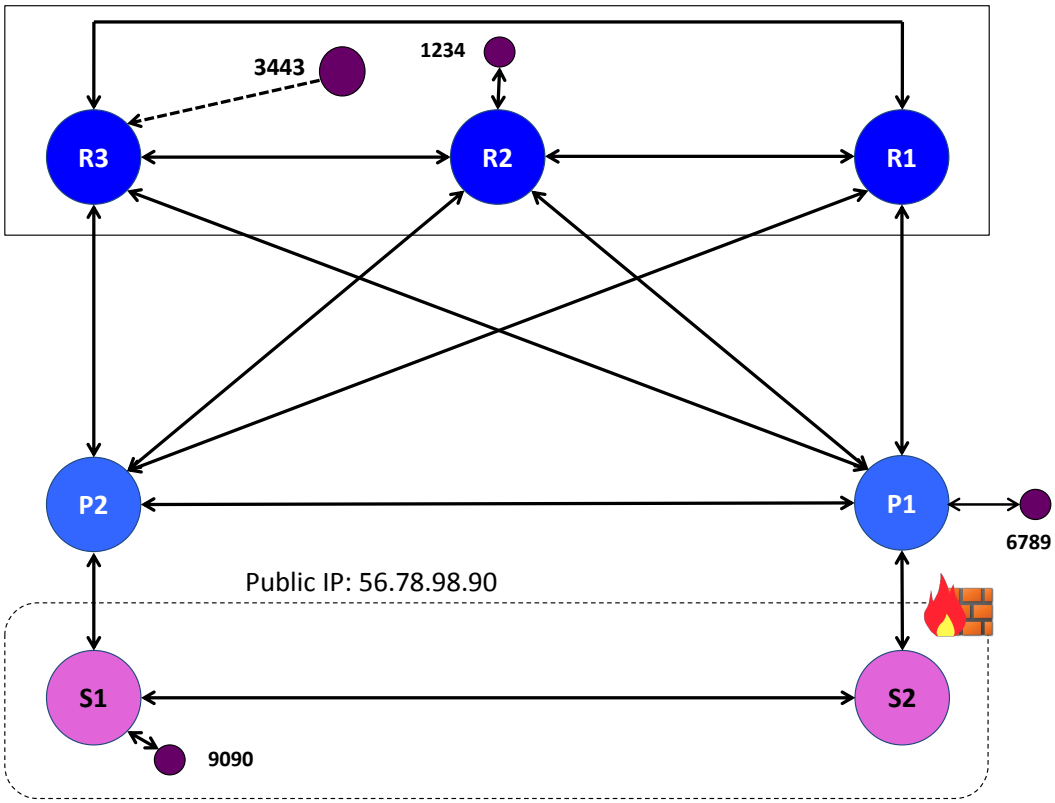

Figure 4.16: *Client connecting to a P node.* A client with destination ID 3443 initiates a connection with a P node (R3) and advertises its ID. The P node updates its routing table

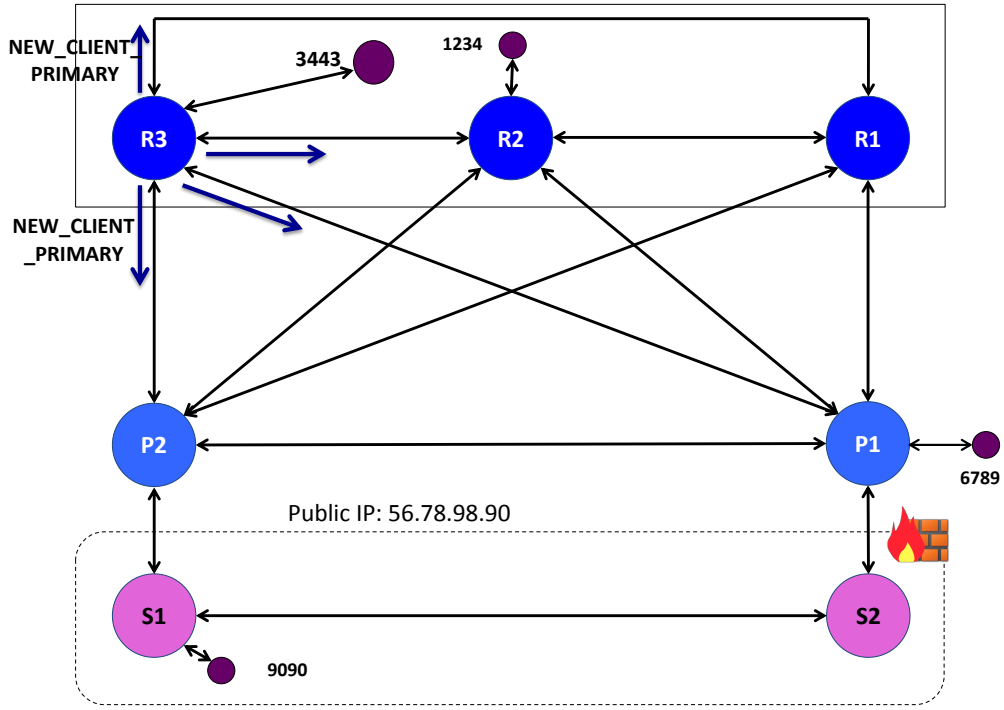

Figure 4.17: *Client connecting to a P node.* The P node sends a `NEW_CLIENT_PRIMARY` command to all other P nodes (Step 1). Each P node on receiving this command adds an entry into its routing table for the new endpoint

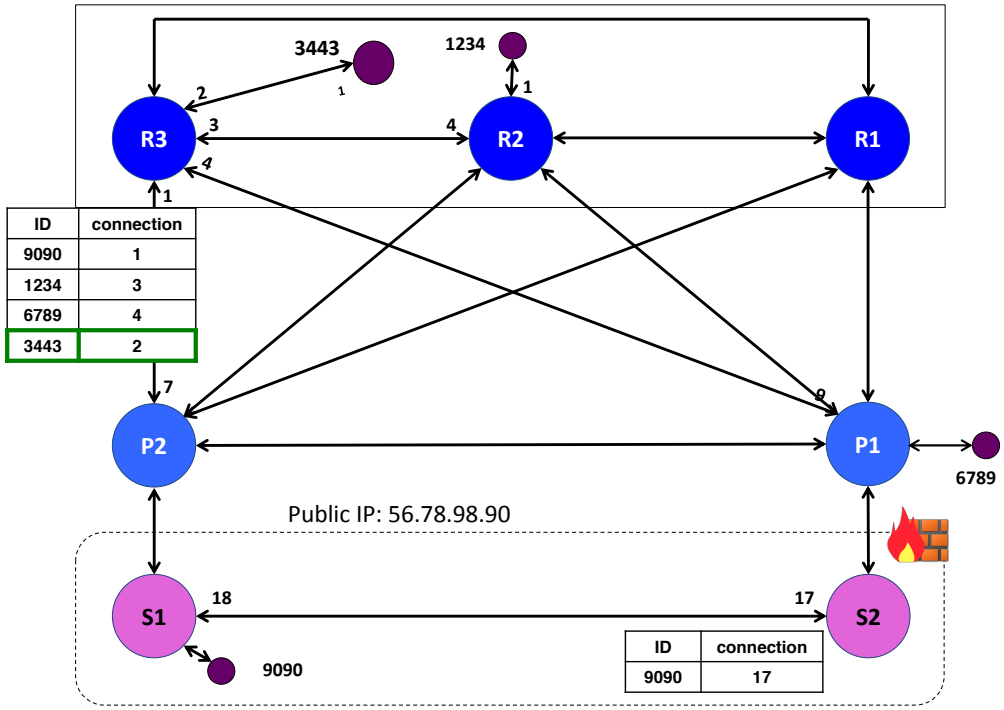

Figure 4.18: *Client connecting to a P node.* Routing Tables of R3 and S2 are demonstrated. S2 only has endpoints in its own private network, whereas R3 has all the GDP endpoints.

4.2.2.2 New GDP client connects to an S node

Step 1 – The S node informs all other S nodes in the same local subnet of the new client: When a new client initiates a connection to a S node and advertises its 256-bit endpoint IDs, the main responsibility of the S node is to forward the IDs to all other S nodes in its private network. This is done by sending a GDPRMP message with the `NEW_CLIENT_SECONDARY` command. The message contains all the endpoint IDs and set the *numClients* parameter accordingly. Upon receiving a `NEW_CLIENT_SECONDARY` command, each S node updates its routing table by mapping the IDs to the file descriptor representing the connection to the source S node.

Step 2 – The S node sends the endpoint IDs to its P relay node: The S node also sends the `NEW_CLIENT_SECONDARY` message to its P relay node. It expects its relay to inform all the other P nodes about the new client. The relay node on receiving the message, updates its routing table by mapping the IDs to the file descriptor representing the connection to the source S node.

Step 3 – The P relay node informs all other P nodes of the new client: The relay P node forwards the `NEW_CLIENT_SECONDARY` message, received from an S node, to all other P nodes in the network. Upon receiving a `NEW_CLIENT_SECONDARY` command, each P node updates its routing table by mapping the IDs to the file descriptor representing the connection to the source P node.

Figures 4.19 – 4.22 depict the above steps taken by a GDP client connecting to an S node. The important point here is that the endpoints hosted by this client are present in the routing tables of all P nodes and all S nodes within the same private network as the client.

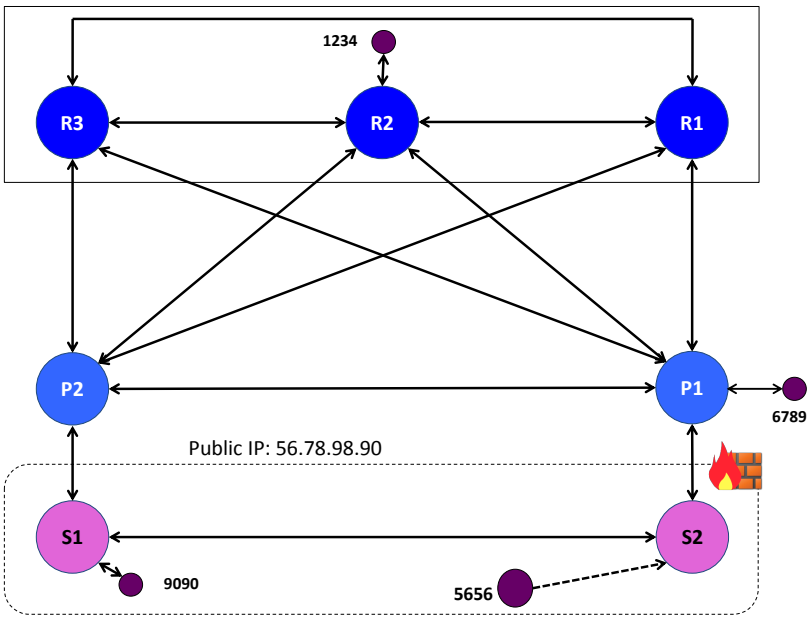

Figure 4.19: *Client connecting to an S node.* A client with destination ID 5656 initiates a connection with an S node and advertises its ID.

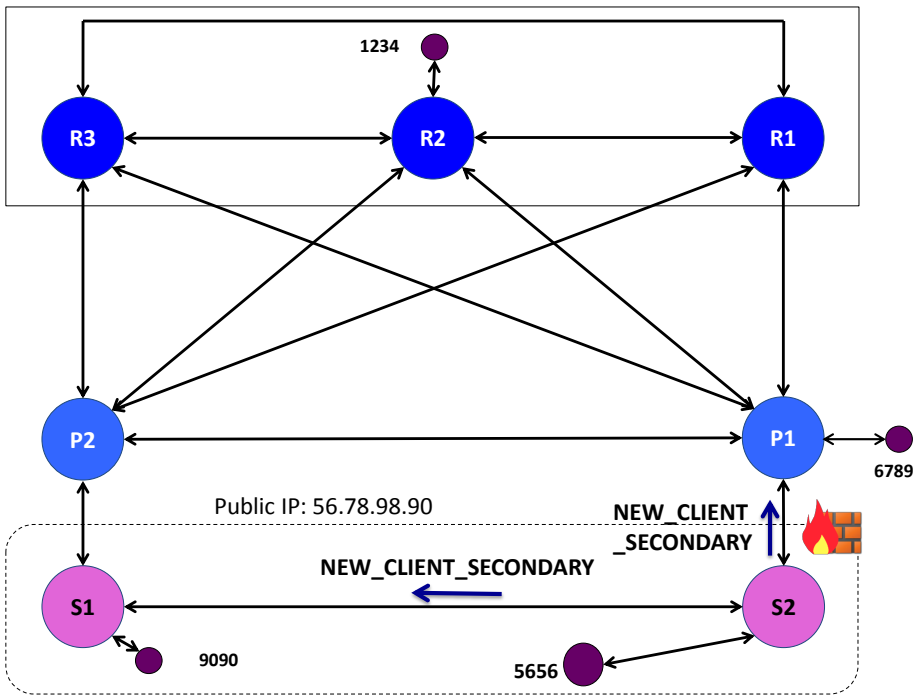

Figure 4.20: *Client connecting to an S node.* The S node sends a `NEW_CLIENT_SECONDARY` command to its relay as well as all other S nodes in the same private network (Steps 1 and 2). The nodes receiving the command update their routing tables.

