Topology Discovery in Autonomic Networks

Parsa Ghaderi

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science at

Concordia University

Montréal, Québec, Canada

September 2022

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Parsa Ghaderi**

Entitled: **Topology Discovery in Autonomic Networks**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair and Examiner
Dr. S. Cespedes

_____ Examiner
Dr. J. Paquet

_____ Supervisor
Dr. J.W. Atwood

_____ Co-supervisor
Dr. L. Narayanan

Approved by _____
Dr. L. Kosseim, Graduate Program Director
Department of Computer Science and Software Engineering

_____ 2022

_____
Dr. M. Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Topology Discovery in Autonomic Networks

Parsa Ghaderi

The network Management Research Group (NMRG) introduced their own version of autonomic networks based on the viewpoint of the Internet Society and following the definition provided by IBM of autonomic systems. NMRG focused on self-optimizing, self-configuring, self-protecting, and self-healing capabilities in the proposed design model of autonomic networks. Later the Autonomic Networking Integrated Model and Approach (ANIMA) working group of the Internet Engineering Task Force (IETF) designed protocols to support the goals set by NMRG. The proposed autonomic network mitigates the human administration influence as much as possible and make the nodes dependent on themselves and the communications with their neighbors. Therefore, autonomic nodes will act as a network management entity that depends on the information they receive/send from/to their surroundings and their knowledge about themselves.

In network management, knowing the network's topology gives nodes a great advantage toward becoming more autonomic. Knowing the topology can help nodes with management tasks such as link failure recovery, routing, and imposing policy. Topology Discovery (TD) is the process of collecting the neighboring information of all nodes and distributing the processed information among them. Topology Maintenance (TM) takes place after the topology map is generated during the TD process. TM updates all nodes upon the changes in the topology map. The TD and TM can be heavy tasks on the network since they require collecting information from all nodes and distributing it among them.

We focus on supporting the benefits of autonomic nodes knowing the network's topology and

suggest efficient methods to collect and maintain the topological information of an autonomic network. Our goal is to minimize the bandwidth consumption by reducing the number of exchanged messages for TD or TM purposes. There have been many approaches proposed to improve the performance of TD and TM. There has been thorough research on TD methodologies but not all the proposed solutions can be applied to autonomic networks.

In this thesis, we review different methods for TD and discuss their compatibility with the proposed autonomic network guidelines. We then propose two new solutions. Our first solution is based on a clustering algorithm that allows the autonomic nodes to join clusters and limits the message passing to intra-cluster communications and inter-cluster communication between cluster-heads. The second proposed solution is based on taking advantage of the secure boot-strapping protocol (BRSKI) for autonomic nodes to generate the topology map of the autonomic network.

Acknowledgments

I want to acknowledge and give my warmest thanks to my supervisors, Dr. Atwood and Dr. Narayanan, who guided and helped me on this journey. The completion of this research could not have been possible without their expertise. With their guidance, I gained the most valuable experiences, not only academic-wise but in life too. I would also like to thank Dr. Carpenter for his help. During the past year, he provided us with answers that without our work would have never been complete.

I want to thank my parents, brother, and other family members for their unconditional love for me. They supported me and my every decision throughout all these years. Your support was what sustained me so far. I would also like to thank my friends. Thank you for all the good memories.

I want to dedicate this work to my late grandfather, whom I lost last year. He was my biggest supporter and always encouraged me to do my studies.

Contents

List of Figures

List of Tables

Acronyms

ACP	Autonomic Control Plane
ANI	Autonomic Network Infrastructure
ANIMA	Autonomic Network Integrated Model and Approach
ARP	Address Resolution Protocol
ASA	Autonomic Service Agent
BGP	Border Gateway Protocol
BRSKI	Bootstrapping Remote Secure Key Infrastructure
DMAC	Distributed Mobility-Adaptive Clustering
GRASP	GeneRic Autonomic Signaling Protocol
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IRTF	Internet Research Task Force
LL	Link Local
LLDP	Link Layer Discovery Protocol
MASA	Manufacturer Authorized Signing Authority
MTU	Maximum Transmission Unit
ND	Neighbor Discovery
NMRG	Network Management Research Group
OSPF	Open Shortest Path First
P2P	Peer to Peer
RIP	Routing Information Protocol
SDN	Software-Defined Network
SHTD	Self-Healing Topology Discovery
SNMP	Simple Network management Protocol
sOFTDP	Secure OpenFlow Topology Discovery Protocol
TD	Topology Discovery
TM	Topology Maintenance
ULA	Unique Local Address

Chapter 1

Introduction

As its name suggests, network management is the process of administering the components of a computer network. Network management allows the network to achieve higher performance, lower costs, and achieve users' goals. Network management provides facilities for nodes to recover from failure and optimize their performance. Networks can be managed centrally or in a distributed fashion. With the increase in the number of network devices, management has become a more challenging task. The complexity of the network depends on the number of nodes and/or the number of roles that they can acquire in the network. As networks grow in size, their dependency on human administrators grows as well, so that the network management meets the pace of the growing infrastructure. As the size of the networks grow and they are exposed to other networks, the chances to experience attacks and having malicious nodes inside the networks can grow. Also, the rate at which network experiences failure of nodes can grow too. Therefore, introducing a network management solution compatible with the type of the network and its purpose is an essential task.

The growth of management-related issues led to the idea of introducing networks with autonomic capabilities. In 2001, IBM introduced autonomic systems. The goal was to achieve self-configuring, self-healing, self-protecting, self-optimizing, and self-managing systems [7]. Paul Horn, former vice-president of IBM, talks about the similarities between autonomic systems and human nervous systems during the introduction of autonomic systems. Horn:

> *The body's autonomic nervous system changes your heart rate and breathing, which allows humans to adapt to any number of situations physically. This is very much how*

you need to think about the middle-ware as it will serve the Internet.

In other words, Horn suggests that the entities in an autonomic system must be capable of adapting themselves to their surroundings and independently making decisions accordingly. This adaptation is the result of constant flow of communication between the entities of the autonomic system.

An autonomic network facilitates the inter-connection of the autonomic systems and inherits the autonomic behaviour from its constructive elements. To date, different adaptations of autonomic networks have been introduced. Different companies provide different design models for autonomic networks based on their needs. The version that we are focusing on was introduced by one of the member research groups of the Internet Research Task Force (IRTF), which was formed to focus on long-term internet-related issues [1]. The IRTF comprises many active research groups, including the Network Management Research Group (NMRG); this group provides a forum for all researchers and enthusiasts to explore new technologies in the internet management field [2]. Besides supervising research groups, the IRTF works with the Internet Engineering Task Force (IETF), a parallel organization. The IETF is in charge of maintaining internet standards by developing protocols or providing guidance on how to use the protocols [18]. NMRG introduced what autonomic networking is, from the viewpoint of IRTF and IETF. The guideline provided by NMRG produced RFC7575 [7] and RFC7576 [21]. The former RFC introduces the design goals of autonomic networks, and the latter is the analysis of the gap between ideal autonomic networks and and the current network abilities. The IETF introduced the Autonomic Network Integrated Model and Approach (ANIMA) working group to develop and maintain documentation and specifications related to autonomic networks [3]. So far, ANIMA produced six core RFCs: RFC8990 to RFC8995, which standardized the designed protocols according to the definition of autonomic networks provided by NMRG. From this point on, wherever we mention autonomic networks, we refer to the version introduced by NMRG and developed by ANIMA.

The proposed design model for autonomic networks introduced a hierarchy of layers to share the network management job across different entities. ANIMA shares the same goals with the

[1] https://irtf.org/
[2] https://datatracker.ietf.org/rg/nmrg/about/
[3] https://datatracker.ietf.org/wg/anima/about/

autonomic systems introduced by IBM: self-configuring, self-healing, self-optimizing, and self-securing nodes inside an autonomic domain.

In network management, Topology Discovery (TD) plays a vital role by providing information about the surroundings of the nodes. The TD process gathers the neighbor information of nodes inside the domain and creates a map of every link between any two entities inside the domain. The primary consumers of the topology map are services for routing, policy imposition, etc. TD helps nodes get a better understanding of their neighbors and surroundings. Most of the time, the network is not static and changes over time, such as in ad-hoc networks or wireless sensor networks. Keeping the topology map updated could be challenging considering some networks' volatility. Topology Maintenance (TM) is the process by which we update the topology map and reflect the updates.

With the growing infrastructure, the result of the TD and TM processes can help us to enhance the performance of the nodes in different situations. However, the TD and TM processes themselves also consume considerable bandwidth. There have been solutions proposed for TD in different types of networks to lower the cost of bandwidth consumption. The efficiency of the TD and TM processes greatly depends on the structure of the network. A well-defined approach with high efficiency in a linear network can suffer from low performance in a tree-like network or a clique (fully connected network). Giving a general solution suitable for all types of networks is a difficult task.

The introduced design model for autonomic networks by ANIMA depends on neighbor nodes constantly communicating with each other. The problem of designing efficient TD methods and efficiency in autonomic networks is not well studied. The proposed solution must comply with the design goals introduced by NMRG and ANIMA. In this thesis, we review the different methods for TD in the literature and their compatibility with autonomic networks. We propose two new solutions and implement them on a testbed in our lab.

This thesis is organized as follows. In Chapter 2 we will review the elements of an autonomic network, and in Chapter 3 we will review the existing TD methods. We specify the problem in detail in Chapter 4 and give our proposed solutions in Chapter 5. The results of the experiments are provided in Chapter 6. We conclude the thesis and outline directions for future work in Chapter 7.

Chapter 2

Autonomic Networks

2.1 Architecture

Autonomic networking was introduced as a new network management scheme to deal with configuration, protection, and recovery in the fast-growing network infrastructure. Carpenter et al. [12] suggest that the entities of the autonomic network can be considered as plug-and-play components to the network. This means that the nodes must be capable of organizing themselves with minimum supervision and administration from outside elements. Autonomic networks focuses on minimizing the dependency on human administrators or other central entities [12]. However, the goal here is not to entirely eliminate the human administrator, or the central control entity [7]. Human element and a central entity are needed for tasks such as deciding on or imposing the policy and oversight [7].

In this chapter, we will discuss the design model of autonomic networks; the architecture, the layers, and the connection between the autonomic network elements.

We talked about the motive and goals behind autonomic networks in Chapter 1. NMRG introduced a hierarchical architecture for autonomic networks [7]. The hierarchy inside the autonomic nodes allows them to have a common infrastructure across a group of autonomic nodes to share services provided on the lowest level. Autonomic nodes have their control loops by using the services provided at higher levels of the hierarchy. This shared infrastructure offers a communication facility between the nodes and information for the node's local functionalities. The middle layer allows

access to the shared infrastructure by acting as the controller entity to the nodes. The shared infrastructure provides various services to the nodes. The middle layer is a collection of all the services provided by the shared infrastructure. The middle layer gets the information from the shared infrastructure and provides it to the higher layers [6]. The higher layer consists of atomic entities and autonomic functions that allow the autonomic node to utilize the services the lower layers provide. The proposed architecture consists of the following three layers.

- Autonomic Network Infrastructure (ANI)
- Autonomic Control Plane (ACP)
- Autonomic Service Agent (ASA)

Figure 2.1 depicts the architecture of an autonomic network from the node's point of view and illustrates its different layers.

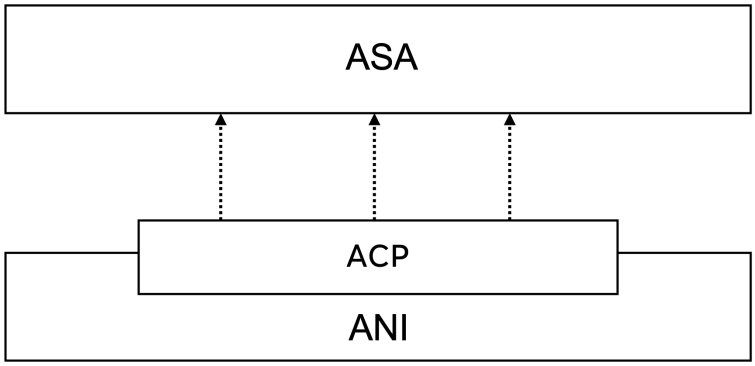

Figure 2.1: Layered architecture of autonomic nodes

In the following sections we will discuss the three layers mentioned above in more detail.

2.1.1 Autonomic Network Infrastructure

As its name suggests, ANI provides the shared infrastructure among the nodes. There are many already-existing autonomic functions. They have their own protocols to communicate, discover services, etc. ANI provides the shared homogeneous infrastructure across all autonomic functions to synchronize values, negotiate parameters, discover services, etc. [7]. The shared set of capabilities

across the autonomic functions inside an autonomic domain is called ANI. ANI is also responsible for holding both node-specific and generic information. ANI provides the required information for autonomic functionalities such as naming and addressing every autonomic node or facilitating communication between the nodes.

2.1.2 Autonomic Control Plane

ACP is a self-configuring communication infrastructure that primarily serves as the control plane for autonomic functions [7]. The ACP is a node-specific entity that provides the path between the ANI and autonomic functions that are trying to use the services from the ANI. ACP is an automatically built communication infrastructure that is secure, resilient, and re-usable [15]. ACP acts as a *virtual out-of-band channel* for autonomic functions [15]. ACP communication is on a hop-by-hop basis, meaning the information and messages are relayed from a node to its neighbors until it reaches its destination. Therefore many interactions between the nodes and services rely on the adjacency table provided by ANI [6, 15]. The ANI is responsible to provide the infrastructures required for actions such as communication and service discovery, and ACP will provide those services to higher layer and the autonomic functions.

2.1.3 Autonomic Service Agent

By using the capabilities of the ANI, we can produce autonomic functions. Autonomic functions form out of atomic entities, which we refer to as ASA [6]. ASAs are installed separately on every autonomic node. There is a one-to-many relation between the autonomic node and the ASA, meaning every autonomic node can install multiple ASAs.

Figure 2.2 illustrates a high-level global view of the autonomic network.

2.1.4 Autonomic Network and Internet Protocol

The components introduced above together form the autonomic networks. Due to some functionalities of the autonomic network, IPv6 is preferred over IPv4 [9]. IPv6 supports some autonomic behavior such as self-configuring addresses on the physical interfaces [6]. In the following section,

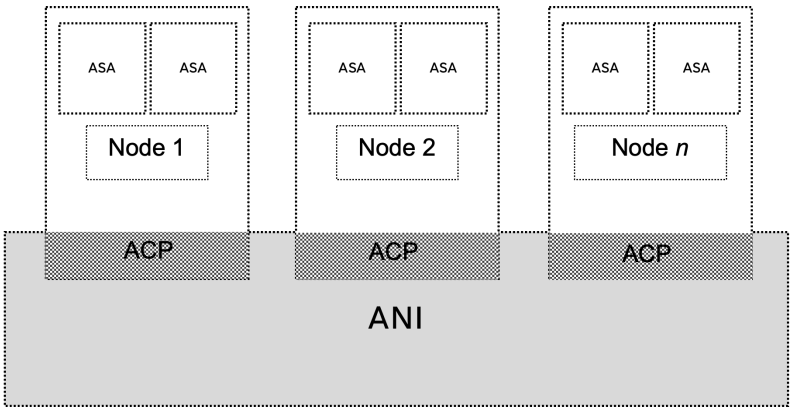

Figure 2.2: High-level global view of an autonomic network

we will discuss IPv6, its address scopes, and why IPv6 is a more suitable internet protocol for autonomic networks.

2.2 IPv6

This section will discuss IPv6 and some of its properties, especially the address scopes that are needed for our work.

IPv6 was introduced as the successor to IPv4 with a larger addressing space, trying to resolve the IPv4 address exhaustion problem. IPv4 can hold 32-bit addressing, while IPv6 is capable of holding 128-bit identifiers for addresses.

2.2.1 IPv6 Addressing

For this research, we discuss two main addressing scopes in IPv6: Link-Local (LL) address and Unique Local Address (ULA).

LL addresses are not routable, but can be generated by a node autonomically. An IPv6 address is represented in 8 groups of 16-bit addresses separated by colons. Each 16-bit group is shown as four hexadecimal digits. Behringer et al. [8] claim that using LL addresses can lead to smaller routing tables and a decrease in risk of attacks by reducing the exposure of the network to malicious nodes. Later, we will discuss how autonomic networks uses this property of IPv6. On the other hand, the ULA is a globally unique address intended for communication within a domain and is

locally routable [13, 16].

2.2.2 IPv6 Link-Local Address Structure

IPv6 LL address configuration is autonomic. In the following we will describe the autonomic behavior of IPv6. The LL address can be generated automatically by using the interface ID. The prefix of an LL address is fixed, *FE80::/10* and the next 54 bits are 0s. The last 64 bits of the address are the interface ID that is generated, using the MAC address (using the EUI-64 format), or it can be 64 random bits. Figure 2.3 from [13] shows the structure of the LL address.

