TRABAJO DE FIN DE GRADO EN INGENIERÍA INFORMÁTICA

Private Networks Intrusion Detection System by Satisfying Network Constraints

Francisco García Martínez

Directed by: Thomas Dean
Co-directed by: José Luis Vázquez-Poletti
08/06/2017

TASSEP PROGRAM

Faculty of Engineering and Applied Science
Queen's University of Canada

&

Facultad de Informática
Universidad Complutense of Madrid

Acknowledgements

I want to express my sincere gratitude to Dr. Thomas R. Dean, Associate Professor of Queen's University's Electrical & Computer Engineering Department and director of this huge and amazing project, without whom any of this would have ever been possible.

Thanks to all my project team members for their patience and kindness throughout this year, special mention to Siam Hasan and Ali ElShakankiry.

Also, special thanks to my co-director in Spain, Dr. José Luis Vázquez-Poletti, Associate Professor of Complutense University's Computer Architecture and Automation Department, for his support, help and availability.

This research project is funded in part by the National Science and Engineering Research Council of Canada (NSERC) and the Department of National Defense (DND), Canada.

Abstract

The great development of newer technologies also carries an important growth in the number of malicious attacks [1]. Even private networks without external Internet connections suffer from those attacks. These private networks play a crucial role in the country's security. Imagine the consequences of turning the power of an entire city down or a denial of service [2] in an air traffic control system. Because of this fact, numerous politicians, including the recently named United States of America's president, Donald Trump, are seriously taking into consideration the huge importance of protecting the private networks from intrusions in order to assure their countries' peace. Some people even believe that efficient Intrusion Detection Systems (IDS) [3] could be a good protection against a possible Third World War. Thus, new and more powerful security solutions need to be developed to protect our organizations' systems.

Key words: Intrusion Detection System (IDS), Data Distribution Service (DDS), Real-Time Publish-Subscribe Protocol (RTPS), private networks, intrusions, constraint engine, network constraints, trees.

Resumen

El gran desarrollo de las nuevas tecnologías conlleva también un aumento considerable en el número de ataques maliciosos [1]. Incluso las redes privadas sin conexión externa a Internet sufren estos ataques. Estas redes privadas juegan un papel crucial en la seguridad de cada país. Imagínense las consecuencias de tirar abajo la corriente eléctrica de una ciudad entera o una denegación de servicio [2] en un sistema de control aéreo. Debido a este hecho, numerosos políticos, incluido el recientemente nombrado presidente de los Estados Unidos de América, Donald Trump, están concienciándose seriamente de la enorme importancia que supone proteger las redes privadas frente a intrusiones para asegurar la paz en sus países. Alguna gente cree incluso que unos Sistemas de Detección de Intrusiones (IDS) [3] efectivos pueden ser una buena medida de protección ante una posible Tercera Guerra Mundial. De esta forma, nuevas y más potentes medidas de seguridad tienen que desarrollarse para proteger los sistemas de nuestras organizaciones.

Palabras clave: Sistema de Detección de Intrusiones (IDS), Servicio de Distribución de Datos (DDS), Protocolo de Publicación-Subscripción en Tiempo Real (RTPS), redes privadas, intrusiones, motor de restricciones, restricciones de red, árboles.

Glossary

- **ABS**: Anomaly-Based System
- **ATC**: Air Traffic Control
- **DAG**: Directed Acyclic Graph
- **DDS**: Data Distribution Service
- **DoS**: Denial of Service
- **DDoS**: Distributed Denial of Service
- **DPI**: Deep Packet Inspection
- **IDS**: Intrusion Detection System
- **IGMP**: Internet Group Management Protocol
- **NIDS**: Network Intrusion Detection System
- **OMG**: Object Management Group
- **PCAP**: Packet capture
- **QoS**: Quality of Service
- **RTPS**: Real Time Publish-Subscribe Protocol
- **SBS**: Signature-Based System
- **SCL**: Syntax Constraint Language

Glosario

- **ABS**: Sistema Basado en Anomalías
- **ATC**: Control de Tráfico Aéreo
- **DAG**: Gráfico Acíclico Dirigido
- **DDS**: Servicio de Distribución de Datos
- **DoS**: Denegación de Servicio
- **DDoS**: Denegación de Servicio Distribuida
- **DPI**: Inspección Profunda de Paquetes
- **IDS**: Sistema de Detección de Intrusiones
- **IGMP**: Protocolo de Gestión de Grupos en Internet
- **NIDS**: Sistema de Detección de Intrusiones en Redes
- **OMG**: Grupo de Gestión de Objetos
- **PCAP**: Captura de Paquetes
- **QoS**: Calidad del Servicio
- **RTPS**: Protocolo de Publicación-Subscripción en Tiempo Real
- **SBS**: Sistema Basado en Firmas
- **SCL**: Lenguaje de Sintaxis de Restricciones

Index

Figures index

1. Introduction

First and vital step to secure a private network is to constantly monitor the ongoing traffic and define a network's "normal behavior". This behavior could be characterized by a set of network constraints [4], so a violation of these constraints indicates a possible intrusion. It is also crucial to detect these intruders as soon as possible to minimize their damage to the network. But in large organizations, in which thousands of packets are transmitted every second to the network, this task might not be easy.

1.1. Objective

The goal of this research project is to present a novel framework to protect private networks with a limited number of protocols against intrusion attacks, by detecting those packets that differ from the normal behavior in real time [5]. For that, we will model the normal traffic patterns as network constraints and present how the framework is evaluating these constraints against traffic from an experimental network using PCAP tools [6]. Figure 1 represents a big picture of the global project's behavior.

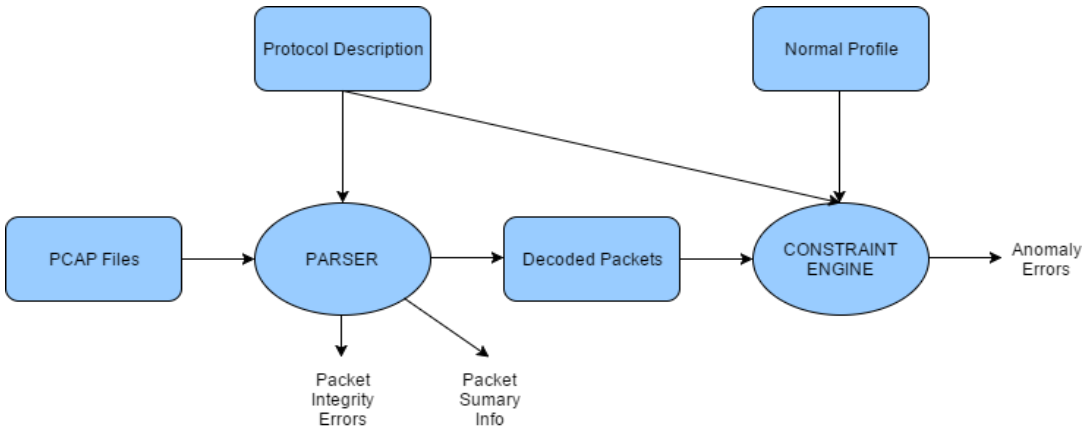

Fig. 1. Global Project Architecture.

Concretely, I will be working on the Constraint Engine and my objective will be to design, implement and correctly test the performance of **three** constraints against sample traffic from our experimental network.

1.2. Method

We are firstly exploring the mentioned framework using two protocols: Internet Group Management Protocol (IGMP) [7] and Real Time Publish and Subscribe Protocol (RTPS) [8], being this last one based in the Data Distribution Service (DDS) mechanism [9]. DDS provides effective support and security for critical services, such as air traffic control management or financial trading. Hence, we start focusing our research on these protocols because they might be used by the kind of applications for what we are aiming our approach. But, once our program is properly working with these two protocols, our final goal is to build a software able to work among any protocol defined, allowing it to, once given the protocol's definition using a specific language that we create, automatically develop and evaluate the set of constraints for that concrete protocol.

Focusing on my particular job, for every defined constraint I will draw a tree that corresponds to its specification to, later on, handily define and create a skeleton for that tree structure in C. Additionally, I will implement all its respective functions to properly work within the system.

1.3. Document Structure

First of all, I will go through all the previous **background** concepts that need to be taken into consideration in order to later better understand the project itself. Here I will talk about concepts such as private networks, deep packet inspection and Network Intrusion Detection Systems, as well as some protocols that will be used. Besides, in this section I will briefly describe our global project framework.

Secondly, I will mention the **network constraints** that we have come up with so far, including a deeper explanation of the constraints I have personally been working on.

The third section of the report contains the information related to the **trees' structures** I have developed to build the constraints and everything related to the **implementation** of the constraints I have mentioned before. Thus, for each of those constraints, I will explain the four-phase mechanism that is followed from the creation to the deletion of their corresponding trees.

In the fourth section it is described how the constraints are **evaluated** with my trees structures' approach, differentiating it from my partner's approach.

Finally, the **Conclusions and Future Work** section includes the results of the evaluation of the constraints I have worked on using my approach, as well as some related work that could be done in the future.

2. Introducción

El primer y vital paso para asegurar las redes privadas consiste en monitorizar constantemente el tráfico en marcha y definir un "comportamiento normal" de la red. Este comportamiento podría caracterizarse por una serie de restricciones de red [4], por lo que una violación de estas restricciones indicaría una posible intrusión. También, resulta crucial detectar estos intrusos tan pronto como sea posible para minimizar el daño que puedan causar a la red. Sin embargo, en grandes organizaciones en las que se mandan a la red miles de paquetes por segundo, esta tarea puede no resultar fácil.

2.1. Objetivos

El propósito de este proyecto de investigación consiste en presentar un novedoso entorno para proteger redes privadas con un número de protocolos limitado ante ataques de intrusiones, a través de la detección de aquellos paquetes que difieran del comportamiento normal en tiempo real [5]. Para ello, modelaremos patrones que representen el tráfico normal en la red como restricciones y presentaremos cómo el entorno está evaluando dichas restricciones ante tráfico obtenido de una red experimental usando herramientas PCAP [6]. La Figura 1 representa a gran escala el comportamiento global del proyecto.

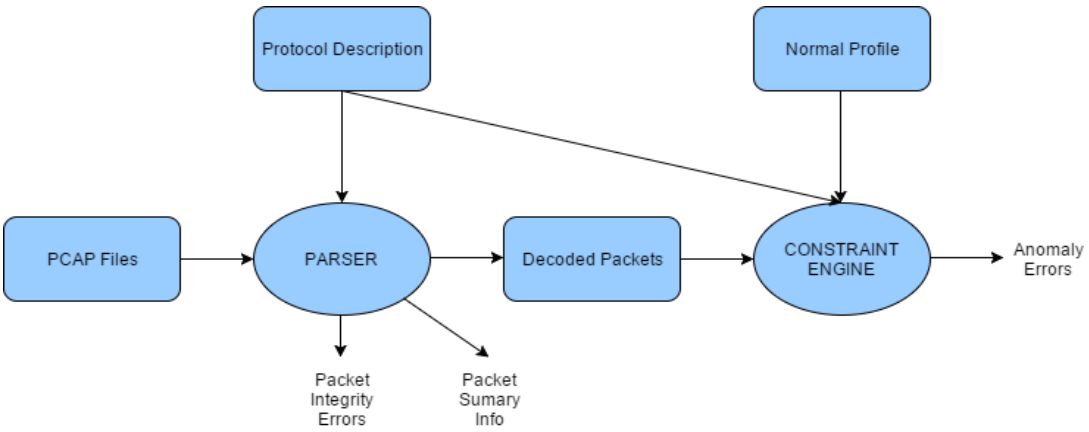

Fig. 1. Arquitectura Global del Proyecto.

Concretamente, yo estaré trabajando en el Motor de Restricciones y mi objetivo consistirá en el diseño, implementación y correcta prueba del rendimiento de **tres** restricciones ante el tráfico de prueba de nuestra red experimental.

2.2. Método

En primer lugar, estamos explorando el entorno mencionado usando dos protocolos: el Protocolo de Gestión de Grupos en Internet (IGMP) [7] y el Protocolo de Publicación-Subscripción en Tiempo Real (RTPS) [8], estando este último basado en el mecanismo del Servicio de Distribución de Datos (DDS) [9]. DDS proporciona un apoyo eficaz y seguridad para tipos de servicios críticos, como pueden ser la gestión de control de tráfico aéreo o el comercio financiero. De ahí que empecemos a centrar nuestra investigación en estos protocolos, ya que podrían ser usados por el tipo de aplicaciones para los que estamos fijando nuestro objetivo. Pero una vez que nuestro programa funcione correctamente para estos dos protocolos, nuestro objetivo final consistirá en construir un software capaz de trabajar con cualquier protocolo que se le defina, permitiéndole, una vez especificada la definición de tal protocolo usando un lenguaje que nosotros mismos crearemos, desarrollar y evaluar automáticamente el conjunto de restricciones para dicho protocolo en concreto.

Centrándonos en mi trabajo en particular, para cada una de las restricciones definidas, dibujaré el árbol que corresponda a su especificación para, más tarde definir y crear a mano un esqueleto para esa estructura de árbol en C. Adicionalmente, implementaré todas sus respectivas funciones para que puedan integrarse y funcionar correctamente dentro de nuestro sistema.

2.3. Estructura del documento

Primero, explicaré brevemente todos los **conceptos previos** que necesitan ser comprendidos para más tarde entender mejor el proyecto en sí. Aquí, trataré conceptos tales como redes privadas, inspección profunda de paquetes y Sistemas de Detección de Intrusiones en Redes, así como algunos protocolos que serán usados. Además, en esta sección describiré por encima el entorno global de nuestro proyecto.

En segundo lugar, mencionaré las **restricciones de red** que hemos encontrado hasta el momento, incluyendo una mayor explicación de las restricciones sobre las que yo he estado trabajando personalmente.

La tercera sección de la memoria contiene toda la información relacionada con las **estructuras de árboles** que he desarrollado para construir las restricciones, y todo lo relacionado con la **implementación** de dichas restricciones que he mencionado anteriormente. Así, para cada una de esas restricciones, explicaré el mecanismo de cuatro-fases que es seguido por ellas desde la creación hasta la eliminación de sus árboles correspondientes.

En la cuarta sección se describe cómo las restricciones son **evaluadas** con ese enfoque de las estructuras de árboles, diferenciándolo del enfoque que daba mi compañero para su evaluación.

Finalmente, la sección de **Conclusiones y Trabajo Futuro** incluye los resultados de la evaluación de las restricciones sobre las que he trabajado usando mi enfoque, así como trabajo relacionado con el tema que puede ser desarrollado en el futuro.

3. Background

Handful open source IDS [10], such as Snort, Suricata or Bro [11], are currently available but, in spite of providing fantastic features and incredibly large rule sets, they all have a common flaw: they only support a few famous protocols, like HTTP [12] or TCP [13], and it results a truly tough task to add new rules to the data set. Due to the fact that most of the private networks use their own protocols built for specific purposes, these Intrusion Detection Systems fail to be effective. That is why we focus our approach on developing an IDS able to cope and produce satisfactory results with any protocol.

3.1. Global Project Framework

Our previously research group's proposed framework is divided into three modules: Scanning, Parsing and Constraint Engine [5]. Figure 2 shows a simplified idea of the interaction between those three modules. Basically, the data flow consists of reading and parsing the packets, and converting them into data structures able to be used by the Constraint Checker.

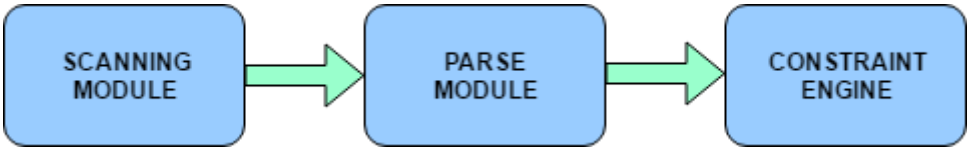

Fig. 2. Modules of Intrusion Detection Framework.

The scanning module is in charge of reading the packets from PCAP files using Wireshark [14] and extracting the essential data. Currently, two IP protocols are supported: IGMP and User Datagram Protocol (UDP) [15]. For every single packet, the Scanner passes the following information to the Parser: source IP, destination IP and arrival time. Besides, it will pass the source and destination ports if there is a UDP packet.

The Parsing module receives these packets along with their information, parses them and converts them into C data structures. An alert is generated if the Parser fails to parse any packet. Some examples of failures could be malformed data, like non-sensitive date or time values, or errors in lengths, such as attempted buffer overflows [16]. On the contrary, if the packets are successfully parsed, they are passed along to the Constraint Engine, where I am going to focus my work.

The Constraint Checker is an engine that implements a several number of constraints related to the definition of the protocols. Currently, it is being hand-written by the research group, but it is designed as a template to be later automatically generated from a specific language specification which we are working on. My mission in the Constraint Engine is to develop an approach to define the network constraints as tree structures [17] and, later on, integrate my code with the Parser module and evaluate those trees against sample RTPS data generated by our experimental network. Concretely, I should be able to test my approach for three constraints.