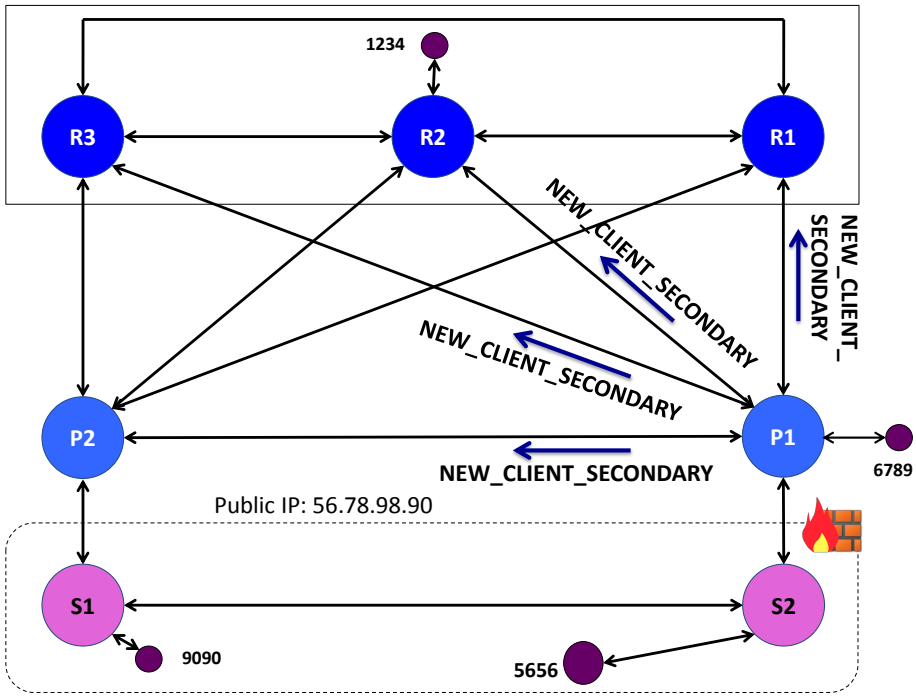

Figure 4.21: *Client connecting to an S node.* The relay forwards the `NEW_CLIENT_SECONDARY` command to all other P nodes (Step 3). The P nodes on receiving this command update their routing tables

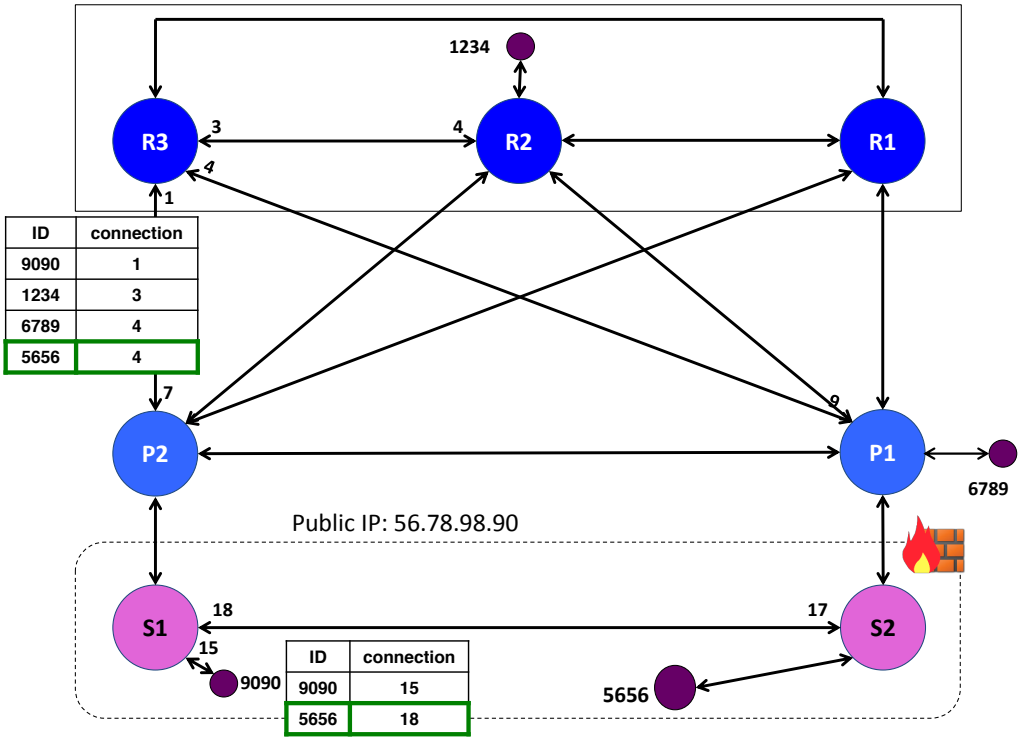

Figure 4.22: *Client connecting to an S node.* The final network. Routing tables of S1 and R3 are shown. S1 has a new entry for the endpoint as it is part of the same private network. R3 keeps track of all GDP endpoints, regardless of their locations.

4.2.3 Node Failure

Since all the clients and routers are interconnected using TCP connections, if one node fails, all nodes directly connected to the failed node are informed about the failure due to TCP's in built mechanism of handling end point failures. For information about this mechanism, please refer to section A.1, Appendix A. It is the responsibility of the GDPRMP to update all directly connected node's states when a failure is detected.

When a client node fails, the failure is detected by the GDP router to which the failed client was previously connected. The protocol that follows is exactly the same as when a new client joins the network except that routers should be told to remove the corresponding endpoint IDs from their routing tables as opposed to adding new entries. Instead of sending `NEW_CLIENT_PRIMARY` or `NEW_CLIENT_SECONDARY`, the router nodes send equivalent client removal commands, `WITHDRAW_CLIENT_PRIMARY` and `WITHDRAW_ CLIENT_SECONDARY`. On receiving these commands, the router nodes update their routing tables by removing the entries corresponding to the failed endpoint IDs.

If a router node fails, the protocol is more intricate. In such a case the GDP clients directly connected to the failed router are notified of the failure and they have to repeat the process of finding another entry point into the network (using Zeroconf or the default Berkeley root nodes). We now discuss the GDPRMP protocol followed by the router nodes that were directly connected to the failed router.

4.2.3.1 Primary Node Failure

When a P node fails, the following nodes detect the node failure:

1. All other P nodes (including the Berkeley root nodes) in the network. Since the P nodes form a fully connected mesh network, all the other P nodes were directly connected to the failed P node.
2. All the S nodes previously treating the failed P node as their relay. These nodes may or may not be part of the same local network.

All other P nodes acknowledge this failure by removing all endpoint IDs from their routing table that mapped to the file descriptor, representing the connection to the failed P node. From our join protocol, it can be seen that these entries include endpoints directly connected to the failed P node as well as endpoints connected to the S nodes that used the failed P node as their relay.

It is the responsibility of the directly connected S nodes to find a new relay amongst all the available P nodes. The following steps are followed by GDPRMP to find a new relay.

Step 1 – Each directly connected S node choses a new relay: Each S node possesses a list of all the P nodes in the network that were provided to it earlier by the failed node during the join process. Each directly connected S node performs a pinging process similar to the one in the join process to find the closest P node. It then contacts this node using an `UPDATE_SECONDARY` command. This message mainly includes the following:

1. The physical location of the S node (public IP, private IP and port)
2. List of IDs of all endpoints hosted by clients directly connected to the S node.

Step 2 – The new relay updates its states and informs other P nodes of the updation: Upon receiving an `UPDATE_SECONDARY`, the new P relay performs 3 tasks:

1. It first obtains the list of all endpoint IDs, directly connected to the source S node. It updates its routing table by adding a new entry for each ID. The entry maps the ID to the file descriptor representing the connection to the source S node.
2. It then updates its Public to Private map by adding the source S node's location.
3. It then forwards the `UPDATE_SECONDARY` message to all other P nodes in the network.

Step 3 – Each P node updates its routing table: When a P node receives an `UPDATE_SECONDARY` message from another P node, it obtains the list of endpoint IDs inside the message and adds an entry for each ID in its routing table. The entry maps the ID to the file descriptor representing the connection to the source P node.

Step 4 – The new relay sends the source S node an Acknowledgment: The new chosen relay also sends an `UPDATE_SECONDARY_ACK` message back to the source S node. This message contains the list of all P nodes in the network. The purpose of this message is similar to the

JOIN_SECONDARY_ACK message. The new relay P node uses this message to give the source S node, the most up to date list of the physical locations of all the P nodes in the network.

Step 5 – Each S node (directly connected to the failed node previously) updates its list of backup P nodes: Upon receiving an UPDATE_SECONDARY_ACK message, each S node updates its backup list of P nodes.

Figures 4.23 – 4.28 depict the steps performed by GDPRMP when a P node fails/leaves the network. Throughout these steps, it can be seen all other connected S nodes within the same private network are unaffected, as they had chosen their relays independently making the design flexible and self-organizing.

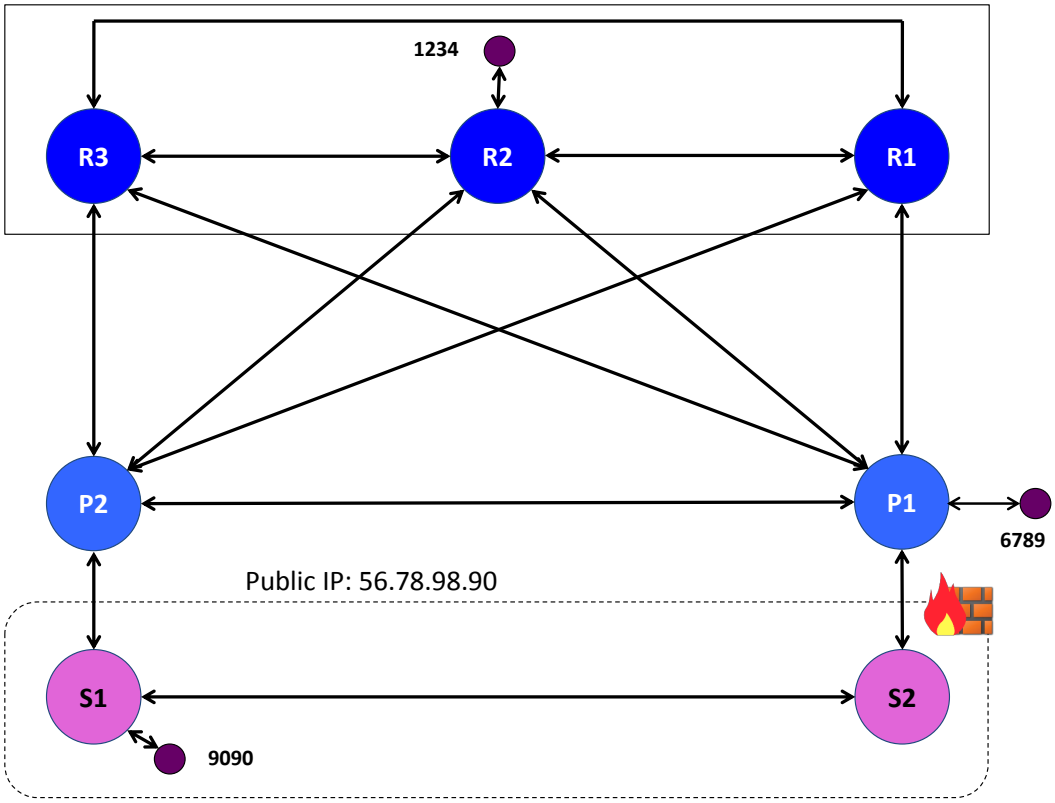

Figure 4.23: *P Node Deletion.* Initially when all P nodes are running

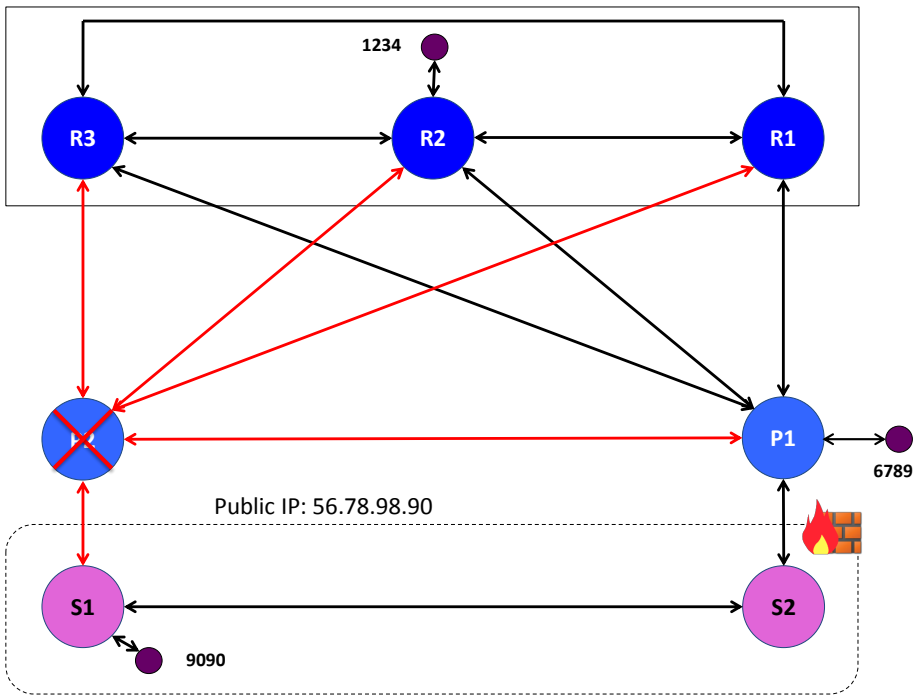

Figure 4.24: *P Node Deletion.* For some reason P2 fails. All nodes directly connected to P2 are informed of the failure due to TCP's in built failure detection mechanism (RST packets)

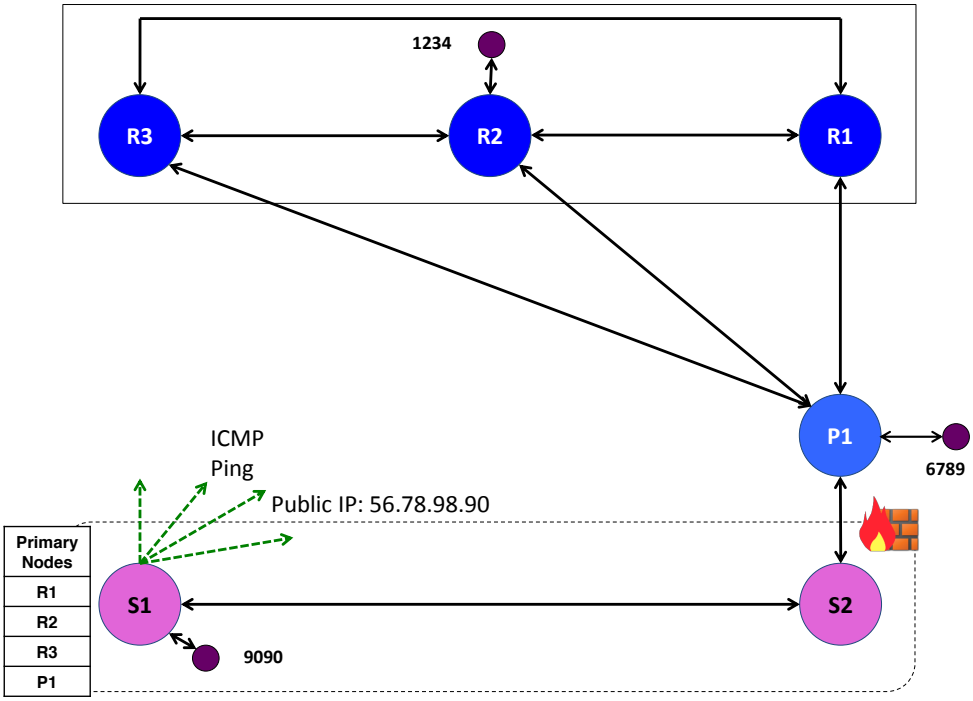

Figure 4.25: *P Node Deletion.* S1 has to find a new relay. It has a list of backup P nodes which it uses to find a new relay. S1 pings each of the P nodes and choses the one that has the smallest round trip ping echo and response time (Step 1).