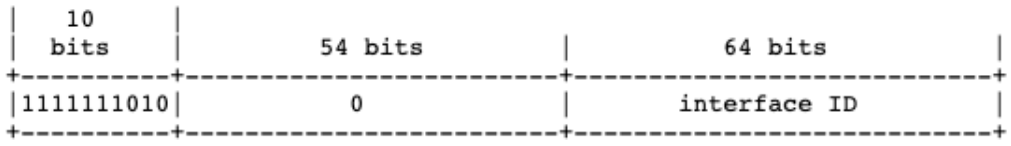

Figure 2.3: Link local Address structure [13]

2.2.3 IPv6 Unique Local Address (ULA)

The ULA is globally unique, but is used for local networking purposes. The uniqueness gives it an advantage that even if leaked outside the local network, it still raises no conflicts with other address spaces or domains [16]. Haberman et al. [16] also suggest that if networks are combined or expanded, since the address is globally unique, we still do not face any conflicts in addresses . ULA has a prefix *FC00::/7* followed by 1 bit local/global identifier, 40 bits Global ID, 16 bits subnet ID, and 64 bits interface ID. Having a specific prefix allows quick filtering [16]. Figure 2.4 from [16] demonstrates the structure of IPv6 ULA address.

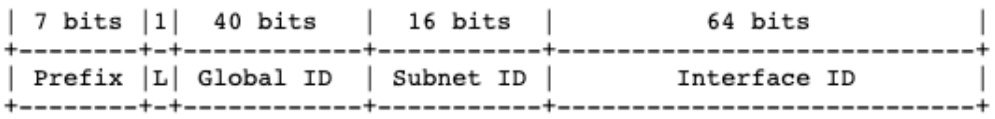

Figure 2.4: Unique local IPv6 address structure [16]

2.2.4 IPv6 Autonomic Behavior

RFC4861 [31] introduces Neighbor Discovery (ND) protocol, which allows nodes to discover their neighbors by determining their LL address. The ND protocol also has an autonomic behavior. The ND protocol does not only return a list of the neighbors; it serves more than that. With its autonomic behavior, ND protocol provides the solution to the following network properties [31]: prefix discovery, parameter discovery, router discovery, address auto-configuration, address resolution, next-hop determination, neighbor unreachablility, duplicate address detection, and redirect. The prefix discovery allows nodes to find the reachable network destinations on each link. Parameter discovery allows the node to figure out the link specifications such as the Maximum Transmission Unit (MTU). Address configuration allows nodes to set the addresses of the interfaces in a stateless manner [31]. Next-hop determination allows nodes to place traffic on a neighbor link toward the final destination. The duplicate address detections allow nodes to verify if an address they want to use is already in use, to prevent a collision.

ND protocol also introduces five types of ICMP messages that facilitate the above-mentioned services. The ICMP messages are as follows [31]: router solicitation, router advertisement, neighbor solicitation, neighbor advertisement, and redirect.

The router solicitation message allows nodes to force the routers to advertise their presence ahead of their scheduled advertisement. The IPv6-enabled routers then send out router advertisement messages upon receiving the router solicitation or reaching the scheduled time. Router advertisement messages include more than the address of the router. They provide nodes with information such as prefixes, address configuration parameters, hop-limit, etc. As its name suggests, the neighbor solicitation ICMP messages are used for finding the LL address of the neighbor or checking whether or not they are still alive. A neighbor solicitation request is followed by a neighbor advertisement response. A node can also send out neighbor advertisements without receiving the neighbor's solicitation, to announce its presence to neighbors. As a part of their autonomic behavior, routers can inform the nodes about a better next-hop by sending redirect ICMP messages to nodes. According to the definitions, some of the ICMP messages, such as redirect or router advertisements, are meant to be used by routers rather than hosts.

2.2.5 Autonomic Network and IPv6

So far, we got familiar with IPv6 and its different address scopes. As discussed in Section 2.1.2, ACP is configured based on the information it receives from its neighbors. By using LL address, the communication between the neighbors takes place without going through the routing table. As discussed in Section 2.1.1, ANI provides communication between the nodes. The peers could be non-neighbor nodes in the domain. Hence ANIMA proposed using both the LL and the ULA addresses for communication purposes such as negotiation and synchronization.

In the following section, we will discuss utilizing these services by a new communication protocol called GeneRic Autonomic Signaling Protocol (GRASP).

2.3 GeneRic Autonomic Signaling Protocol

As discussed in Section 2.1.1, ANI is responsible for providing a shared platform for communication services such as negotiation, synchronization, and discovery. The GenRic Autonomic Signaling Protocol (GRASP) is designed to allow easy expression of these features. The communication between the entities of an autonomic domain, starting from the early stages of bootstrapping to the final steps of ending any communication channel, is facilitated by GRASP. GRASP is an extensible protocol that enables the communication between different autonomic functions and autonomic nodes. Different protocols and mechanisms in autonomic networks, including our suggested methods and implementations, use GRASP to enable autonomic nodes to communicate. This section is dedicated to describing GRASP and its functionalities. GRASP is a communication protocol designed to provide an API to express the architectural concepts according to NMRG's vision of an autonomic network. GRASP introduced a new data structure capable of being labeled and taking values. This data structure is called a *GRASP Objective*. Each GRASP objective will be mapped on to an ASA. GRASP allows us to register ASAs and use the name of the ASA as the unique identifier for future referencing.

2.3.1 GRASP Objectives

In order to provide the information needed for the autonomic functions, GRASP introduced a data structure capable of holding different identifiers and values. Figure 2.5 shows the structure of the objective.

```
objective = [objective-name, objective-flags,
             loop-count, ?objective-value]
```

Figure 2.5: GRASP objective data structure [9]

The two main pieces of information that the GRASP objectives data structure holds are *name* and *value*. The *name* is used as a unique identifier of the GRASP objectives, and the *value* holds the actual value of the GRASP objectives. *Value* can hold any data type, which allows GRASP to support exchanging any data structure across the domain.

Along with *name* and *value*, there are multiple flags accompanying each GRASP objective. The *objective_flags* are

- *Neg*: set to True if the GRASP objective supports negotiation
- *Synch*: set to True if the GRASP objective supports synchronization
- *Dry*: set to true if the GRASP objective supports dry-run negotiation

The other essential field in a GRASP objectives's data structure is *loop_count*, which indicates the maximum number of steps for negotiation purposes.

GRASP restricts the *name* of the GRASP objectives to be in *String* format. The *loop_count* can hold any value (*integer*) between 0 to 255. All GRASP objectives, regardless of their value, must be passed across the domain between the peers in the same byte format. GRASP utilizes *CBOR*, a binary data serialization format [10]. Before the initiator of the communication sends the GRASP objectives, it will convert the GRASP objective's value to CBOR object and then transmit it to the peers. On the receiving side, the receiver will convert the *CBOR object* to the basic format of a *GRASP objective*.

Each GRASP objective must be registered by an ASA. GRASP introduces another data structure called a *tagged objective*. The tagged objective only holds two values, the registered objective and a pointer to the ASA object on which the GRASP objective is registered. The tagged objective is the actual data structure being exchanged inside the autonomic domain.

Finally, GRASP assigns a random port number to each objective that is registered on an ASA. As long as the autonomic node is up and running and the ASA is registered on that autonomic node, the port number will not be affected. Later we will discuss how we can utilize this port number.

2.3.2 GRASP Messages

GRASP supports a variety of messages for different purposes. Each follows a specific format along with a specific set of flags. The messages are exchanged for discovery, negotiation, or synchronization purposes. As discussed in [9], the structure of any GRASP message is as shown in Figure 2.6. Each GRASP message has a `Message_type` identifier. Below we have listed the available types of messages:

- Discovery messages
    - Discovery request message
    - Discovery response message
- Negotiation messages
    - Request negotiation
    - Negotiation
    - Confirm wait
    - End negotiation
- Synchronization
    - Request synchronization
    - Synchronization
    - Flood synchronization

As shown in Figure 2.6, each message holds a `session_id`. The `session_id` is randomly generated by the initiator of the connection whether it is discovery, negotiation or synchronization and the receiving peer must use the same `session_id` for any further messages concerning that communication. The `session_id` is randomly generated and double checked by GRASP to avoid any sort of collision.

```
grasp-message = (message .within message-structure) / noop-message

message-structure = [MESSAGE_TYPE, session-id, ?initiator,
                     *grasp-option]

MESSAGE_TYPE = 0..255
session-id = 0..4294967295 ; up to 32 bits
grasp-option = any
```

Figure 2.6: GRASP message structure [9]

2.3.3 GRASP Discovery

The discovery process allows the autonomic nodes to find peers who have registered a specific GRASP objective on a specific ASA. As a result of the discovery process, the GRASP discovery function will return one or more locators, which hold the value of the ULA address of the corresponding autonomic nodes and the port number on which the objective is accessible. The discovered peer is listening for incoming requests for the same GRASP objective (with the same name) and registered on the ASA with the same name. In order for the GRASP discovery to find a peer (or multiple peers), the name of the GRASP objective and its respective ASA from the peer must match the names from the GRASP discovery initiator. The locator also holds the value of the port number that has been assigned to the objective we are looking for on the corresponding autonomic node. Since the port number is randomly assigned to each objective on each node, it cannot be guessed nor be collected by any other means other than requesting it directly from the peer node. The GRASP discovery request is sent on all interfaces.

The locator is an object of class `asa_locator` that holds the value of the ULA or LL address of the peer, the interface on which the autonomic node must reach out to the peer and an instance of the GRASP objective. The GRASP objective will also hold the port number by which it can be

accessed.

The discovery process is used for finding peers and is often followed by negotiation over the value of the objective. Therefore, for discovery purposes, the objectives must have the *NEG* flag set to true.

The GRASP discovery is a hop-by-hop process, meaning the request will be relayed from one node to another. On this path, if any node satisfies the request, a response will be sent back via the same path and set of nodes that the request took to reach the corresponding peer. On the way back to the initiator, all nodes will update their cache with the locator of the corresponding peer. The GRASP discovery process starts by sending out a discovery message from the initiator on all interfaces. In the first step, all neighbors will receive the discovery request message. Now one of the three following outcomes might occur. The autonomic node receiving `Discovery_Request` holds the requested objective and will send back a discovery response message. As for the second possibility, the receiver does not own the objective and does not have any information cached about other owners that satisfies the discovery request. Therefore, the request will be relayed to its neighbors, except on the interface on which the request was received. As for the third possibility, the neighbor who received the discovery request has previously cached the peers' locators that hold the requested objective. In this case, if the cached information has not yet expired, a response message will be generated, containing the peers' locators from the cached information. The discovery function will return a list (of size one or greater) that contains the locators of the peers that hold that discovery Objective. Later, the locators will be used to initiate negotiations with peers.

The discovery probes will continue spreading into the domain until they find a suitable peer or until they expire. To prevent non-ending search in a large domain, where probes continuously go deeper into the domain to look for the objective, *timeout* was introduced as a parameter of the discovery probe by the GRASP discovery function.

The GRASP discovery is a one-way request. Meaning that, if node u discovers node v by running GRASP discovery, it is not guaranteed that v also finds u.

2.3.4 GRASP Negotiation

The result of the GRASP discovery process will be used for negotiation purposes. For the GRASP negotiation, a negotiation request (M_REQ_NEG) must be sent from the negotiation initiator to its corresponding peer. This request will be followed by a negotiation response or an end_negotiation message from the peer. If the responding peer accepts the offer, the negotiation will end with an Accept message (O_ACCEPT) in an END_NEG message. Otherwise, their peers can continue to negotiate over the value of the Objective by exchanging negotiate messages (M_NEGOTIATE). If the responding peer does not accept the offered value, a decline response message (O_DECLINE) will be sent in an END_NEG message. During the negotiation process, either party can end the negotiation by accepting or declining offered values.

The logic of the negotiation can vary depending on the purpose of the autonomic function. Therefore, it is the responsibility of the network administrator to set the logic behind the negotiation. The two peers will continue negotiating until they reach an agreement or the maximum number of *negotiation_steps* is reached. After the negotiation, no other messages such as ending confirmation will be transmitted.

Figure 2.7 is the sequence diagram of the GRASP negotiation process.

The GRASP negotiation takes a tagged objective, the target peer's locator, as inputs. The locator is obtained from the GRASP discovery process, or the initiator will use the cached information to retrieve the locator of the peer. The mentioned objective must have the *NEG* flag set to true.

As discussed above, GRASP negotiation takes the locator of the corresponding peer as a parameter. Normally, GRASP negotiation comes after the GRASP discovery to obtain the locator of the required peer. But, if we assume that in some way, we are able to collect the port number and the ULA address of the autonomic node that the GRASP objective is registered on, we can create the locator ourselves and establish the negotiation. If the locator is not provided by as a parameter to GRASP negotiation, the discovery will run first and the first peer in the list of found peers will be selected.

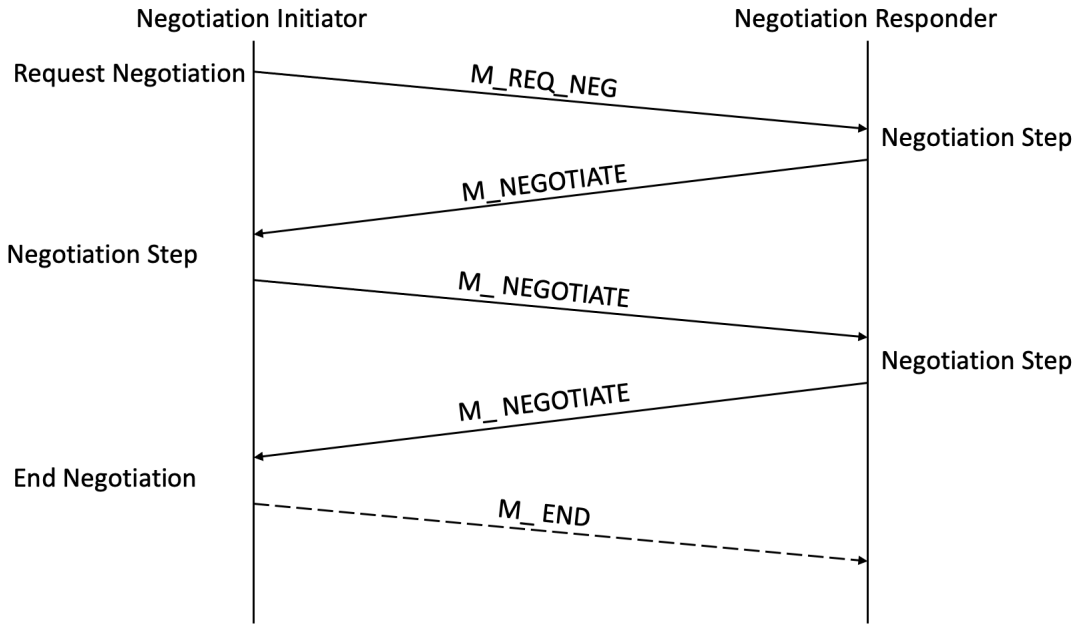

Figure 2.7: Negotiation sequence diagram

2.3.5 GRASP Synchronization and Flood

Negotiation and synchronization are the two main communication methods introduced in [9]. While in negotiation, the focus is on reaching an agreement between two nodes, the synchronization process is not followed by multiple messages negotiating the objective. There are two types of synchronization:"unicast synchronization" and "flood and synchronization". In the case of unicast synchronization, the synchronization initiator opens a connection with the synchronization responder (initiator already aware of the locator of responder) and sends a `M_REQ_SYNCH` message and receives a `M_SYNCH` in response. No further messages will be exchanged. In the case of unicast synchronization, if the synchronization initiator does not know the peer's locator, a *Rapid* mode will be used. In this case, the initiator sends a GRASP discovery message containing the synchronization objective, this enables the discovery message to act as synchronization request. This type of request, allows the discovery responder to reply a synchronization message instead of discovery response to the initiator of the discovery request [9].

When a large group of nodes wish to synchronize the value of the same objective, we use

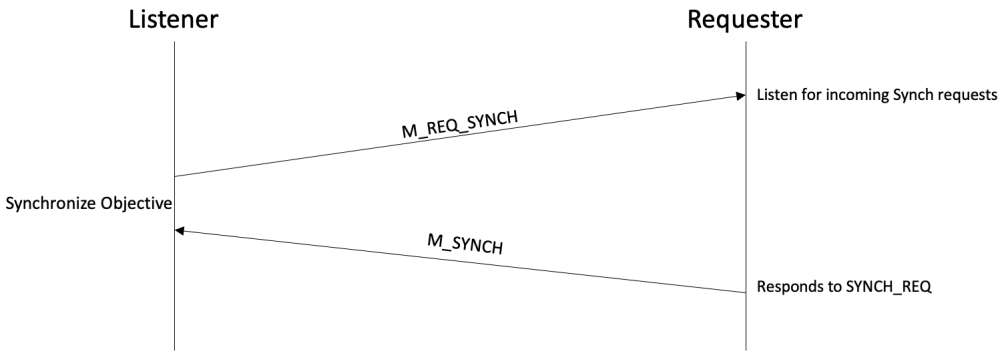

Figure 2.8: Synchronization sequence diagram

GRASP flood and GRASP synchronization. The flood initiator will send an unsolicited flood synchronization message on all interfaces. The flood message is a multicast message to all neighbors. The receiver of the flood will relay this message to its neighbor (other than on the receiving interface). To prevent infinite flooding, GRASP adds a `loop_count`. Upon receiving the flood message, the receiver will decrease the value of `loop_count` by one and then relay it to other neighbors. Any autonomic node will use LL multicast for this purpose [9]. In Section 2.2 we pointed out that the LL multicast is bound to only one hop, but for flooding purposes, we must use LL multicast. Earlier, we discussed that nodes will *NOT* relay the exact same LL multicast messages to other nodes. Each node alters the initial LL multicast, in this case by changing the `loop_count`. Therefore the message is not the same and has been altered.