Our initial Parser, inherited from previous penetration testing research [18], used a general engine that is parameterized by a grammar graph. This graph is generated from a Syntax Constraint Language (SCL) protocol description [19]. SCL derives from ASN.1 [20], which contains XML markup [21] to provide both context sensitive parsing and general constraints. Our grammars for IGMP and RTPS were validated against multiple sources of network data to ensure that the grammar was correct. However, the Parser turned out to be too slow for our intrusion detection prototype when evaluated against packet data from an industrial partner. Consequently, we decided to develop a new Parser, which is a hand written, designed with the intention of being automatically generated in the future. Thus, we have manually followed a rigorous approach to translate the SCL to produce a hand coded parser.

3.2. Private Networks

In networks such as our home Internet or a cafe's Wi-Fi we can download apps, send emails, play games, use Facebook or watch YouTube videos. To sum up, there isn't any restriction about the protocols used or the type of packets sent through the network. Not even about the type or number of devices connected to it. Consequently, it results a really tough work to monitor the ongoing traffic or determine what is "normal" in those networks. These are called public or conventional networks.

On the other hand, private networks or intranets are networks created by a single organization that controls its security policies and network management. This means that the devices connected to them are clearly identified, and that the packets that they send belong to a certain type of allowed packets. They use a limited number of protocols. Examples of these networks are telecommunication, nuclear and power plants, or aircraft control systems, which represent critical infrastructures.

Private networks increase their security by preventing external access, protecting their industrial control operation from outside attacks. However, a private network is

not fully secured, as malware could be injected by USB drives or installation discs. For instance, the worldwide known case of the Stuxnet worm used to attack a nuclear plant in Natanz was distributed by an USB flash drive [22]. As a result, even more protection needs to be developed. Perimeter checking, such as firewalls and authentication policies [23], are useful defensive techniques, but they can't monitor the ongoing traffic. The only way of performing that duty is using an Intrusion Detection System [24].

3.3. Network Intrusion Detection Systems

Network Intrusion Detection Systems (NIDS) are the most efficient way of defending against network-based attacks. They consist in a device or software application that monitors a network for malicious activity or policy violations. NIDS are placed at strategic point or points of the organization and performs an analysis of passing traffic on the entire subnet. These systems are widely used among large-scale IT infrastructures [25]. Basically, there are two main types of NIDS: signature-based (SBS) and anomaly-based (ABS).

On the one hand, signature-based IDS refers to the detection of attacks by looking for specific patterns, such as byte sequences in network traffic, or known malicious instruction sequences used by malware. They maintain a database of signatures of previously known attacks and compare them with the analyzed data. Thus, an alarm would be raised when the signatures were matched. The main advantage of this technique is that signatures are very easy to develop and understand if we know what network behavior we are trying to identify. Moreover, pattern matching can be done more efficiently as they can perform this matching with just a small rule set. For instance, if the system that is to be protected only communicates via DNS [26] and ICMP [27], all other signatures can be ignored. However, SBS only detect attacks whose signatures are previously stored in the database, so zero-day attacks [28] cannot be detected.

On the other hand, anomaly-based IDS were primarily created to detect unknown attacks, due to the fast development of malware [29]. In contrast to SBS, they build a statistical model describing the normal network traffic and, and any abnormal behavior that deviates from the model is identified. As anomaly based detection is based on defining the network behavior, it requires a training phase to develop the database of general attacks and a careful setting of threshold level of detection. This accepted network behavior is prepared or learned by the specifications of the network administrators. The major drawback of ABS is defining its rule set. As a consequence, the efficiency of the system depends on how well it is implemented and tested on all protocols. Hence, we introduce the concept of constraints in our system because it represents more concise actions than rules [30]. Furthermore, even using dynamic rules of existing IDS, it results a really tough task to create a flexible

and expressive rule set which can represent a real complex attack scenario [5]. The major advantage over SBS is that a novel attack for which a signature does not exist can be detected if it falls out of the normal traffic patterns.

3.4. Deep Packet Inspection

There is a large number of organizations protecting their networks with firewalls, simply limiting their traffic through the ports 22, 80 and 443, corresponding with the SSH [31], HTTP [9] and HTTPS [32] protocols respectively. Any packet transmitted by a protocol which does not correspond to any of these three ones, is automatically discarded. However, attackers have found the way of intruding those networks hiding their functionality under either the SSH, HTTP or HTTPS protocols [33]. As a result, they are able to snick into the network's allowed traffic without being noticed and correspondingly rejected by the firewalls.

Because of these attackers, a simply firewall is not enough to protect the networks, and more sophisticated techniques need to be developed. Here come to scene the term 'Deep Packet Inspection'. DPI is an advanced method of packet filtering that functions at the Application layer of the OSI (Open Systems Interconnection) reference model [34]. "The use of DPI makes it possible to find, identify, classify, reroute or block packets with specific data or code payloads that conventional packet filtering, such as firewalls, which examines only packet headers, cannot detect" [35]. By looking deeply into the packet fields, it is possible to identify the real data that is being sent in each packet. This makes it effective against buffer overflow attacks, denial of service (DoS) attacks and certain types of malware.

3.5. Data Distribution Service

The Data Distribution Service (DDS) is an Object Management Group (OMG) protocol whose goal is to provide reliable real-time and scalable communications at a high performance by using a publish-subscribe mechanism. Air traffic control, financial trading or other big data applications are examples of environments where a protocol like DDS is needed for their purposes. Besides, this standard is widely used among worldwide recognized institutions such as the NASA [36] or the Canadian Air Traffic Control System [37], as well as in the mining industry to check mining equipment and in the automobile sector to develop simulators [38].

As shown in Figure 3, DDS provides an infrastructure layer that enables many different types of applications to communicate with each other.

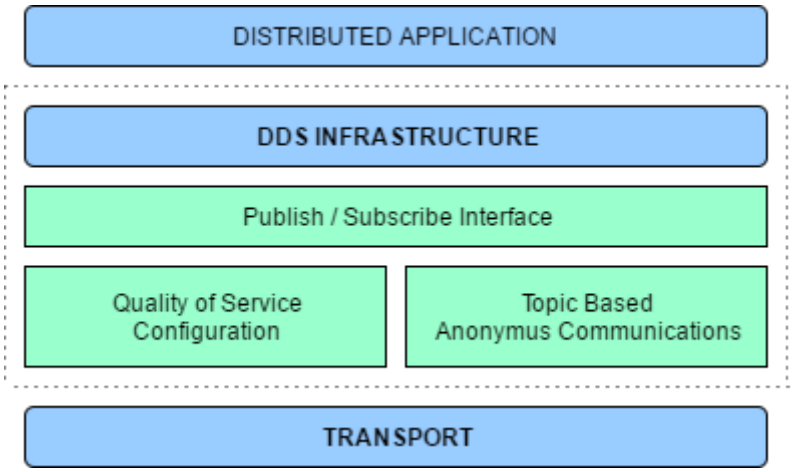

Fig. 3. DDS Infrastructure.

Pardo-Castellote's paper says: "Publish-subscribe applications are typically distributed applications with endpoint nodes that communicate with each other by sending (or publishing) data and receiving (or subscribing) data anonymously. Usually the only property that a publisher needs in order to communicate with a subscriber is the topic name and the definition of the data. What is more, the publisher does not need any information about the subscribers and vice versa". Resultantly, the only needed thing to listen to certain data is the Topic Key of the data published and the Domain Id.

As said in the RTI paper, "A publish-subscribe infrastructure is capable of delivering data to the appropriate nodes without having to set up individual connections between them". Machines interested in receiving packets related to a certain topic just need to become a subscriber for that topic. Analogously, nodes willing to send data about a certain topic just need to become a publisher of that topic, and the DDS infrastructure will automatically assure that the published information is distributed to all subscribers interested in that topic, without the publisher needing to specify those destination ports.

The specification for DDS is broken up into two separate layers: Data-Centric Publish-Subscribe (DCPS) and Data Local Reconstruction Layer (DLRL) [13]. For our purpose, we will only concentrate on the DCPS portion of the specification. RTI's paper defines DCPS as following: "DCPS is the lower layer API that an application can use to communicate with other DDS-enabled applications". This communication mechanism provides them the ability to specify several Quality of Service (QoS) parameters, such as rate of publication, rate of subscription or how long data is valid for. These parameters allow the system designers to build a customized distributed application based on their requirements. In DCPS, applications must use APIs to create entities (objects) in order to establish publish-subscribe communications between each other. DCPS has the following primary

entities: Domain, Domain Participant, Data Writer, Publisher, Data Reader, Subscriber and Topic. Figure 4 shows how entities in DDS are related.

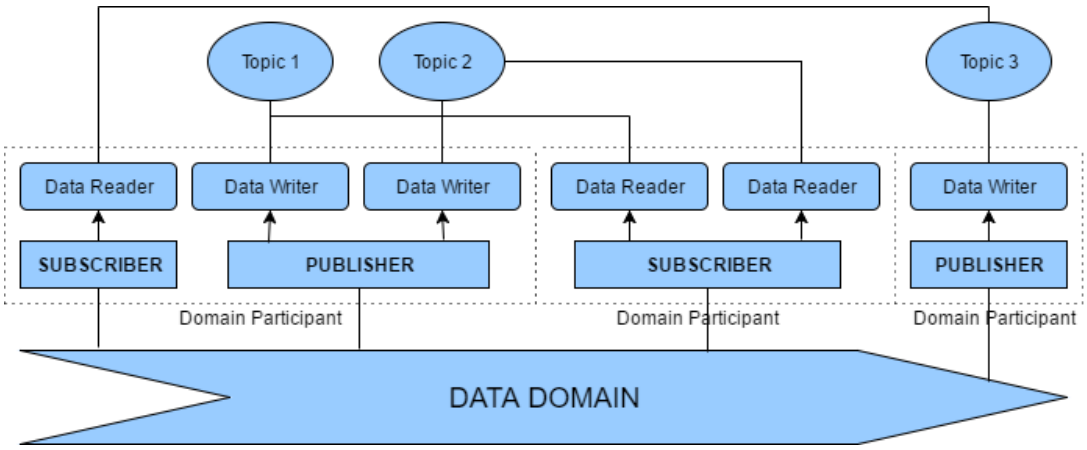

Fig. 4. DDS Entities.

The Domain is the basic element needed to bind individual applications and start a communication between them. DDS has the capability of supporting both a single Domain of application's data-centric [13] communications, and multiple Domains. If the system is using a single Domain, all the nodes will communicate within this Domain. On the contrary, if we have a system that can scale with effective data isolation, we might choose multiple Domains. Therefore, when specific data is published in one Domain, it will not be received by Subscribers residing on any other Domains. Moreover, multiple Domains are also a great way to control the introduction of a new functionality into an existing system.

An application uses a Domain Participant to represent its activity within a Domain. This allows developers to define default QoS parameters for all Data Writers, Publishers, Data Readers and Subscribers in that Domain. Thus, this makes it easy to specify default behavior in the Domain.

Obtained from Pardo-Castellote's paper, "Data Writers are the primary access point for an application to publish data into a DDS domain". An application uses Data Writers to send data. These are associated with a single Topic. An application can have multiple Data Writers and Topics. Besides, it can have more than one Data Writer for a particular Topic. However, before being able to perform a write call, Data Writers need to be created and configured with the correct QoS settings.

The Publisher is just a container to group and manage together individual Data Writers. It represents the DCPS object responsible for the actual sending of data. Whereas a Data Writer can only be owned by a single Publisher a Publisher can own multiple Data Writers [39]. Hence, the same Publisher may be sending data for

several different Topics of different data types. “When user code calls the write() method on a Data Writer, the DDS data sample is passed to the Publisher object which does the actual dissemination of data on the network” [39]. A developer can specify the QoS behavior for a Publisher and have it applied to all the Data Writers in that Publisher’s group. Figure 5 illustrates how these entities are connected to establish the communication.

Defined in the RTI paper, “A Data Reader is the primary access point for an application to access data that has been received by a Subscriber”. Also, a Data Reader is associated with a single Topic and, once again, an application can have multiple Data Readers and Topics [39]. Additionally, it can have more than one Data Reader for a single Topic. Data can be accessed by the take() or read() methods. The take() call removes the data from the middleware after returning it, while the read() call allows the same data to be retrieved several times. Once created and configured with the correct QoS, an application can be notified that data is available in one of three ways:

1. Listener Callback Routine: consists in setting up a listener callback routine that DDS will run immediately when data is received.
2. Polling the Data Reader: consists in “polling” or querying the Data Reader to determine if data is available.
3. Conditions and WaitSets: the application waits until a specified condition is matched and then it accesses the data from the Data Reader.

Just as Publishers are used to group together multiple Data Writers, Subscribers are used to group together several Data Readers. Subscribers own and manage Data Readers. Again, although a Data Reader can exclusively be owned by a single Subscriber, a Subscriber can own many Data Readers. Thus, the same Subscriber may receive data for many different Topics of different data types. When data is sent to an application, it is first processed by a Subscriber; the DDS data sample is then stored in the appropriate Data Reader. Analogously to how Publishers work, this allows you to configure a default set of QoS parameters that will apply to all the Data Readers in that Subscriber’s group.

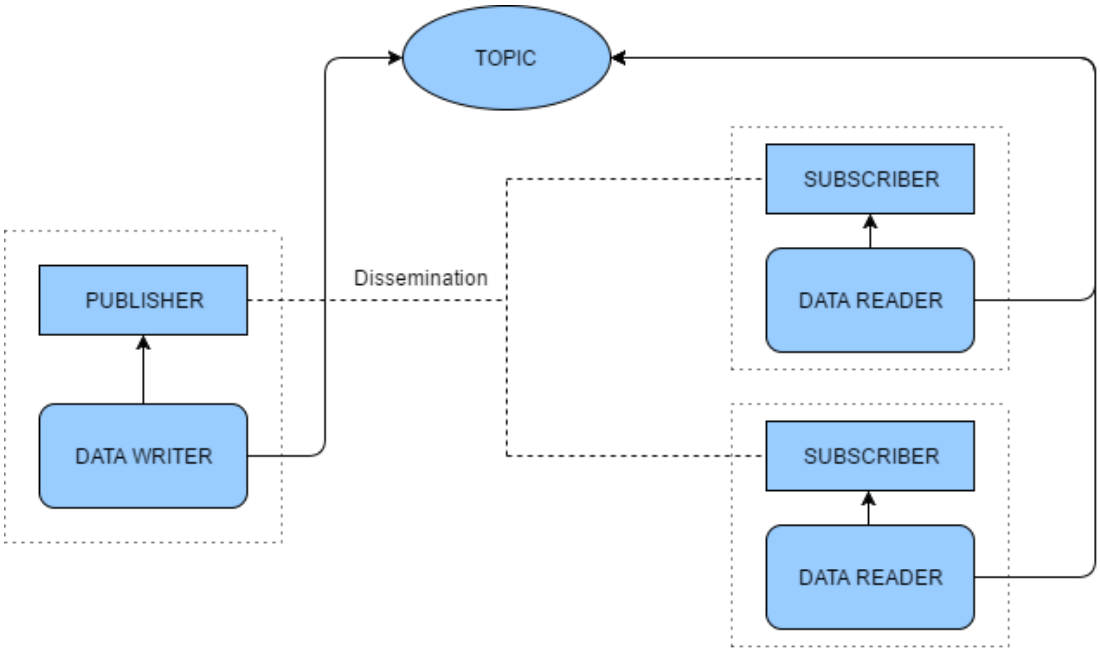

Fig. 5. DDS Objects Communication Overview.

Topics represent the basic connection point between Publishers and Subscribers. To establish the communication, the Topic of a given Publisher on one node must match the Topic of an associated Subscriber on any other node. A Topic is comprised of a Topic Name, which uniquely identifies the Topic within a Domain, and a Topic Type, which has the definition of the data contained within the Topic. Topics must be uniquely defined for a particular Domain. Even if two Topic Types were exactly the same defined, they would be considered two different Topics within the DDS infrastructure if the Topic Names do not match.

Taken from the RTI's DDS paper, "Within the definition of the Topic Type, one or more data elements of it can be chosen to be a Key". The DDS middleware will use this Key to match the data. Thus, by specifying a Key, an application can then retrieve data from DDS that either matches a specific Key, or the next Key of a sequence of Keys. In conclusion, Keys provide scalability [13].

DDS provides developers with the QoS capability to specify different parameters for each individual Topic, Writer or Reader. This constitutes the essence of data centricity in DDS. Thanks to these QoS parameters, developers can widely design their system. Some examples of these parameters are: Deadline, Durability, Latency Budget, Liveliness, Ownership, Partition, etc [13].

3.6. Real-Time Publish-Subscribe Protocol

The Real-Time Publish-Subscribe (RTPS) protocol started to be used in industrial automation and was specifically developed to support the unique requirements of

data-distribution systems [40]. The industrial automation community defined requirements for a standard publish-subscribe protocol were very similar to the ones defined for the DDS targeted application domains. The OMG commented in 2008: "As a direct result, a close synergy exists between DDS and the RTPS wire-protocol, both in terms of the underlying behavioral architecture and the features of RTPS."

The RTPS protocol was designed to be able to run over connectionless transmission models, such as UDP. As described in the OMG paper, "In fact, it is sufficient that the transport offers a connectionless service capable of sending packets best-effort. That is, the transport need not guarantee each packet will reach its destination or that packets are delivered in-order. Where required, RTPS implements reliability in the transfer of data and state above the transport interface." To sum up, this does not exclude RTPS from being implemented on top of reliable transports, but it just allows the protocol to be used in a wider range of transports. Therefore, its main features make RTPS an excellent match for a DDS wire-protocol [12].