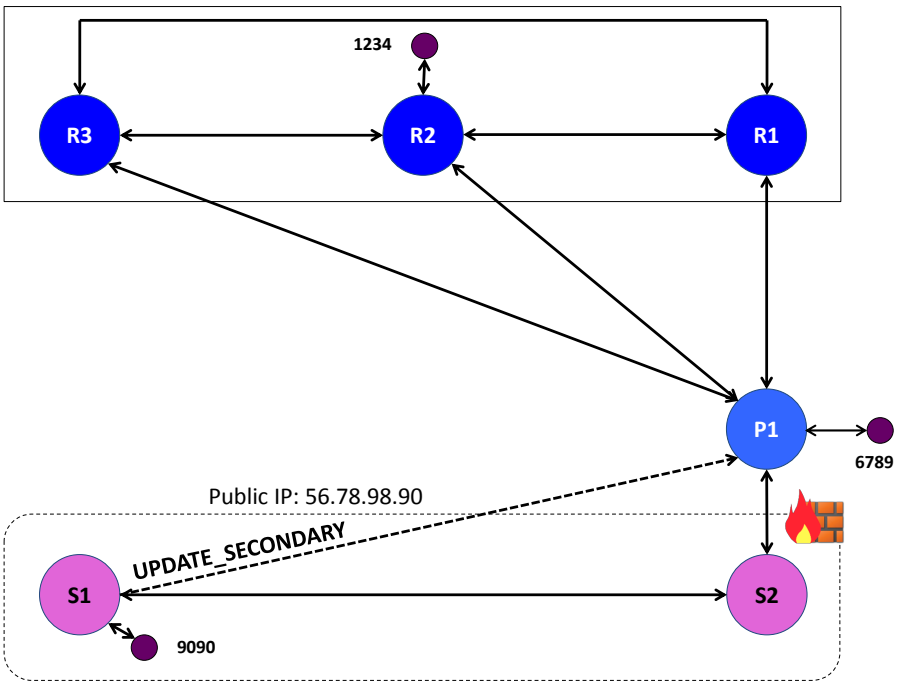

Figure 4.26: *P Node Deletion.* S1 choses P1 as its relay and sends the node an `UPDATE_SECONDARY`, containing the list of all directly connected GDP endpoints (9090). When P1 receives this message it updates its routing table and Public to Private map (Step 2).

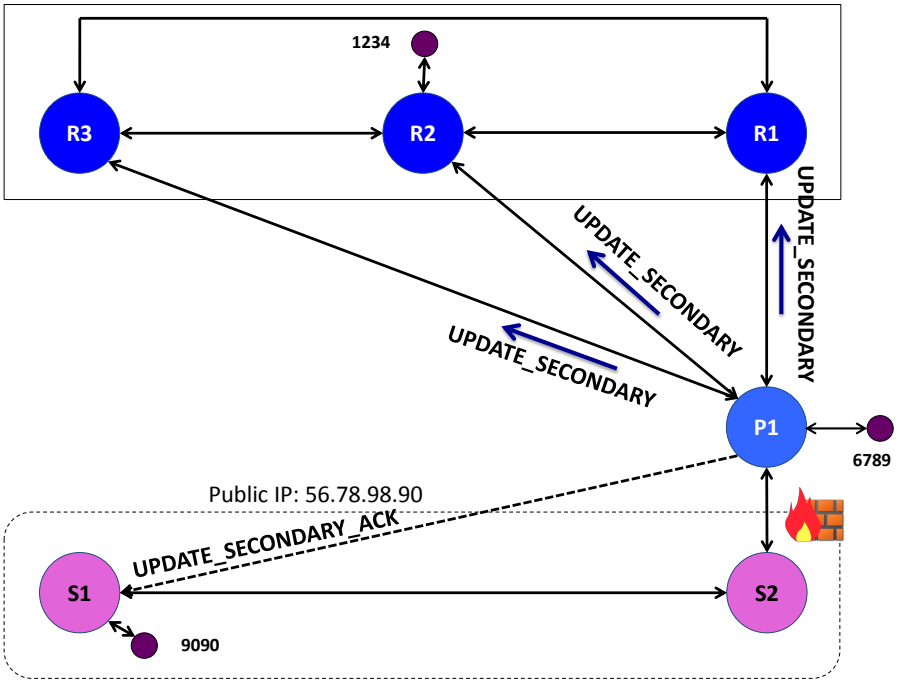

Figure 4.27: *P Node Deletion.* The new P relay (P1) informs all other P nodes about the updation by forwarding the `UPDATE_SECONDARY` message (Step 2). P1 also sends S1 an acknowledgment containing the most up to date list of all P nodes (Step 4).

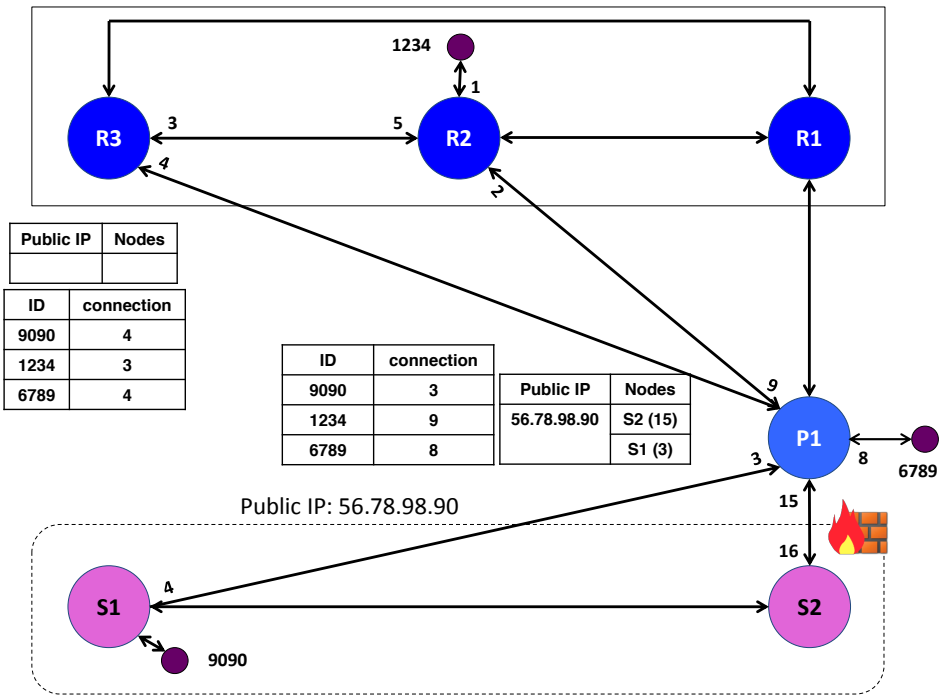

Figure 4.28: *P Node Deletion.* The final network. The routing tables and Public To Private maps of P1 and R3 are shown as examples

4.2.3.2 Secondary Node Failure

When an S node fails, the following nodes detect the node failure:

1. All other S nodes behind the same NAT/firewall as the failed S node. This is because all the S nodes within the same local network form a fully connected mesh network.
2. The P node, which served as a relay to the failed S node.

All other S nodes within the same local subnet acknowledge this failure by removing all endpoint IDs from their routing table that mapped to the file descriptor, representing the connection to the failed S node. These entries only include endpoints hosted by clients connected directly to the failed S node.

The relay P node has three tasks:

1. *Update its routing table:* The P relay removes all entries in its routing table that included clients directly connected to the failed S node.
2. *Update its Public To Private hash Map structure:* When an S node fails, its entry in the Public to Private map becomes invalid. The relay P node updates its map by removing the entry.
3. *Inform all other P nodes in the network to update their routing tables:* The P relay node sends a `WTIHDRAW_SECONDARY` command to each other P node. This message contains the list of IDs of the endpoints hosted by clients directly connected to the failed S node previously. Upon receiving this message, each P node removes the corresponding entries from its routing

table.

Apart from the node deletion protocol we have currently, we would also in addition eventually require a periodic anti-entropy protocol in which all P nodes contact each other to compare notes on their routing tables and current list of S nodes and clients. This is important to ensure that the system is always in a consistent state where all the data structures maintained by each node are up to date. Moreover, the failure handling mechanism that we have implemented currently follows a reactive recovery model [18]. In order to prevent positive feedback loops in the system during failures, we wish to move to a periodic recovery mechanism. This periodic protocol has not yet been implemented and is part of the future work for this project.

5

System Implementation Using Click

This section describes our implementation of the GDP routers on the Click Modular Router Platform. The platform is used for building flexible and configurable software routers. A router in Click is made up of packet processing modules called elements. In order to build a router configuration, users choose a collection of elements and connect them into a directed graph. Users are also allowed to write new elements or modify existing elements are per their requirements [3].

A Click router design is thus divided into two phases. In the first phase, users write element classes, which are configuration independent. In the second, users design a particular router configuration by choosing a set of elements and the connections between them. Click follows an Object Oriented Design approach, in which elements are written in C++ in the form of classes using an extensive support library. Router configurations, however, are written in a programming language called Click. This language is wholly declarative. It enforces hard separation between the roles of elements and configurations, leading to better element design. Its main advantage is that it makes router configurations human-readable. It can also be manipulated easily using automatic tools. Thus, it keeps the system as a whole both simple and clear [3].

In section 5.1, we describe the Click Configuration file, used to create our GDP router and how it differentiates between a P node willing to serve as a relay to S nodes and a P node that cannot be a relay due to memory/performance reasons. Section 5.2 next depicts our Click element that we designed. In section 5.3, we illustrate the implementation of the most important data structure maintained by the routers, the routing table. Finally, we address the topic of Keepalive in TCP connections in section 5.4.

5.1 Click Configuration File

A Click configuration file is written in the Click language. It is used to specify a given router's configuration using various types of parameters that are parsed by Click's internal parser tool. Below is the click configuration file for our GDP router:

```
define($SADDR 128.32.33.47, $SPORT 8007, $BOOTADDR 128.32.33.68,
$BOOTPORT 8007, $CANBEPROXY true, $WPORT 15000, $DEBUG 1)

gdp::GDPRouter($SADDR, $SPORT,BOOTADDR, BOOTPORT, $CANBEPROXY, $WPORT, $DEBUG)
```

The GDPRouter is our click element, written in C++, whose implementation is discussed in the next section. The input parameters are defined using define(), provided by the Click language. The value of these parameters can be defined dynamically by specifying them in the command line or they can be statically modified inside the Click configuration file. Our GDP routers accept four parameters as input.

1. $SADDR: This parameter is used to specify the IP address of the host running our GDP router. The IP address of a host running on Linux can be obtained using the terminal command ifconfig. This parameter can also accept the name of the link interface directly. An example is 'eth0' which represents the first Ethernet interface. This type of interface is usually an NIC connected to the network by a category 5 cable IO.
2. $SPORT: This parameter specifies the port on which the user wants to run the GDP router. The router uses the specified $SADDR and $SPORT to create a TCP server socket, listening for incoming connections.
3. $DADDR: This parameter specifies the IP address of an existing P node that will serve as the bootstrap node to the new router. It can be the IP address of any one of the Berkeley root servers.
4. $DPORT: This parameter specifies the port on which the Bootstrap is running and listening for connections. Its value can be the port of any of the 3 Berkeley root nodes.
5. $CANBEPROXY: This parameter specifies whether the new node is willing to serve as a relay/proxy or not. It is up to the user running the router to decide if the host running the router can allow the router to serve as a relay. This parameter becomes meaningless if the router turns out to be an S node.
6. $DEBUG: This is an internal parameter, which allows the user to print debug messages when the router is running.

Now that we have introduced an option for a given P node to serve as relay or not, we need to explain how this impacts the GDPRMP protocol. In the GDPRMP message header, we introduce an additional parameter called CanBeProxy, which takes one byte, making the total header size to be 39 bytes instead of 38. This parameter is set to 1 if the node can serve as a relay and 0 otherwise.