2.3.6 Discovery Unsolicited Link-Local

For security purposes, GRASP introduces Discovery Unsolicited Link-Local (DULL) for communication in insecure environments. Some actions, such as finding neighbor ACPs through join proxy (discussed in Section 2.4), are insecure since the joining node can be malicious. To prevent this, GRASP limits the communication radius to only one hop in DULL. This means that while using DULL, autonomic nodes will only communicate with neighbors, and any messages in DULL will not be relayed to other neighbors.

The DULL initiator can send discovery or flood synchronization messages. The receiver of those requests will check the loop count of the messages, and if they are greater than one, the message

from the initiator will be discarded. The receiver will also not relay any LL multicast messages since they are only bound to communication between neighbors.

2.4 BRSKI

Carpenter et al. [12] suggests that the autonomic nodes could be looked at as *Plug-and-Play* devices. Since self-securing is one of the characteristics of autonomic networks, each node must be able to find a way to communicate securely and start its ACP autonomously. As mentioned in Section 2.1.2, ACP is a self-configuring entity in the AN, and BRSKI is the bootstrapping protocol used to help the autonomic nodes configure their ACP. Multiple autonomic nodes with different roles are involved during bootstrapping, i.e., pledge, registrar, join proxy, and Manufacturer Authorized Signing Authority (MASA). BRSKI allows nodes to reach the mutual authentication state by exchanging X.509 certificates. MASA, an entity that resides outside the network, plays an essential role in helping nodes reach mutual authentication. BRSKI issues X.509 Certificates with authenticating entities for joining the network to authorize bootstrapping ACP and secure communication with neighbors. X.509 certificate is a public key certificate format mainly used in secure transport protocols, e.g., SSL. The detailed security aspects of BRSKI are outside the scope of this thesis. The main focus is the message flow among the domain entities upon joining.

Upon joining the domain, a new node has no fully operational ACP. The new node that is not yet registered inside the domain is referred to as a pledge. Every domain has a central entity called the registrar, responsible for authenticating any pledge wishing to join the domain. The registrar accepts or refuses the pledge's request to join the domain. The pledge must get in touch with the registrar to get Authenticated, but it has no knowledge of the domain and therefore does not know the locator of the registrar to get in touch with it. In Section 2.4.2 we will talk about the message flow in BRSKI for the pledge to get accepted in the domain.

2.4.1 Manufacturer Authorized Signing Authority

The MASA is not a part of the domain but plays an essential role in the process of authentication of the pledge. The MASA is a representative of the manufacturer of the pledge device. The MASA

keeps track of all the device owners, including current and previous, to verify the ownership. Also, the manufacturer embeds a certificate inside the pledge device that allows the MASA to authenticate the device. The pledge does not directly get in touch with the MASA. The registrar is responsible for asking the MASA to authenticate the device and its owner. The MASA will authenticate the registrar to the pledge as well. So, the MASA plays a vital role in pledge and registrar reaching mutual authentication.

2.4.2 Message Flow in BRSKI

Upon joining the domain, the pledge is looking for the registrar. The pledge only has a limited understanding of its surrounding neighbors. Figure 2.9 shows the sequence diagram of how the BRSKI protocol works.

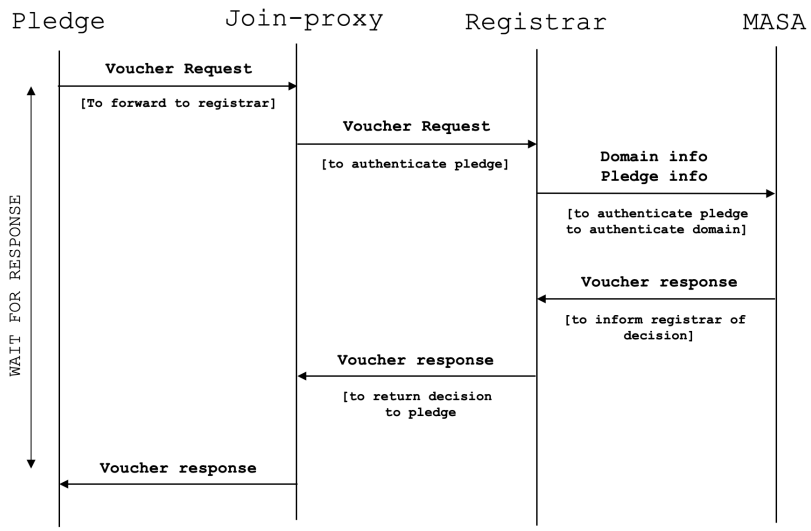

Figure 2.9: BRSKI sequence diagram [30]

In order to get authenticated, the pledge must send a `voucher_request` to the registrar. The pledge can not use GRASP discovery method to locate the registrar due to security measures. It can only be in contact to its directly connected neighbors on the link level and only through LL address before the registrar authenticates it. At this stage, to mitigate the security risks, the communication between the pledge and its neighbors, which are authenticated members of the network, is limited.

For security concerns, the registrar cannot flood its presence since exposing itself might be a security risk that allows malicious nodes to abuse the information. Richardson et al. [30] introduce a new role that can fill the gap between pledge and registrar with a low-security risk called join proxy. The join proxy is the middle node connecting the pledge to the registrar, which is already an authenticated part of the domain. The pledge knows that to get to the registrar, it must first go through the join proxy. Hence, it will start looking for a join proxy among its neighbors using DULL. Upon finding the join proxy, the pledge will send a `voucher_request` message destined to the registrar and relayed through the join proxy. Figure 2.10 illustrates the structure of `voucher-response`. If multiple join proxies are discovered, the first one discovered will be used. Figure 2.10 depicts

```
{
    "ietf-voucher-request:voucher": {
        "assertion": "proximity",
        "nonce": "62a2e7693d82fcda2624de58fb6722e5",
        "serial-number" : "JADA123456789",
        "created-on": "2017-01-01T00:00:00.000Z",
        "proximity-registrar-cert": "base64encodedvalue=="
    }
}
```

Figure 2.10: Voucher request structure [30]

the structure of a voucher request that contains information about the pledge device, such as "serial number". A nonce is stored in the `voucher_request` to guarantee the freshness of the request. As the pledge receives the confirming voucher request, it will act as join proxy [6, 30].

The join proxy will forward the `Pledge_Request_Voucher` to the registrar. The registrar will run a set of preliminary authentication operations of the request and forward it to the MASA after adding additional information about itself for authentication purposes.

In Section 2.4.1, we discussed how the pledge does not contact the MASA directly, and it is the responsibility of the registrar to forward the `Pledge_Request_Voucher` to the MASA.

The MASA will also check the `Pledge_Request_Voucher` and the additional information from the registrar. Upon successful authentication of the pledge device and the registrar, MASA will inform both pledge and the registrar in a single message. The part meant for the pledge device will be encrypted using the same keys that the MASA embedded in the pledge. If the request is not valid, meaning that the pledge device or the registrar is fraudulent or the information sent to MASA

is incomplete, a message indicating that the request is invalid will be sent back to the registrar and pledge. If the registrar is willing to accept the pledge, it will send back the `Voucher_Response` through the same join proxy. The pledge will receive the response, and if it is a valid one, the pledge will then join the domain and configure its ACP. Henceforth the pledge is a part of the domain and can communicate securely with other entities using the certificate issued by the registrar.

Chapter 3

Topology Discovery

Topology discovery is the process of finding the connections between every pair of entities in a network. In Chapter 1 we mentioned the importance of TD in network management. The output of this process can allow the autonomic nodes to understand their surroundings better and make the processes such as routing or failure-recovery faster and more efficient.

The other use case of the TD results is in Software Defined Network (SDN). The SDN is a new approach toward central network management. SDN proposes separating the control plane from the data plane to enhance the network management experience [11]. By controlling the software of the network devices, e.g., SDN-enabled switches or routers, controlling and monitoring the network entities will take place much more easily through a central SDN controller. Islam et al. [19] suggests the benefits of knowing the topology of the network for multicasting in SDN. For classical multicast, the destinations are not known. The individual routers only know there are one or more interested receivers connected to one of its interfaces. Essentially, a set of shortest-paths are developed, from the source to the entire set of destinations. Islam et al. [19] discusses in context of an SDN, the centralized controller can have a view of the overall topology, and can therefore set up the switches to provide different optimizations. So, the operation of SDN in general depends on the central controller knowing the actual topology of the network, and multicast SDN can make use of topology information to make application-dependent optimizations.

Donnet et al. [14] categorizes TD in three main layers. The overlay TD, the network level (internet level) TD, and link-level TD. The network level TD focuses on the higher level TD in the

internet. Networking is not just about the physical connection between two end nodes in a single domain. Today's internet is much larger and is about connecting different networks. Routing can be done inside a domain between two entities in the same subnet by using well-known protocols such as OSPF [26] (a link-state routing protocol) or RIP [17] (a distance-vector routing protocol), or it can be done between two independently operating networks with different addresses and subnets by using protocols such as BGP [22]. In the case of link-state routing protocols, the nodes are fully aware of the topology of the network and use that information to decide on the shortest path. As for distance-vector protocols, nodes will have a partial information about the topology available to them. The overlay topology focuses on the P2P connection of a system [14], and the link-level TD focuses on the physical connection between two nodes. The link-level TD is the basis of many network management tasks [14]. The link-level TD describes how devices in a network are connected. The link-level TD will provide all connections between two entities.

In Chapter 2, dependency of autonomic nodes on their neighbors and their constant communication were discussed. The constant communication takes place for different purposes such as enhancing routing or updating/exchanging configuration parameters. Since the physical connection between the autonomic nodes is critical, methodologies that investigate physical connectivity between the nodes are more studied in this research.

Another way to categorize TD is based on how the TD method collects, stores, and distributes the information. All the mentioned actions could be done in a centralized or a decentralized manner. In a central approach, all the information is gathered by a single node and stored on the same node. The role of the central management entity is predetermined by the network administrator or decided by an election process. In distributed computing the leader election is the process in which all the nodes participate in choosing a single node as their leader. They all send data to or communicate with that single selected node instead of broadcasting messages to all nodes. The selected leader node can be distributing the data and responding to requests or facilitating the communication between two nodes [32]. Heuristics depend on random factors or node-specific information such as the node id (integer) in the network. For example, the node with greatest id value can be selected as the leader. However, the efficiency of the election process correlates with the size of the network. As the network size grows, the time and the number of messages required for this process

also grows. The second approach is a distributed approach, where different nodes take part in collecting the topology information from the other nodes. The processed information can be stored in a distributed manner among several nodes. The storage of the topology map can be different in distributed systems. All nodes responsible for holding the topology map can have replicas of the the topology map. This type of data storage is referred to as replication in distributed systems terms; or, each responsible node can hold a part of the topology map. The aggregated stored topological data from all the nodes, which are storing a part of the map, gives the full topological map of the network.

This chapter is dedicated to reviewing different methodologies for TD and checking whether or not they can be applied to autonomic networks. We can categorize the physical TD methodologies based on the features of the network, such as wired or a wireless network or being centrally managed or not. Depending on the type of network, different methodologies can be used.

Different methods can be used in different stages of TD or TM. For example, we can propose a new method for the setup phase of the topology map, and we can use the diagnostic tools for TM purposes. In the following sections, different approaches will be briefly discussed, and some of the most well-studied methodologies will be explained along with their limitations.

3.1 Existing Diagnostic and Management Tools

Topology discovery has been investigated in many contexts, and many of the existing methodologies depend on traditional tools and protocols such as Simple Network Management Protocol (SNMP), Internet Control Message Protocol (ICMP), and Address Resolution Protocol (ARP) [1, 34].

SNMP is used for managing a network and controlling the devices by collecting information and organizing it. ICMP is used for diagnostic and monitoring purposes in IP networks [29]. ARP is used to map the IP address to the physical address of the interfaces, i.e., MAC address [28]. Yin et al. [34] discuss the limitations of the above internet protocols for TD purposes.

3.1.1 ICMP for TD

ICMP requires prior knowledge of the network. This means that the supposed network controller must have a complete list of the entities of the network. That is why ICMP is not a suitable approach for TD. The controller can use ICMP messages to check whether or not the entity is alive. As described in Chapter 2, autonomic nodes must be able to configure themselves upon joining the network without any prior knowledge of the network entities and must only obtain setting parameters through communicating with their authenticated neighbors [7]. There are also security concerns regarding ICMP messages, e.g., firewall blocking and spoofing ICMP messages. Therefore, we suggest using ICMP messages during the TM process for networks that are not dynamic and have a stable connection.

3.1.2 ARP for TD

The ARP is used for resolving the IP address to the MAC address of the interfaces. Nevertheless, ARP cannot find the connection between the entities in the network. It can help us find the existing entities, but as mentioned, it cannot generate the map of the network that represents the connection between the entities. Also, in reasonably large networks, ARP will be inefficient as it cannot represent all the entities due to the size of the ARP table [34]. However, if needed, ARP can be used to discover neighbors' IP addresses and map them to their respective MAC addresses.

3.1.3 SNMP for TD

SNMP is a standard IP protocol used for management purposes by gathering and organizing the information of the network. For SNMP to work, both peers (the SNMP server and the client) must support the SNMP as SNMP is not always enabled for security reasons. Much redundant information can be transmitted as SNMP was not explicitly designed for TD purposes. Also, SNMP requires prior knowledge of the network.

3.1.4 `traceroute` for TD

Other tools can be used to discover the network's topology, like `traceroute`. Like ICMP, `traceroute` is also a diagnostic tool in an IP network that allows us to find the path and round trip time from the source to destination. Like previous methods, `traceroute` also requires prior knowledge of the network IP addresses, and it is not capable of representing the full connectivity between the nodes.

3.1.5 IPv6 Network Discovery Protocol

In Section 2.2 we discussed the IPv6 address structure and its advantages over IPv4 in autonomic networks, which is why NRMG chose IPv6. Another approach to finding the network's topology is gathering information from the ND protocol. IPv6 allows nodes to find their neighbors and actively checks their reachability by sending specific ICMP messages discussed in Section 2.2. Most communications in ND protocol is limited to communication between the neighbors by sending simple ICMP messages. Therefore, it does not consume a considerable amount of bandwidth. However, the information is still local to a node and its neighbors. We require a central node to collect this information and process it to generate the topology map of the network.

3.2 TD in Centrally Managed Networks

In the previous sections, the limitations of the traditional tools were discussed. Earlier in this chapter, SDN was introduced. There has been a thorough investigation of different TD methodologies in SDN networks [35]. This section is dedicated to investigating some of the most well-known TD and TM approaches in SDN and checking whether or not the methods apply to autonomic networks.

3.2.1 OpenFlow Discovery Protocol

McKeown et al. [25] initially introduced the OpenFlow protocol for research purposes to allow researchers to run tests in heterogeneous networks regardless of their vendors. OpenFlow works on the forwarding level of the switches or routers and allows the controller to specify the path

each packet takes. OpenFlow protocol is the primary communication protocol between the SDN controller and the other entities [4]. OpenFlow is a widely used method for TD purposes, but it has some limitations. OpenFlow is vulnerable to many attacks, e.g., link-fabrication, switch spoofing, etc. [4].

Azzouni et al. [4] introduced a secure TD method based on OpenFlow protocol in SDN called Secure OpenFlow Topology Discovery Protocol (sOFTDP). Figure 3.1 will help us demonstrate how the proposed method works.

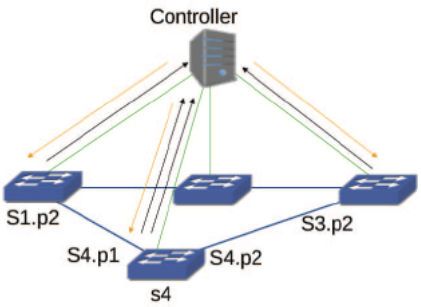

Figure 3.1: Different steps of sOFTDP [4]

The security measurements taken for the sOFTDP are out of the scope of this thesis. We concentrated more on the message flow between the entities in order to demonstrate how the topology map is generated. The proposed solution by Azzouni et al. [4] starts by sending out LLDP packets. LLDP packets are used for informing neighbors of the node's information. This information may include chassis ID, port ID, etc. It is sent out from each interface periodically.