3.7. Internet Group Management Protocol

The Internet Group Management Protocol (IGMP) is a communications protocol that runs on top of IP used by hosts and routers on IPv4 networks to create multicast group memberships, where messages can be send to all the members of that group at a single time. It can be used for one-to-many networking applications such as gaming or online video streaming [41]. Currently, there are three versions of the protocol [11].

IGMP provides communication between the client computer and the router on the network layer, just like other network management protocols such as ICMP. A client (host) requests membership to a group through its local router to become part of the communication, whereas a router just listens to these requests and periodically sends out subscription queries. In our approach, we are concerned about IGMP's basic operation and common messages sent between the hosts and router. The following messages could correspond to the normal IGMP's behavior:

- General Query: sent periodically from to router to the IP address 224.0.0.1 (all hosts group) to check if there is any host that wants to join a certain group and receive their own particular multicast traffic, or simply check if the hosts of that group are still alive and willing to continue listening. An interval can be set for periodically query the hosts to join a group.

- Group-specific Query: similar to a General Query, but just targeting a particular group instead of all the groups of the attached network.

- Membership Report: once the router sends a General or Group-specific Query, those hosts interested in listening to the multicast group's traffic or new hosts willing to join the group send back their membership reports to the router. If a host does not reply to the router's query means that is no longer interested in that group's messages. Besides, it is possible that a host sends its Membership Report to a router without previously receiving a query [42].

- Leave Report: when a host wants to leave a group, it will send a "leave message" to its specific group address. If the host sends a general leave group message to the all-router multicast group (224.0.0.2), that means that all the host in the subnet want to leave the group. Therefore, if after that the router sends two group-specific queries to ensure the absence of all members in the group, there would be no response.

4. Network Constraints

Network constraints are what define the normal behavior of the network. Although some constraints that define the structure of each packet are handled by the parser, constraints involving multiple packets are handled by the constraint engine. Usually, IDS use rules or events to represent the normal flow pattern of the network traffic. However, as I mentioned before, we decided to introduce the concept of constraints in a network because we believe that they represent more concise actions than rules. Thereby, we consider network constraints one of the key factors of an IDS and what makes our research approach different from previous IDS research.

As a primary version of our approach, we have developed a set of 11 constraints [5] based in the RTPS and IGMP protocols' documentation, sample network traffic and Distributed Denial of Service (DDoS) attacks based on DDS. Although most of the existing IDS are capable of handling this type of attacks, they require creating highly complicated and inflexible rules. Because of that reason, we propose much easier-to-understand and more-flexible-to-modify constraints to get the network fully secured against that type of attacks and reduce false alarms. Thus, our goal is to use these constraints to monitor the ongoing network traffic and generate alerts in case of violation.

Like I mentioned before, 11 constraints have previously been identified by the research group so far:

1. IGMP – Frequency of Join Report: A host is only allowed to send two successive Join Reports to a specific group.
2. IGMP – Frequency of Membership Report: Between two queries, a host can send its Membership Report only once to its specific group address.
3. IGMP – Destination of Membership Report: After receiving a General Query, a host will eventually send its Membership Report.
4. IGMP – Validity of Membership Report: After leaving a group, a host will not send any Membership Report to any multicast group unless it joins again.
5. RTPS – Validity of participants: All the Publishers and Subscribers in DDS must be a valid member of an IGMP multicast group. These participants should have sent a Membership Report before showing their interest in a Topic.
6. RTPS – Arrival of Participants: After joining a multicast group, participation should be announced from that host within a constant period of time.
7. RTPS – Participant's dual role: A participant cannot be Publisher and Subscriber of the same Topic at the same time.
8. RTPS – Subscriber turning Publisher: A Topic Key can only be published from a specific set of hosts.
9. RTPS – Publishing frequency: In synchronous mode, a Publisher has to publish in a fixed frequency rate.

10. RTPS – Quality of Service: The QoS policy between a Publisher and a Subscriber cannot be altered during run-time of the application.
11. RTPS – Validity of communication: Only a valid Publisher and a valid Subscriber can communicate.

Although we had firstly defined three as the number of constraints that I should develop, I have finally contributed to the project by coming up with a new approach to represent and evaluate **five** out of those eleven constraints. Its purpose was to compare the results of evaluating sample RTPS and IGMP traffic data against these five constraints with my approach to my partner Siam Hassan's approach. These constraints are the following:

C1. IGMP – Frequency of Join Report: A host is only allowed to send two successive Join Reports to a specific group.

According to the IGMP manual, a host can send a Membership Report to the router whenever it wants to join a multicast group without previously having received a Membership Query. We will call this particular type of Membership Report "Join Report". Additionally, the host can send another Join Report to the same router covering the possibility of the previous message being lost or damaged. The host can do that several times within a short period of time. Consequently, this will result in a DDoS attack [43], causing other hosts become unresponsive if we do not stop it.

So here is the foundation of our first constraint. Limiting a host to send two successive Join Reports to a router avoids the possibility of denying the service to other legitimate users in the multicast group and saturating the network.

C5. RTPS – Validity of participants: All the Publishers and Subscribers in DDS must be a valid member of an IGMP multicast group. These participants should have sent a Membership Report before showing their interest in a Topic.

An illegitimate host can secretly listen to a conversation by becoming a Reader of a certain Topic and gain access to unauthorized data. Analogously, a host can interfere and push malicious data into the network by becoming an illegitimate Publisher on a Topic.

Constraint 5 makes sure that all the Publishers and Subscribers of the private network are authorized. Thereby, considering that DDS depends on multicast groups, every host should be a valid multicast member and become a Participant of a Topic before it can read or write data related to that Topic.

C7. RTPS – Participant's dual role: A participant cannot be Publisher and Subscriber of the same Topic at the same time.

As we already know, a Topic is the link that connects Publishers and Subscribers. Referring to the DDS manual, it is perfectly possible that a Participant can become a Publisher and a Subscriber of the same Topic at the same time. Hence, a Participant can listen to data on a certain Domain, alter this data and, eventually, forward it to the multicast group. This fact can lead to two possible attacks: a malicious Publisher sending illegitimate data; or a denial of service caused by increasing the network's traffic.

Accordingly, Constraint 7 prevents these scenarios by making sure that a current Publisher of a certain Topic in a Domain does not become a Subscriber of the same Topic in that Domain, and vice versa.

C8. RTPS – Subscriber turning Publisher: A Topic Key can only be published from a specific set of hosts.

RTPS is used in Air Traffic Control (ATC) [44] worldwide to provide securer air traffic management and communications. For instance, there must be just a specific set of radar information publishers, or a specific machine which allows a plane to land. In ATC, this means very sensible information that must be carefully taken care of to provide a safe and efficient movement of aircrafts in the airspace.

Constraint 8 refers to defining a set of Publishers responsible for sending a restricted type of information. Therefore, any machine publishing restricted information that does not belong to the set of allowed Publishers for that Topic constitutes an intrusion.

C11. RTPS – Validity of communication: Only a valid Publisher and a valid Subscriber can communicate.

During an RTPS unicast communication, DATA packets are constantly being sent from Publishers to Subscribers. In our previous constraints we were checking that either the Participants were members of a valid multicast group or that Publishers were legitimate. Now, we are taking into consideration that when a Publisher sends a DATA packet, its destination is a valid Subscriber.

For this type of constraints, it is not enough to check the validity of the Publisher, as an authorize host could be publishing "legitimate" information to intruders. Hence we must check that every DATA packet sent in our network is coming from a trustable Publisher and reaching an allowed Subscriber.

5. Implementation

Once the packets are scanned and successfully parsed by the previous project's modules (Figure 1), they are passed to the Constraint Engine. According to its protocol, it already receives the necessary parameters of each packet to deal with the constraints. Its basic function is to evaluate the data structures received by the Parser and prompt an alert when a constraint is violated. Figure 6 shows a simplified framework of constraint based IDS.

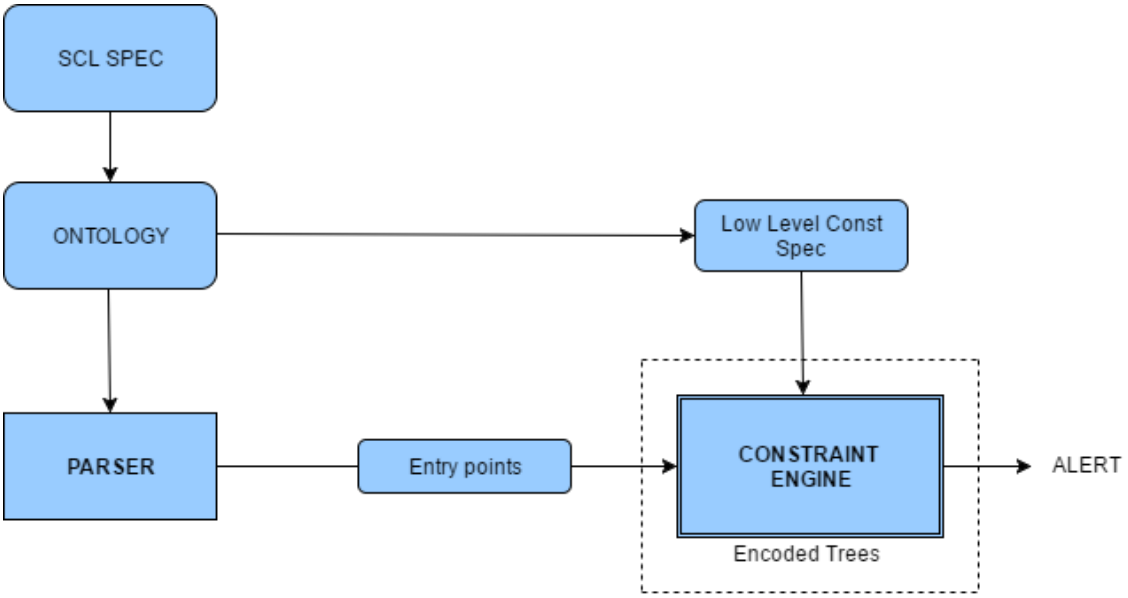

Fig. 6. IDS Framework.

As the project is coded in C and, unlike C++, there was not any tree or Directed Acyclic Graph (DAG) [45] structure available to use, my first approach was to develop our own tree class and tree structure myself, as well as all its functionality. Moreover, the trees were not always binary trees [46], and the number of children of each node could vary depending on the constraint, making it more difficult to represent our data. Every time a packet arrived, I had to loop through the tree to populate the leaf nodes, as well as to evaluate it (Figure 7 shows how the constraint trees look like). Hence, this approach resulted to be O(log n) and, therefore, considerably costly in time [47].

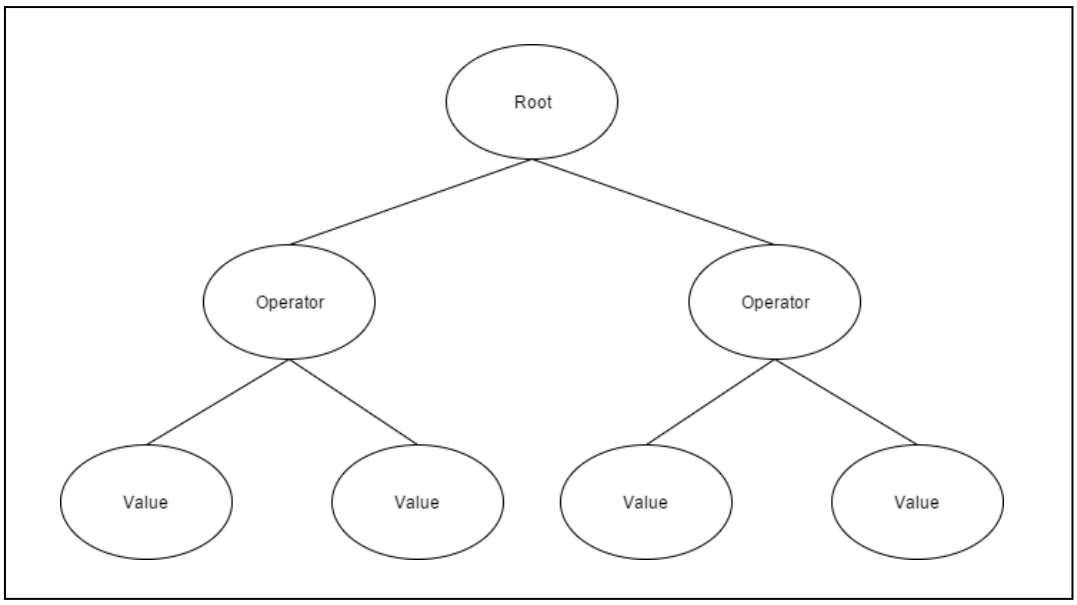

Fig. 7. Constraint's tree skeleton.

As a result, I had to develop a better approach. This approach is still using trees to represent and solve the constraints. However, instead of creating a tree class, I have coded a new DAG structure (Figure 8), which represents a constraint as an array of nodes by just keeping a pointer to the Root Node and the length of the structure. With this structure I can uniquely represent a general skeleton for each single constraint. Also, every time I need to create a new instance for the tree I just have to allocate some memory and do a memcopy() [48], without having to walk through the tree to copy it.

```
struct Dag
{
  struct node* root; //pointer to the first element of the array
  int iLength; //length of the array
};
```

Fig. 8. Snippet of the DAG structure.

Consequently, I have also developed a node structure (Figure 9) to be able to represent the differences between the operator and the leaf nodes. These differences refer to their values and number of children. Each node is composed by a value, an integer that indicates the position where its children start to be allocated in the array, an integer referring to its number of children and a type of node. Knowing the position and number of children allows us to build the hierarchy of the tree. The "eType" field indicates whether the node is an Operator Node or a Leaf node. Depending on the Leaf Node's type (integer or char) its "vValue" field may vary. Finally, I distinguish between the various types of Operator Nodes: "andOp", "orOp", "eqOp" and "notEqOp" correspond to their logical expressions "&&", "||", "==" and "!=". Whereas the "viol" and "aff" operators only refer to Root Nodes. The "viol" operator means that that tree

should prompt an alert if its evaluation is not satisfied, while the “aff” operator means that an alert should be prompt if the tree’s evaluation is satisfied.

```
enum eNodeType {
    op,
    leafStr,
    leafInt
};

enum eOperator{
    viol, //violation tree root
    aff, //affirmation
    andOp,
    orOp,
    eqOp,
    notEqOp
};

union data{
  const char* sData;
  uint32_t iData;
  enum eOperator eOp;
};

struct node{
   union data vValue;
   int iPosChildren; //position of the array where the its children start to be allocated
   int iNumChildren; //number of positions that the children occupy in the array
   enum eNodeType eType; //operator, leafString or leafInteger
};
```

Fig. 9. Snippet of the Node structure.

Due to this structure, I am able to know the exact position of every node in the array for each constraint, making every operation made on a node O(1), accessing directly to the position without having to loop through the tree. This will be easier to understand if we take a look to Figure 10 (Constraint 1 tree). Thereby, the four constraints’ trees I have developed and their corresponding snippets of code using this tree structure are as follow:

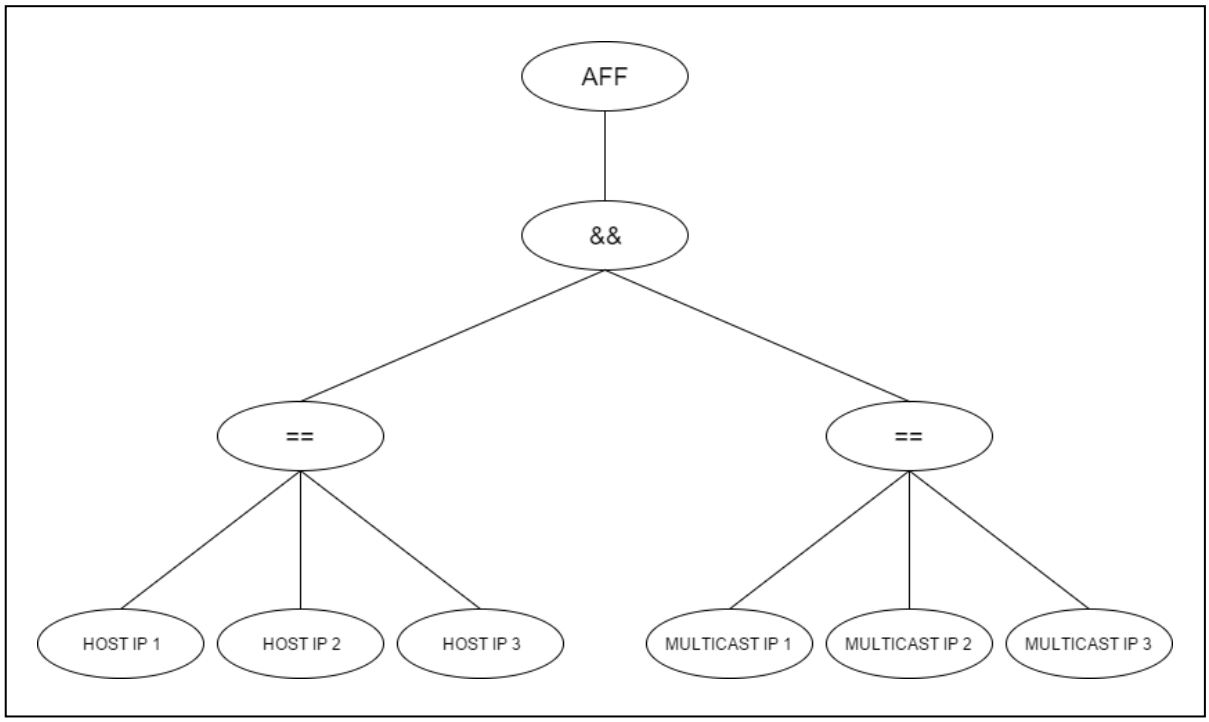

Fig. 10. Tree drawing of Constraint 1.

```
struct node constraint1[10] = {
        {.vValue = {.eOp = aff}, .iPosChildren = 1, .iNumChildren = 1, .eType = op}, //root: Affirmation node
        {.vValue = {.eOp = andOp}, .iPosChildren = 2, .iNumChildren = 2, .eType = op}, //1: and
        {.vValue = {.eOp = eqOp}, .iPosChildren = 4, .iNumChildren = 3, .eType = op}, //2: eq
        {.vValue = {.eOp = eqOp}, .iPosChildren = 7, .iNumChildren = 3, .eType = op}, //3: eq
        {.vValue = {.iData = 0}, .eType = leafInt}, //4 ip host 1
        {.vValue = {.iData = 0}, .eType = leafInt}, //5 ip host 2
        {.vValue = {.iData = 0}, .eType = leafInt}, //6 ip host 3
        {.vValue = {.iData = 0}, .eType = leafInt}, //7 ip multicast 1
        {.vValue = {.iData = 0}, .eType = leafInt}, //8 ip multicast 2
        {.vValue = {.iData = 0}, .eType = leafInt} //9 ip multicast 3
    };
```

Fig. 11. DAG structure for Constraint 1.