When a new node sends a JOIN command to its bootstrap, if the bootstrap determines the new node is a P node, it will store the node in its list of P nodes along with this parameter indicating whether the new node can act as a relay if needed. Similarly, all other P nodes also store this parameter in each entry of their respective lists of P nodes, when the new P node sends them an ADD_PRIMARY command. When a new S node asks its bootstrap or later its relay for the list of all P nodes in the network, the bootstrap (and the relay) only sends the new S node, the list of the physical locations of P nodes that can serve as relays based on this parameter. This technique ensures that an S node is never made aware of P nodes that cannot serve as relays.

5.2 Implementation of `GDPRouter` Element on Click

On initialization, Click creates a single running user level thread, which is responsible for processing every message received by a router from a given input port and forwarding a message to the appropriate output port of the router. Based on the message type, each message at the receiving node is handed to an up-call in the routing system, which does the required processing. This approach is used to implement an event driven message-passing system.

We have built our GDP router by forming TCP connections with other GDP routers as well as with GDP clients. These connections have been established using socket programming in C++. For more information about how to perform TCP socket programming in Linux, please refer to section A.1, Appendix A. Click offers a built-in callback mechanism that indicates whether a given socket is readable, writable or both. For initiating the call back mechanism for a given socket, Click provides two APIs:

1. `add_select(int fd, int mask)` : Click will register interest in readability and/or writability (depending on the value of mask) on file descriptor fd associated with the given socket. When fd is ready, Click will call the `selected(fd, mask)` method.
2. `selected(int fd, int mask)` : Click uses this method to handle a file descriptor event.

Our GDP router has been written as a Click element called `GDPRouter` using Object Oriented programming. For each element, Click defines a method called `configure()`. This method obtains the parameter values from the configuration file. Click also provides a method to perform router initialization called `initialize()`. In the initialization stage, two main tasks are performed:

1. The router uses its source IP and port to create a TCP server socket, which listens for incoming connections. The system calls `socket()`, `bind()` and `listen()` are used as discussed in section 2.1. The returned file descriptor is associated with Click's up-call using the API `add_select()` method.
2. The router creates a TCP client socket and connects to its bootstrap, using the bootstrap IP and listening port, with the `connect()` system call. The returned file descriptor is associated with Click's up-call using the API `add_select()` method. It then creates a GDPRMP `JOIN` message and sends the message over this client socket.

Click maintains a queue of events whose call back is associated with Click's `selected()` method. For example, when the server socket receives an incoming connection, Click generates an event and stores that event along with the file descriptor inside this event queue. The single running thread obtains an event from the queue and calls the `selected()` method, passing it the corresponding file descriptor for the event.

Our `selected()` method performs the following tasks:

1. It compares whether the file descriptor is the same as the one representing the router's server socket

- If the two file descriptors are equal, then we have a new incoming connection and we accept the connection using the `accept()` system call and associate a call back for the new socket created using the `add_select()` method.
- If the two file descriptors are not equal, then we know we have an event associated with an existing connection with another router or a client. This event represents that there is data available to read on the corresponding input queue.

2. The next task is to read the data from the input queue corresponding to the file descriptor. Since our sockets are TCP stream sockets, data is read as an array of characters. It is our responsibility to convert this raw data into GDP Packets.
3. For each GDP packet, we call a method `processPacket()`, which is responsible for differentiating GDPRMP packets from forwarding packets, processing the packets and sending the corresponding response to the correct output queue.

Figure 5.1 depicts our `GDPRouter` element design. Based on the type of GDP packets received, we have different types of methods for processing forwarding packets and different types of GDPRMP packets. The `regulatedWrite()` converts the responses generated from the methods into raw character bytes. The arrows between the methods show the flow of messages received by the router and the response to those message sent by the router. Every TCP connection in our router is represented by a TCP stream socket and every socket has an input and output queue associated with it. The input/output queues contain the GDP messages as raw character bytes. Every time a message arrives at an input queue, a corresponding event is generated and placed in the event queue. The `selected()` method reads the raw character bytes from the corresponding input queue and converts the data into GDP packets. Based on TCP's congestion control mechanism data is sent in a controlled manner over the link interface from the output queues.

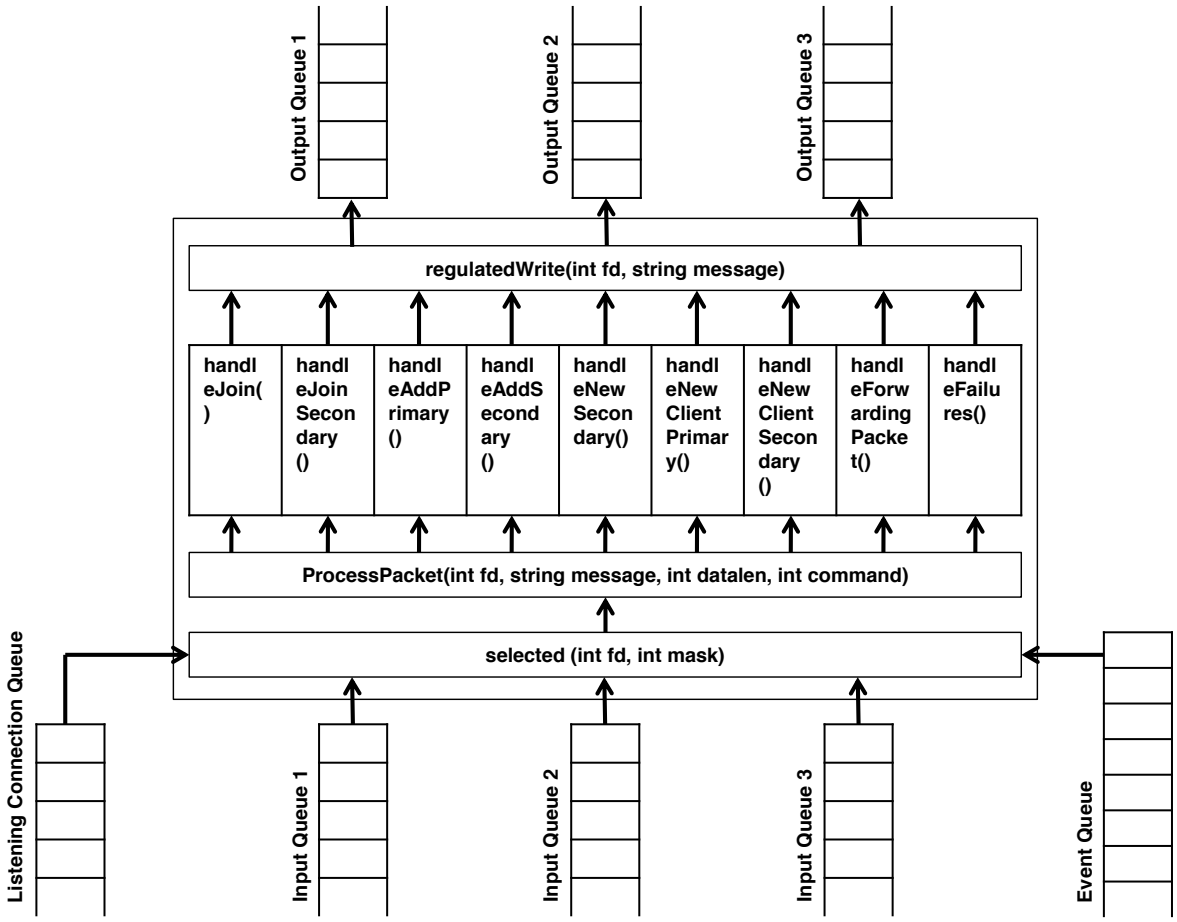

Figure 5.1: GDP Router Element on Click. The Input/Output queue are for each 2-way TCP connection with other nodes. The Listening Connection Queue is the input queue for the server listening socket in the router waiting for incoming connections. Arrival of each message on any input queue or listening connection queue generates an event stored in the Event queue. The `selected()` obtains the raw data from a given queue and converts it into GDP packets. It sends the GDP packets to the `processPacket()` method which processes the packet based on its type. Each method processes a specific packet type and generates an appropriate response. The `regulatedWrite()` method stores the responses in the appropriate output queue in the form of raw character bytes.

5.3 Routing Table Design

We now discuss the implementation of the Routing Table, which is used by the GDP routers to route messages from one client to another. We had earlier mentioned that a simple routing table design is a hash map table, which maps 256-bit IDs to appropriate link interfaces. However this design imposes the following limitation.

Consider the case when a P node fails in the network. We mentioned that all directly connected S nodes find a new relay. They contact this new relay by sending an `UPDATE_SECONDARY` command which contains the list of IDs of all endpoints hosted by clients directly connected to the S nodes. Let us look at this scenario more closely. Figure 5.2 shows two S nodes connected to 7 clients, each holding one endpoint. Both the S nodes use P1 as their relay. For simplicity, we have not shown the root nodes and their connections to S1, S2, P1 and P2. Figure 5.3 next shows what happens when P1 fails. P2 is notified of P1's failure and as per our protocol, P2 removes all entries from

its routing table corresponding to endpoint IDs mapped to the connection to P1. Suppose using the ICMP pinging mechanism, both the S nodes chose P2 as their relay. Each S node then sends P2 an UPDATE_SECONDARY command containing IDs of all endpoints directly connected. Figure 5.4 finally shows the new routing table of P2. P2 had to re-add 14 entries in its routing table corresponding to the endpoints connected to the two S nodes. Also, it is P2's responsibility of forwarding the UPDATE_SECONDARY messages to all other P nodes, who will also have to modify the 14 entries of their routing tables as well.

In reality an S node may be connected to more than, say 1000 clients instead of just 7 and P1 might be serving as a relay to more than just two S nodes. Also, each client may be hosting more than one endpoint. Thus when a P node fails we see that the size of each message sent by each S node to its new relay will be more than 1000*32 = 32000 bytes, since each endpoint ID is 256 bits or 32 bytes. This is a significant overhead on the network bandwidth consumed by the S nodes, especially as the number of directly connected clients (and endpoints) increase. Further when the new relay informs the other P nodes about the changes, it will have to replicate each UPDATE_SECONDARY message received before sending it to the nodes. This will lead to an exponential increase in the network bandwidth consumed by each node, including the new relay.

Also the new relay has to change more than 1000 entries in its routing table, upon receipt of each UPDATE_ SECONDARY command. Since Click is running in a single thread, this can tremendously slow down processing of each GDPRMP UPDATE_SECONDARY packet received by the new relay. During this time messages received from other nodes will wait in the corresponding input queues, adding a significant delay between the time a message was received and the time a corresponding response was generated. Since every other P node has to perform this step later, all the P nodes in the network will eventually encounter this issue and will have a significant reduction in their processing performance.

Both these issues are further aggravated if more than one P node serving as relay, fail at the same time. In order to address this scalability issue, we propose a new design for the routing table. Each node, instead of maintaining a single hash map, maintains 3 hash maps. These hash maps have been created in Click using the map data type from the C++ STL library [13]. The following are the descriptions of the three hash maps maintained by the GDPRouter element in Click.

1. ClientAdvertisments: This map is used to store the destination IDs of the directly connected clients to the router. It is similar to the previous routing table where the key is the 256-bit ID of the directly connected endpoints and the value is the file descriptor representing the connection with the corresponding client.

2. remoteClientAdvertisments: This map is used to store the destination IDs of the clients that are not directly connected to the router, but are connected to other routers and hence are remote to the given router. The key is the 256-bit ID of the endpoint hosted by a client and the value is the physical location (public IP, private IP and port) of the router to which the corresponding client is directly connected.

3. routingTable: This map is used to match a router's physical location to the appropriate file descriptor representing the connection to the router. For a given P node, say x, the map will contain physical locations of all P nodes in the network mapped to the file descriptors representing the connection to the nodes. The map will also contain physical locations of all

S nodes mapped to the file descriptor representing the connections to the S node if x is serving as relays to the nodes or the file descriptor representing the connections to the P nodes serving as relays to the corresponding S nodes. For an S node, the map will contain the location of all other S nodes within the same private network and the location of its relay mapped to the appropriate file descriptors.

In order to explain the usage of these maps, let us look at figures 5.5 – 5.7 which illustrate the same scenario as figures 5.2 – 5.4 but using the new routing table design. In figure 5.5, we see the three maps maintained by P1 as well as P2. In the maps, the location field contains the physical location of the corresponding router nodes (public IP, Private IP and port). For simplicity, this is represented by the node names in the figure. When P1 fails, both S1 and S2 choose P2 as their new relay and send an `UPDATE_SECONDARY` command to P2 (figure 5.6). This message no longer needs to include the endpoint IDs as P2 is already aware of the clients directly connected to the S nodes using its `remoteClientAdvertisment` map. When P2 receives the messages, it just updates its corresponding entries in `routingTable` to map the location of the S nodes to the new-formed connections (figure 5.7). Thus with this new architecture, we have saved network bandwidth and the number of entries to be modified by P2 in its routing table. It should be noted that this significant improvement over network bandwidth and router processing performance during failures is not free of cost. We have introduced some small amount memory overhead while shifting from a single map to a 3 map design. Also, in order to route a given GDP client message, a router will have to perform lookup on either one (`clientAdvertisments`) or two maps (`remoteClientAdvertisments` and `routingTable`). The previous design always ensured a single lookup.