Upon a new switch joining the network, the controller and the switch will start exchanging `OFPT_HELLO`. This message is followed by a `feature_request` from the controller. The `feature_response` includes the status of the ports, which are all set to *down* initially. Consider Figure 3.1, when all four switches join the controller and then set up the links between themselves, and their port status will change from *down* to *up*. The switches will then inform the controller about their ports' status change. The controller will then start sending out LLDP packets to the *up* ports on the switches with a specific path starting from one switch and ending in the controller to determine whether or not there is a path between the switches. After receiving the LLDP packets, the controller can generate the topology map.

To figure out the paths between any two entities in the network, a specific LLDP packet must be generated. For each path between any pair of nodes, this process must take place. Therefore this method is highly inefficient and time-consuming. Also, it requires a central node commanding the other nodes to take actions, which is in contradiction to the desired autonomic behaviour.

3.2.2 Self-healing TD Protocol for SDN

Ochoa-Aday et al. [27] introduced the Self-Healing TD (SHTD) Protocol that allows nodes decide for themselves and act autonomously during the failure recovery phase. This method categorizes nodes and ports (interfaces) on each device. Each node in this SDN can hold one of the following states: non-discovered node, leaf node, v-leaf node, and core node. The SDN's controller goal is to create a control tree in which each node knows its parent node and path to the controller. A non-discovered node is a node that has not yet received any requests (`topoRequest`) to join the control tree. A leaf node is an external node, and a v-leaf node has all its ports connected to other leaf or v-leaf nodes. The remaining nodes will be considered as core nodes [27].

As discussed by Ochoa-Aday et al. [27], each port can have one of the following states: standby port, parent port, child port, and pruned port. A port is a standby port when is not a part of the control tree. A port is considered a parent port when it is an upstream port and initially received a `topoRequest` message on that node. A port is a child port when it is a downstream port and has received an `echoReply` message. A pruned port is a child port that has received a `topoReply` message and now is connected to a leaf or a v-leaf node.

The TD process starts with a probing method. Initially, all nodes before being discovered are in a non-discovered state. First, the SDN controller will send out `topoRequest` messages to its directly connected subordinates. The receiving node will then reply to the source of the `topoRequest` with an `echoReply`. The `echoReply` helps to determine the Round Trip Time (RTT) and also contains information that allows the receiving node to announce to its neighbors which node it has joined. The receiving node will then start to relay the `topoRequest` message to other neighbors on all interfaces except the one on which it received the request. Depending on the state of the receiving port, whether it is a `standby` port or a `non-standby` port, nodes will act differently. In the former case, the node that receives the `topoRequest` messages will set

the node that sent the request as its parent, and in the latter case, the receiving node will drop the request. This is done to avoid sending redundant information by nodes that have already joined the tree. By doing so, the network will gradually create a tree, and nodes that have joined the tree will periodically send out neighboring information to the SDN controller through their parent. So far, this method follows a probing method for creating the topology map of the network. Ochoa-Aday et al. [27] proposed a solution for self-recovery that is autonomous.

Upon failure of a node in the network, neighbors will take different measures according to the state of their ports. If any standby, child, or pruned ports fail, the failure will be directly reported to the SDN controller since their parent port is still alive and a part of the control tree. If a parent port fails, the node must autonomously decide on recovery on the forwarding level. Since the node has been separated from the control tree, it has no routes to the controller; therefore, it must first find a new parent. The failure recovery starts by sending out `topoUpdate` messages on all interfaces (except the one that has failed) to find a new path to the SDN controller. If the ports that receive the `topoUpdate` have any active parent ports, they will reply with a `replyUpdate`; otherwise, they will relay the `topoUpdate` to their other neighbors. In this method, the nodes will take action by communicating with their neighbors, similar to autonomic networks. The autonomic failure recovery process is similar to the self-recovery goal introduced by ANIMA. But, since the context is defined in SDN, the network needs a central management entity that performs heavy tasks on the network and still has control over the nodes inside the network. Also, SHTD protocol is more focused on the self-healing purpose than on generating the topological map of the network.

3.2.3 Other TD Methods

There has been much research on TD in wireless networks, especially in ad-hoc linear sensor networks such as the proposed method for linear sensor networks by Jawhar et al. [20]. These methods mostly depends on the signal strength to detect close neighbors. Some of the proposed solutions in ad-hoc linear sensor networks such as Jawaher et al. [20] or Ali et al. [2], are purpose-built solutions, designed for linear wireless networks. We are looking for a more general solution capable of supporting any arrangement of the nodes. For the purpose of this thesis, we have investigated mostly wired-connected networks.

3.3 Distributed Methods for TD

As the network grows in size, the performance of centralized TD algorithms drops. Meaning that since the central entity must handle more messages, the wait time for other nodes will increase to get their response from the central entity, and hence decrease the performance of the overall network. For a more scalable approach, we must look into a distributed solution. In distributed systems, nodes can have more freedom to make their own decisions and act more autonomously. Also, by distributing the data and the processing power across the nodes, the network can experience better performance. In this section, we focus on the distributed method for TD. In distributed methodologies, each node or a selected group of nodes is responsible for collecting, processing, and distributing the neighboring information of all nodes or a selected group of nodes. For example, by using the link-state routing protocols, each node can generate a topological map of the network, but there is a certain limit to this approach. Clustering is known to be the first stage of solving many distributed problems. By breaking the problem space and the data into multiple pieces, we can balance the workload of a distributed network among multiple nodes. Even for centrally managed networks, clustering can be a good solution to improve their performance. The following section provides a definition of clustering and its use cases for TD.

3.3.1 Clustering

In previous sections, we discussed autonomic networks and TD, the two main concepts in this research. According to centrally managed methods mentioned above, centralized TD can be a heavy task on the network due to the significant number of messages that are exchanged and the considerable amount of bandwidth it consumes. Clustering is the process of grouping a set of nodes together based on a similar feature under the same cluster representative, which we call cluster head. Nodes in the same cluster can have similar behavior, similar functionality, same goal, reachable within a specific radius of the cluster head, etc. It helps the network to break into non-overlapping clusters and distributing the collection of data, process of data, and distribution of the data among the clusters. Clustering gives us the advantages such as increased processing speed, network extensibility, easier management, minimizing storage for communication informations,

optimizing bandwidth consumption, etc. [5].

Dali et al. [33] categorizes the clustering algorithms based on four main criteria: One-hop or multi-hop, synchronous or asynchronous, location-based or non-location-based, stationary or mobile. Considering the formation of nodes in the network, the network's purpose, and the nodes' features, a clustering scheme will be chosen.

The initial step in every clustering method is to identify the characteristics of the underlying network and then the network will move to the next phase, which is cluster head selection. The selection of cluster heads can depend on many features. One of the most common and easy methods for cluster head selection is the use of the device ID. The device with the lowest or the highest ID number in a range of k hops, will be selected as cluster head. Lin et al. [24], and Amis et al. [3], use the lowest and the highest device ID to select their cluster heads respectively. But, depending on the network a selection mechanism based on more than one attribute can be used for choosing the cluster head. Zhang et al. [23] introduce a metric based on multiple factors to select the cluster heads. After the cluster heads are selected, they will make their neighbors aware of the change in their state, and wait for nodes to join their clusters. After the clusters have been shaped, nodes will move to the next phase, which is cluster maintenance. During the cluster maintenance phase, nodes can leave their cluster and join other clusters, new nodes can join or leave the network. Reconstruction of the clusters is a costly task for the network. It takes time and a considerable number of messages. Therefore, the cluster reconstruction must be justified perfectly to not affect the performance of the network.

3.3.2 Cluster-based TD in Vehicular Ad-Hoc Networks

Zhang et al. [23] propose a k-hop, location-based clustering protocol for TD in a Vehicular Ad Hoc Network (VANET). By definition, a VANET has a dynamic topology, meaning the nodes leave and join the network non-deterministically often. Zhang et al. [23], use the highest connectivity, vehicle mobility, and host ID as metrics to select the cluster head. The vehicle mobility metric is location-based, yielded from the GPS information (location), velocity, and transmission range. Since nodes tend to send beacon packets periodically to neighbors to announce their presence, they are made aware of each others' presence. Using all the above metrics, the average link expiration

time between 2 mobile nodes in a VANET will be calculated. Zhang et al. [23]. provide a detailed description of how to calculate the link expiration in their paper. Zhang et al. [23] argue that the vehicle's number of neighbors is important in selecting the cluster head since the proposed solution focuses on reducing the number of small clusters. Each vehicle node has a predefined link expiration metric threshold. If the calculated link expiration is greater than the predefined link expiration metric threshold, the node will announce itself as cluster head. Before announcing it to other neighbors, a winning metric will be generated based on the average link expiration metric calculated previously, and the number of neighbor vehicles. The winning metric will be broadcast to neighbor vehicles in the broadcasting range of the node. The receiving nodes have one of the two following states.

- The node has already announced itself as cluster head and broadcast its winning metric to other neighbors
- The node has not announced itself as cluster head

In both states, the node will continuously listen for incoming winning metrics from other nodes. As for the nodes that are in the former state, if their winning metric is less than the one received, they will update the value of their winning state to the received value, and as for the nodes in the latter state, they will set their winning metric to the one received. The receiving nodes will then start to broadcast the winning metric of the cluster head to their neighbors to make sure all nodes in the k-hop distance of the cluster head will receive the winning metric. Once the nodes have broadcast the information to other neighbors, any node willing to join will respond to the cluster head with the convergence packets, including device ID, cluster head ID, and their neighboring list. The process of sending convergence packets starts from the furthest nodes, referred to as cluster-border nodes. Each node waits until it receives the information from the nodes that are further from the cluster head, and after receiving their convergence packets, the node will update its cluster information and send the convergence packets to the cluster head. Upon receiving the convergence packet from all nodes, the cluster head will generate a complete list of the neighboring nodes and send it back to the nodes inside its cluster. The cluster head will provide the local connectivity table to the nodes. The link connectivity table contains the following information: distance to cluster head, neighbors of each node, gateway to neighbor clusters. Member nodes will use a link-state routing

protocol to generate the cluster's topology map based on the information from the link connectivity table. The member nodes will also generate an inter-routing table using Dijkstra's shortest path by using the information from link connectivity table to other clusters. Autonomic nodes have no prior knowledge of the network. The proposed solution in this section embeds a link expiration metric threshold and the range of the cluster head into the device based on the network's and device's characteristics. Hence, it contradicts the design goal for autonomic networks proposed by NMRG.

3.3.3 Distributed Clustering for Ad-Hoc Networks

The proposed solution in this section is a clustering scheme rather than focusing on TD. In the next chapter we will discuss the reason behind introducing this clustering scheme.

Basagni [5] introduces a one-hop clustering algorithm that allows nodes to decide roles themselves without any controller. The algorithm is a single-hop, non-location based, asynchronous clustering scheme, which can support mobile behaviour in nodes as well. Each node will decide whether or not it is a cluster head based on its own weight and its neighbors' weight. The heaviest node among neighbors that are exactly one hop away will be considered the cluster head.

Each node goes through the initial phase and determines if it is a cluster head. The node will let its neighbors know if it is a cluster head. The autonomic nodes with heavier neighbors will wait to receive updates from their heavier neighbors to check which heavier node to join if one or multiple heavier neighbors declared themselves as cluster heads. This decision is made upon receiving updates from all of their heavier neighbors.

In the situation where none of the heavier neighbors announce themselves as cluster heads, and they want to join other clusters, the node will announce itself as the cluster head. After any change in the roles of the nodes, update messages will be sent to the neighbors. The changes include a node announcing itself as cluster head, a node joining a cluster, and a node being aware of link failure or new links.

After each node successfully decides its role, it moves to the maintenance phase. Each node will listen for updates from their neighbors' roles or check their reachability. Upon receiving any updates during the maintenance phase, one of the situations mentioned below might happen. For clarification purposes, please see Figure 3.2.

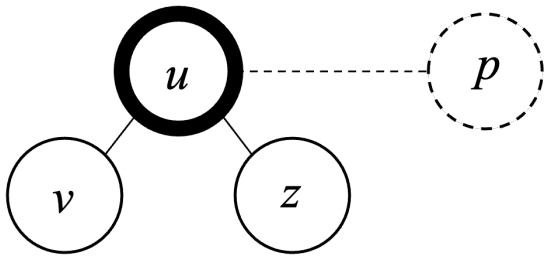

Figure 3.2: Node *u* is the cluster head to nodes *v* and *z*. Node *p* has not yet joined the network.

- *u* is aware of node *v* leaving the network: Node *u* removes node *v* from the neighbor list and updates the map.
- *v* is aware of *u* leaving the network or changing its role to a non-cluster head node: *v* will first update its neighbor list (if *u* has left the network) and then starts looking for a new cluster head or announces itself as the cluster head if there are no other heavier cluster heads among the neighbors.
- *v* is aware of *z* leaving the network: Since *z* is neither *v's* cluster head or *v* is not *z's* cluster head, node *v* simply removes *z* from the neighbor list.
- *u* is aware of a new link to *p* it will first update its neighbor list: If *p* is heavier than *u* and announces itself as cluster head, *u* will join *p* and update its neighbors.

Chapter 4

Problem Statement

Having access to the topological map of the network is an essential part of network management. Having an updated version of the topology map allows the administrator, nodes, and any management entity in the network to make faster and more efficient decisions. Decisions such as failure recovery, updating routing tables, applying policies, etc., are some of the actions that depend on having access to an updated topology map. The importance of TD in network management was discussed in more detail in Chapter 1.

To our knowledge, there has been no prior research on TD in autonomic networks. This is the first time that TD and TM solutions and their challenges and benefits are being investigated in autonomic networks. A primary goal for this research is to investigate whether TD is helpful to autonomic networks. It must be investigated what type of information we should provide to the nodes, how we are providing it, and how autonomic nodes will use that information.

In Chapter 1, it is discussed that the autonomic network is expected to be self-configuring, self-protecting, self-healing, and self-optimizing. These characteristics can directly depend on the topology of the network. For example, similar to the described TD method in Section 3.2.2, if a link to a specific node goes down, the autonomic node will move to a failure recovery state. If the disconnected autonomic node has a good understanding of the network's topology and it has multiple options to chose from, it can look for possible neighbors that have the path with the least cost to the controller and update the routing respective to the information it obtained from the topology map. Knowing the topology of the network here will reduce the cost of message passing

between the autonomic node and the controller. The mentioned example is one of the examples out of many that show how TD can help autonomic networks to enhance their performance.

At this point, the next step is to find a way to collect, process, and distribute the topological information efficiently. As discussed in Chapter 3, centrally managed methods are more likely to be used for TD purposes such as being easier to implement and maintain. However, such methods require that each node, no matter how many hops away, must be in touch with the network's controller. In Chapter 1 and Chapter 2, we discussed how autonomic nodes are in constant communication with their neighbors, as a design goal proposed by NMRG. Therefore, it is important to propose a solution that mostly depends on communication with the neighbors. The TD approach we propose should attempt to minimize the number of messages that being exchanged for TD and TM purposes.

By decreasing the number of TD and TM messages, we can use less bandwidth for control and management purposes and leave most of the bandwidth for data transfer. By decreasing the time for all nodes to converge to the same state regarding the topology map, all nodes can make decisions faster and will not face conflicts.

Another important aspect of this thesis is investigating how the topology map must be stored. In Chapter 3, most TD methodologies store the topology map in a central node. However, storing the topology map in a centralized fashion in autonomic networks may lead to lower network efficiency and more bandwidth consumption. Therefore, we need to find the best possible solution for ways of storing the topology map in autonomic networks as well.

All the goals mentioned above focus on increasing the efficiency of the network and leaving more bandwidth for the data to be transferred. In Chapter 5, we will discuss our proposed solutions that will help an autonomic network to gather, generate and maintain information on the domain's topology.

Chapter 5

Proposed Solution

In previous chapters, we discussed autonomic networks and TD, the two main concepts in this research. The autonomic network is the infrastructure that we are using, and our goal is to generate the topology map of the network by using the facilities provided by the infrastructure.

According to methods reviewed in Section 3, TD can be a heavy task on the network due to the significant number of messages that are exchanged and the considerable amount of bandwidth it consumes. In Chapter 3 different TD approaches have been introduced. The TD approaches must be designed based on the characteristics of the underlying network. For the purpose of this research, autonomic network constitutes the infrastructure on which we are working. Different attributes of autonomic networks were discussed in Chapter 2. Autonomic network elements constantly communicate with their directly connected neighbors to exchange information such as configuration parameters, etc. Therefore, a more local approach for gathering the topological information of each node is more appropriate. However, as described in Section 2.4, in a fully working autonomic network, BRSKI is in charge of the security aspects of the network. Upon joining the domain, each node must contact the registrar through a join proxy to verify its authenticity. Therefore, in a fully working autonomic network, nodes are aware of the presence of the registrar, a central entity in charge of the security of the network. We have proposed solutions for both scenarios explained above. In this thesis, we described both centralized and decentralized solutions for TD in autonomic network. As for the purpose of this research, we consider the network to be stationary during the TD setup phase but during the TM phase nodes are allowed to leave or join the network.

To be able to implement the solutions and communicate between the nodes, we used an implementation of GRASP that was written in the Python programming language by B. Carpenter [1]. We used the communication services from GRASP such as GRASP discovery and GRASP negotiation mainly, to achieve our implementation goals.