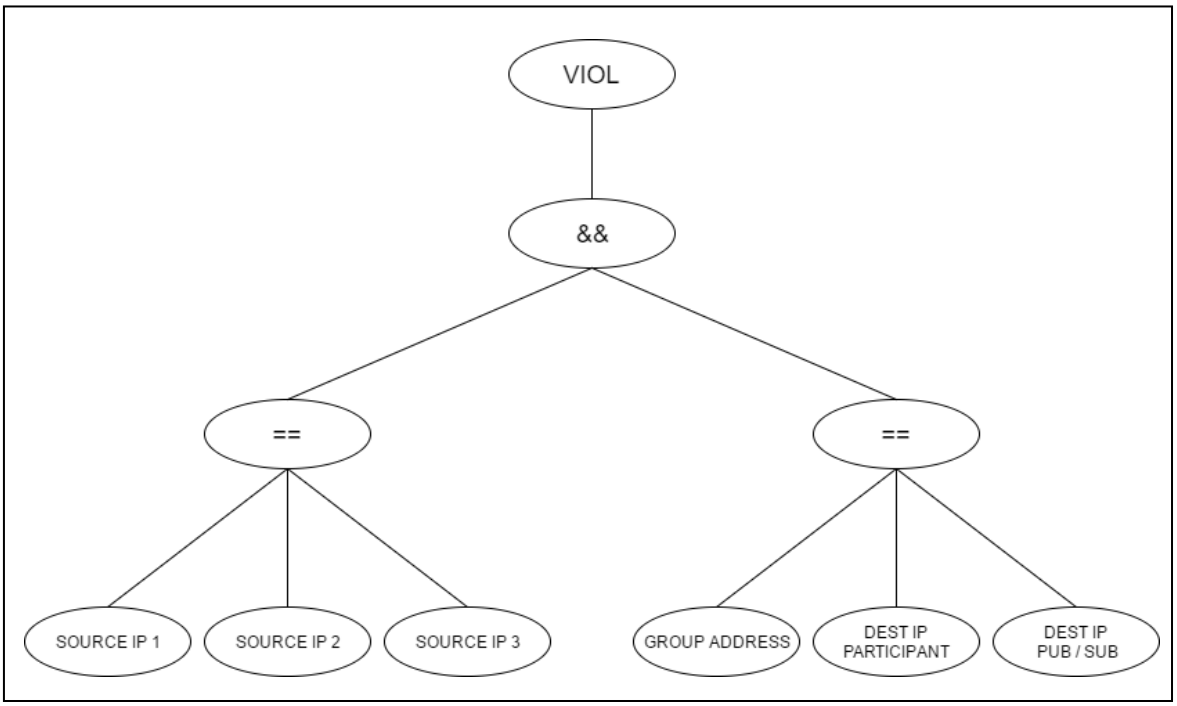

Fig. 12. Tree drawing of Constraint 5.

```
struct node constraint5[10] = {
        {.vValue = {.eOp = viol}, .iPosChildren = 1, .iNumChildren = 1, .eType = op}, //root: Violation node
        {.vValue = {.eOp = andOp}, .iPosChildren = 2, .iNumChildren = 2, .eType = op}, //1: and
        {.vValue = {.eOp = eqOp}, .iPosChildren = 4, .iNumChildren = 3, .eType = op}, //2: eq
        {.vValue = {.eOp = eqOp}, .iPosChildren = 7, .iNumChildren = 3, .eType = op}, //3: eq
        {.vValue = {.iData = 0}, .eType = leafInt}, //4 source ip 1
        {.vValue = {.iData = 0}, .eType = leafInt}, //5 source ip 2
        {.vValue = {.iData = 0}, .eType = leafInt}, //6 source ip 3
        {.vValue = {.iData = 0}, .eType = leafInt}, //7 group address
        {.vValue = {.iData = 0}, .eType = leafInt}, //8 destination ip participant
        {.vValue = {.iData = 0}, .eType = leafInt} //9 destination ip publisher/subscriber
    };
```

Fig. 13. DAG structure for Constraint 5.

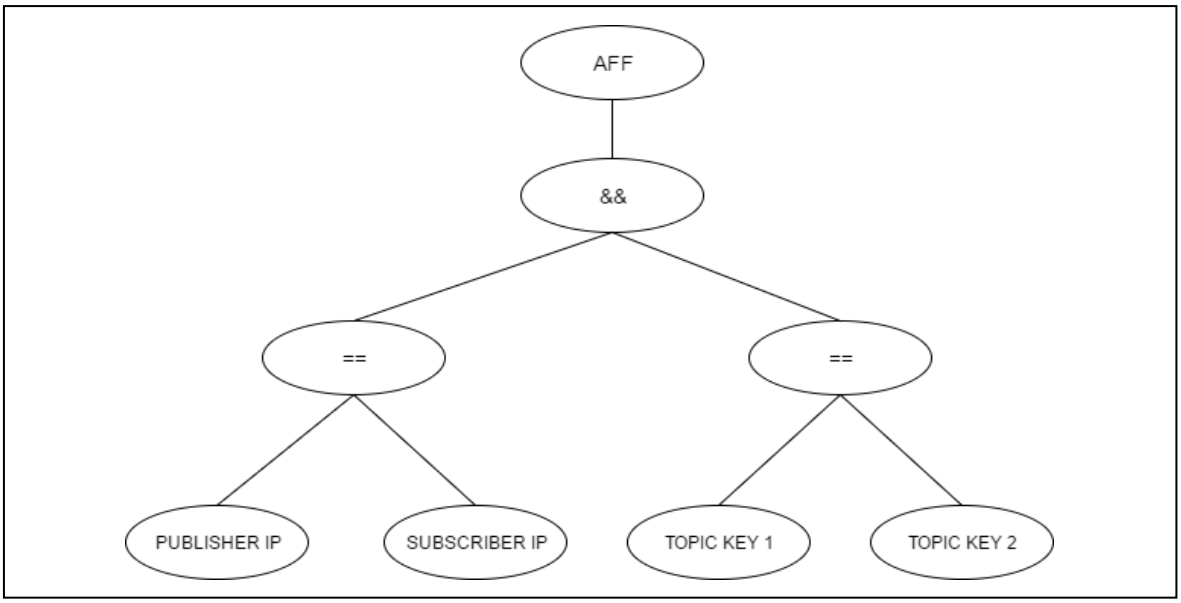

Fig. 14. Tree drawing of Constraint 7.

```
struct node constraint7[8] = {
        {.vValue = {.eOp = aff}, .iPosChildren = 1, .iNumChildren = 1, .eType = op}, //root: Affirmation node
        {.vValue = {.eOp = andOp}, .iPosChildren = 2, .iNumChildren = 2, .eType = op}, //1: and
        {.vValue = {.eOp = eqOp}, .iPosChildren = 4, .iNumChildren = 2, .eType = op}, //2: eq
        {.vValue = {.eOp = eqOp}, .iPosChildren = 6, .iNumChildren = 2, .eType = op}, //3: eq
        {.vValue = {.iData = 0}, .eType = leafInt}, //4 source ip pub
        {.vValue = {.iData = 0}, .eType = leafInt}, //5 source ip subs
        {.vValue = {.iData = 0}, .eType = leafInt}, //6 topickey 1 1
        {.vValue = {.iData = 0}, .eType = leafInt}, //7 topickey 2
    };
```

Fig. 15. DAG structure for Constraint 7.

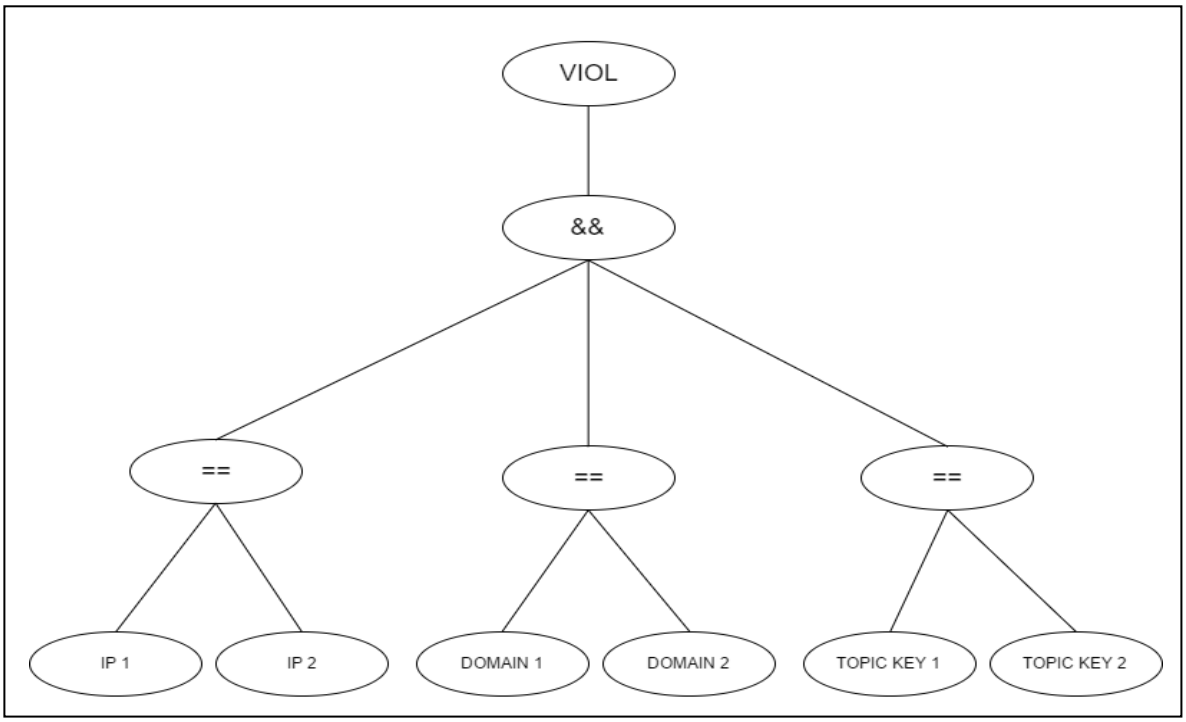

Fig. 16. Tree drawing of Constraint 8.

```
struct node constraint8[11] = {
        {.vValue = {.eOp = viol}, .iPosChildren = 1, .iNumChildren = 1, .eType = op}, //root: Violation node
        {.vValue = {.eOp = andOp}, .iPosChildren = 2, .iNumChildren = 4, .eType = op}, //1: and
        {.vValue = {.eOp = eqOp}, .iPosChildren = 6, .iNumChildren = 2, .eType = op}, //2: eq
        {.vValue = {.eOp = eqOp}, .iPosChildren = 8, .iNumChildren = 2, .eType = op}, //3: eq
        {.vValue = {.eOp = eqOp}, .iPosChildren = 10, .iNumChildren = 2, .eType = op}, //4: eq
        {.vValue = {.iData = 0}, .eType = leafInt}, //6 ip 1
        {.vValue = {.iData = 0}, .eType = leafInt}, //7 ip 2
        {.vValue = {.iData = 0}, .eType = leafInt}, //8 domain 1
        {.vValue = {.iData = 0}, .eType = leafInt}, //9 domain 2
        {.vValue = {.iData = 0}, .eType = leafInt}, //10 topic key 1
        {.vValue = {.iData = 0}, .eType = leafInt} //11 topic key 2
    };
```

Fig. 17. DAG structure for Constraint 8.

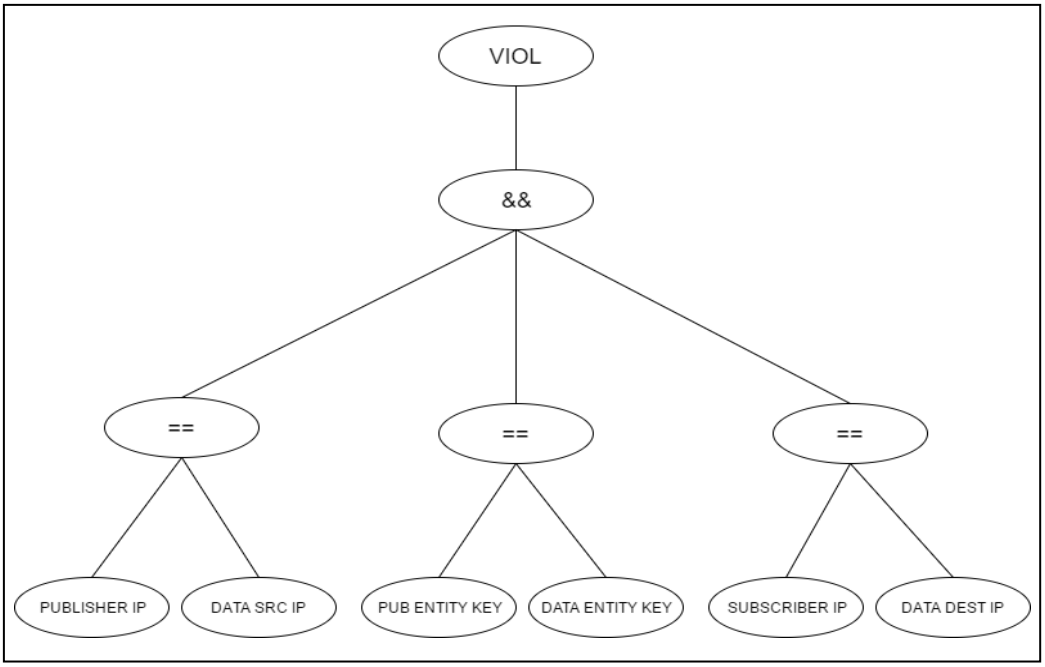

Fig. 18. Tree drawing of Constraint 11.

```
struct node constraint11[11] = {
        {.vValue = {.eOp = viol}, .iPosChildren = 1, .iNumChildren = 1, .eType = op}, //root: Violation node
        {.vValue = {.eOp = andOp}, .iPosChildren = 2, .iNumChildren = 3, .eType = op}, //1: and
        {.vValue = {.eOp = eqOp}, .iPosChildren = 5, .iNumChildren = 2, .eType = op}, //2: eq
        {.vValue = {.eOp = eqOp}, .iPosChildren = 7, .iNumChildren = 2, .eType = op}, //3: eq
        {.vValue = {.eOp = eqOp}, .iPosChildren = 9, .iNumChildren = 2, .eType = op}, //4: eq
        {.vValue = {.iData = 0}, .eType = leafInt}, //5 Publisher source IP
        {.vValue = {.iData = 0}, .eType = leafInt}, //6 DATA source IP
        {.vValue = {.iData = 0}, .eType = leafInt}, //7 Publisher entity id
        {.vValue = {.iData = 0}, .eType = leafInt}, //8 DATA entity id
        {.vValue = {.iData = 0}, .eType = leafInt}, //9 Subscriber source IP
        {.vValue = {.iData = 0}, .eType = leafInt}, //10 DATA destination IP
    };
```

Fig. 19. DAG structure for Constraint 11.