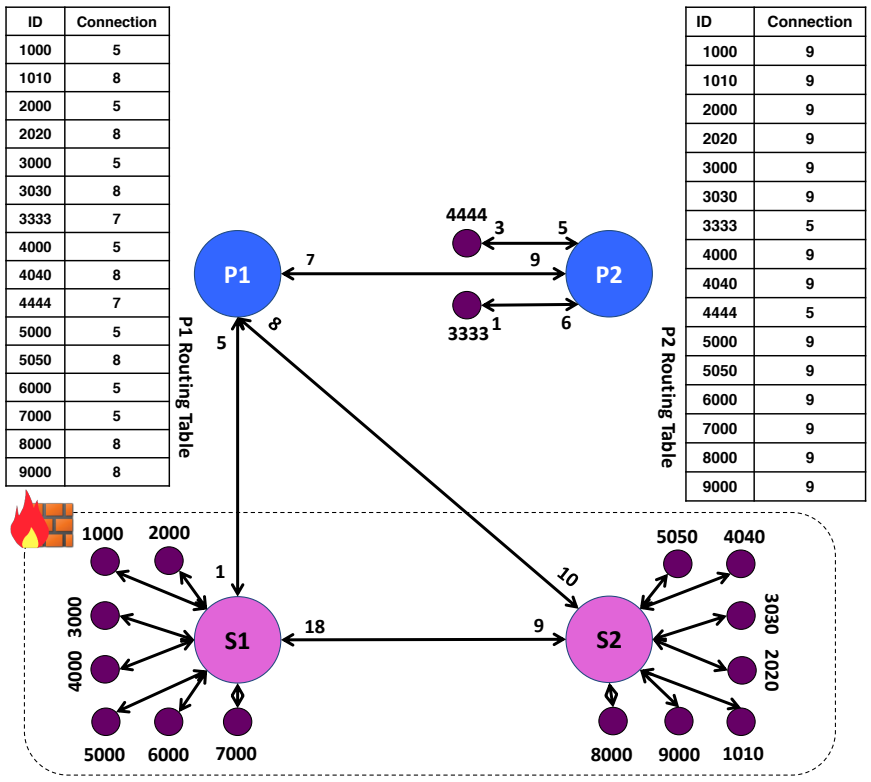

Figure 5.2: *A simplified GDP system with the previous Routing Table design.* There are 2 S nodes, each connected to 7 clients (each client has a single endpoint). The routing tables for P1 and P2 nodes are shown.

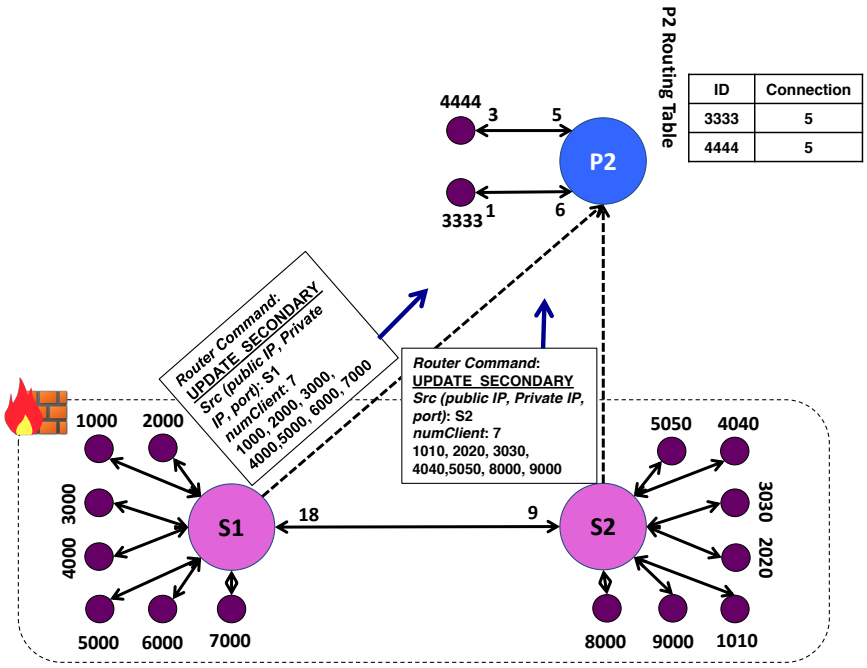

Figure 5.3: *A simplified GDP system with the previous Routing Table design.* When P1 fails, P2 removes all entries from its routing table corresponding to endpoint IDs mapped to connection with P1. Each S node establishes a TCP connection with P2 and sends an `UPDATE_SECONDARY` message containing IDs of all directly connected endpoints.

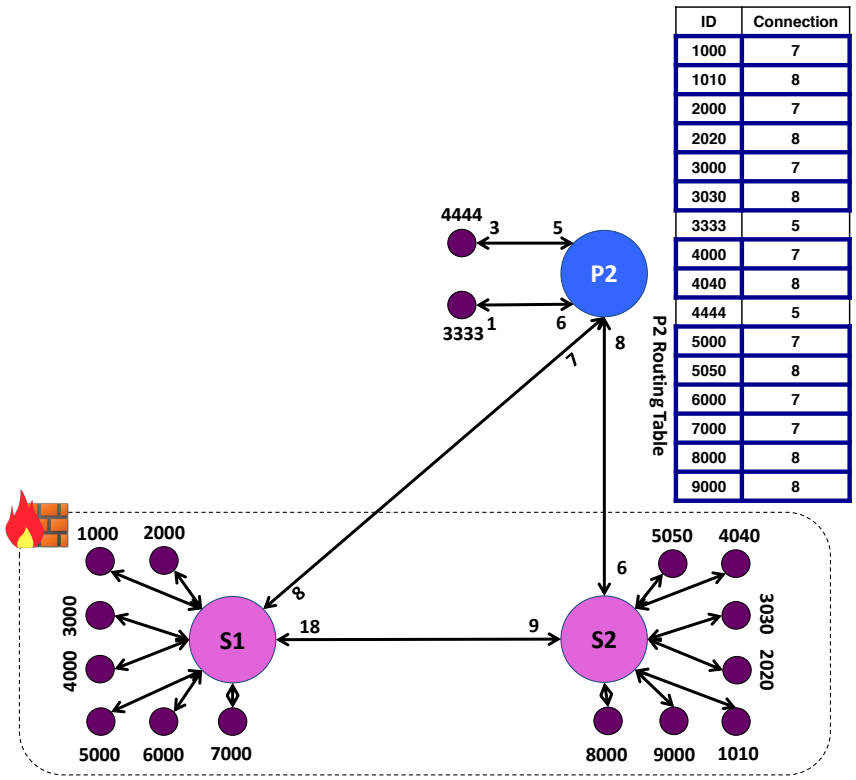

Figure 5.4: *A simplified GDP system with the previous Routing Table design.* P2 changed 14 entries in its routing table (marked in blue)

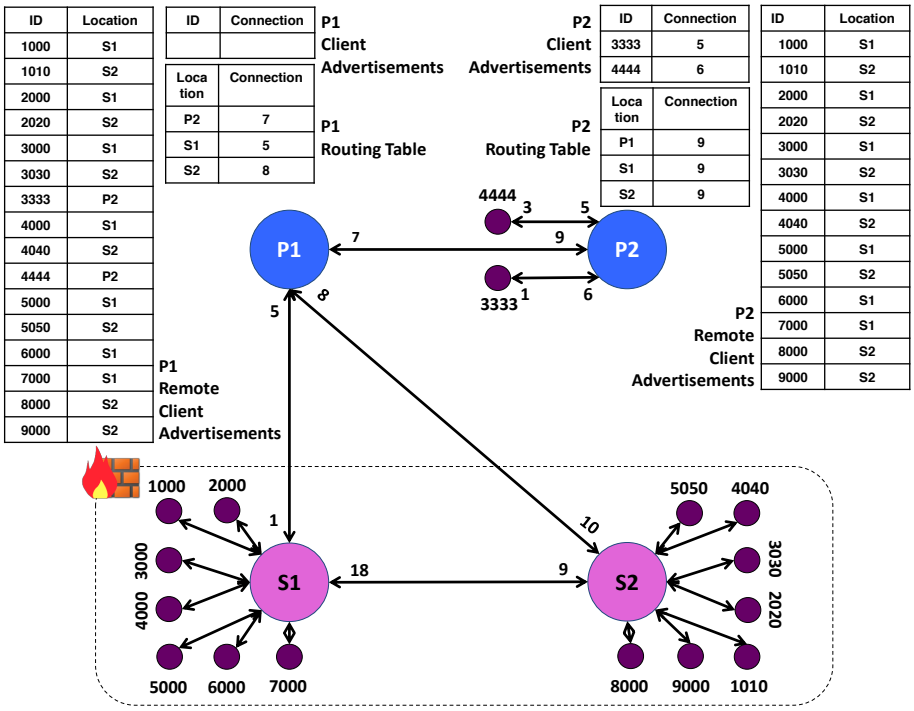

Figure 5.5: *A simplified GDP system with the new Routing Table design.* Each node maintains 3 maps (`clientAdvertisments`, `remoteClientAdvertisments`, `routingTable`). These data structures are shown for P1 and P2

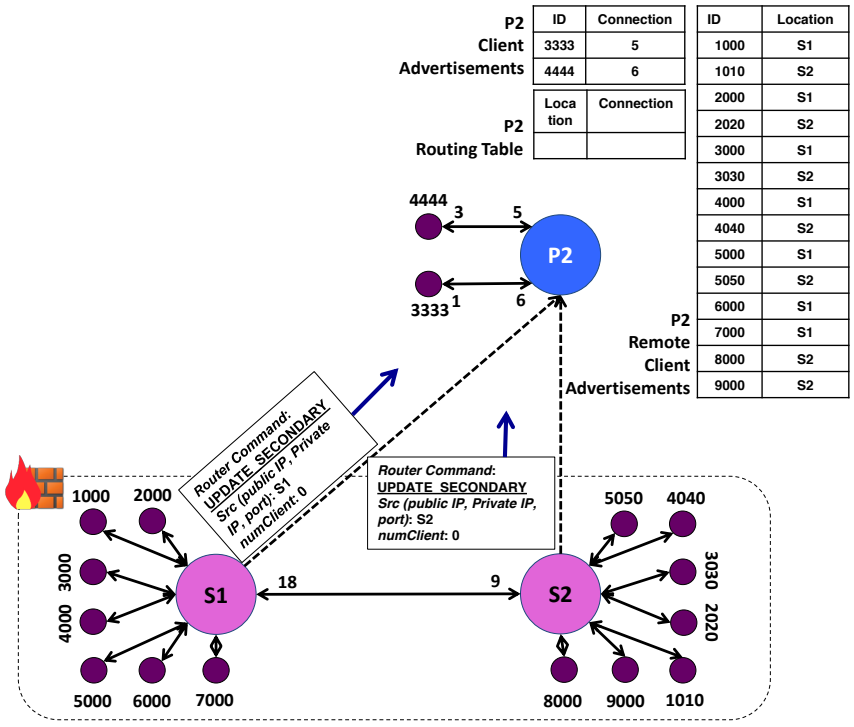

Figure 5.6: *A simplified GDP system with the new Routing Table design.* When P1 fails, each S node establishes a TCP connection with P2 and sends an `UPDATE_SECONDARY` message. The message doesn't have any endpoint IDs reducing the message size from the previous design.

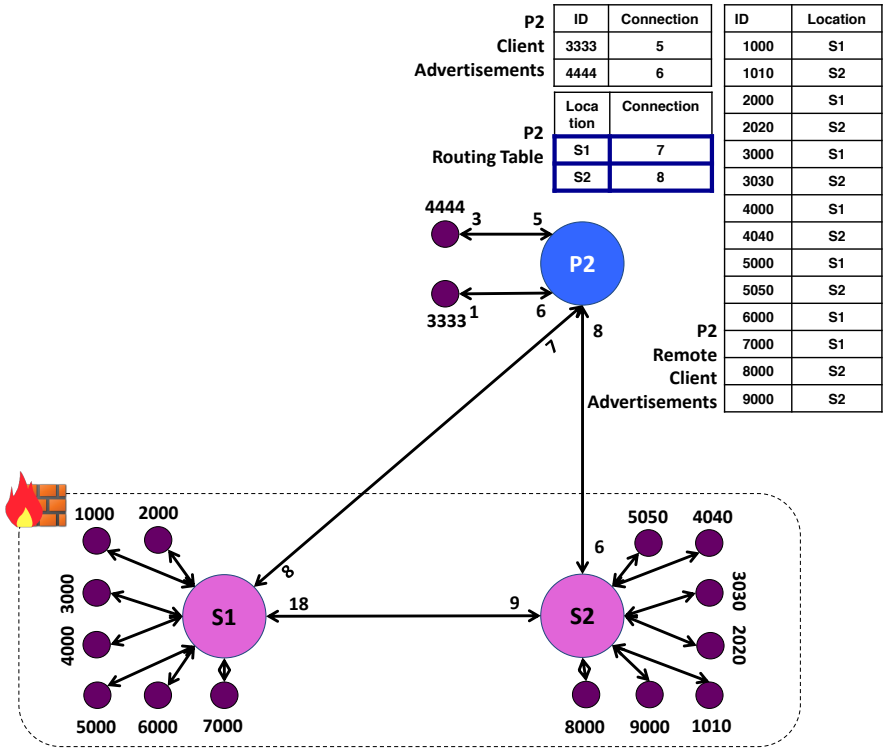

Figure 5.7: *A simplified GDP system with the new Routing Table design.* P2 only changes two entries in its routing Table map as opposed to 14, improving processing speed.

5.4 Keepalive

Keepalive is a technique used to ensure that a TCP connection between two hosts is operating and not broken. For the purpose of this project, we mainly use Keepalive to prevent inactivity from disconnecting a given TCP channel. It's a common issue, when a host is behind a NAT or a firewall, and is disconnected due to lack of messages being sent/received over the TCP channel. This behavior is caused by the connection tracking procedures implemented in NAT-enabled routers, which keep track of all connections (from all applications running in the host) that pass through them in their NAT translation tables. Because of the physical limits of these machines, they can only keep a finite number of connections in their memory. The most common and logical policy is to keep newest connections and to discard old and inactive connections first. [14]. Although Linux provides a keepAlive option when a TCP connection is established, this option has not been very reliable as some firewalls may filter TCP Keepalive messages [15].