5.1 Solution Based on Clustering

Our first proposed solution for TD is based on clustering approach. The goal here is to group the nodes into non-overlapping clusters and present a node from each cluster as its representative. Definition of clustering and different clustering methods have been provided in Section 3.3. As noted in Chapter 1, we base our solution on the version introduced by NMRG and standardized by ANIMA.

For the purpose of this research we decided to employ a clustering scheme similar to the described clustering methods in Section 3.3.3, which is single-hop, asynchronous, non-location based. As for the mobility aspect of the clustering scheme, we propose a hybrid model. Depending on the state of the nodes, the clustering algorithm considers the network to be stationary during clustering setup phase and supports nodes leaving or joining during the maintenance phase. The described clustering scheme is single-hop since autonomic nodes are already aware of their neighbors LL addresses and depend on the updates they receive from neighbors. Since autonomic nodes work independently, we consider them as asynchronous. The actual location (e.g. yielded from GPS) of the nodes is not a factor in the TD process and the physical connection is important. Thus, the proposed clustering solution for TD is a non-location-based clustering scheme.

After each node joins the network, the ACP can provide the autonomic functions with information about the neighbors. The ACP utilizes IPv6 ND protocol to provide a list of all physically connected neighbors and will return a list of the LL address of all neighbors for each node. The ACP can provide the ULA address of the neighbors to the autonomic node as well.

At the early stage, when the node boots up, the node has preliminary information about its surroundings but does not know anything beyond its directly-connected neighbors. In order to

[1] https://github.com/becarpenter/graspy

enable the nodes to exchange their neighbor list, we store a list of the neighbors, provided by the ACP, as a part of the value of a GRASP objective. Since the value of this objective is going to be negotiated, the negotiation flag of this objective is set to true upon registration on an ASA. Along with the neighboring information, a weight is assigned to each node. The weight of each node can depend on different attributes, such as the number of interfaces, the aggregated bandwidth, etc. Similar to Zhang et al. [23], the number of interfaces (indicating the number of direct neighbors) is chosen as a metric to calculate the weight of the autonomic nodes. Having larger clusters will reduce the number of clusterheads. Therefore, it results in a simpler clustered network and more efficient inter-cluster communication. We assume the weight of a node does not change during its existence in the network.

Other cluster-related attributes are stored along with the neighbor list and weight, such as the port number assigned to each objective registered on the ASA, a list of members of the node's cluster (if the node is selected as clusterhead), the locator of the node's clusterhead (if the node is not a clusterhead), and a flag indicating the node is a clusterhead. All the node related information is stored in variable, local to each node. The variable is a map container (dictionary in Python), which is called `node_info`. The key of the map (dictionary) is the name of the node's attribute and the value for those keys are the value of node's attributes. The structure of `node_info` is shown in Figure 5.1. As depicted in Figure 5.1, each node has a $weight$ yield of the product of

```
node_info = {
  'weight': NUMBER_OF_INTERFACES*random(0, 1),
  'clusterhead': False,
  'clusterhead_location': None,
  'cluster_set': [],
  'neighbors': NEIGHBORS_LIST,
  'ports':{
    'obj1': 0,
    'obj2': 0,
    'obj3': 0,
    ...
  }
}
```

Figure 5.1: Structure of `node_info`

the number of interfaces with a random number. The randomness allows us to have better tests and experiments. The $clusterhead$ is initially set to $False$, but after the initial clustering phase, the value changes. If the node is a clusterhead, the value of $clusterhead$ will change to $True$. Later, by checking the value of $clusterhead$, neighbors will understand that the node is a clusterhead. If a node joins a cluster, the value of $clusterhead\_location$ will change to the ULA address of the cluster's clusterhead. A clusterhead will store the members of its cluster in the `cluster_set` list. The node will store an updated list of its neighbors in the $neighbors$ list. The $neighbors$ lists are maintained by the ACP and are always up to date. As discussed in Section 2.3.1, a port number will be assigned to each GRASP objective upon being registered on the ASA. The port numbers are initially set to 0, but as the objectives get registered, their respective port numbers will also update.

The structure of a GRASP objective that holds the value of `node_info` is depicted in Figure 5.2. This objective will later be used to exchange information between the nodes including neighboring information, cluster-related information, etc.

```
CLUSTERING_OBJECTIVE = {
    name :         'topo_obj',
    discoverable : True,
    neg :          True,
    dry :          False,
    synch :        False,
    loop_count:    10,
    value :        cbor.dumps(node_info)
}
```

Figure 5.2: Topology discovery GRASP objective

The objective from Figure 5.2 will be registered on an ASA of an autonomic node. After an objective is registered, its value can be modified and updated. As discussed Section 2.3.1, upon registering an objective on any ASA, a random port number is assigned to that objective. As long as the objective is registered on the same ASA and on an active autonomic node, this port number does not change. After the `topo_obj` is registered on the ASA, the value of its port number will be stored in the `node_info` attribute, initially. Then, the value of the `topo_obj` will be updated by serializing `node_info` using `cbor`. At this stage, the local variables and objectives required for a communication are defined and created. In the following we will describe how nodes are going to

utilize them.

The state machine from Figure 5.3 demonstrates a node's different states during the TD and TM processes. The state machine below, shows the path each node takes during TD and TM.

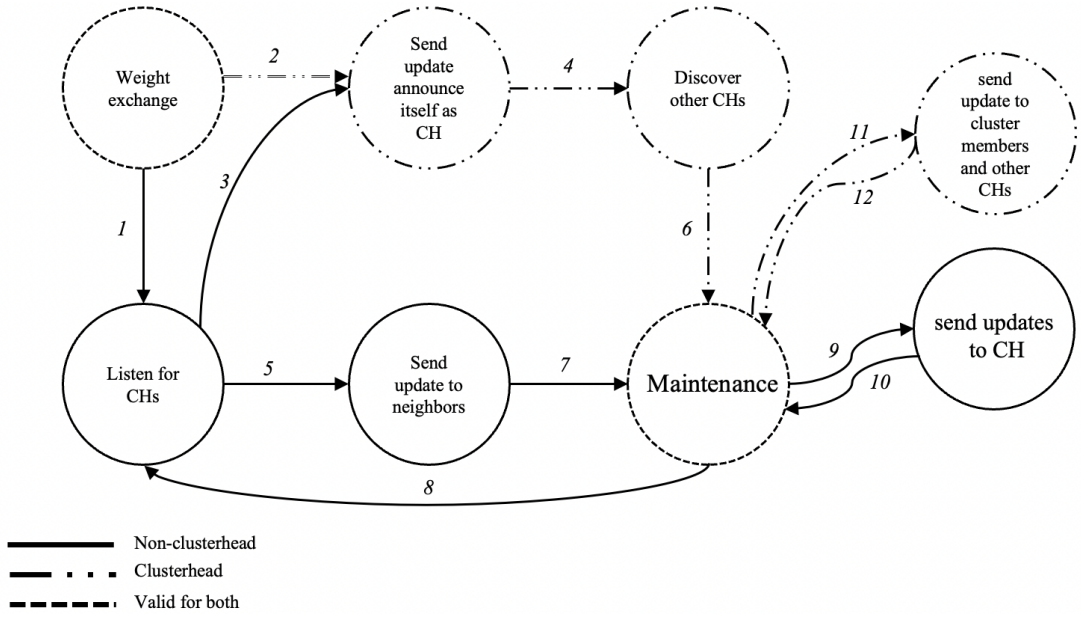

Figure 5.3: TD using clustering state machine

On each arrow of Figure 5.3 there is a number. The numbers refer to an event. The events that might happen are listed below.

(1) Once the weight of all neighbors is received via GRASP negotiation, the node will run the initial procedure to decide its role. If heavier neighbors exist, the node will move to the next state to listen for clusterhead announcements from its neighbors.

(2) Once the weight of all neighbors is received via GRASP negotiation, the node will run the initial procedure to decide its role. If the node is the heaviest node among neighbors, the node will proceed to the next state to announces its change of role to clusterhead.

(3) All heavier nodes joined another cluster themselves, the node introduce itself as clusterhead.

(4) The node is a clusterhead and tries to discover other clusterheads.

(5) The node joins a cluster and the neighbors know it has joined which clusterhead.

(6) The node is a clusterhead and goes to maintenance phase; to start listening for any types of update e.g. link-failure, new links detection, etc.

(7) The node is not a clusterhead and goes to maintenance phase; starts listening for any types of update e.g. link-failure, new links detection, etc.

(8) The link to clusterhead fails; move to looking for new cluster state again.

(9) The node is not a clusterhead and reports update to its clusterhead.

(10) The node is not a clusterhead and after reporting returns to maintenance phase.

(11) The node is a clusterhead and notices an update; update the topology map, updates its peer clusterheads and then updates its subordinate nodes.

(12) The node is a clusterhead and after reporting updates to clusterheads and non-clusterheads will return to maintenance phase.

5.1.1 Setup Phase

Before the clustering process begins, each node will exchange its weight with its neighbors through GRASP negotiation. During this cluster setup phase, nodes are to be stationary. After a node has received all the weights from its neighbors, it will move to the setup phase. The pseudo code for the setup phase is provided in Algorithm 1.

During the setup phase, as described in Section 3.3.3, each node will autonomically decide whether it is a clusterhead. If the node has the heaviest weight among its neighbors, it will announce itself as clusterhead by letting its neighbors know about its role change.

Autonomic nodes are responsible for letting their neighbors know about any updates. This method of updating is referred to as the *push model*. There are two different approaches when it comes to updating others. The two approaches are *push model* and *pull model*. In the former, each entity in the system will send an update to its peers upon changes, whereas for the latter approach, an entity will request updates from its peers. The pull model is usually done periodically, while the push model is *trigger-based*. An observer constantly checks for changes in *trigger-based* systems.

Algorithm 1 Initial phase for clustering setup on node u

```
role_decide = False                                   ▷ Used only for internal use
while ∀ v ∈ neighbors, Weight(v) is Null do           ▷ Did not receive weight from neighbors
    WAIT                                              ▷ Wait until the weights from all neighbors are received
end while
for ∀v ∈ neighbors do
    if Weight(v) ≥ Weight(u) then
        heavier_neighbors.append(v)                   ▷ Store heavier nodes in a list called Heavier_neighbors
    else if Weight(u) > Weight(v) then
        lighter_neighbors.append(v)                   ▷ Store lighter nodes in a list called Lighter_neighbors
    end if
end for
if SizeOf(heavier_neighbors) is 0 then
    is_clusterhead = True                             ▷ variable is_clusterhead is used only for internal use
    clusterhead = True                                ▷ If u is clusterhead, set boolean is_clusterhead to True
    role_decided = True
    for ∀v ∈ neighbors do
        Send(v) : I am clusterhead
    end for
end if
cluster_heads = DiscoverClusterheads()                ▷ The result will be stored in cluster_heads list
```

If the observer notices any changes, it will follow a pre-defined protocol to take action about the change that has been made. Since the changes in a network are not deterministic, following the trigger-based approach and push model is more reasonable. Upon any changes, the autonomic nodes are responsible for reporting updates to their neighbors or clusterhead. After a node changes its role to clusterhead, it will start looking for other clusterheads by running GRASP discovery for other clusterheads. In the implementation provided for this solution, each node initially exchanges weight and node specific information by running GRASP negotiation over `topo_obj`, which is a GRASP objective. Upon changing its role to clusterhead, a new GRASP objective will be registered on the same ASA, called `cluster_obj`. This objective will be used by clusterheads to discover each other and negotiate `topology_map` of the network.

After the initial phase the clusterheads will update their neighbors. Non-clusterhead nodes will start listening for incoming updates from their neighbors to check which one has changed its role to clusterhead. Clusterhead nodes will start listening for updates to keep track of the nodes trying to join them.

Algorithm 2 shows the pseudo code for handling clusterheads announcements on nodes who

were not decided as clusterhead during initial state.

Algorithm 2 listening to updates from neighbors on non-clusterhead node u

```
if Not is_clusterhead then
    while ∀v ∈ heavier_neighbors, RoleDecided(v) is False do
        wait                    ▷ Wait until all heavier neighbors decide and announce their role
    end while
    heaviest_candidate = u
    for ∀v ∈ heavier_neighbors do
        if Weight(v) > Weight(heaviest_candidate) and isClusterhead(v) then
            heaviest_candidate = v
        end if
    end for
    if heaviest_candidate is not u then
        is_clusterhead = False
        clusterhead_location = v
        role_decided = True
        for ∀q ∈ neighbors do
            Send(q) : I joined v        ▷ Let other nodes know which clusterhead node u joined.
        end for
    else if candidate_to_join is u then
        is_clusterhead = True
        clusterhead = True ▷ none of the heavier nodes announced themselves as clusterhead,
therefore, node u announces itself as clusterhead
        role_decided = True
        for ∀q ∈ neighbors do
            Send(q) : I am clusterhead
        end for
    end if
end if
```

Consider node u a non-clusterhead node. Node u will wait until it receives updates from all its heavier nodes. The reason is that the chosen clustering scheme is a one-hop clustering method. The heaviest node in a neighborhood of nodes will be selected as the clusterhead. Therefore, each node must wait for its heavier nodes to announce themselves as clusterheads. According to Algorithm 2, each node will decide whom to join among its heavier neighbors, which have announced themselves as clusterheads. The heaviest neighbor, a clusterhead, will be selected as the clusterhead, and u will join it. Upon joining a clusterhead, u will announce this update to all its neighbors so that the role of u will be clear to all neighbors, especially the lighter neighbors. The decision-making of lighter nodes depends on whether their heavier nodes are clusterheads. By announcing to all neighbors that

u has joined a cluster and is not a clusterhead, lighter nodes can remove u from their list of possible clusterhead options to join. In a situation in which none of the heavier neighbors of u announce themselves as clusterheads and decide to join other clusters themselves, u will announce itself as clusterhead and make its neighbors aware of it.

In a very rare situation, where the weight of all the neighbor clusterheads is the same, the metric that is used to join a clusterhead is the size of its `cluster_set`. The non-clusterhead node will join a clusterhead that has a smaller `cluster_set`. The goal of clustering methods is to maximize the size of each cluster as much as possible. Therefore, joining a smaller cluster will allow the distribution of the cluster sizes to be normal, to prevent one cluster to grow considerably larger in size than other clusters.

5.1.2 Maintenance Phase

After the roles have been decided, nodes will move to the maintenance phase. During the maintenance phase, depending on their role, nodes will take different measures towards taking action according to the updates they receive. The messages a node receives will be either from their direct neighbor or from a peer clusterhead only. The non-clusterhead nodes will only receive messages from their neighbors, which include their clusterhead as well. The clusterheads on the other hand, receive messages from both their neighbors and their non-neighbor clusterhead peers.

Algorithm 3, Algorithm 4, and Algorithm 5 demonstrate how different types of nodes will react to incoming requests. Each of these algorithms according to their context will run in a separate thread in parallel to other processes. Based on the information embedded in the values of the incoming requests and the role of the receiving node, nodes will chose their next steps.

Consider node u is receiving an update message during the maintenance phase from node v. If v is either u's clusterhead or its non-clusterhead neighbor, the update sent from v will be stored in a map container called `neighbor_info`. If v is a new neighbor and there is no record of v in `neighbor_info`, a new record will be added to `neighbor_info` and then the value of the incoming request will be stored in the `neighbor_info`. If v is the clusterhead of u, one of the following situations might happen. If the update from v does not include any information about v changing its role from clusterhead to non-clusterhead, u will update its topology map according

Algorithm 3 Handling incoming requests from v at non-clusterhead node u during maintenance

if $v \in neighbors$ **And** $v \notin neighbor\_info$ **then**
  $neighbor\_info.append(v)$ ▷ New node added to neighbors_info map.
end if
$neighbor\_info[v] = incoming\_request$
if $clusterhead\_location$ **is** v **And** $IsClusterhead(v)$ **then**
  Update $topology\_map$ ▷ Clusterhead sent an update.
end if
if $clusterhead\_location$ **is** v **And Not** $IsClusterhead(v)$ **then** ▷ Current clusterhead announce joining another cluster, therefore it is no longer a clusterhead and u must look for a new clusterhead.
  Call $FindClusterhead$ ▷ Call Algorithm 2
  return
end if
if **Not** $clusterhead\_location$ **is** v **And** $Weight(v) > Weight(clusterhead)$ **And** $IsClusterhead(v)$ **then**
  $clusterhead = v$ ▷ Changing the clusterhead.
  for $\forall q \in neighbors$ **do**
    $Send(q) : I\ joined\ v$ ▷ The current and the previous clusterheads of u are one-hop away, therefore, they will be notified of the update by this message like other neighbors of u.
  end for
end if

to the update sent from v, its clusterhead. However, if the incoming update from v, which is u's current clusterhead, announces that v has changed its role from clusterhead to non-clusterhead, u will change its state from maintenance to finding a new clusterhead. If v is not u's clusterhead, but it is a clusterhead in the network, and its weight is greater than the weight of u's clusterhead, u will join v and let its neighbors, including its clusterhead, know by sending them update messages.