Additionally, I have created a Hash Table [49] structure for each type of constraint to keep track of the existing trees alive. Every Hash Table is formed by a key and a value. In my approach, I have created a “hashnode” structure where the key for each Hash Table is composed of the parameters that create that constraint and identify a single instance of the constraint, and the value is a pointer to the position where the Tree Root for that constraint is allocated. As a result, my Hash Table is an array of “hashnodes”. Thereby, this Hash Table allows me to easily find out whether a tree exists or not and, if it already exists, modify or delete it. Figure 20 shows how this structure looks like for Constraint 1. Similarly, it works for the other three constraints.

```
struct hashnodeC1
{
    uint32_t m_key1; // Source IP
    uint32_t m_key2; // group address
  struct Dag* m_pdata;
};

int insertValueC1(hashnodeC1 hashTable[], uint32_t key1, uint32_t key2, struct Dag* data);
int deleteValueC1(hashnodeC1 hashTable[], uint32_t key1, uint32_t key2);
struct Dag* GetTreefromC1(hashnodeC1 hashTable[], uint32_t key1, uint32_t key2);
int GetPosOfC1(hashnodeC1 hashTable[], uint32_t key1, uint32_t key2);

hashnodeC1 hashC1[SIZEC1];
```

Fig.20. Snippet of the Hash Table structure for Constraint 1.

Thanks to my partner's previous research, we have come up with a four-phase-procedure to solve the constraints. Thereby, when a new packet arrives and matches any of our constraints, it is classified to its corresponding phase. These four phases are Instantiate, Bind, Evaluate and Delete.

1. **Instantiate**: every constraint has an action that triggers the creation of a new tree and, consequently, makes us take into consideration different aspects of the machine causing it and follow up its behavior. This action is often related to a specific type of message. Thus, the Instantiate phase refers to the creation of a new tree skeleton and population of the first leaf nodes with the corresponding values of a packet that is sending a "trigger message" that matches any of our constraints. Besides, it is understood that such tree should not previously exist.

2. **Bind**: there are single packet constraints (already managed by the Parser) and multiple packet constraints. Within multiple packet constraints we can distinguish between two-packet constraints and more-than-two-packet constraints. The reason why we make this distinction is because of the Bind phase. This phase takes place when the arriving message corresponds to an expected message to fill in the remaining leaves from a previously created tree in the Instantiate phase. However, the constraint's tree is not ready to be evaluated yet, because is still waiting for more packets. Hence, exclusively more-than-two-packet constraints contain this Bind phase. On the contrary, two-packet constraints would go directly for evaluation when one of those messages arrives.
   In addition, usually just taking a look at the drawing trees it is easy to know if that constraint has a Bind phase or not. So, if the number of children of the last level Operator Nodes is greater than two, it means that the tree has a Bind phase. Although sometimes it might need the Bind phase even if it the number is two.

3. **Evaluate**: similarly to the Bind phase, when an expected packet matching an Instantiation tree arrives, its remaining leaves are populated. The difference with

the Bind phase is that the constraint does not have to wait for more packets to be evaluated. As a result, when this phase takes place, the tree is ready for evaluation. Furthermore, it will prompt an alert if the evaluation results to be a violation of the constraint.

4. **Delete**: although we might think that every tree is being deleted after its evaluation, this assumption is not always valid. It could be possible that a tree is deleted right after being evaluated, but that does not usually happen. Normally, the trees are kept alive until a specific message or action says the contrary (this concept will better be understood when we will go through each single constraint's case). When that happens, we will both free the memory of the tree itself and its reference in the Hash Map.

According to this four-phase procedure, the five constraints I have been working on were developed as described below:

5.1. CONSTRAINT 1: A host is only allowed to send two successive Join Reports to a specific group.

Just a reminder that a Join Report refers to a Membership Report sent by a host without previously having received a Membership Query from the router. This constraint was caused by the fact that a host might want to join a multicast group without have been queried by the router responsible for that group.

Instantiate: this phase takes place when a host sends an IGMPv2 or IGMPv3 Report to a multicast group address and such address does not match to a previous Membership Query made from that address. Resultantly, if it did not exist yet in the Constraint 1 Hash Table (which would mean that the Join Report is not the first one sent by that host and its tree has already been instantiated), we create a new Constraint 1 tree and we populate its first leaf nodes. Besides, we insert it in the correspondent Hash Table. Figure 21 represents the part of the code corresponding to Constraint 1's instantiation and Figure 22 shows how the tree would look like at the end of this phase.

```
int instantiateC1(uint32_t key1, uint32_t key2)
{
    if (GetPosOfC1(hashC1, key1, key2) == -1)
    {
        struct Dag * tmp1;
            tmp1 = (struct Dag *) malloc(sizeof(struct Dag));
            tmp1->root = (struct node *) malloc(d1.iLength*sizeof(struct node));
            memcpy(tmp1->root, d1.root, d1.iLength*sizeof(struct node));
            tmp1->root[4].vValue.iData = key1;
            tmp1->root[7].vValue.iData = key2;
        return insertValueC1(hashC1, key1, key2, tmp1); //insert DAG in the HashTable
    }

    return -1;
}
```

Fig. 21: Snippet of code of Instantiate phase for Constraint 1.

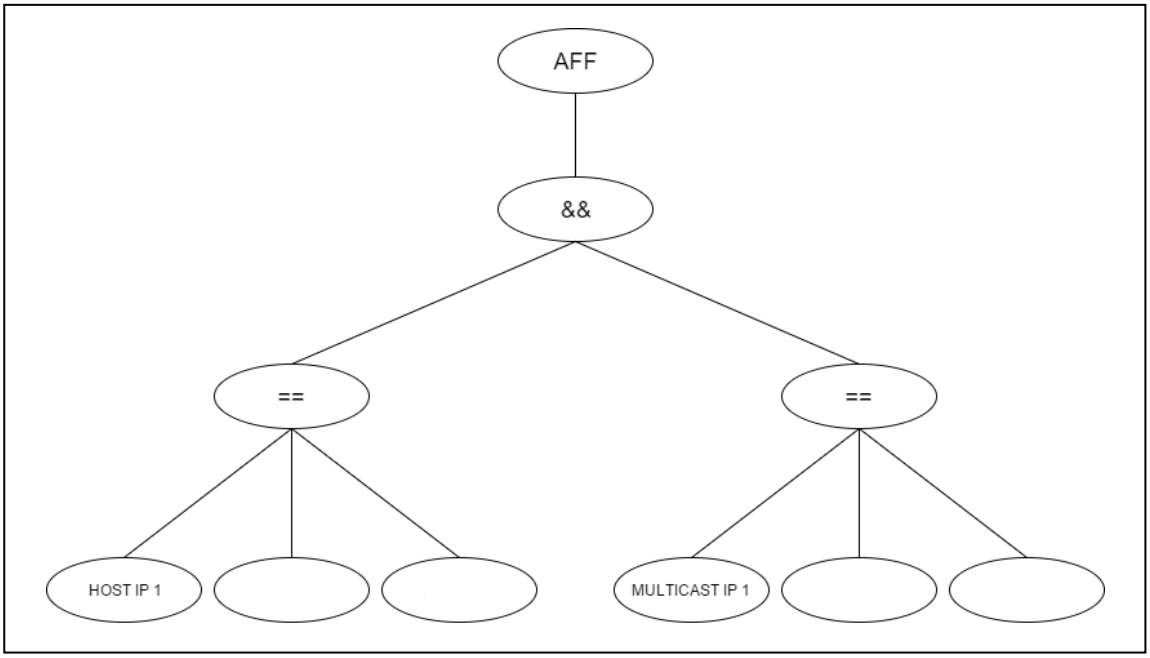

Fig. 22: Tree drawing after Instantiate phase for Constraint 1.

Bind: similarly to the Instantiate phase, it takes place when a host sends an IGMPv2 or IGMPv3 Report to a multicast group. However, this time it already exists a tree created due to a previous Join Report from that host to that router. Hence, we do not have to Instantiate a new tree, but simply insert the values of the respective Leaf Nodes of the tree that already exists. Figure 23 represents a snippet of the Bind phase for Constraint 1 and Figure 24 shows the actual state of its tree.

```
int bindC1(uint32_t key1, uint32_t key2)
{
    struct Dag* treeRoot = GetTreefromC1(hashC1, key1, key2);
    if (treeRoot->root[5].vValue.iData == 0  && treeRoot->root[8].vValue.iData == 0)
    {
        treeRoot->root[5].vValue.iData = key1;
        treeRoot->root[8].vValue.iData = key2;
        return 1;
    }

    return -1;
}
```

Fig. 23: Snippet of code of Bind phase for Constraint 1.

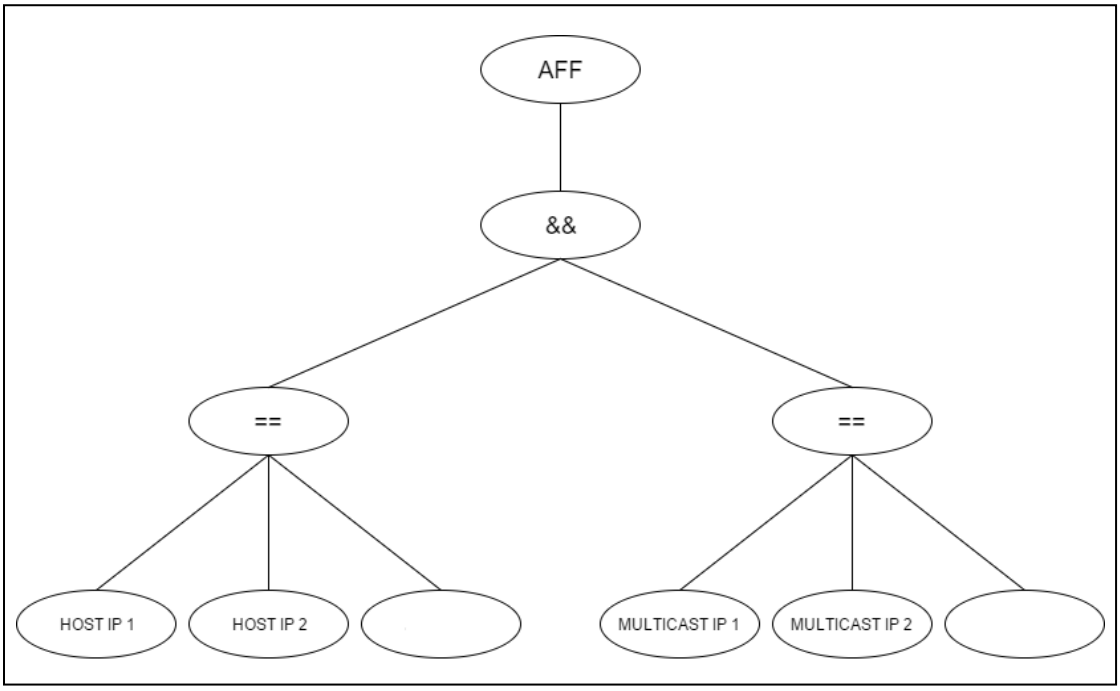

Fig. 24: Tree drawing after Bind phase for Constraint 1.

Evaluate: once again, the trigger action refers to when a host sends an IGMPv2 or IGMPv3 Report to a multicast group. But this time, the tree has already been instantiated and bound. As a consequence we fill in the last values of the Leaf Nodes as shown in Figure 25, and we send the tree for evaluation. The reason why we evaluate an identical copy of our tree is because our evaluate() function modifies the tree's values. This action will prompt an error if there is a failure in the evaluation function. Figure 26 represents the snippet of code for this phase of Constraint 1.

```
int evaluateC1(uint32_t key1, uint32_t key2)
{
    iC1++;

    struct Dag* treeRoot = GetTreefromC1(hashC1, key1, key2);
        treeRoot->root[6].vValue.iData = key1;
        treeRoot->root[9].vValue.iData = key2;
        struct Dag * tmp2;
        tmp2 = (struct Dag *) malloc(sizeof(struct Dag));
        tmp2->root = (struct node *) malloc(d1.iLength*sizeof(struct node));
        memcpy(tmp2->root, treeRoot->root, d1.iLength*sizeof(struct node));
        int result = evaluate(tmp2);
        free(tmp2);
    if (result == 1)
    {
        return 1;
    }

    struct in_addr addr;
    addr.s_addr = htonl(key1);
    char *token = inet_ntoa(addr);

    if (traceFileCons)
        fprintf(traceFileCons, "C1 :: Frequent membership reports from: %s\n", token);
    else
        printf("C1 :: Frequent membership reports from: %s\n", token);
    return -1;
}
```

Fig. 25: Snippet of code of Evaluate phase for Constraint 1.

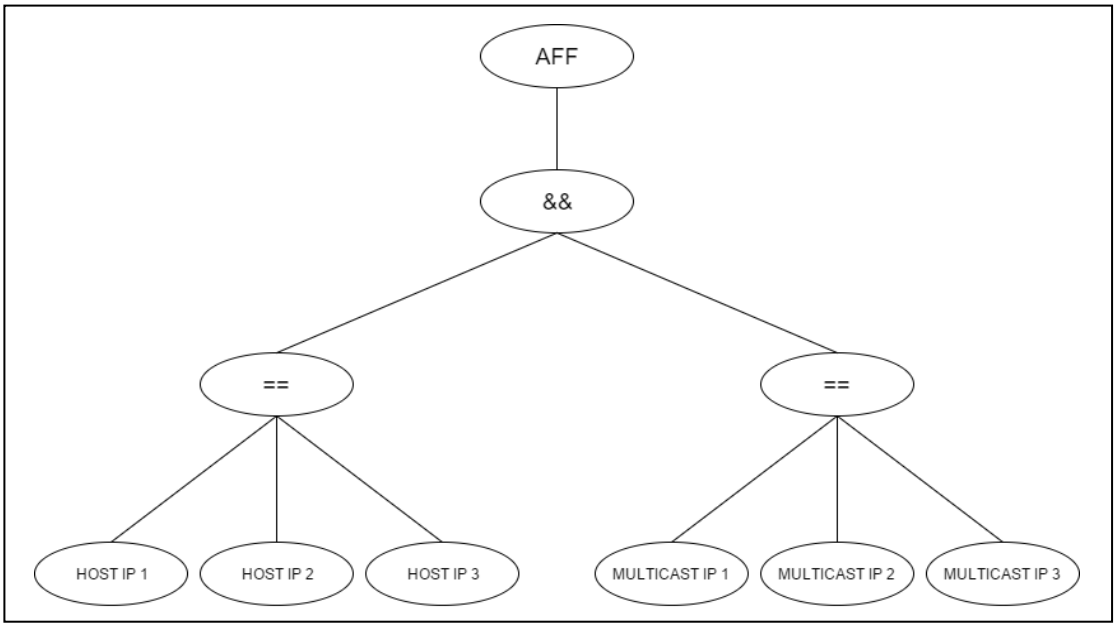

Fig. 26: Tree drawing after Evaluate phase for Constraint 1.

Delete: At this point we remove the tree from its Hash Table as well as the reference to its memory position. This action is caused by the arrival of a Membership Query from the router to the host which corresponded to Constraint 1 tree's keys. In other words, if

a host that has previously sent a Join Report to a router receives a Membership Query from the same router, we will delete the tree. Figure 27 represents the snippet of code corresponding to Constraint 1's deletion phase.

```
int destroyC1(uint32_t key1, uint32_t key2)
{
    return deleteValueC1(hashC1, key1, key2); //remove DAG from the HashTable
}
```

Fig. 27: Snippet of code of Delete phase for Constraint 1.

5.2. CONSTRAINT 5: All the Publishers and Subscribers in DDS must be a valid member of an IGMP multicast group. These participants should have sent a Membership Report before showing their interest in a Topic.

The inception of Constraint 5 is to avoid illegitimate hosts participating in a DDS communication. Equally to Constraint 1, as it involves more than to packets to determine if there is any violation of this constraint, we will again have the four phases.

Instantiate: if a host sends a Membership Report to a multicast group address from which had previously received a Membership Query, this will lead us to create a new tree for Constraint 5 and, consequently insert it in the correspondent Hash Table. Figure 28 shows how the tree looks like after being instantiated. Actually, the type of message that triggers the Instantiate phase for Constraint 5 (Membership Report) is the same that the one that caused the Instantiate, Bind and Evaluate phases for Constraint 1, as we can appreciate in Figure 29.