We have implemented our very own Keepalive mechanism using a Click Timer element object. A Click Timer object triggers the execution of a given code after a specific time. [16]. We initialize a Click Timer object in our `GDPRouter` element. Periodically, the code in the timer, loops through all the available connections using the `routingTable` map and on each connection, sends a GDPRMP message with the `PING` command. This message only contains the GDPRMP headers, with no data. We have an option in our element class that allows the user to set the period manually. Thus by using Click's inbuilt timer mechanism, we prevent a TCP connection from being inactive for a long duration. Below is our C++ code that is executed periodically in Click's `run_timer()` method.

```
char version = (char)(VERSION);
char cmd = (char)(GDPRMP);
string dst = _GDPRouterAddress;
string src = _GDPRouterAddress;
WritablePacket *sendPacket = createGDPPacket(version, cmd, src);
char* pingOffset = (char *)(sendPacket->data()) + GDPRMP_HEADER_SIZE;
routePacket* pingPacket = (routePacket *)(pingOffset);
pingPacket->type = PING;
pingPacket->src = _myInfo;
pingPacket->numClientAdvertisments = 0;
pingPacket->numSecondaryAdvertisments = 0;
pingPacket->numSecondary = 0;
pingPacket->numPrimary = 0;
pingPacket->isTypeAssigned = false;
for (map<routingTableEntry, int>::iterator it = routingTable.begin();
it != routingTable.end(); it++)
int sendFD = it->second
int sentPacket = regulatedWrite(sendFD,
sendPacket->data(), sendPacket->length());
sendPacket->kill(); _pingTimer.reschedule_after_sec(PING_INTERVAL);
```

6

Evaluation

In this section, we evaluate our new GDP routers by comparing them with the previous GDP routers that were running. Our evaluation is in terms of the end-to-end latency for a given client GDP PDU. We define end-to-end latency as the total time taken by a given GDP client message to reach its destination client from its source client. This latency includes the processing time of each router along the path and the queue time of the message in each router node's input/output queue. We start by first describing in section 6.1, the Swarm Box platform, on which we deployed our routers for testing. Section 6.2 next describes our experimental set up. Finally, section 6.3 displays our performance results and provides a discussion for the results.

6.1 Swarm Box

Swarm Boxes are portable middleware devices developed in the Swarm lab, Berkeley. The goal of these devices is to make IoT applications scalable by separating the sensing functionality from the actuation. As part of the GDP vision, we expect users to set up these boxes in their local networks. Users will then be able to create their IoT applications by using these devices as a middleware platform for storing their sensor data securely. Thus these devices become part of the local environment and perform local computation, avoiding the need for a remote public cloud. We expect that these devices will be distributed with our GDP router code as well as the code for setting up a log server, if required.

Currently, two versions of the Swarm Box exist. The first version features a 4th generation i5 processor, dual ethernet ports (one of which supports IEEE1588), 4-8GB Dram, SSD, BLE and WiFi in a fanless enclosure. The second version is similar, with the exception of having a 5th generation i5 processor, single ethernet and smaller form factor. This version is a NUC [17]. Both the versions support connectivity to the Beagle Bone Black embedded device via Ethernet, USB or Wifi. Figure 6.1 shows the two versions.

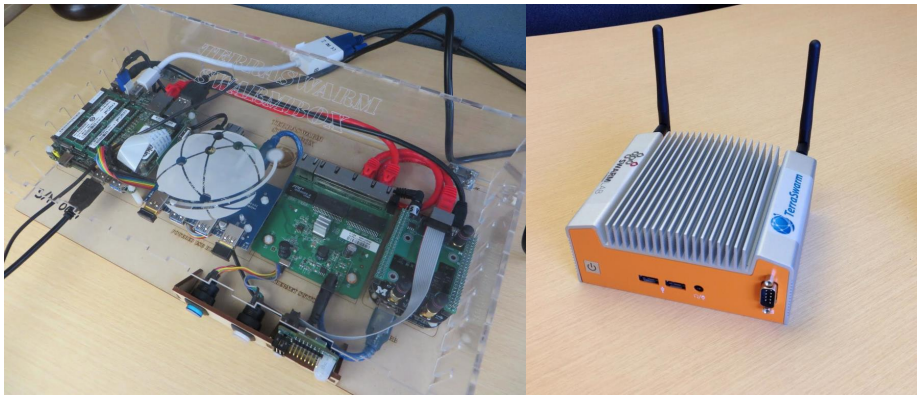

Figure 6.1: Swarm Boxes (Version 1 to the left and Version 2 to the right)

6.2 Experimental Setup

We perform 3 experiments to evaluate our router design. For each experiment, we limit the GDP forwarding packets between clients to be 100 bytes.

1. Our first experimental setup consists of two GDP routers, each connected to 10 fictitious clients, each hosting a single endpoint with a 256-bit ID. The clients and the router nodes have been deployed on a single version 2 Swarm box. The aim of our experiment is to measure the overall end-to-end latency for all messages received by a given client. We first install the previous GDP routers that were written in Python. These routers were designed to run in single thread mode. We next repeat the experiment with our new GDP routers deployed on Click. By deploying the entire infrastructure on a single Swam Box, we wish to avoid noise due to network latency. Thus the main factors that contribute to the end-to-end latency are the queue time of a given client message in the input/output queues of each router along the path from the source to destination client and the routing table lookup time at each router. Each client continuously sends a certain number of GDP client messages, say x, to each of the 19 other clients. We then plot the maximum time taken for a given client, amongst all the clients, to receive the messages (9x) from all the other clients versus the number of requests sent by each client (x). We next perform the same experiment with our new GDP routers. This experiment gives us a high level sense of how our new routers perform as compared to the previous router design.

2. Our second experiment illustrates a more comprehensive comparison between the two router designs. In this experiment, we analyze the end-to-end latency for each packet received by a destination client from a given source client. For this, we have two single clients (each hosting a single endpoint), one source and one destination. The source client sends 10,000 GDP packets to the destination client. Each client is connected to a separate GDP Router. The routers are directly connected to each other. Thus each packet, from the source to the destination client, passes through two router nodes. We first perform the experiment by using the previous GDP routers, written in Python and then replace them with our new GDP routers. In both cases, we plot the end-to-end latency histogram of all the GDP client packets received by the destination. All the nodes in these experiments are deployed on a single version 2 Swarm box.

3. In our third experiment, we analyze the performance of our new GDP routers using a more realistic scenario, in which we have two distinct private networks, willing to interact with each

other. Each local network consists of a single client (hosting a single endpoint) connected to an S node. For this experiment, we make sure that GDPRMP assigns each S node a different P relay. We once again send 10,000 GDP packets from one source client to a destination client. However, this time the packets move through two S nodes and two P nodes, in order to reach the destination from the source. The P nodes are still deployed on a single version 2 Swarm box, residing in the Swarm lab, Berkeley. For the purpose of this experiment, we deployed the private networks on two Amazon EC2 servers located in West-Oregon. Both the servers have distinct IP addresses and our local networks reside behind two separate NATs. In the end, we show the end-to-end latency histogram of all the GDP packets received by the destination client.

6.3 Performance Results

Figure 6.2 depicts our performance results from our first experiment. It can be seen that our new GDP routers perform significantly better than the previous routers, as we increase the number of packets sent by a given client to every other client. This gap becomes more important when we consider the fact that our new GDP routers have extra feature add-ons as compared to the previous design (dynamic operations such as join/fail, differentiation between P and S nodes, keepAlive etc.)

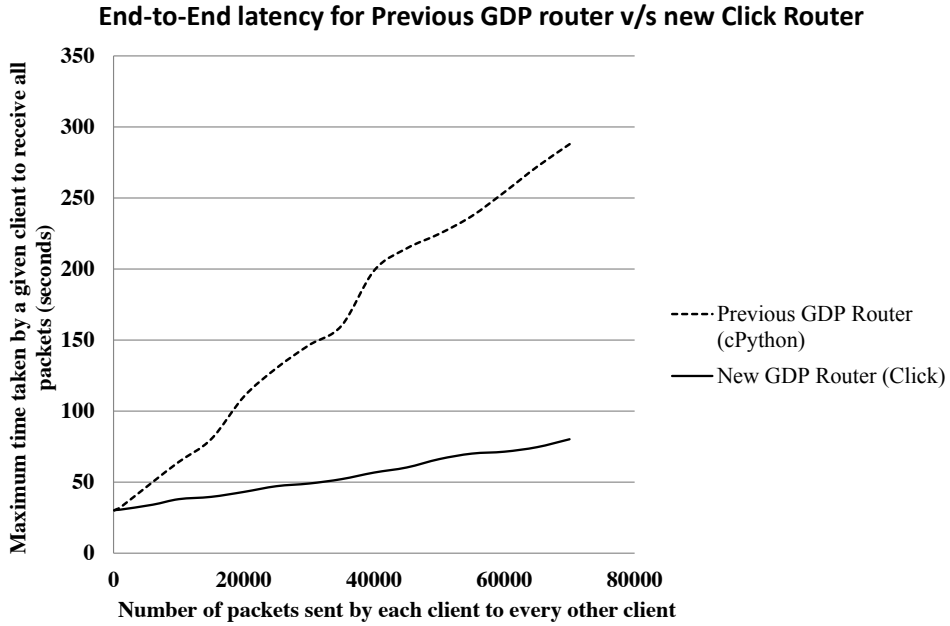

Figure 6.2: End-To-End latency results for the previous GDP router design and our new GDP routers on click. It can be seen that there is a drop on the end-to-end latency for the click routers as they have been implemented in C++ (unlike that previous routers, that were written in Python)

This performance gap is due to the reason that the previous router code was written in Python whereas Click is written in C++. Python adds an extra layer of abstraction above C in order to make the code written more user-readable. This layer adds significant performance overhead. Due to this layer, the processing time per packet on each router node, written in Python, is higher compared to that in C++. As a result a large number of packet are left waiting in the input queue and due to TCP's in-built congestion control [19], the sender of a packet on a given TCP connection may at times have to wait, until the receiver's input queue has enough space to accommodate new

packets. Our second experiment gives a concrete proof for this.

Figures 6.3 and 6.4 depict the performance results obtained from our second experiment. In Figure 6.3, we have illustrated the latency histogram for 10,000 GDP packets received by a single destination from a single source client. Each packet passes through two previous GDP routers, written in Python. It can be seen that there are two main spikes (latencies with the highest packet count) in the histogram. The second spike is due to the increase in amount of wait period in the router nodes' input/output queues, during congestion. Figure 6.4 illustrates the latency histogram using our new GDP routers deployed on Click. There is only one spike and the latency corresponding to it (0.1434 milliseconds) is much smaller compared to the first spike in figure 6.3 (0.5397 milliseconds).

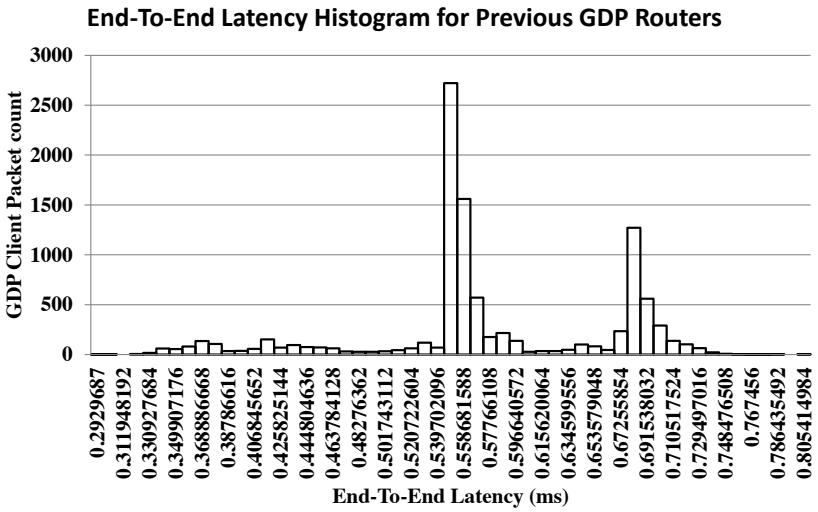

Figure 6.3: Latency Histogram recorded for 10,000 packets sent from a single source client to a destination client, via two previous GDP Routers (written in Python). The mean End-To-End latency is 0.57732 milliseconds and the Standard Deviation is 0.083302 milliseconds.

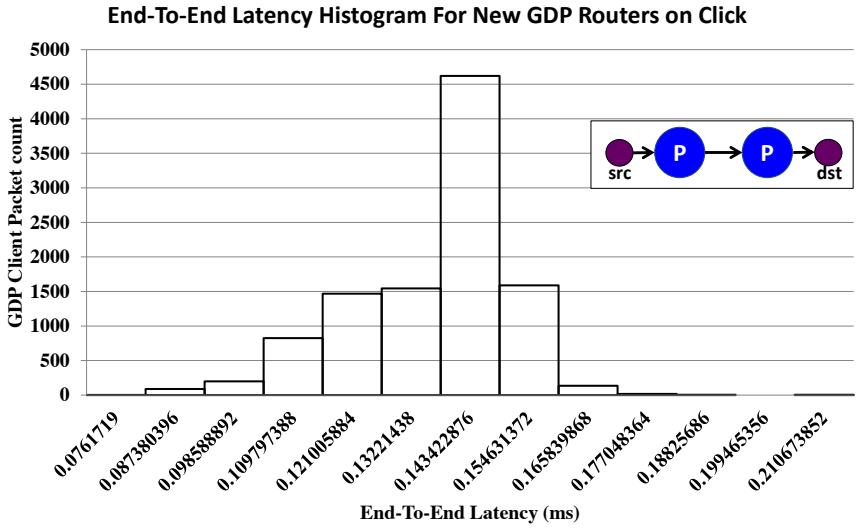

Figure 6.4: Latency Histogram recorded for 10,000 packets sent from a single source client (src) to a destination client (dst), via two new GDP Routers on Click. The mean End-To-End latency is 0.1367 milliseconds and the Standard Deviation is 0.015108 milliseconds.