If u is a clusterhead, depending on the role of v, one of Algorithm 4 or Algorithm 5 will be selected to execute. If v is **NOT** a clusterhead, then Algorithm 4 will execute. Otherwise, if v is a clusterhead, Algorithm 5 will execute.

Similar to Algorithm 3, the value of the incoming requests is stored in a different container. If the incoming request is from a clusterhead node, then the value of the incoming request is stored in $cluster\_heads$, a container similar to $neighbor\_info$ that is responsible for storing the updates from peer clusterheads. The clusterheads can still receive updates from neighbors, regardless of being their clusterhead or not. The neighbor information is stored in $neighbor\_info$. For both mentioned containers, if the incoming request comes from a new clusterhead or a new neighbor,

Algorithm 4 Handling incoming requests from the non-clusterhead node v at clusterhead node u during maintenance

Require: **Not** $IsClusterhead(v)$
  if $v \in neighbors$ **And** $v \notin neighbor\_info$ **then**
    $neighbor\_info.append(v)$ ▷ New node added to neighbors.
  end if
  $neighbor\_info[v].update(incoming\_request)$ ▷ Updating information of v.
  if $v \in cluster\_set$ **And Not** $Clusterhead(v)$ **is** u **then**
    $cluster\_set.remove(v)$ ▷ v is no longer a member of u's cluster_set.
    update $topology\_map$
    for $\forall q \in cluster\_heads$ **do**
      $Send(q) : topology\_map$
    end for
    for $\forall p \in cluster\_set$ **do**
      $Send(p) : topology\_map$
    end for
  end if
  if $Clusterhead(v)$ **is** u **then**
    if $v \notin cluster\_set$ **then**
      $cluster\_set.append(v)$ ▷ A new node just joined u's cluster_set.
    end if ▷ If v is already recognized as
    update $topology\_map$ ▷ Update topology map according to the value of the new incoming request.
    for $\forall q \in cluster\_heads$ **do**
      $Send(q) : topology\_map$
    end for
    for $\forall p \in cluster\_set$ **do**
      $Send(p) : topology\_map$
    end for
  end if

a record in the containers will be created for them, and then their value will be stored. If v is not a clusterhead but it has u as its clusterhead, if v is not a member of u's cluster_set, it means v has recently decided to join u. Node v will be added it the cluster_set of u. Upon changing the cluster_set, the topology map will be updated and distributed among member of cluster_set and other clusterheads. If v is a member of u's cluster_set, or v is a known peer clusterhead, the update received from v will be applied on the topology_map and then the updates will be sent to all members of cluster_set and other discovered clusterheads.

A clusterhead changing its role is a costly task on the network. Clusterheads together create the backbone of the network. Clusterheads are selected based on more stable features, which means

Algorithm 5 Handling incoming requests from a clusterhead node v at clusterhead node u during maintenance

Require: $IsClusterhead(v)$ **is** $True$
  if $v \in neighbors$ **And** $v \notin cluster\_heads$ **then** ▷ A (new) neighbor has announced itself as clusterhead. Upon meeting the conditions, the clusterhead may change it role and join the new neighbor, otherwise, two clusterheads can co-exist as neighbors.
    if $v \notin neighbor\_info$ **then**
      $neighbor\_info.append(v)$
    end if
    $neighbor\_info[v].update(v)$
    if $Weight(v) > Weight(u)$ **And** $SizeOf(clusterSet(v)) > SizeOf(cluster\_set)$ **then**
      $is\_clusterhead = False$
      $clusterhead\_location = v$
      for $\forall p \in neighbors$ **do**
        $Send(p) : I\ joined\ v$
      end for
      return
    end if
  end if
  if $v \notin cluster\_heads$ **then** ▷ A new clusterhead v found u and is sending updates to u; This case also include the newly joined neighbor clusterhead as well;
    $cluster\_heads.append(v)$
  end if
  $cluster\_heads.update(v)$
  update $topology\_map$ ▷ update topology according to the value of update from v
  for $\forall q \in cluster\_heads$ **do**
    $Send(q) : topology\_map$
  end for
  for $\forall p \in cluster\_set$ **do**
    $Send(p) : topology\_map$
  end for

they are less likely to change their roles [23]. Basagni's [5] solution for clustering is only dependent on the weight of the neighbors. This may lead to inefficient clustering, a large number of messages exchanged, etc. Consider a star topology as shown in Figure 5.4. Node u is the clusterhead of all the connected nodes. Upon node v joining the network, according to the DMAC algorithm, u joins v, and all u's subordinates will lose their clusterheads. Since nodes 1 to n do not have any other neighbors than u who have now joined v, they will announce themselves as clusterheads. This leaves the network with $n + 1$ clusterheads. This autonomic decision-making based on a single factor is not efficient. Therefore, for a clusterhead to join another cluster, the weight of a new neighbor is not the only factor for changing its role.

If a clusterhead u is notified by the presence of another clusterhead v, it will check both the weight and size of its cluster set. If both have greater values, then, in that case, u will withdraw from its role as clusterhead and join v. This case happens only for newly added nodes. Because during the initial phase, as discussed in Section 5.1.1, only the heaviest node among all of its neighbors will announce itself as a clusterhead. Only in this case co-existence of two clusterhead will be allowed.

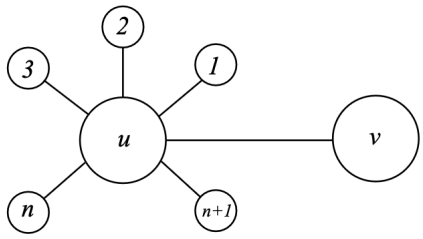

Figure 5.4: Star topology with u as clusterhead and nodes 1 to n a member of u's cluster. Node v is heavier than node u

Similarly, for a clusterhead to change its role and join another clusterhead there must be some metrics more than just comparing the weight of the nodes. Consider v to be a clusterhead. Upon establishing a new link between u and v, two clusterheads became neighbors. If v is heavier than u and the size of its `cluster_set` is greater than the size of u's clusterhead, u will change its role to a non-clusterhead node and join v. It will announce joining v by sending updates to all neighbors, which include the new clusterhead v and its previous members of `cluster_set`, since they are all neighbors. In the case that the aforementioned condition is not met, the two clusterheads will co-exist.

After clusterheads have discovered each other, they will start negotiating over the value of the `cluster_obj` objective, which holds the value of the current version of the map of each cluster. After the first round of negotiation among clusterheads, each clusterhead will review the list of all the discovered clusterheads again and check whether or not the current version of the map is the same as the one previously received by the peer clusterhead.

It may take more than one iteration, based on arrangement of the nodes and their distance, for all clusterheads to have the same topology map. In the worst-case scenario, consider a linear network of *n* nodes. Since we are using a one-hop clustering approach, clusterheads are at most three hops

away from each other, as shown in Figure 5.5 and Figure 5.6.

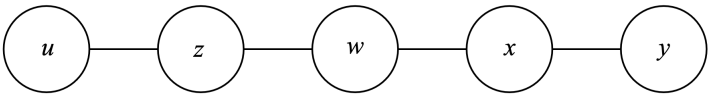

Figure 5.5: Sample graph with clusterheads only 2 hops away; nodes *u* and *w* and *y* are clusterheads and node *z* has joined node *u* and node *x* has joined *y*.

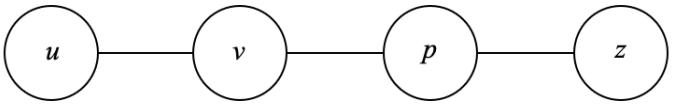

Figure 5.6: Sample graph with maximum clusterheads distance; nodes *u* and *z* are clusterheads, *v* and *p* joined *u* and *z*, respectively.

The number associated with each node in Figure 5.5 represents the weight of the nodes. In Figure 5.5, the weights of the nodes are sorted as $Weight_u > Weight_z > Weight_x > Weight_y > Weight_w$.

According to the proposed clustering scheme, nodes z, and x will wait for heavier neighbors u and y respectively, to announce their roles as clusterhead. Node w will wait for nodes z and x to announce their roles. In the first iteration of updates, nodes u and y will announce themeselves as clusterheads and nodes z and x will join them respectively. Although node w has the least weight in the network, but since all of its heavier neighbors have joined another clusters, it will announce itself as clusterhead. As you can see, the maximum distance between clusterheads is two hops. In Figure 5.6 the nodes can be sorted as follow based on their weights: $Weight_u > Weight_z > Weight_p > Weight_v$. However, according to Figure 5.6, after the initial phase nodes *u* and *z* will announce themselves as clusterheads. Therefore, this type of network gives us the maximum distance possible between any two clusterheads that is three hops away in a one-hop clustering approach. In the worst-case scenario in a linear network, clusterheads are three hops away. If there are *n* clusterheads, it will take *n-1* iterations for the last node's data to reach the first node. In this scenario, we consider the message passing process to be atomic, meaning that they are not interrupted.

Upon any changes happening in the network, the clusterhead is made aware of the changes itself, or the subordinate nodes will report it to their clusterhead. For updating other nodes we use the push model. The clusterheads maintain the latest version of the topology map for each node based on the latest negotiations. After receiving an update message that affects the topology map, the clusterhead will first update its own version and then check with all neighbor clusterheads and subordinates. If their versions differ, the clusterhead will send an update messages to them. Algorithm 6 is the pseudo-code for clusterheads sending update.

Algorithm 6 clusterheads sending updates

```
for ∀v ∈ cluster_heads do
    if topologyMap(v) ≠ topology_map then
        topology_map.update(topologyMap(v))
        send(v) : topology_map
    end if
end for
for ∀z ∈ cluster_set do
    if latest_map_received[z] ≠ topology_map then
        topology_map.update(topologyMap(v))
        send(v) : topology_map
    end if
end for
```

If the update includes a subordinate node leaving the domain, similar to the proposed algorithm in Section 3.3.3, clusterhead will simply take care of that by removing it from its neighbors list and the cluster. If the clusterhead is the one leaving the network, again following the same behavior as described Section 3.3.3, the subordinates will change their state to initial and either look for a new clusterhead or announce themselves as clusterheads.

As a part of maintenance, each node is responsible for checking whether or not their peers (neighbors or other clusterheads) are reachable. Neighbor reachability or clusterhead reachability can be done in two ways. Each node will have three tries before it concludes its peer is out of reach. If the peer does not respond to the updates the node sends to it, after three failed attempts, the node will recognize that neighbor is unreachable and remove it from the list of neighbors or clusterheads, updating the topology and neighboring information and send an update to the accessible peers. The other method to check peer reachability is by using the ICMP messages. ND protocol and ping can be used to check the reachability of the neighbor nodes, but only ping can be used to check the

availability of peers who are more than one hop away. Similar to the previous method, after three failed attempts in responding to ICMP messages, the peer would be considered out of reach, the topology will be updated, and an update will be sent to the other accessible peers. This process takes place periodically on a separate thread, depending on the node's role. Clusterheads will use Ping to reach out to other clusterheads, and all nodes will use ND protocol to maintain their neighboring list and observe changes in their neighboring list.

5.2 Utilizing BRSKI for TD

The solution discussed in Section 5.1 was a distributed solution in which nodes did not have to report anything to a central controller since no central management entity was assumed. As discussed in Section 2.4, before starting any secure communication between any two nodes in a fully working autonomic network, the BRSKI protocol must be executed. BRSKI allows pledges and the registrar to achieve mutual authentication and securely communicate to each other or other verified autonomic nodes in the domain. The details of BRSKI protocol can be found at Section 2.4.

In this section, we describe a centralized solution for TD. We propose taking advantage of the BRSKI protocol and the participating nodes in its process, i.e., the registrar. This solution collects the neighboring information in the registrar during the authentication process. Centrally managed networks have some advantages over distributed networks. They are easier to manage, monitor, maintain, etc. In centrally managed networks, the network manager role is either predetermined by the administrator or nodes in the network must go over the election process that is discussed in Section 3. BRSKI introduced four new roles to the autonomic network: registrar, pledge, join proxy, and MASA [30]. We take advantage of the registrar and add a new network management task, i.e., collecting, processing, and distributing the topology map of the network. The proposed solution greatly impacts the number of messages exchanged for TD purposes and can enhance the TM efficiency.

The BRSKI protocol has been standardized by the IETF [30]. Altering its semantics would produce a non-conformant implementation. We consider two scenarios: one with conformant BRSKI semantics, and one with non-conformant semantics. It will be up to the implementer to decide

whether or not to use the non-conformant approach in order to gain the advantages.

5.2.1 TD with BRSKI (without altering BRSKI semantics)

In the first scenario, where the semantics of BRSKI cannot be altered, the TD setup phase takes place right after the authentication finishes. At this point, the pledge has received credentials and certificates from the registrar, and both the registrar and the pledge have authenticated each other mutually. Upon successful authentication, the pledge will become a part of the domain and bootstrap its fully working ACP. Figure 5.7 is the sequence diagram for the proposed solution.

To collect the TD information in this scenario, the registrar will register an objective called `td_obj` to later use for negotiation purposes with newly joined authenticated nodes. The `td_obj` will also be registered on the newly authenticated pledges after they have started their ACP. On the registrar, the value of `td_obj` holds the full topology of the network and the port number on which `td_obj` can be accessed. On the newly authenticated pledge, the value of `td_obj` holds its neighbor list, the port number of `td_obj` on which it can be reached, and an updated version of the topology. Initially, the value of the topology map is set to null since the pledge has not yet communicated with the registrar. Before authentication, the pledge had the list of its neighbor LL addresses for each interface. However, after authentication, it can obtain the ULA address of its neighbors by requesting it from its ACP. Upon having a complete list of its neighbors, the newly authenticated pledge will establish a negotiation session with the registrar to send its `td_obj`. The pledge communicates with the registrar through a join proxy. Therefore, it does not have the locator of the registrar and must discover it by running GRASP discovery. The join proxy already has the locator of the registrar since it is an authenticated member of the network. If the newly authenticated pledge runs GRASP discovery to find the locator of the registrar, the join proxy will return the cached record of the registrar locator. The GRASP discovery message does not need to go all the way to the registrar, and the newly authenticated pledge can use its neighbor's cached information as long as the cached information is not expired. Upon finding the locator of the registrar, the newly authenticated pledge will start a GRASP negotiation session with the registrar to report its neighbors list and get the network's topology from the registrar.

On the registrar's side, upon receiving the negotiation request from the newly authenticated

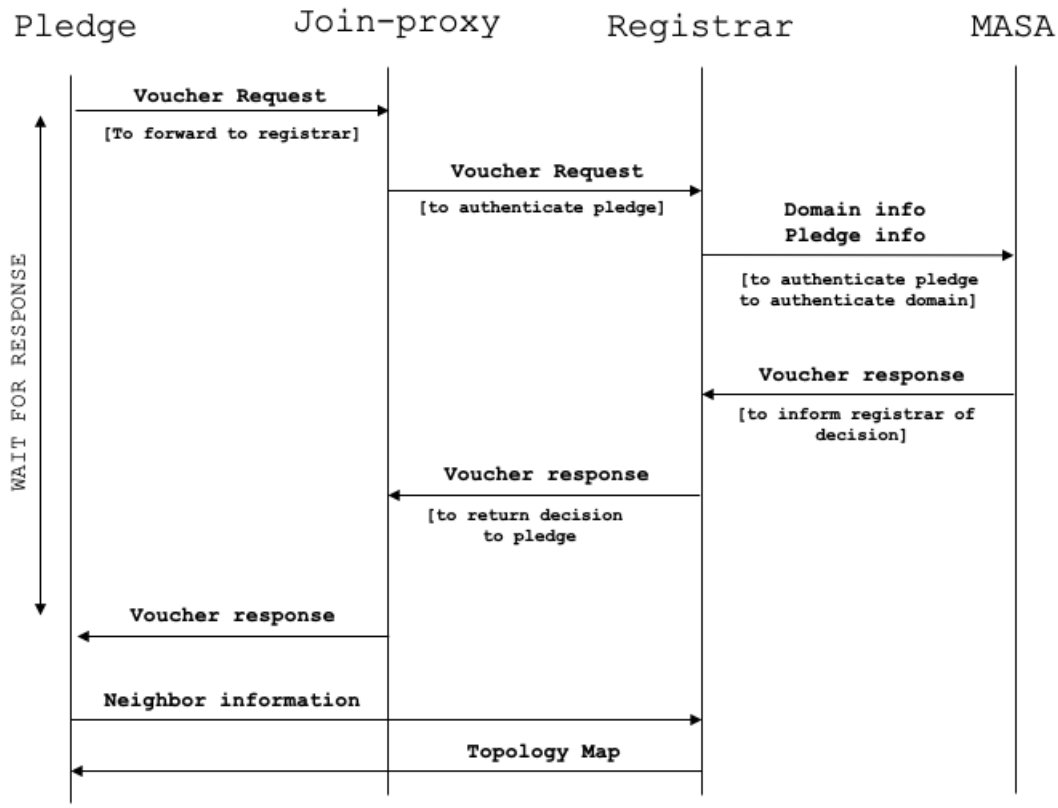

Figure 5.7: Using BRSKI for TD after authentication of the pledge

pledge, the topology map will be updated according to the offered value by the negotiator initiator.