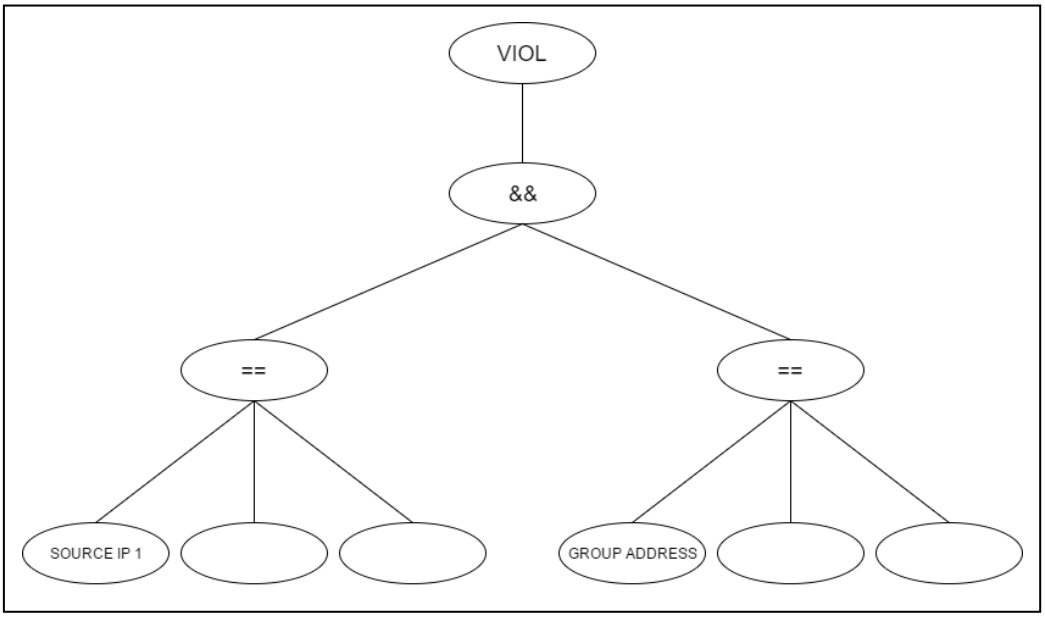

Fig. 28: Tree drawing after Instantiate phase for Constraint 5.

```
void V2Report_IGMP(struct V2Report_IGMP * v, HeaderInfo * h) {
    if (learnmode == 1)
        return;

    if (HasValuegroupaddressLists(v->groupaddr) == -1)
    {
        if (instantiateC1(h->srcIP, v->groupaddr) != -1) //IsInTree
        {
            if (bindC1(h->srcIP, v->groupaddr) != -1) //IsInTree
                evaluateC1(h->srcIP, v->groupaddr);
        }
    }

    if (v->groupaddr == GenerelQueryAddress)
    {
        for (int i = 0; i < SIZEFactgroupaddressLists; i++)
            instantiateC5(h->srcIP, factsgroupaddressLists[i].m_key);
    }
    else
    {
        instantiateC5(h->srcIP, v->groupaddr);
    }
}
```

Fig. 29: Snippet of code of handling an IGMPv2 Report.

Bind: before a host can write or read data related to a certain topic within a Domain, it must foster the Domain Participant's QoS specifications [13]. This assures that all Data Writers, Data Readers, Publishers and Subscribers share the same QoS parameters in the corresponding Domain. Thereby, the Bind phase takes place when a host that matches to a previous Instantiation tree for Constraint 5 sends a message to become a member of the Domain Participant. Figure 30 represents the state of Constraint 5's tree after the Bind phase.

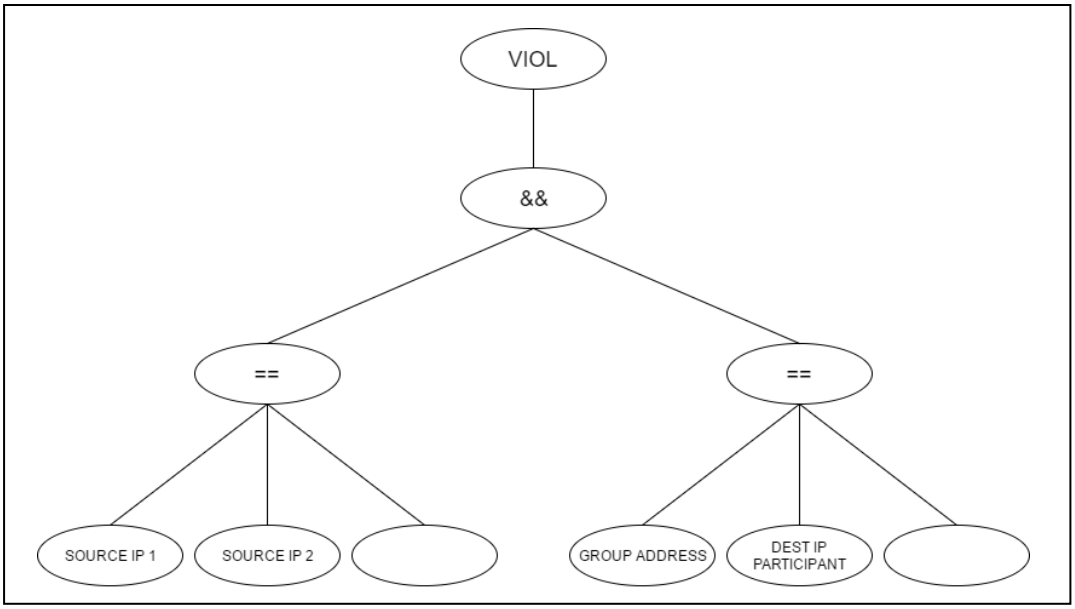

Fig. 30: Tree drawing after Bind phase for Constraint 5.

Evaluate: Constraint 5 is evaluated every time a Data(w) or a Data(r) are sent, checking that either the Publisher or the Subscriber for that Topic has previously joined the IGMP group and the Domain Participant. Thus, it is important to notice that we might have to go through the Evaluate phase for Constraint 5 without having done the corresponding Instantiate or Bind phases before. In that case, we would first check if there already exists its respective Instantiation tree for Constraint 5, prompting a violation of the constraint if it did not exist. Figure 31 shows how the Constraint 5's tree looks like after evaluation in the case of having gone through the previous instantiation and bound.

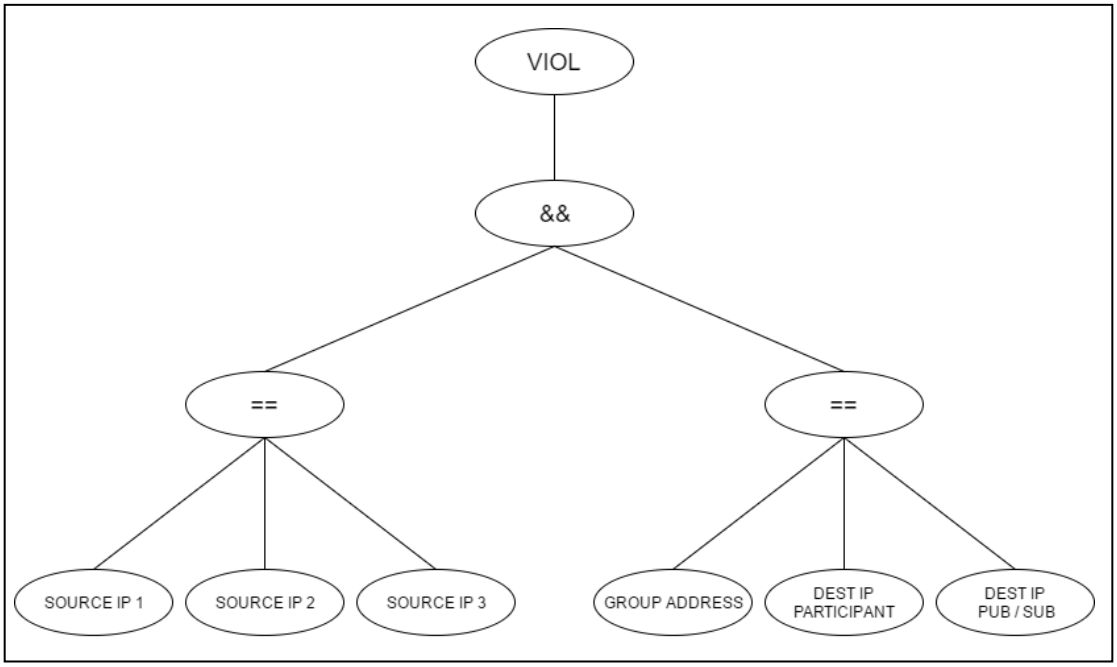

Fig. 31: Tree drawing after Evaluate phase for Constraint 5.

Delete: the trigger action for the Constraint 5's tree deletion is a Leave Report message from the appropriate host to the correspondent multicast group. So, this means that the host does not want to belong to that IGMP group anymore and we have to remove it from its respective Constraint 5's Hash Table.

5.3. CONSTRAINT 7: A participant cannot be Publisher and Subscriber of the same Topic at the same time.

Constraint 7 just involves two packets: a Publisher packet and a Subscriber packet. Therefore, we do not have a Bind phase this time. However, we need to differentiate between two possible situations: if a Publisher comes first and instantiates the tree, so the following Subscriber will be the evaluation packet (C7a); or, on the contrary, a Subscriber arrives first and the Publisher is the one evaluates the tree (C7b).

Instantiate: we instantiate a C7a's tree every time a Data(w) packet (which corresponds to a Publisher) arrives and it does not match to a corresponding instantiation tree created by the same host sending a Subscriber packet. Analogously, we instantiate a C7b's tree every time a Data(r) packet (which corresponds to a Subscriber) arrives and it does not match to any previously instantiated tree created by the same host sending a Publisher packet. In any case, we have to insert them in their respective Hash Table. However, their drawing trees look exactly the same. Figure 32 shows how both C7a's tree and C7b's tree look like after instantiation.

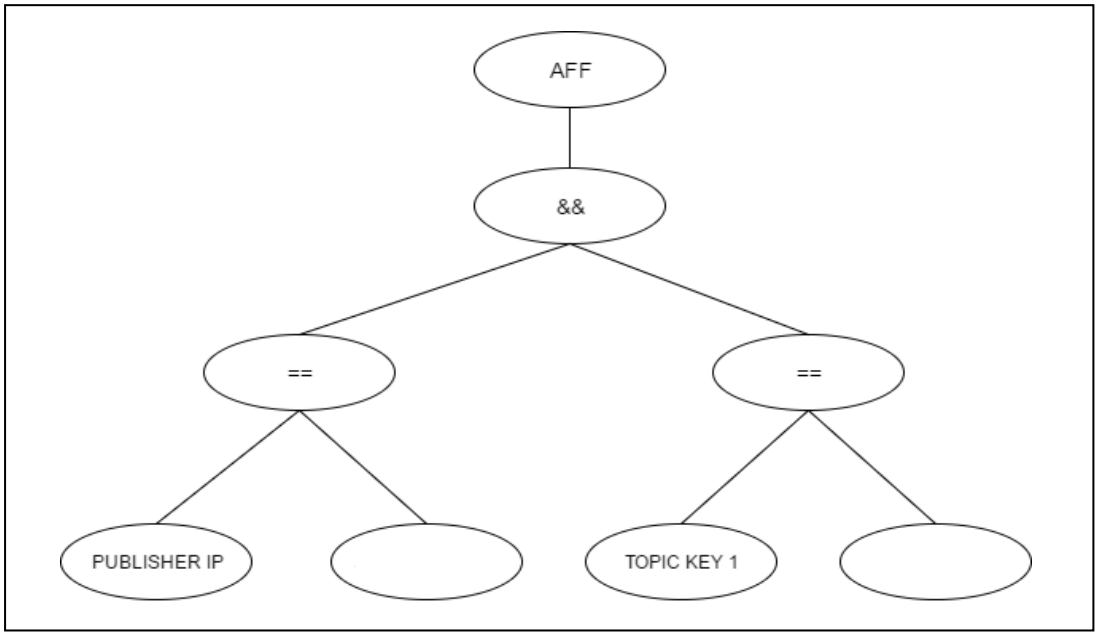

Fig. 32: Tree drawing after Instantiate phase for Constraint 7.

Evaluate: similarly to how we have instantiated the trees works the Evaluate phase. This is also triggered by a Publisher or a Subscriber packet. On the one hand, we evaluate C7a when a new Publisher packet arrives and it already exists its correspondent C7b's instantiation tree. On the other hand, we will send C7b for evaluation if a Publisher packet arrives and matches to its correspondent C7a's instantiation tree. Figure 33 illustrate the evaluation trees for both C7a and C7b.

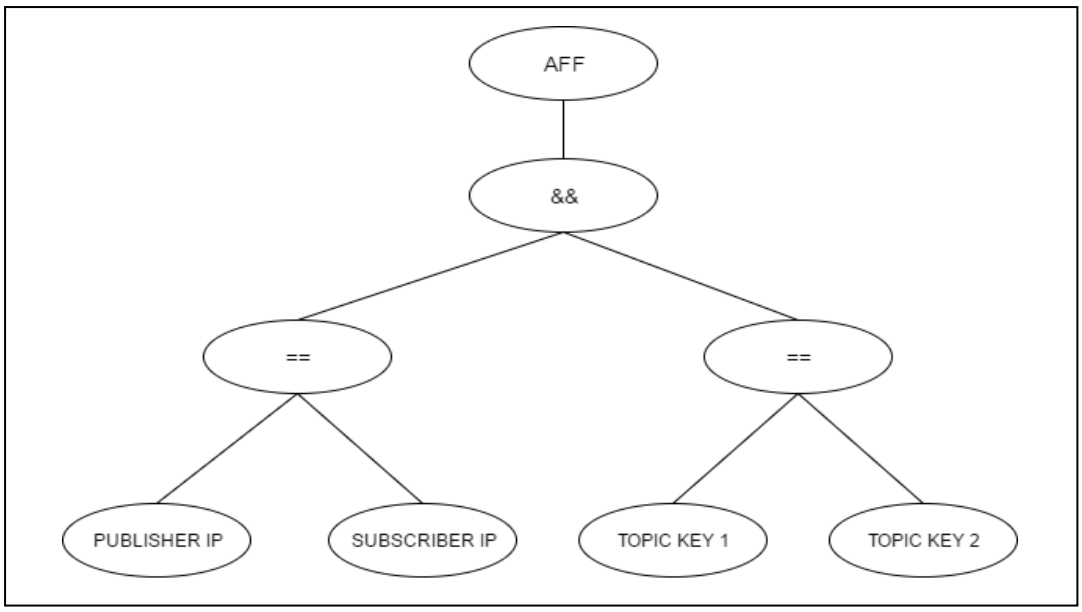

Fig. 33: Tree drawing after Evaluate phase for Constraint 7.

Delete: when a host sends a Leave Report and it matches to a previously created Constraint 7's tree we will both delete the respective tree and its reference in the Hash Table.

5.4. CONSTRAINT 8: A Topic Key can only be published from a specific set of hosts.

For every Topic there is a list of legitimate Publishers, which are the only ones allowed to push data about that Topic. As a result, Constraint 8 just needs to check if a Data(w) message comes from one of those legitimate hosts. It might seem that this is a single packet constraint and, consequently, it should be evaluated by the Parser. However, here is where we introduce the concept of "Learning Mode" [52]. Every other constraint I have mentioned before is part of the "Checking Mode", but our Constraint Engine also possesses a "Learning Mode" which has been developed by my partner. This "Learning Mode" allows our Constraint Engine to, as its own name implies, to learn from "facts" [52] and include them to our engine. Therefore, when we have a new Publisher packet we will include it to our "facts", letting us know in the future that such host is a valid Publisher. Hence, our Constraint Engine is able to update itself and learn from those updates, dealing with constraints that, as happens with Constraint 8, might appear to be single packet constraints and, as a consequence, validated by the Parser. Thus, as it does not involve more than two packets to be evaluated, this constraint only has three out of our four phases.

Instantiate: whenever we are in "Learning Mode" and we receive a message from a host wanting to become part of the Domain Participant, we will instantiate a new

Constraint 8's tree. Like this, we will also insert it in its respective Hash Table. Figure 34 represents Constraint 8's tree after instantiation.

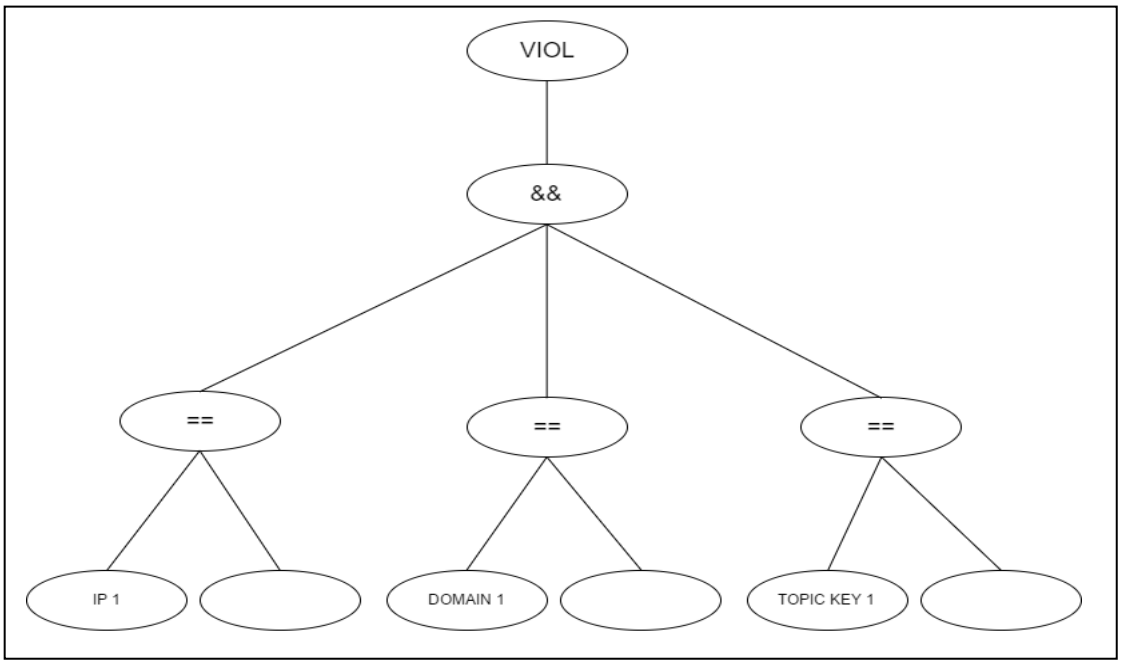

Fig. 34: Tree drawing after Instantiate phase for Constraint 8.