Finally, figure 6.5 illustrates the latency histogram corresponding to our third experiment. It can be seen that, although there are 4 router nodes in the path of each GDP Router Packet, our system shows signs of stability with most of the packet latencies ranging between 30 – 50 milliseconds. The mean packet latency, as computed from the histogram is 42.9696 milliseconds and the Standard Deviation is 13.5439 milliseconds. The stability is due to the fact that we have implemented our routing tables and other data structures using hash maps. This gives us an O(1) lookup time during forwarding packets, improving packet processing performance.

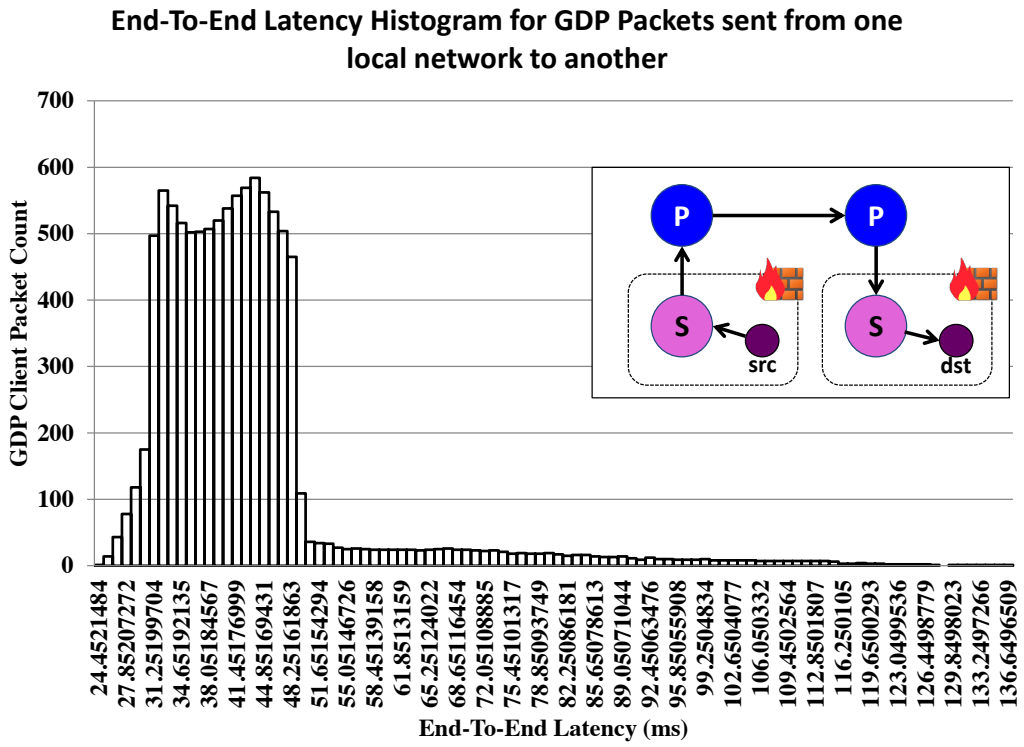

Figure 6.5: Latency Histogram of 10,000 GDP packets sent from one source client (src) located in a private network to a destination client (dst) in another private network. The path taken by each packet consists of two S nodes residing in both the client's private networks and two P nodes, serving as relays to the S nodes. The mean End-To-End latency is 42.96957 milliseconds and the Standard Deviation is 13.54392 milliseconds.

7

Future Work

Currently, we have deployed our Berkeley root nodes in 3 Swarm boxes and have also reached a point where we can create local networks behind a given NAT/firewall by installing our software routers in different swarm boxes and using them to connect various types of sensors and actuators in a local environment. However, we are yet to perform a quantitative analysis on how the system as a whole performs when subjected to periodic network churns. Churning refers to the process of continuous arrivals and departures of nodes in a given network [18]. Ideally, we want the router nodes to perform lookups quickly and consistently under churn rates at least as high as those observed in typical P2P systems such as BitTorrent. A suitable metric of churn is a router session time: the time between when a given router node joins the network until the next time it leaves. Median session times observed in deployed networks range from as long as an hour to as short as a few minutes [18]. We intend to determine how our system performs, in terms of latency of lookups and the percentage of client requests processed successfully, when the session time of the nodes in the network vary.

In terms of the features, we envision the GDP to eventually be a completely self-organizing and self-configuring system. Currently our GDP clients can successfully use Zeroconf to locate entry points into the network. Similarly, we would also want routers to dynamically locate their bootstrap nodes, without the need to statically provide the location of the Berkeley root nodes. However, since our design poses the constraint that a bootstrap node for a new router has to be a P node, a filtering mechanism needs to be introduced to make sure Zeroconf only provides a new router node, location of close-by P nodes. Another feature we need to add is to account for local networks behind firewalls, which only support outgoing HTTP connections. For this project, we assumed that GDP routers residing behind firewalls could initiate TCP connection with non-NATed routers at will. This assumption breaks if the firewall is only equipped to support connections on specific ports, such as HTTP (port 80). In such as scenario, our GDP routers will have to initiate persistent HTTP connections with nodes outside their firewalls instead of TCP connections on randomly selected ports.

Another important area of future work is replication. Currently our routers don't support multiple copies of a single GCL for example. In order for the GDP to be a durable system, we would want our GDP routers to support replication of different types of services and then based on a client's location, direct its request for a given service in a way that can preserve locality. On the other hand, we would also require implementation of publish-subscribe semantics. In order to

implement subscriptions efficiently, it will most likely be necessary to use some kind of multicast scheme that intelligently delivers data to all subscribers for a particular GDP service (a log for example).

Finally, we also want to address the issue of scalability. Currently, all the P nodes in the GDP network form a fully connected network. Not only that, each P node has all the GDP endpoints, connected to every P or S node, in its routing table. This design has limitations in terms of scalability. We first need to perform experiments in order to quantify the maximum level of scalability this design can achieve, in terms of the maximum number of routers and clients that can form a stable network. We also need to quantify the parameters that define a stable network such as the rate of node failure, average end-to-end latency, network bandwidth consumed per client request etc. Finally, we intend to convert the fully connected network formed by the P nodes into a Distributed Hash Table (DHT) system. For service discovery and routing, we will then use techniques from Plaxton routing, which is currently deployed in P2P systems such as Chord, Pastry, Tapestry and Bamboo [20, 21, 22, 18]. Although this may increase the number of network overlay hops (From 2 in our case) for a given client message, it will certainly improve system scalability.

8

Conclusion

This project was a stepping-stone towards a distributed implementation of the GDP that also supports creation of private networks behind NATs/firewalls. We began by first shifting the previous version of the GDP router running in python, to the Click platform. We then introduced a joining and deletion protocol for GDP router nodes and clients, which was missing in the previous design. After that we introduced support for router nodes residing in a private network by differentiating between NATed and non-NATed router nodes. We finally shifted from basic routing table architecture to a more intricate design that helps in handling node failures more efficiently. We next evaluated our final system and compared it with the previous GDP router design. Our results demonstrated that our efforts to create a hierarchical routing system are worthwhile, but work remains to be done in order to develop a more efficient and stable system that will make the GDP a fully distributed self-scaling, self-organizing and a self-configuring system.

9

References

[1] Zhang, B., Mor, N., Kolb, J., Chan, D. S., Goyal, N., Lutz, K., Allman, E., Wawrzynek, J., Lee, E. and Kubiatowicz, K. The Cloud is Not Enough: Saving IoT from the Cloud. In *proceeding of the 7th USENIX Workshop on Hot Topics in Cloud Computing, (HotCloud 15), (July 2015)*, USENIX.

[2] Global Data Plane | SWARM. `https://swarmlab.eecs.berkeley.edu/projects/4814/global-data-plane`.

[3] Kohler, E., Morris, R., Chen, B., Jannotti, J. and Kaashoek, M. F. In *ACM Transactions on Computer Systems, (TOCS), (August 2000),* ACM, v.18.n.3, pp. 263-297.

[4] Thomson, S., Narten and Jinmei, T. "IPv6 Stateless Address Autoconfiguration", *(September 2007),* IETF 4862, Request for Comments.

[5] Abbes, H. and Dubacq, J. C. Analysis of Peer-to-Peer Protocols Performance for Establishing a Decentralized Desktop Grid Middleware. In *Euro-Par 2008 Workshops - Parallel Processing, VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas de Gran Canaria, Spain, August 25-26, 2008, Revised Selected Papers (2008)*, pp. 235-246.

[6] Apple Inc. `https://support.apple.com/en-us/HT201311`

[7] Apple Inc. `https://developer.apple.com/library/mac/documentat ion/Cocoa/Conceptual/NetServices/Articles/about.html`

[8] Chadwick, D. Network Firewall Technologies. In *Advanced Security Technologies for Insecure Networks, IOS Press, (2000),* pp. 149-166.

[9] Kurose, J. and Keith, R. Network Address Translation (NAT). In *Computer Networking, (2010),* v.5, pp. 359-262.

[10] Kho, W., Baset, S. A. and Schulzrinne, H. Skype Relay Calls: Measurements and Experiments. In *proceedings of IEEE Global Internet Symposium, (2008),* IEEE.

[11] Weaver, N., Sommer, R. and Paxson, V. Detecting Forged TCP Reset Packets. In *proceedings of the Network and Distributed System Security Symposium, NDSS 2009*

[12] Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B. and Kubiatowicz, J. Pond: the OceanStore Prototype. In *proceedings of 2nd USENIX Conference on File and Storage Technologies, (FAST '03), (2003),* USENIX, pp. 1-14.

[13] CPlusplus.com. `http://www.cplusplus.com/reference/map/map/`

[14] Seggelmann, R., Tuexen, M. and Williams, M. "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", *(February 2012),* IETF 6520, Request for Comment.

[15] Braden, R. "Requirements for Internet Hosts – Communication Layers", *(October 1989)*, IETF 1122, Request for Comment.

[16] Click Modular Router | Timer Class Reference
`http://read.cs.ucla.edu/click/doxygen/class_timer.html`

[17] LogicSuply | NUC Evolved
`http://www.logicsupply.com/media/resources/spec-sheets/Logic-Supply-ML100-Series-Industrial-NUC-Computer-SpecSheet.pdf`

[18] Rhea, S., Geels, D., Roscoe, T. and Kubiatowicz, J. Handling Churn in a DHT. In *proceedings of the USENIX Annual Technical Conference, (ATEC '04), (June 2004),* USENIX.

[19] Allman, M., Paxson, V. and Blanton, E. "TCP Congestion Control", *(September 2009),* IETF 2581, Request for Comment.

[20] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, (SIGCOMM 01), (October 2001),* ACM, pp. 149-160.

[21] Rowstron, A. and Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg, (2001)*, pp. 329-350.

[22] Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. Joseph, A. D. and Kubiatowicz, J. Tapestry: A resilient global-scale overlay for service deployment. In *IEEE Journal on Selected Areas in Communication 22, 1 (January 2004),* pp. 41-53.

[23] Mitchell, S., Villa, N., Weeks, M. S. and Lange, A. The Internet of Everything for Cities. *Cisco, 2013*. Available:
`http://www.cisco.com/web/strategy/docs/gov/everything-for-cities.pdf`

[24] Mazie Res, D., Dabek, F. and Perterson, E. Using TCP through Sockets, *(2000)*. Available: `http://www.scs.stanford.edu/07wi-cs244b/refs/net2.pdf`

[25] Stoica, I., Adkins, D., Zhuang, S., Shenker, S. and Surana, S. Internet Indirection Infrastructure. In *proceedings of SIGCOMM 2002*, pp. 73-86.

10

Appendix A: System Building Blocks for the GDP Router

This section intends to provide some background information about the various system-level blocks used to build our GDP routers. We provide information on TCP socket programming in Linux, working of firewalls and NAT-enabled routers and Zeroconf.

A.1 TCP Socket Programming

Sockets are a mechanism for exchanging data between processes. These processes can either be on the same machine, or on different machines connected via a network. Once a socket connection is established, data can be sent in both directions until one of the endpoints closes the connection. There are two types of sockets based on the transport layer protocol (TCP v/s UDP) they follow, stream sockets and datagram sockets.

Our GDP routers make use of stream sockets to form TCP connections with other GDP nodes. Stream sockets treat communications as a continuous stream of characters. They use TCP (Transmission Control Protocol), which is a reliable, stream-oriented protocol. In Linux, a new socket is created using the `socket()` system call. This system call creates a new socket and returns an entry into the file descriptor table. This entry is represented by a small value integer and is used for all subsequent references to this socket [24].

Networking using stream sockets follows a client-server model. Figure 10.1 depicts the set of system calls used by each end point to set up a two-way TCP connection between them.

On the server's end the following steps are performed:

1. A new socket is created using the `socket()` system call.
2. The `bind()` system call is used to bind the socket to a specific IP address and port on which the server will run.
3. Next, the `listen()` system call is used to allow the process to listen on the socket for connections.
4. The server process next uses the `accept()` system call to accept an incoming client connection. It returns a new file descriptor, and all communication on this connection should be done

using the new file descriptor.