Suppose it is the first time that the newly authenticated pledge is contacting the registrar. In that case, the port number of `td_obj` will also be recorded and mapped to the newly authenticated pledge's ULA address. After recording all the information from the offer, the registrar will send the updated topology map to the negotiation initiator. The newly authenticated pledge will collect the registrar's response and update the topology map's value in the `td_obj`.

As soon as the pledge gets authenticated and receives the voucher response and the updated version of the topology map, it moves to the maintenance phase and starts listening for events. As discussed in Section 2.4, each pledge, upon getting authenticated, will act as a join proxy to other nodes. During the maintenance phase, the join proxy will be notified by two kinds of events. First, a pledge is trying to join the network and using the node as a join proxy. In this case, the join proxy is aware of the presence of the pledge but will take no further actions regarding reporting this event

to the registrar since the pledge is not yet authenticated, and if the pledge gets authenticated, the registrar will send an updated version of topology to all nodes. Therefore, sending an update to the registrar and notifying the registrar of the existence of a new pledge will be considered a redundant message. The second type of event is noticing the presence or leave of an *authenticated* neighbor. In this case, since there are no observers for the network to report such cases to the registrar directly, the authenticated node notified by this event will report it to the registrar, and the registrar will process the update and send out updated topology information to all authenticated nodes.

Initially, the registrar can start flooding the updates since every node will receive the same replica of the topology map or send out update messages individually to each node. In the former case, the number of redundant messages will increase, and the nodes must listen for flood messages only from the registrar on a separate thread. In the latter case, the nodes can listen for any incoming update messages from authenticated nodes and not just from the registrar.

Similar to SDN, in this centralized approach, the registrar can process the incoming information, and based on the network's topology, it can form clusters, create control trees, etc. For example, upon collecting the neighbor list of a node, the registrar can make clusters and let the node know that it is a clusterhead or will join another cluster. In this case, the registrar will update only the clusterheads. Acting as the central network manager will reduce the number of connections the registrar needs to establish and the number of requests it receives, which leads to reducing the workload on the registrar.

5.2.2 TD with BRSKI (with altering BRSKI semantics)

The second proposed solution based on BRSKI is only applicable if the semantics of BRSKI is altered. The modification that needs to be done is on the level of the exchanged messages between the pledge and registrar. The pledge knows the LL address of its neighbors before getting authenticated and joining the network. At this stage, the pledge can prepare a list containing information about the connectivity between its interfaces and neighbors. The pledge must map the LL addresses of its interfaces to the LL address of the connected interfaces. Figure 5.8 demonstrates the sequence diagram for this proposed solution.

By the time the pledge wants to generate the `voucher_request` message, it should include

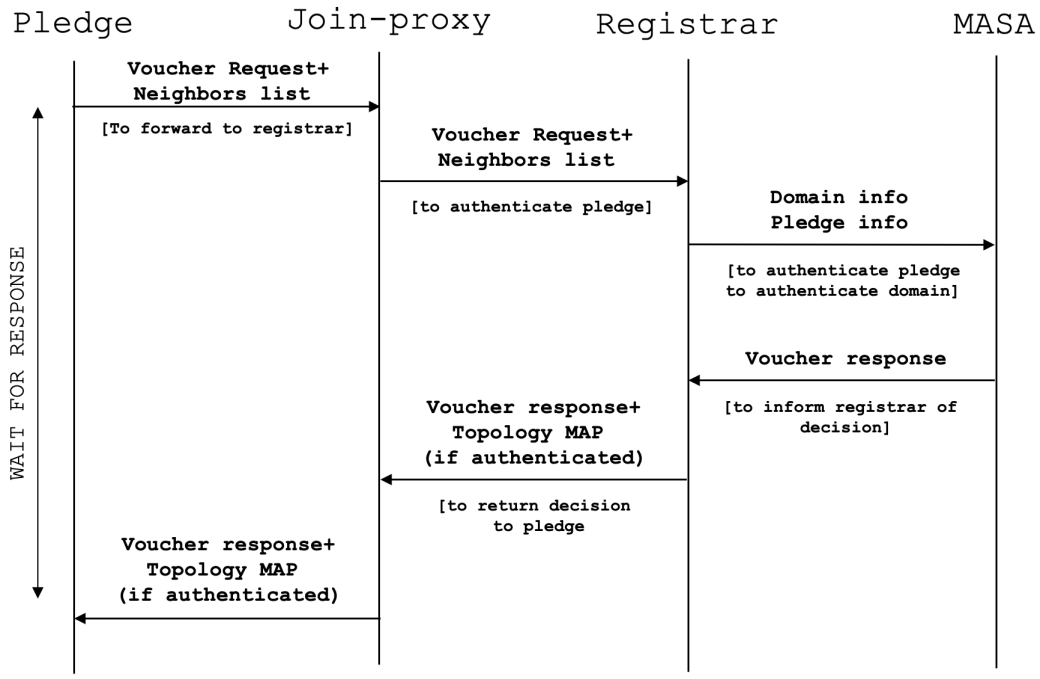

Figure 5.8: Using BRSKI for TD during authentication of the pledge

the mentioned list of neighbors in the `voucher_request`. The rest of the process continues as usual BRSKI protocol until the `voucher_request` reaches the registrar. The pledge will look for join proxy in its neighborhood. Upon finding a join proxy, the pledge will send out the voucher request to the join proxy so it can be relayed to the registrar.

The registrar will receive the `voucher_request` and extract MASA's certificate from it, and then tries to authenticate the pledge by sending a request to the MASA. If MASA authenticates the certificate and the registrar authenticates the pledge itself, it will then extract the neighbor list added to the `voucher_request`. Similar to Section 5.2.1, the registrar has a `td_obj` objective registered that is intended for communications over the topology map between itself and other nodes. Upon extracting the neighbor list, the registrar will update the topology map's value of the `td_obj` and the port number value of pledge's `td_obj`.

The registrar will modify the `voucher_response` message, include the updated map in the `voucher_response`, and send it back to the join proxy. The join proxy will then relay the response from the registrar to the pledge. If the `voucher_response` is an approval message,

the pledge will first extract the registrar's certificate from it and then extracts the updated topology map of the network. At this point, the pledge has been authenticated and has the updated topology of the network simultaneously. This solution will save us a round of communication between the pledge and the registrar. If the MASA or the registrar rejects the pledge, the registrar will ignore the neighbor list of the pledge. Similar to the original BRSKI protocol, the registrar sends the unsuccessful authentication response to the pledge through the join proxy.

After the authentication and clustering setup phase, the maintenance phase will occur, similar to the process described in Section 5.2.1.

Chapter 6

Experimental Results

This chapter describes our experimental results to validate and test our TD protocols. We set up a testbed and configured it as three different topologies. We ran the two proposed solutions described in Chapter 5 and measured the number of messages exchanged. To verify the results, the final topology map stored in each node was compared with the actual topology after the results were gathered.

6.1 Testbed

We tried three topologies for test purposes. The three topologies are shown in Figure 6.1, Figure 6.2, and Figure 6.3. Since the linear topology Figure 6.1, starts with letter L, we call the linear topology, "Topology L". The topology presented in Figure 6.2 looks like letter A, therefore, we call it "Topology A". Similar, the topology presented in Figure 6.3 looks like a rotated letter Y, therefore we call it "Topology Y".

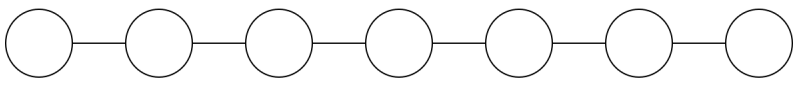

Figure 6.1: Topology L

Each topology has its unique challenge. As discussed in Section 5.1.2, topology L can be challenging for a centralized approach. In a cyclic topology such as topology A, the chance of a

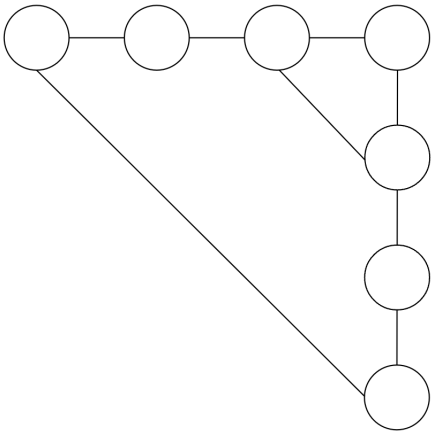

Figure 6.2: Topology A

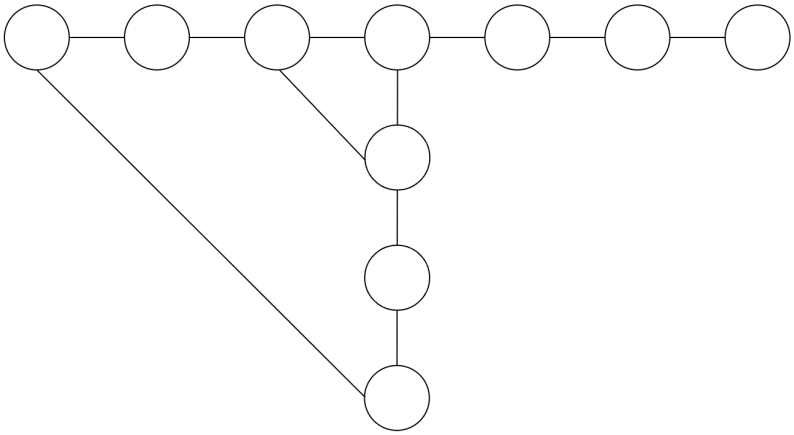

Figure 6.3: Topology Y

node receiving two inconsistent update messages for the same purpose from each interface is high. The asynchronous characteristic of the network, can raise the chance of such conflicts. Therefore, the challenge is handling the duplication of incoming data. Topology Y is a combination of the two topologies with higher connectivity and complexity than the other two topologies. Adding an inner smaller cycle in the existing cyclic topology can raise the chance for such unwanted events.

All the participating machines in the testbeds run Ubuntu 18.04 and higher as their operating system and are wired connected and are all IPv6 enabled. We used Python 3.8 to implement the algorithms.

The provided library for GRASP by Dr. B. Carpenter [1] is also implemented in Python.

6.2 Results of Clustering Approach

6.2.1 Weight Exchange Phase

As for the clustering solution, we initially assume the authentication process has already been done by BRSKI and all the nodes are authenticated. Therefore, the messages concerning BRSKI are not counted for the results. The clustering scheme is asynchronous. Initially, nodes have no prior knowledge about each other's weight. There is a limited number of methods for a node to obtain its neighbor's weight. The neighbor's weight can be stored either by extracting it from an incoming request for initiating a GRASP negotiation, or the weight will be extracted from the response of the peer to a GRASP negotiation initiation request. The weight exchange in this clustering scheme is limited to only neighbors. Since they are only one hop away, this action takes only a short amount of time, because it does not need to go through routing and be relayed from one node to the other.

We now describe two methods to realize the weight exchange. In the first method, all nodes initiate a request for weight exchange simultaneously, because none of the nodes in the network have any knowledge of their neighbor's weight. This can lead to exchanging a considerable number of messages just for exchanging weights.

Alternatively, in the second method, some nodes can start earlier and request the weight of their neighbors sooner than others. In this case, some nodes obtain their neighbors' weight from initiating a negotiation request before they do, so not only they obtain their neighbors' weight, but also they will send their own weights as a part of the request to the peer neighbors. This allows both participating peers to have their significant other neighbor's weight by establishing a single negotiation session between them. This short delay in starting the nodes will considerably decrease the number of messages exchanged. The delay can be simulated by adding a random weight time between 0 seconds to 10 seconds. We implemented both experiments, the first experiment is called "Simultaneous", and the second is called "With delay". Table 6.1, shows the number of messages that are exchanged in each topology. The results for starting the process "Simultaneously" or "With

[1] https://github.com/becarpenter/graspy

delay" are separated.

	Topology L	Topology A	Topology Y
Simultaneous run	24	32	46
With delay run	14	20	24

Table 6.1: Number of messages exchanged simultaneously or starting randomly from some nodes

The number of messages exchanged in the same topology is noticeably different when running tests simultaneously or with a short delay.

In the first scenario, since nodes do not have any prior knowledge of their neighbors, they are all considered to start negotiating with their neighbors simultaneously. However, in the second scenario, if some nodes have some knowledge about their neighbors, it will help them to save messages. The selection of the nodes who have previously communicated with their neighbors is random.

6.2.2 Role Selection and Announcement Phase

As discussed in Section 5.1.1, nodes will either announce themselves as clusterhead or try to join clusterheads. The number of clusterhead announcements, join announcements, and topology map exchange between clusterheads and cluster members are presented in Table 6.2. Since the number of nodes in each topology is limited, the number of possible outputs is also limited. The unique results obtained during each test run are recorded in each cell of Table 6.2. The results are grouped based on the type of the topology and the number of clusterheads. For some of the topologies, the number of messages remain the same, regardless of the number of test runs. In Table 6.2 some of the cells are left blank, because depending on the topology not having all the number of clusterheads are possible. Also, some of the cells has more than one results, which are obtained as a result of multiple runs. We added a random factor for the weight of the nodes, so that during each test run a unique result is generated.

Figure 6.4 demonstrates an example for the order in which messages are sent during the clustering setup phase. The order of the weights is as follows: $W_a > W_f > W_b > W_g > W_e > W_d >$

Number of clusterheads	Topology L	Topology A	Topology Y
2	-	18	-
3	18	22	29
4	30	-	39
5	-	-	55,58,60

Table 6.2: Total number of exchanged messages during clustering phase and topology map exchange for first time

W_c. During the first iteration of the initial phase of clustering, nodes a and f will announce themselves as clusterheads since they are the heaviest nodes among their neighbors and send clusterhead announcements. One clusterhead announcement is sent from a, and two clusterhead announcements are sent from f. Node b will join a and let its neighbors know. Nodes e and g join f, since f is heavier than them and heaviest clusterhead among their neighbors, and send three join messages in total. Node c wanted to join node b, but it did not receive a clusterhead announcement from b; similarly node d did not receive a clusterhead announcement from node e. Therefore, since node d has no heavier neighbor that announced itself as clusterhead, it will announce itself as clusterhead and send two clusterhead announcement messages to its two neighbors. During the final stage, node c will join d since d is the only neighbor that announced itself as a clusterhead. The dotted lines above the nodes demonstrate the clusterhead announcements, and the solid lines below the nodes represent the join messages.

Number of clusterheads	Topology L	Topology A	Topology Y
2	-	1	-
3	2	2	1
4	3	-	2
5	-	-	3

Table 6.3: Maximum number of iterations for a clusterhead to synchronize with other clusterheads

After the clusterheads have been identified and non-clusterhead nodes have joined their neighbor clusterheads, each clusterhead generate the topology map of its own local cluster. The clusterheads then try to discover each other and synchronize their topology map value by establishing GRASP negotiation sessions and negotiating the value of the map of their clusters. Depending on how many clusterheads have been detected and how they are spread across the network, it takes one or more

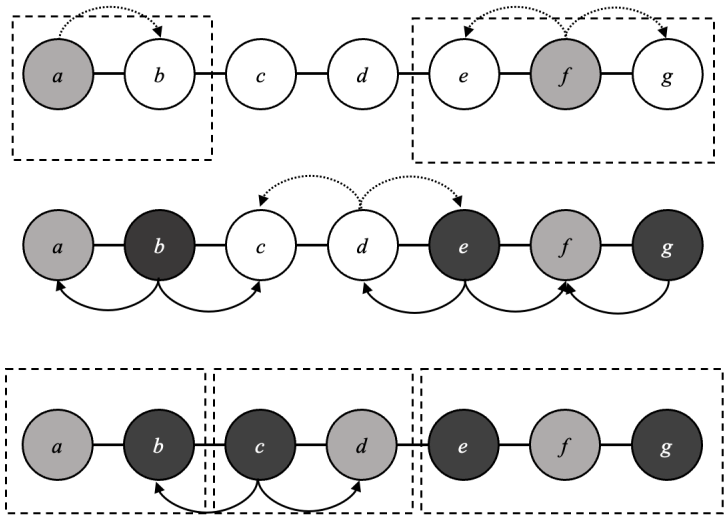

Figure 6.4: Illustrating the clusterhead announcement and join messages sent from each node in topology L.

steps to synchronize their values. The maximum numbers of iterations needed for any clusterhead to synchronize completely with other clusterheads during the setup phase is presented in Table 6.3. Depending on where the clusterheads are located in the topology, nodes can have different numbers of iterations to synchronize with their peer clusterheads. During each iteration, every clusterhead will establish a GRASP negotiation session with its peer clusterheads, which were found as a result of a GRASP discovery process.