Evaluate: after having instantiated our tree for Constraint 8, the following incoming Publisher packets from that host in that Domain will be sent for evaluation, without going through a Bind phase. Also, notice that, as happened with Constraint 5, we might have to evaluate Constraint 8 without having previously instantiated its tree. In other words, we have to check if every Publisher is trustable, resulting this in a violation of Constraint 8 if that Publisher did not belong to a previous instantiated tree. Figure 35 shows the state of Constraint 8's tree after evaluation.

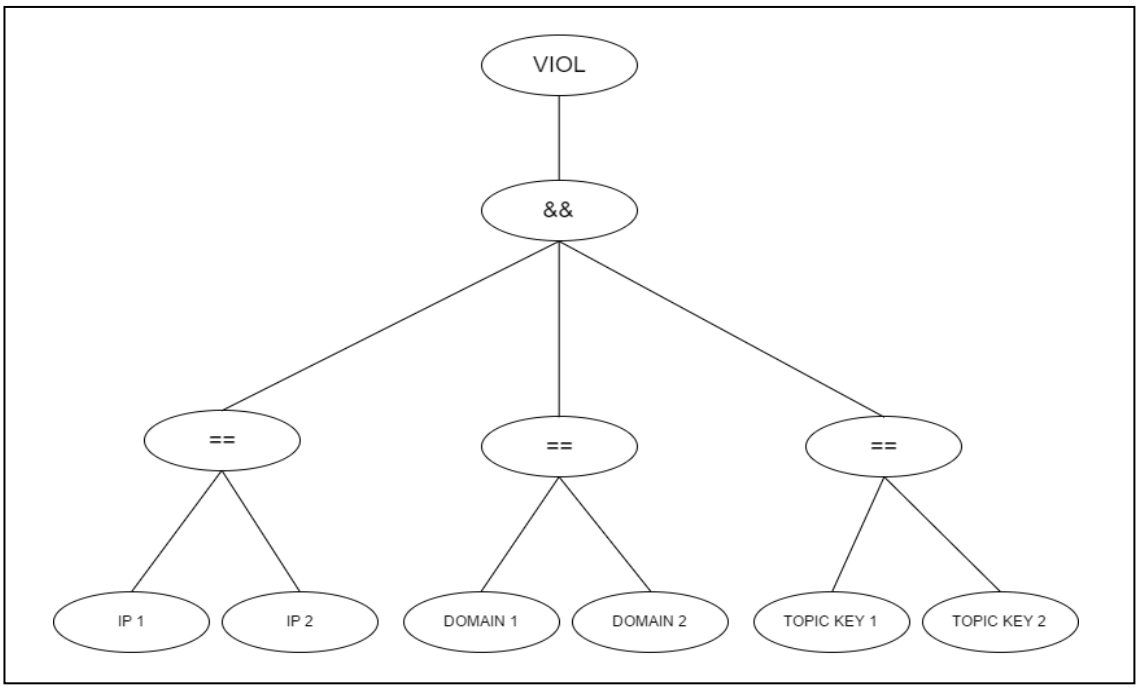

Fig. 35: Tree drawing after Evaluate phase for Constraint 8.

Delete: as it happens with most of the constraints, the Delete phase takes place when that specific host sends a Leave Report to the multicast group. Consequently, we have to update our "facts" this time apart from removing the tree and its reference in the Hash Table.

5.5. CONSTRAINT 11: Only a valid Publisher and a valid Subscriber can communicate.

Instantiate: the packet that triggers the instantiation of Constraint 11 is a Data(w) packet sent by a host that is notifying the router that wants to become a new Publisher. As a consequence, when this type of packet arrives we will store in the tree the packet's respective IP and entity key to, later on, insert it in the correspondent Hash Table. This will mean that the host is a validated Publisher. Also, the search keys to find that tree in the Hash Table will be the Topic Key and the Publisher IP, so those are the values we will have to look for in the Bind phase. Figure 36 shows the state of the Constraint 11 tree after being instantiated.

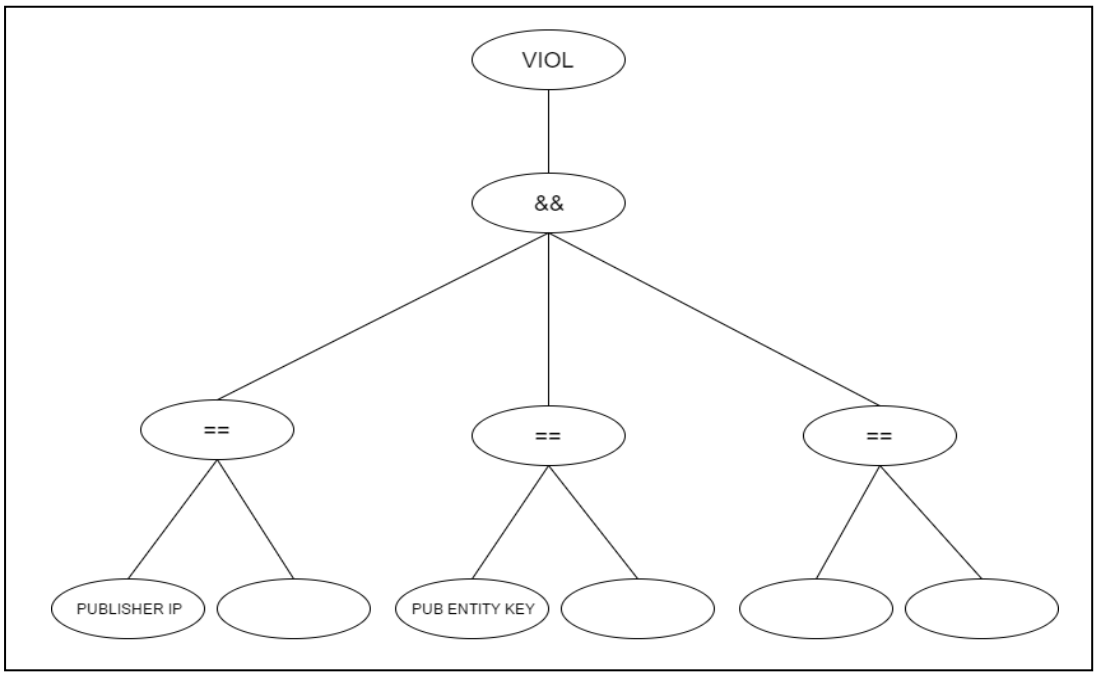

Fig. 36: Tree drawing after Instantiate phase for Constraint 11.

Bind: the Bind phase takes place when a Data(r) packet from a Subscriber arrives. As mentioned before we will have to look for the Topic Key and the Publisher IP that that Subscriber is willing to listen to. Depending on the result of that search we will have two different options: if we cannot find the tree it would be an error, or if, on the contrary it does, we would have to go for an intermediate phase called "Split" [52]. This consists on making an exact copy of the instantiation tree, populate the new values and update the new search keys, because the evaluation packet for Constraint 11 will be looking for the Publisher IP, Publisher port and Subscriber IP instead of the Topic Key and Publisher IP. Consequently, we will also delete the previous instantiation tree. Figure 37 helps visualizing the state of the tree after this phase.

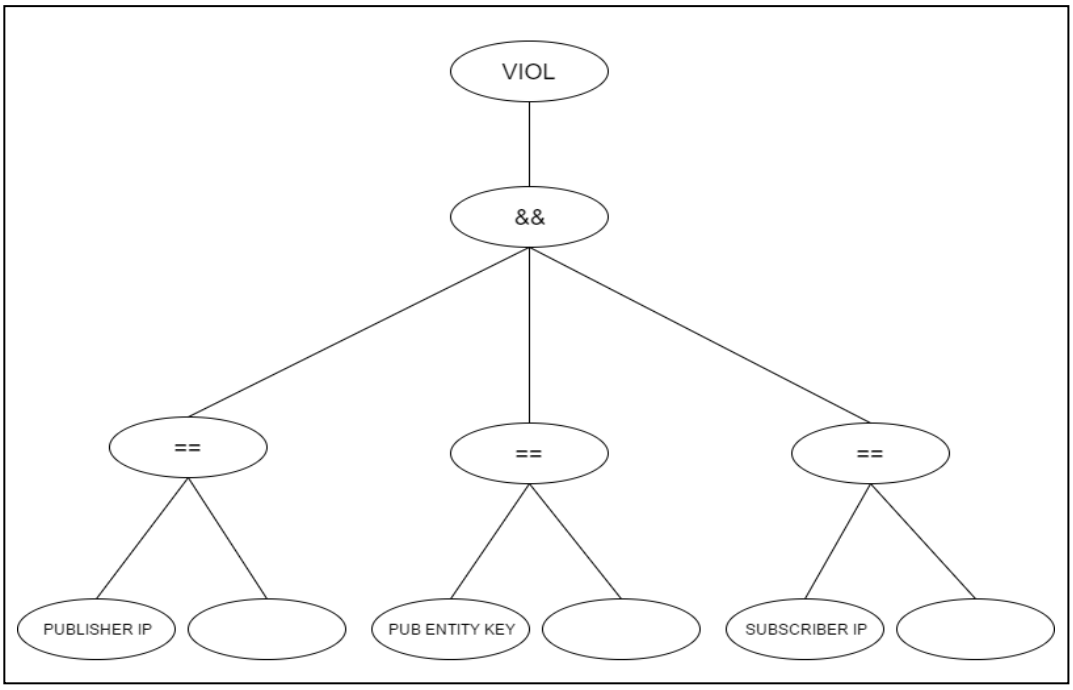

Fig. 37: Tree drawing after Bind phase for Constraint 11.

Evaluate: when a DATA packet is sent from a Publisher to a Subscriber Constraint 11 will go for evaluation. Thus, we will search for the Bind phase's keys and store the remaining Leaf nodes. If after finishing populating the tree all its nodes are validated, we can assure that the DATA was communicated from a legitimate Publisher to an authorized Subscriber. We can visually see represented the Constraint 11 tree after the evaluation phase takes place in Figure 38.

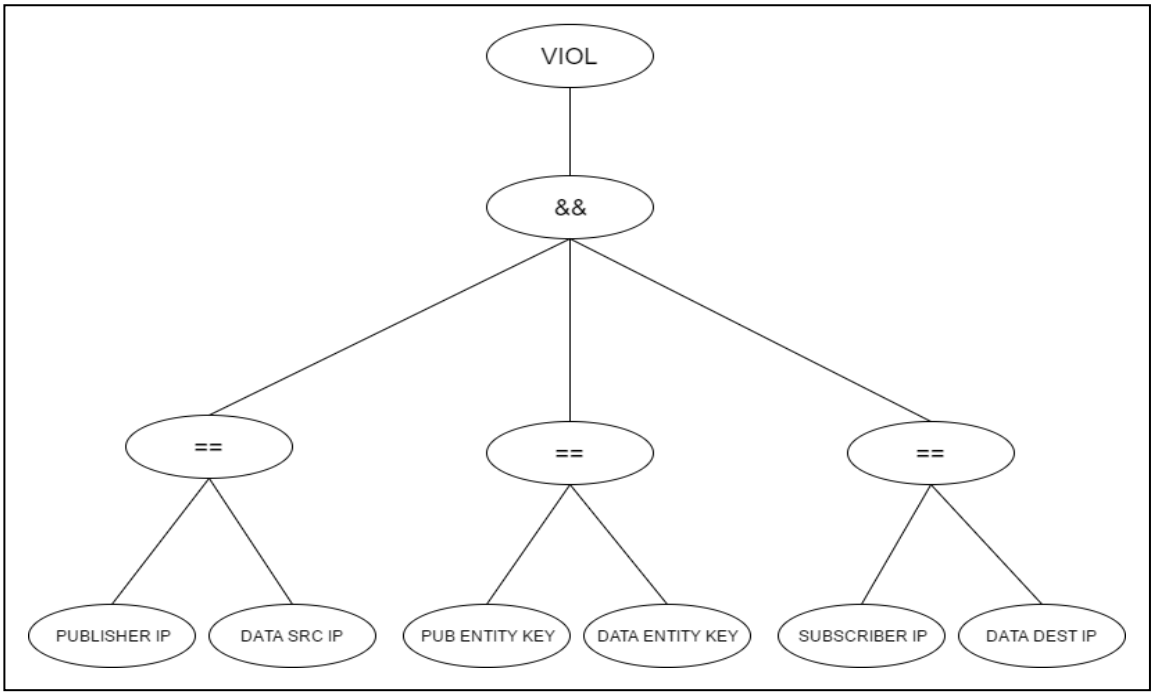

Fig. 38: Tree drawing after Evaluate phase for Constraint 11.

Delete: the corresponding tree for Constraint 11 will be destroyed when any of the respective participants leave the multicast group. Resultantly, that tree will also be removed from the Hash Table.

6. Evaluation

Assuming that all the trees corresponding to a same constraint look alike could mean that keeping the full tree structure is no longer needed. In other words, if we know that, for example, the third node of the Constraint 7's tree is always an "EQUAL" node, why do we have to loop through it and check that node's value every time we evaluate Constraint 7? Besides, why do we have to keep all the Operator Nodes for every instance of a constraint's tree if we already know what their values are always going to be? It should only be necessary to save the leaves' values to evaluate the constraint.

In fact, that is Siam's approach: to assume that the Operator Nodes for a certain constraint never change, so we do not have to keep them and save memory for them every time we create a new tree for that constraint. In addition, he is evaluating the trees by simply comparing the leaf values with the operators that "we assume do not vary", without looping through all the tree structure. However, we are not completely sure about that assumption.

Thus, the purpose of this research is to compare the results of my evaluation to Siam's, testing our Constraint Engine against sample generated traffic data. As a result, if the assumption that we are making is eventually true, we could test whether getting rid of the Operator Nodes and avoid walking through the trees is time costly significant.

Therefore, focusing in my approach, it consists on keeping a copy of the full tree structure every time we create a new instance of that tree and walking through every node of the tree when it is sent for evaluation without obviating whether the Operator nodes are always the same. So, in order to evaluate the trees using my approach I created a structure that behaves similarly to a stack, shown in Figure 39. Its basic functioning consists in pushing the Operator nodes to the stack until we find an Operator node whose children are Leaf nodes. In that case, we will evaluate its children and update the Operator node's value with that result. Following, we will start going backwards in the stack evaluating the Operator nodes of the higher levels with the results obtained in the previous lower levels of the tree.

```
struct stacks
{
    int m_ipos;
    int elems[256];
}sts;
```

Fig. 39: Snippet of the auxiliary stack structure.

In order to make the evaluation function easier, I have developed three auxiliary functions, which we can appreciate in Figures 40, 41 and 42. The first of them simply checks whether a certain node's children are Leaf nodes. Thanks to that function, I can

easily identify which nodes belong to the last Operator nodes level and stop pushing the following nodes to the stack. Secondly, the CompareValues() function receives two nodes and an logic operator as parameters, and returns the result of evaluating those two node's values against that parameter. This function results quite helpful to simplify the comparison between the Leaf nodes differentiating between the several types of logic operators. Finally, the getResults() method uses the previous function to gather the result of comparing the values of all of a certain node's children according to that node's operator, so I can compare more than two Leaf nodes and obtain a single result that will be used by the higher level Operator nodes to continue evaluating the tree.

```
int HasLeafNode(int pos, struct Dag *d)
{
    struct node Curr = d->root[pos];
    return (d->root[Curr.iPosChildren].eType != op);
}
```

Fig. 40: Snippet of the auxiliary function HasLeafNode().

```
int CompareValues(struct node n1, struct node n2, enum eOperator op)
{
    assert(n1.eType == n2.eType);

    switch (op)
    {
        case andOp:
        {
            if (n1.eType == leafStr)
                return n1.vValue.sData && n2.vValue.sData;
            return n1.vValue.iData && n2.vValue.iData;
        }
        case orOp:
        {
            if (n1.eType == leafStr)
                return n1.vValue.sData || n2.vValue.sData;
            return n1.vValue.iData || n2.vValue.iData;
        }
        case eqOp:
        {
            if (n1.eType == leafStr)
                return n1.vValue.sData == n2.vValue.sData;
            return n1.vValue.iData == n2.vValue.iData;
        }
        case notEqOp:
        {
            if (n1.eType == leafStr)
                return n1.vValue.sData != n2.vValue.sData;
            return n1.vValue.iData != n2.vValue.iData;
        }
        default:
            assert(0);
            return 0;
    }
}
```

Fig. 41: Snippet of the auxiliary function CompareValues().

```
int getResults(int pos, struct Dag * d)
{
    struct node curr = d->root[pos];
    if (curr.iNumChildren == 1)
    {
        int res = d->root[curr.iPosChildren].vValue.iData;
        if (curr.vValue.eOp == viol)
            return (res == 1); //True if Violation tree and the previous evaluations were true
        return (res == 0); //True if Affirmation tree and the previous nodes were not satisfied
    }

    int i = curr.iPosChildren;
    struct node n1 = d->root[i];
    i++;
    struct node n2 = d->root[i];
    if (CompareValues(n1, n2, curr.vValue.eOp) == 0)
        return 0;
    i++;

    while (i < curr.iNumChildren)
    {
        n2 = d->root[i];
        if (CompareValues(n1, n2, curr.vValue.eOp) == 0)
            return 0;
        i++;
    }

    return 1;
}
```

Fig. 42: Snippet of the auxiliary function getResults().