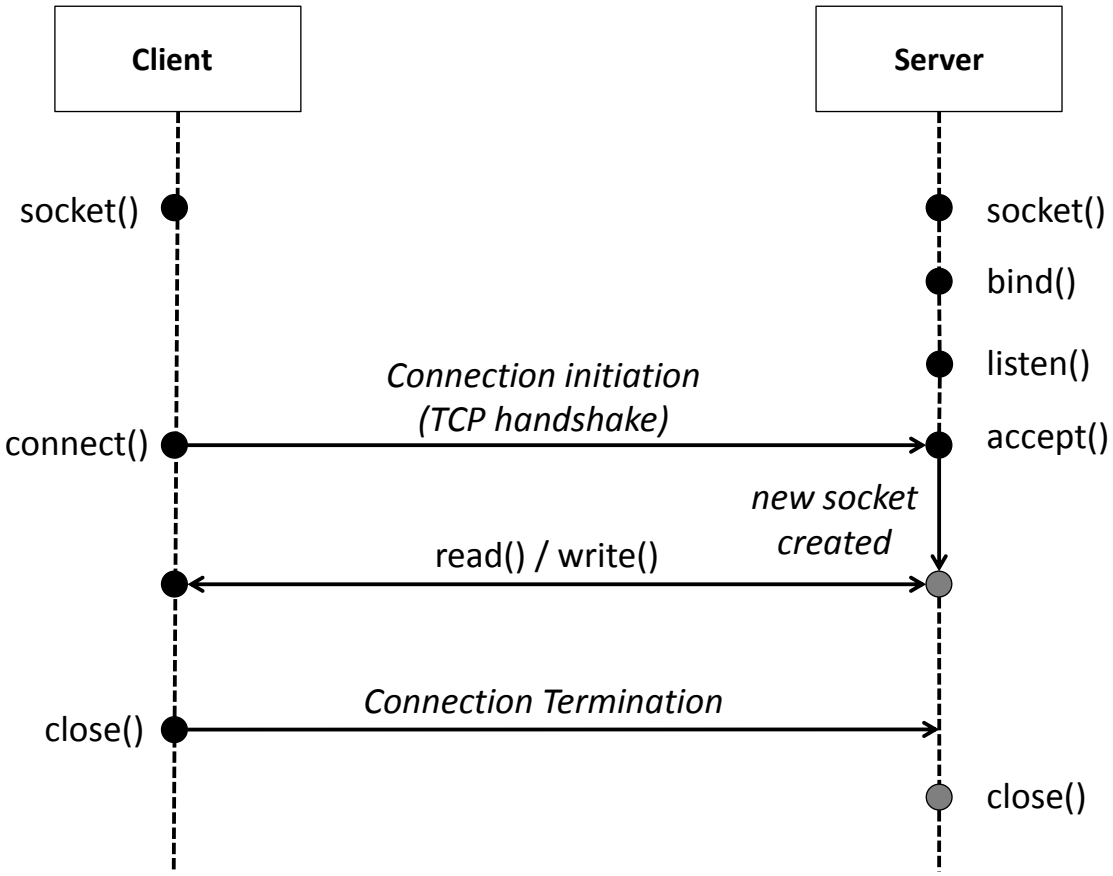

Figure 10.1: The set of system calls used by two host in order to set up a two-way TCP connection between them.

On the Client's end the following steps take place:

1. A new socket is created using the `socket()` system call.
2. The `connect()` system call is used to establish a TCP connection with the server. We provide the IP address and port, on which the server is running and listening for connections, to this system call.

Once a TCP connection has been successfully established between the client and the server, both the ends can use `read()` and `write()` system calls to read and write data respectively, in the form of character bytes. In the end, either one of them terminates the connection by issuing a `close()` system call. The other end is immediately informed about the connection termination and it closes its end of the connection using the `close()` system call as well. TCP socket programming also provides a built-in mechanism to detect failure of an endpoint (client or server). When an end point process is killed for some reason, the other end is notified immediately. The operating system running on the killed process sends the other end point an RST (reset) packet informing that the connection on the crashed process's end has been closed [11]. If say the operating system of the failed process crashes as well, the running end point detects the failure when it tries to send

data to the failed end point since the socket at the failed host no longer exists.

A.2 Firewalls and NATs

Most organizations today have an internal network, interconnecting their computer systems. The systems share mutual trust and form a private network. However, in most cases it is beneficial to connect this private network to the Internet. This however leads to vulnerabilities to insecure software, weak authentication, spoofing and cracker programs [8]. A firewall provides a secure gateway to interconnect a private network with the Internet. Network Address Translation (NAT), on the other hand, is a technique used to map an entire private network to a single IP address. It plays an important role in conserving global address space allocations in face of IPv4 address exhaustion [9].

Figure 10.2 depicts the working of a NAT enabled router. The router has an interface that is part of a private network. It also has a public interface which allows it to behave as a single device with a single public IP address. Thus, the router is hiding the details of the private network from the outside world. It also maintains a NAT translation table, which is used to route data to/from the private hosts. Suppose the host with IP address 10.0.0.1 requests for some piece of data from a host with IP address 128.32.33.68.

NAT Translation Table	
WAN side	LAN side
50.34.45.67, 5001	10.0.0.1, 3345

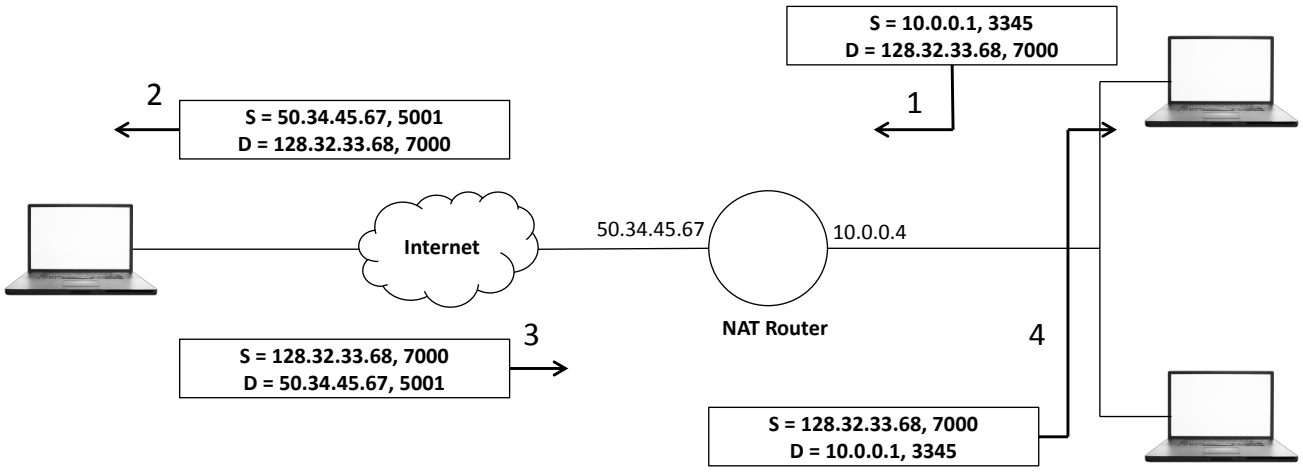

Figure 10.2: Network Address Translation Protocol.

The host 10.0.0.1 assigns a source port number 3345 and sends a query IP datagram packet to the NAT router. The NAT router then generates a new source port number 5001 (this number should not have been assigned earlier). It replaces the packet's source IP with its public IP and the source port number 3345 with 5001. It also adds an entry into its NAT translation table. The

public host running on 128.32.33.68, unaware that the NAT router has changed the source IP and port, responds with a datagram whose destination address is the public IP of the NAT router and destination port number as 5001. When the NAT router receives this response, the router indexes the NAT translation table using the destination IP and port to obtain the location of the private network host (10.0.0.1, 3345). The router updates the destination IP and port in the datagram and forwards it to the host in the private network [9].

In order to cater to the most general case, we have assumed that any host residing outside the Firewall/NAT of a given private network cannot establish a connection with a host residing in the private network. However, a host in a private network can initiate a connection with a host outside the private network, as long as the host being connected to is itself not behind another firewall/NAT. Once the connection has been established, both the end points can send and receive data over the connection at will. As per our assumption, two hosts behind two separate firewalls/NATs cannot connect with each other directly. In such a scenario, we would require a relay or a proxy node, which can serve as a common point of contact between the two hosts. This relay needs to be chosen such that it resides in the public domain (not behind a NAT). The two hosts establish a connection with this relay node and accordingly route traffic to each other through this node. Figure 10.3 depicts such a scenario where the relay node maintains a table that maps a destination IP and port to the appropriate link interface (file descriptor) to route external traffic.

Mapping Table in the public proxy (128.32.33.68)	
Destination (IP, port)	Link File Descriptor
50.34.56.78,8001	6
63.56.78.90,7001	7

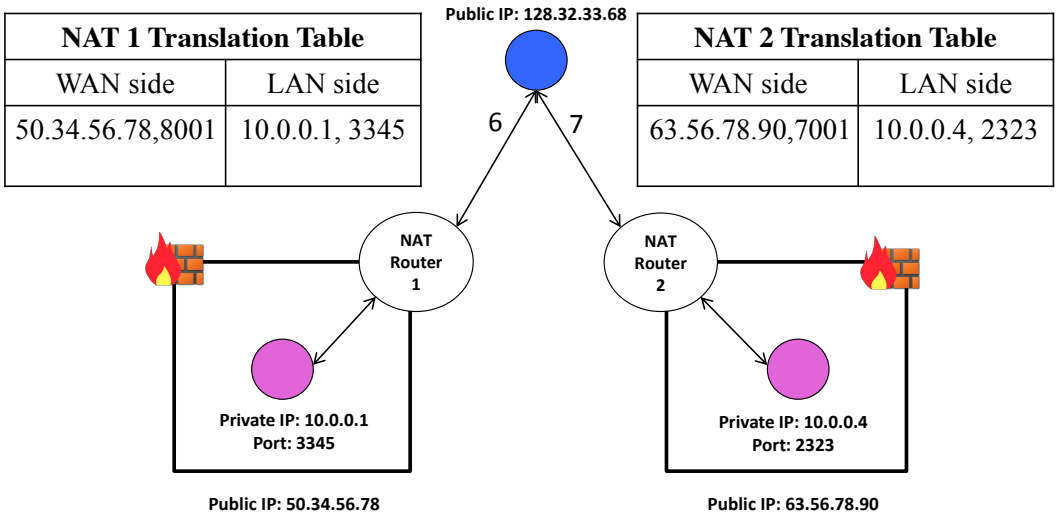

Figure 10.3: Interaction of two nodes behind two separate NAT enabled routers. The two nodes are assigned IP addresses of 10.0.0.1 and 10.0.0.4 in their respective networks and are running on ports 3345 and 2323. Both the nodes connect to a common relay outside both the NAT networks. The relay maintains a forwarding table which maps an IP address to the file descriptor representing the connection to the corresponding private network. In the diagram, integer value 6 represents the file descriptor at the relay end for the connection with the network, having public IP as 50.34.56.78 . Similarly, integer value 7 represents the file descriptor for the connection to the network, having public IP 63.56.78.90.

Skype is an immensely popular P2P application, which uses relays for establishing calls between hosts in private networks [10]. For each user residing in a private network, Skype assigns a non-NATed super peer. The users initiate a connection with their assigned super peers. When one user wants to talk to another user, the super peers select a third non-NATed super peer. The super peers then ask both the users to initiate a connection with this third super peer. Once the users establish the connections, this super peer serves as a relay which routes traffic between the two users. It will be seen, that our GDP routing protocol is very similar to Skype except for that in our protocol we don't ensure that two hosts residing behind two NATs always have to use a single relay. When two private hosts connect to two different non-NATed nodes, these public nodes continue to communicate with each other to route message from one private host to another. Although, this will add an extra hop delay in the communication between the private hosts, it reduces the overhead of having each private host to initiate a new connection with another non-NATed node every time it wants to talk to another host.

A.3 Zeroconf and Avahi

Zero-configuration networking (Zeroconf) is a type of network application technology that automatically creates a usable computer network based on the Internet Protocol Suite (TCP/IP) when computers or network peripherals are interconnected. Its main advantage is that it does not require manual operator intervention or special configuration servers [4].

In general, Zeroconf covers three main areas [7]:

1. *Addressing (allocating IP addresses to hosts running services):* This is achieved by self-addressing, which involves picking a random IP address in the link-local range and testing it. If the address is not in use, it becomes the host's local address. If it is already in use, the host chooses another address at random and tries again.
2. *Naming (using names to refer to hosts instead of IP addresses):* In this phase, each service offered by a given host is represented by a unique name. This name is chosen by the company/ organization willing to provide the service. Zeroconf maintains a mapping between the service name and the local IP address and port of the host running the service. When a host registers a service, Zeroconf automatically advertises the availability of the service so that any queries for the service name are directed to the host running the service.
3. *Service discovery (finding services on the network automatically):* When a client wishes to locate a service with a given name, it contacts Zeroconf providing it with the name of the service. Zeroconf then issues a multicast DNS (mDNS) query that is sent over the local area network using IP multicast. When a host running the service sees the query, it provides a DNS response with its own address and port on which it is running.

As part of the project, we make use of Avahi, an open-source Zero-configuration networking implementation. It allows programs to publish and discover services and hosts running a special Avahi daemon on a local network with no specific configuration [5]. Its main usage today is in the famous AirPrint application created by Apple Inc [6]. AirPrint is a feature in Apple's OS X operating system used to connect printers via a wireless LAN network.

For the purpose of this project, we use Avahi by enabling our GDP routers to advertise themselves as a special service, which new clients can automatically discover. A new client will use Zeroconf to discover a router node in its local area network that can serve as the closest entry point into the GDP network. Avahi will locate all the GDP router nodes running in the client's local area network and send their locations (IP address and port) to the client in the form of a list. The client will contact each node in the list until it has successfully established a TCP connection with one of the routers. In case there are no GDP routers running in the client's local subnet, we have installed 3 default root nodes, which any client can contact, in the event when Avahi returns an empty list.