6.2.3 Maintenance Phase

After the clusters have been formed and clusterheads have synchronized their value for the topology map for the first time, every node will listen for events. Events can include a new node joining/leaving the network or an update message from a subordinate cluster member to its corresponding clusterhead. Establishing a new link or removing an existing one are considered as joining or leaving a node. The reason is, we are using a one-hop clustering scheme. If a neighbor node is not accessible by a direct link we can consider it leaving the network. Though it might still be connected to the network via another link and join another cluster. To consider a node as neighbor and alive, we need a direct and stable link between any two neighbor nodes.

We measured the number of messages exchanged during TM. The results are presented separately in Table 6.4, Table 6.5, and Table 6.6.

Results are shown in the format of (x, y) where x represents the number of iterations needed for the clusterheads to synchronize their value for the topology map again. y represents the number of update messages sent from or to the non-clusterhead nodes by their respective clusterheads. For the purpose of testing and experimenting, for joining clusterheads, we added a single clusterhead node with no other nodes in its `cluster_set`. In realworld examples, a new link between two nodes from separate networks can be established. In that case, clusterheads from both networks will discover each other and update their topology map information of the newly joined network. Joining two networks may lead to extra iterations for clusterheads to update their topology information and spread the update across the network.

Number of clusterheads	Type of node	Type of change	Topology L
3	Clusterhead node	Join	(3, 6), (3 4)
		Leave	(2, 5), (2, 6)
	Non-clusterhead node	Join	(2, 4), (2, 5)
		Leave	(2, 4), (2, 5)
4	Clusterhead node	Join	(3, 3)
		Leave	(2, 3), (2, 2)
	Non-clusterhead node	Join	(3, 4)
		Leave	(2, 2)

Table 6.4: Propagating updates in the network; number of messages include updates to clusterheads and members of each cluster.

Number of clusterheads	Type of node	Type of change	Topology A
2	Clusterhead node	Join	(2, 5), (5, 4)
		Leave	(2, 4), (3, 5)
	Non-clusterhead node	Join	(1, 4)
		Leave	(1, 5)
3	Clusterhead node	Join	(5, 3), (4, 4)
		Leave	(2, 3), (2, 4)
	Non-clusterhead node	Join	(2, 4)
		Leave	(2, 3)

Table 6.5: Propagating updates in the network; number of messages include updates to clusterheads and members of each cluster.

Number of clusterheads	Type of node	Type of change	Topology Y
3	Clusterhead node	Join	(6, 7), (4, 6)
		Leave	(2, 6), (2, 5), (4, 5)
	Non-clusterhead node	Join	(2, 7)
		Leave	(2, 6)
4	Clusterhead node	Join	(5, 6), (8, 5), (4, 6)
		Leave	(3, 7), (3, 8)
	Non-clusterhead node	Join	(3, 6)
		Leave	(3, 5)
5	Clusterhead node	Join	(5, 5), (6, 6)
		Leave	(3, 5), (6, 6)
	Non-clusterhead node	Join	(4, 6)
		Leave	(4, 5)

Table 6.6: Propagating updates in the network; number of messages include updates to clusterheads and members of each cluster.

6.3 Results of TD during BRSKI and TD after BRSKI

The suggested solutions based on BRSKI have only one major difference. If TD takes place after BRSKI, a negotiation session from the pledge to the registrar is established to exchange the node's cluster-related info and receive the updated version topology map. However, if TD and BRSKI take place simultaneously, the number of messages exchanged for the initial process remains the same as for the BRSKI process. Table 6.7 shows the number of messages exchanged for TD after BRSKI. The first row of Table 6.7 contains all the messages concerning the TD. To have a fair comparison between the two BRSKI-based solutions, the second row which holds the number of the messages exchanged during the BRSKI authentication phase is added. During the maintenance phase of both BRSKI-based solutions, a fixed number of messages will be exchanged since the authentication is already done. The number of messages during maintenance phase for both solutions is the same as the results shown in the first row of Table 6.7. Each node will reach out to the registrar to report its neighboring information and updates. The registrar upon receiving the neighboring information or updates, will reflect those updates onto the topology map and forwards an instance to all the nodes.

Table 6.8 shows the number of messages exchanged in a scenario where the TD takes place after BRSKI. Depending on which node is selected as the registrar, the number of exchanged messages is different. In Table 6.8, the minimum, maximum, and average number of messages exchanged

	Topology L	Topology A	Topology Y
Number of messages exchanged regarding TD	27	31	54
BRSKI-only messages	22	20	34
Sum	49	51	88

Table 6.7: TD after BRSKI

	Topology L	Topology A	Topology Y
min	34	35	64
max	37	53	70
average	35	41	67

Table 6.8: TD during BRSKI

during this process is mentioned. The messages include registrar enrollment and topology map update messages. After each node joins the network, upon getting accepted, the registrar has to update its local version of the topology map and send it to all authenticated nodes.

6.4 Discussion

Table 6.9 shows a comparison among the number of messages in all three solutions on all topologies for the topology setup phase. For the clustering method, it should be mentioned that after clusters are formed, the clusterheads must go through some iterations and GRASP negotiation sessions to synchronize their value of the topology map. The messages for the clustering approach are calculated by adding the messages exchanged during weight exchange, role announcement (sending clusterhead announcements or join announcements), and messages exchanged among clusterheads and cluster members to synchronize their value of the topology map.

	Topology L	Topology A	Topology Y
TD with BRSKI	35	41	67
TD after BRSKI	27	31	54
Clustering	32	38	55

Table 6.9: Comparing the results of all methods with all topologies

Table 6.9 shows the number of messages exchanged during the TD setup phase by each topology

running each approach individually. The first observation would be the correlation between the size and connectivity of the topology to the number of exchanged messages, regardless of the topology. By taking a closer look at the results, we can observe that the rate at which the number of messages grows is different. The number of exchanged messages in our three approaches, has the following order:

$$TD\,After\,BRSKI < Clustering < TD\,While\,BRSKI$$

As discussed in Chapter 3, choosing the TD and TM approach is highly dependent on the nodes' formation and connectivity. Different approaches can be used depending on the network's type, purpose, etc. Therefore, comparing obtained results from each approach does not conclusively prove the advantage of one approach over the other.

Chapter 7

Conclusions and Future Work

Topology discovery plays an important role in network maintenance and network management. It allows nodes to understand their surroundings better and make decisions such as failure recovery beforehand. Autonomic networks proposed by the IRTF and the IETF are expected to achieve self-managing networks, in other words they are self-healing, self-protecting, self-configuring, and self-optimizing. Providing the topological information of the network to the autonomic nodes can help them make decisions with fewer messages exchanged.

Our literature survey revealed that the existing TD methods are categorized into two main groups: centralized approaches and decentralized approaches. The TD approach to be used in a given situation depends on the type of network and how often nodes in the network require topology information. In this thesis, we proposed both a centralized and a distributed approach to topology discovery and maintenance in an autonomic network.

Our centralized approach takes advantage of a secure bootstrapping protocol in autonomic networks. BRSKI introduced roles to the network to carry out tasks related to security, such as authentication of the devices. For this study, we took advantage of two roles introduced by BRSKI: the registrar and the pledge. We implemented two different methods of using BRSKI for TD purposes. The first method takes place after BRSKI, when nodes have securely reached mutual authentication. Upon reaching mutual authentication, the pledge, now an authenticated member of the network, sends its neighboring information to the registrar. Now the registrar is in charge of collecting the

neighboring information of all nodes and generating and distributing the topology map of the authenticated nodes. The other method requires modifying BRSKI by adding extra information to the BRSKI messages. The pledge joins the network by sending a `voucher_request` to the registrar. If we can embed the neighboring information of the node in the same `voucher_request`, we can authenticate the node and collect its neighboring information simultaneously at the registrar. Then the registrar would be able to generate/update the topology map of the authenticated network and distribute the updated version among nodes.

Our second proposed solution utilizes a clustering scheme. We studied several proposed clustering methodologies in the literature. By evaluating the infrastructure of autonomic networks, we selected a distributed clustering scheme that allows nodes to act autonomously. The proposed clustering scheme is a one-hop, non-location based, and asynchronous scheme. The clustering scheme is considered stationary during the cluster setup phase, but it supports the nodes' leaving or joining during the maintenance phase. Each node will be assigned a weight based on a metric, e.g., the number of active interfaces in the network. The weight will be shared with neighbors. Generally speaking, nodes with heavier weight will announce themselves as a clusterhead to its neighbors. Each node that has not announced itself as a clusterhead will join one of the neighbors that has introduced itself as a clusterhead. If the non-clusterhead node has heavier neighbors, but none has introduced itself as clusterhead, then the node will announce itself as clusterhead. Upon clusters being shaped, clusterheads will discover each other and try to synchronize their cluster's topology map by going through multiple iterations of negotiation. For TM, clusters have been created, nodes will listen for any events, such as new nodes joining or a node leaving the network. The node will send the update to its clusterhead, and the clusterhead will distribute it among other clusterheads and members of its `cluster_set`.

After reviewing the benefits of TD in autonomic networks, we set the goal to decrease the number of messages exchanged for TD purposes as much as possible as the size of the network grew. The proposed solutions provide an efficient approach to collect, generate, update, and distribute the topological map. The solutions presented are all compatible with the definition of autonomic behavior provided in RFC 7575 [7]. All solutions except "TD with BRSKI" are compatible with the standardization of autonomic networks provided by the ANIMA Working Group (RFC 8990 to

RFC 8995).

As part of our contribution to the ANIMA working group, we have reported some bugs regarding the implementation of GRASP to Dr. B. Carpenter, who provided us with the implementation of GRASP that was used in this thesis. Subsequently, these bugs were fixed and updates were issued.

We now discuss some future directions for research. To obtain more conclusive results, the suggested solutions should be tested on more topologies and a scalability analysis should be conducted. We focused on minimizing the number of messages, but the solutions can still be optimized to lower the number of messages even more during each stage of TD and TM. In the topology maintenance phase, we may allow two neighboring nodes to become clusterheads; we suggest that each node periodically goes through the setup phase to lessen the possibility of having too many such neighboring clusterheads.

In this thesis, we focused on wired networks, and a limited form of mobility in which nodes can join and leave the network in the topology maintenance phase. Furthermore, all our implementations were on wired networks. In future work, it would be important to consider whether these solutions are also suitable for wireless and mobile networks.

Also, we propose offering nodes with a service-based topology map. The service-based topology map provides the nodes with information on which autonomic nodes in the network support which objective. By mapping the autonomic nodes to the objectives they support, we can allow nodes to skip the discovery process, which will save a considerable number of messages from occupying the bandwidth.

As described in Section 5.2, the registrar can act as a central entity and create/maintain clusters to enhance the performance of the network. We suggest a hybrid method combining centralized and distributed TD and TM in future work. By using the hybrid method, the workload of the network will be distributed across the nodes. For example, the clustering setup can happen using the BRSKI-dependent methods, and the TM can happen using the clustering approach.

Bibliography

[1] R. A. Alhanani and J. Abouchabaka, "An overview of different techniques and algorithms for network topology discovery," in *Proceedings of the 2nd World Conference on Complex Systems (WCCS)*. IEEE, 2014, pp. 530–535.

[2] S. Ali, A. Ashraf, S. B. Qaisar, M. Kamran Afridi, H. Saeed, S. Rashid, E. A. Felemban, and A. A. Sheikh, "Simplimote: A wireless sensor network monitoring platform for oil and gas pipelines," *IEEE Systems Journal*, vol. 12, no. 1, pp. 778–789, 2018.

[3] A. Amis, R. Prakash, T. Vuong, and D. Huynh, "Max-min d-cluster formation in wireless ad hoc networks," in *Proceedings of the Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 1, 2000, pp. 32–41 vol.1.

[4] A. Azzouni, R. Boutaba, N. T. M. Trang, and G. Pujolle, "sOFTDP: Secure and efficient openflow topology discovery protocol," in *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–7.

[5] S. Basagni, "Distributed clustering for ad hoc networks," in *Proceedings of the 4th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99)*, 1999, pp. 310–315.

[6] M. H. Behringer, B. E. Carpenter, T. Eckert, L. Ciavaglia, and J. C. Nobre, "A Reference Model for Autonomic Networking," no. 8993, 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8993

[7] M. H. Behringer, M. Pritikin, S. Bjarnason, A. Clemm, B. E. Carpenter, S. Jiang, and L. Ciavaglia, "Autonomic Networking: Definitions and Design Goals," RFC 7575, 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7575.txt

[8] M. H. Behringer and Éric Vyncke, "Using Only Link-Local Addressing inside an IPv6 Network," RFC 7404, 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7404

[9] C. Bormann, B. E. Carpenter, and B. Liu, "GeneRic Autonomic Signaling Protocol (GRASP)," RFC 8990, 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc8990.txt

[10] C. Bormann and P. E. Hoffman, "Concise Binary Object Representation (CBOR)," RFC 8949, 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8949

[11] M. Boucadair and C. Jacquenet, "Software-Defined Networking: A Perspective from within a Service Provider Environment," RFC 7149, 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7149

[12] B. E. Carpenter, M. C. Richardson, F. Toerless Eckert, J. C. Nobre, S. Jiang, Y. Li, and C. Bormann, "Autonomic networking gets serious," in *The Internet Protocol Journal*, vol. 24, no. 3. IPJ, 2021, pp. 2–19.

[13] D. S. E. Deering and B. Hinden, "IP Version 6 Addressing Architecture," RFC 4291, 2006. [Online]. Available: https://www.rfc-editor.org/info/rfc4291

[14] B. Donnet and T. Friedman, "Internet topology discovery: a survey," *IEEE Communications Surveys & Tutorials*, vol. 9, no. 4, pp. 56–69, 2007.

[15] T. Eckert, M. H. Behringer, and S. Bjarnason, "An Autonomic Control Plane (ACP)," RFC 8994, 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8994

[16] B. Haberman and B. Hinden, "Unique Local IPv6 Unicast Addresses," RFC 4193, 2005. [Online]. Available: https://www.rfc-editor.org/info/rfc4193

[17] C. Hedrick, "Routing Information Protocol," RFC 1058, 1988. [Online]. Available: https://www.rfc-editor.org/info/rfc1058

[18] IAB, L. Daigle, and O. Kolkman, "RFC Streams, Headers, and Boilerplates," RFC 5741, 2009. [Online]. Available: https://www.rfc-editor.org/info/rfc5741

[19] S. Islam, N. Muslim, and J. W. Atwood, "A survey on multicasting in software-defined networking," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 355–387, 2018.

[20] I. Jawhar, N. Mohamed, and L. Zhang, "A distributed topology discovery algorithm for linear sensor networks," in *1st IEEE International Conference on Communications in China (ICCC)*, 2012, pp. 775–780.

[21] S. Jiang, B. E. Carpenter, and M. H. Behringer, "General Gap Analysis for Autonomic Networking," RFC 7576, 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7576

[22] Y. R. K. Lougheed, "Border Gateway Protocol (BGP)," RFC 1163, 1990. [Online]. Available: https://www.rfc-editor.org/info/rfc1163

[23] H. E.-S. L. Zhang, "A novel cluster-based protocol for topology discovery in vehicular ad hoc network," *Procedia Computer Science*, vol. 10, pp. 525–534, 2012.

[24] C. Lin and M. Gerla, "Adaptive clustering for mobile wireless networks," *Proceedings of the IEEE Journal on Selected Areas in Communications*, vol. 15, no. 7, pp. 1265–1275, 1997.

[25] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[26] J. Moy, "OSPF Version 2," RFC 1247, 1991. [Online]. Available: https://www.rfc-editor.org/info/rfc1247

[27] L. Ochoa-Aday, C. Cervello-Pastor, and A. Fernandez-Fernandez, "Self-healing topology discovery protocol for software-defined networks," *IEEE Communications Letters*, vol. 22, no. 5, pp. 1070–1073, 2018.

[28] D. C. Plummer, "An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware," RFC 826, 1982. [Online]. Available: https://www.rfc-editor.org/info/rfc826

[29] J. Postel, "Internet Control Message Protocol," RFC 792, 1981. [Online]. Available: https://www.rfc-editor.org/info/rfc792

[30] M. Pritikin, M. Richardson, T. Eckert, M. H. Behringer, and K. Watsen, "Bootstrapping Remote Secure Key Infrastructure (BRSKI)," RFC 8995, 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8995

[31] W. A. Simpson, D. T. Narten, E. Nordmark, and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)," RFC 4861, 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc4861

[32] S. Vasudevan, J. Kurose, and D. Towsley, "Design and analysis of a leader election algorithm for mobile ad hoc networks," *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP)*, pp. 350–360, 2004.

[33] D. Wei and H. A. Chan, "Clustering ad hoc networks: Schemes and classifications," *Proceedings of the 3rd Annual IEEE Communications Society on Sensor and Ad Hoc Communications and Networks*, vol. 3, pp. 920–926, 2006.

[34] J. Yin, Y. Li, Q. Wang, B. Ji, and J. Wang, "SNMP-based network topology discovery algorithm and implementation," *Proceedings of the 9th International Conference on Fuzzy Systems and Knowledge Discovery*, pp. 2241–2244, 2012.

[35] A. Zacharis, S. V. Margariti, E. Stergiou, and C. Angelis, "Performance evaluation of topology discovery protocols in software defined networks," *Proceedings of the IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pp. 135–140, 2021.