As a consequence, the actual evaluate() function takes advantage of the previously described auxiliary functions and returns the final evaluation result of a given tree. The way it works is the following: it starts pushing the Root node of the tree to the stack, and checks whether the Root's children are Leaf nodes. If they are not Leaf nodes, it pushes them to the stack. Then, I similarly do the same with the following Operator nodes until there is an Operator node whose children are Leaf nodes. In that case, it will stop pushing more nodes to the stack. Thus, I end up just having all the Operator nodes in the stack. What the algorithm does at this point is comparing the values of the leaves of that last Operator node according to the node's logic operator and updating the value of that node with the result of the comparison. Therefore, that Operator node's value will no longer be a logic operator, but a 1 or a 0 instead, depending on the result. Analogously, it does the same with the rest of the Operator nodes, walking the stack backwards and passing the results of each comparison to the higher nodes levels of the tree. Once it gets to the front of the stack and, consequently, the Root of the tree, it generates the result of the function depending on regarding if the Root is a "viol" or an "aff" operator.

```
int evaluate(struct Dag * d)
{
    sts.m_ipos = 0;
    sts.elems[sts.m_ipos] = 0;
    sts.m_ipos++;
    int count = 0;
    while (1)
    {
        count++;
        if (sts.m_ipos == 0)
        {
            assert(d->root[sts.elems[0]].eType == leafInt);
            return d->root[sts.elems[0]].vValue.iData;
        }
        int pos = sts.elems[sts.m_ipos - 1];
        if (HasLeafNode(pos, d) == 1)
        {
            sts.m_ipos--;
            int res = getResults(pos, d);
            d->root[pos].eType = leafInt;
            d->root[pos].vValue.iData = res;
        }
        else
        {
            int i = 0;
            for (i = 0; i < d->root[pos].iNumChildren; i++)
            {
                int index = i + d->root[pos].iPosChildren;
                sts.elems[sts.m_ipos] = index;
                sts.m_ipos++;
            }
        }
    }
    return 0;
}
```

Fig. 43: Snippet of the evaluate() function.

Resultantly, whenever a packet received on the PCAP file correctly parsed by the Parser is performing an action that causes that packet to be sent for evaluation against any of our constraints, it will call the previously mentioned evaluate() function and, consequently prompt an alert if it fails to be evaluated properly (if the function returns a 0).

In order to gather all the sample IGMP and RTPS traffic to evaluate my approach, our research group has used the Euroscope Simulator and ATC Display [50], which were also adapted from the Canadian Automated Air Traffic System (CAATS) [51] to use the RTPS protocol to communicate between the air traffic control centers, devices and controllers. Thanks to Euroscope, we are able to get virtual RTPS data generated by pilots flying for virtual airlines using flight simulators and air traffic controllers.

7. Conclusions and Future Work

7.1. Results and Conclusions

The purpose of this research project was to come up with and test a new tree-based approach for solving constraints in an Intrusion Detection System. Although my goal was to develop and test three out of our eleven IGMP and RTPS constraints, I have finally done **five**, so we could test the complete performance of my engine's approach. Adding Constraints 8 and 11 to the first required three constraints made possible to successfully test my tree structures through all of our four evaluation phases, including the "Split" condition later introduced by my partner, and the learning from "Facts".

As a result, after running my Constraint Engine against a 1.6GB PCAP file for ten times we obtained an average of the following results: **16.112 s** of **time**, with a **102.525 MB/s** of **throughput** and **828.203 Mb/s bandwidth**. Just remarking that a bandwidth of at least 180 Mb/s was already acceptable, so my engine was running almost 5 times faster. Thus, my approach turned to be just slightly slower than the non-tree-structure-based approach, with around 1.5s difference in time and around 300Mb/s less of bandwidth, which allows us to conclude that dealing with my tree structures does not significantly reduce the program's performance.

We have also demonstrated that our system is capable of dealing with multiple-packet constraints using UDP and keeping the information of the packets, apart from successfully perform a deep packet inspection of them.

The following table shows the average results of running my Constraint Engine against the 1.6GB PCAP file for ten times.

Packets Parsed	10.707.581
Packets Failed	113.435
Total Packets	10.821.016
Failure Rate	1,05 %
C1 Evaluated	0
C5 Evaluated	12
C7a Evaluated	36

C7b Evaluated	0
C8 Evaluated	66
C11 Evaluate	3193212
Time	16,112 s
Throughput	102,525 MB/s
Bandwidth	828,203 Mb/s

Table 1. Results after running a 1.6 GB PCAP file for ten times.

7.2. Future Work

Our future work firstly consists in automatically generate all the C implementations for the constraints using our research's group grammar in order to, in the future, provide a custom IDS for any private network with the possibility of easily adding new protocols to it. Also, new constraints should be found out and developed.

Furthermore, just as Bro is nowadays for the research community, our future-open-source IDS could be handful for private organizations willing to introduce their own protocols in the system, as well as widely-used tool for any network security and intrusion detection research.

8. Conclusiones y Trabajo Futuro

8.1. Resultados y Conclusiones

El propósito de este proyecto de investigación consistía en idear y probar un nuevo enfoque basado en árboles para la evaluación de restricciones en un Sistema de Detección de Intrusiones. Aunque en un principio my objetivo era desarrollar y probar tres de nuestras once restricciones IGMP y RTPS, finalmente he desarrollado **cinco**, mediante las cuales podíamos probar el correcto funcionamiento de mi enfoque del motor de restricciones. La implementación de las Restricciones 8 y 11, además de las tres requeridas anteriormente, hizo posible probar con éxito mis estructuras de árboles a través de las cuatro fases de evaluación, incluyendo la condición de "Split" que introdujo posteriormente mi compañero, y el aprendizaje de "Facts".

Como resultado, después de correr mi Motor de Restricciones diez veces ante un archivo PCAP de 1.6GB, hemos obtenido una media de los siguientes resultados: **16.112 s** de **tiempo**, con un **rendimiento** de **102.525 MB/s** y un **ancho de banda** de **828.203 Mb/s**. Sólo remarcar que un ancho de banda de por lo menos 180 Mb/s ya se consideraba aceptable, por lo que mi motor va 5 veces más rápido. Así, mi aproximación resultó ser solo ligeramente más lenta que la aproximación sin usar estructuras de árboles de mi compañero, con una diferencia de alrededor de 1.5s en tiempo de ejecución y 300 MB/s menos de ancho de banda, lo que nos permite concluir que trabajar con las estructuras de árboles no reduce el rendimiento del programa de forma significativa.

Con este proyecto también hemos demostrado que nuestro sistema es capaz de trabajar con restricciones multi-paquete usando UDP y guardando la información de los paquetes, aparte de realizar un inspección de paquetes profunda sobre ellos.

La siguiente tabla muestra la media de los resultados obtenidos tras ejecutar diez veces el archivo PCAP de 1.6GB sobre mi Motor de Restricciones.

Paquetes Parseados	10.707.581
Paquetes Fallidos	113.435
Paquetes Totales	10.821.016
Tasa de Fallo	1,05 %
C1 Evaluada	0

C5 Evaluada	12
C7a Evaluada	36
C7b Evaluada	0
C8 Evaluada	66
C11 Evaluada	3193212
Tiempo	16,112 s
Rendimiento	102,525 MB/s
Ancho de Banda	828,203 Mb/s

Tabla 1. Resultados después de correr el archivo PCAP de 1.6 GB diez veces.

8.2. Trabajo Futuro

Nuestro trabajo futuro consiste primeramente en generar automáticamente todo el código C correspondiente a la evaluación de las restricciones, usando la gramática desarrollada por nuestro grupo de investigación para, en el futuro, proveer un IDS personalizable para cualquier red privada con la posibilidad de añadir fácilmente cualquier protocolo a dicha red. Además, nuevas restricciones deberían ser encontradas y desarrolladas.

Aún más, de la misma manera que Bro es hoy en día para la comunidad de investigación, nuestro futuro IDS open-source podría ser muy útil para organizaciones privadas que deseen introducir sus propios protocolos al sistema, así como una herramienta usada internacionalmente para cualquier investigación acerca de seguridad en redes y detección de intrusiones.

9. Appendix

9.1. User Manual

IMPORTANT: a Linux operative system is needed to run the program.

9.1.1. Learning Mode

Go to the *IDS/parserGenerator/F_Constraints* folder and open the *Constraints.c* file with a text editor. Comment out (or make sure that it is commented out) the *"#define LEARN*" line. Save the file.

```
#include "Constraints.h"
#include "hashmap.h"
#include "StringHash.h"
#include <arpa/inet.h>
#include "node.h"
#include <inttypes.h>

#define LEARN
```

Fig. 44: Snippet of code for Learning Mode.

Open a new terminal and move to the *IDS/parserGenerator/F_Constraints* directory. Compile and link the program using *make*. It is possible than *make* has nothing to be done at all.

```
lubuntu@lubuntu-VirtualBox:~$ cd IDS/parserGenerator/F_Constraints/
lubuntu@lubuntu-VirtualBox:~/IDS/parserGenerator/F_Constraints$ make
```

Fig. 45: Using make command.

Run the program using the following command:
./pcapparse LongStressTest.pcapng

Once you have run it, the program will have "learned" from our *Facts* files and will be ready for execution. It is possible that some of the constraints that do not need to learn from facts for their evaluation were evaluated. You should get something like this:

```
lubuntu@lubuntu-VirtualBox:~/IDS/parserGenerator/F_Constraints$ ./pcapparse LongStressTest.pcapng
argc = 2
**********************************************************************************************
Packets Parsed: 10707581
Packets Failed: 113435
Total Packets: 10821016
Failure rate: 1.05%
C1 Evaluated: 0
C5 Evaluated: 0
C7a Evaluated: 0
C7b Evaluated: 0
C8 Evaluated: 0
C11 Evaluated: 0
```

Fig. 46: Running the program in Learning Mode.

9.1.2. Checking Mode

Go to the *IDS/parserGenerator/F_Constraints* folder and open the *Constraints.c* file with a text editor. Comment (or make sure that it is commented) the *"#define LEARN*" line. Save the file.

```
#include "Constraints.h"
#include "hashmap.h"
#include "StringHash.h"
#include <arpa/inet.h>
#include "node.h"
#include <inttypes.h>

//#define LEARN
```

Fig. 47: Snippet of code for Checking Mode.

Open a new terminal and move to the *IDS/parserGenerator/F_Constraints* directory. Compile and link the program using *make*. It is possible than *make* has nothing to be done at all.

```
lubuntu@lubuntu-VirtualBox:~$ cd IDS/parserGenerator/F_Constraints/
lubuntu@lubuntu-VirtualBox:~/IDS/parserGenerator/F_Constraints$ make
```

Fig. 45: Using make command.

Run the program using one of the two following options:

1. *./pcapparse LongStressTest.pcapng*. This option is thought for performance testing, so we do not specify any trace file where to write the outputs.

```
$ ./pcapparse LongStressTest.pcapng
```

Fig. 48: Running command for 2 arguments.

2. *./pcapparse LongStressTest.pcapng parsertrace [traceFileName]*. This option, apart from showing the results of the execution in the terminal, it also writes the outputs to a trace file. Once the program has finished running, you can check the outputs opening the *traceFile* file which has been created in the same directory.

```
$ ./pcapparse LongStressTest.pcapng parsertrace traceFile
```

Fig. 49: Running command for 4 arguments.

In any case, you should get the same results printed out in the terminal, which should be similar to these:

```
lubuntu@lubuntu-VirtualBox:~/IDS/parserGenerator/F_Constraints$ ./pcapparse LongStressTest.pcapng
argc = 4
**********************************************************************************************************
Packets Parsed: 10707581
Packets Failed: 113435
Total Packets: 10821016
Failure rate: 1.05%
C1 Evaluated: 0
C5 Evaluated: 12
C7a Evaluated: 36
C7b Evaluated: 0
C8 Evaluated: 66
C11 Evaluated: 3193212
```

Fig. 50: Running the program in Checking Mode.

References

[1] Andreea Bendovschi . “Cyber-Attacks – Trends, Patterns and Security Countermeasures”, 2015 (7th International Conference on Financial Criminology 2015, Wadham College, Oxford, UK).

[2] Qijun Gu, Peng Liu. “Denial of Service Attacks”, 2007.

[3] Biswanath Mukherjee, L. Todd Heberlein, Karl N. Levitt. “Network Intrusion Detection”, 1994 (IEEE Network).

[4] Pedro Salgueiro and Salvador Abreu. “Modeling Distributed Network Attacks with Constraints”, 2011.

[5] Siam Hasan, A. ElShakankiry, T. Dean and M. Zulkernine. “Detection of Intrusion in Private Network by Satisfying Network Constraints”, 2016.

[6] Stephen Deck and Hamed Khiabani. “Extracting Files from Network Packet Captures”, 2015.

[7] B. Fenner, H. Be, B. Haberman and H. Sandick. “Internet Group Management Protocol (IGMP) Multicast Listener Discovery (MLD)-Based Multicast Forwarding (IGMP/MLD Proxying)”, 2006. (https://tools.ietf.org/html/rfc4605)

[8] Gerardo Pardo-Castellote. “OMG Data Distribution Service: Real-Time Publish/Subscribe Becomes a Standard”, 2005.

[9] G. Pardo-Castellote, B. Farabaugh and R. Warren. “An Introduction to DDS and Data-Centric Communications”, 2005.

[10] O. Eldow, P. Chauhan, P.Lalwani and M. B. Potdar. “Computer Network Security IDS Tools and Techniques (Snort/Suricata).

[11] V. Paxson. “Bro: A System for Detecting Network Intruders in Real-time”, 1999.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee. “Hypertext Transfer Protocol—HTTP/1.1”, 1999. (https://tools.ietf.org/html/rfc2616)

[13] “RFC 793: Transmission Control Protocol.” (https://www.ietf.org/rfc/rfc793.txt)

[14] “Wireshark.” (https://www.wireshark.org/docs/wsug_html_chunked/ChapterIntroduction.html)

[15] J. Postel. “RFC 768 - User Datagram Protocol”, 1980 (https://tools.ietf.org/html/rfc768)

[16] C. Cowan, P. Wagle, C. Pu, S. Beattie and J. Walpole. “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade”.

[17] “The Tree Data Model” – Stanford University (http://infolab.stanford.edu/~ullman/focs/ch05.pdf). Retrieved: April 8th 2017.

[18] S. Marquis, T. R. Dean and S. Knight. “Packet Decoding Using Context Sensitive Parsing”, in Proceedings of the 16th IBM Centre for Advanced Studies Conference (SCASCON 06), pp. 263–274, 2006.

[19] S. Marquis, T. R. Dean and S. Knight. “SCL: A Language for Security Testing of Network Applications”, in Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative Research (CASCON 05), pp. 16–19, 2005.

[20] “ASN. 1” (http://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx) Retrieved: April 10th 2017.

[21] “XML 1.0 Specification”. World Wide Web Consortium (https://www.w3.org/TR/REC-xml/) Retrieved: April 10th 2017.

[22] R. P. Andrew Clark, Quanyan Zhu and T. Basarauthor. “An Impactaware Defense Against Stuxnet American Control Conference, ACC, Washington DC, USA”, 2013.

[23] A. Buecker, P. Andreas and S. Paisley. “Understanding the IT Perimeter Security”, IBM, 2008.

[24] D. E. Denning. “An Intrusion-Detection Model”. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 1987.

[25] Irfan Gul and M. Hussain. “Distributid Cloud Intrusion Detection Model”, 2011.

[26] P. Mockapetris. “RFC 1035: Domain Names – Implementation and Specification”, 1987 (https://www.ietf.org/rfc/rfc1035.txt)

[27] J. Postel.“RFC 792: Internet Control Message Protocol”, 1981 (https://tools.ietf.org/html/rfc792)

[28] D. Hammarberg. “The Best Defenses Against Zero-day Exploits for Various-sized Organizations”, 2014.

[29] V. Jyothsna, VV. R. Prasad and K. Munivara Prasad. “A review of anomaly based intrusión detection systems”, 2011.

[30] D. Odden. “Rules v. Constraints”, 2007.

[31] T. Ylonen and C. Lonvick. “The Secure Shell (SSH) Protocol Architecture”, 2006. (https://www.ietf.org/rfc/rfc4251.txt)

[32] E. Rescorla and A. Schiffman. “The Secure Hypertext Transfer Protocol”, 1999. (https://tools.ietf.org/html/rfc2660)

[33] B. Wippich. “Detecting and Preventing Unauthorized Outbound Traffic”, 2007.

[34] Rachelle L. Miller. “The OSI Model: An Overview”, 2001.

[35] http://searchnetworking.techtarget.com/definition/deep-packet-inspection-DPI Retrieved: April 11th 2017.

[36] “RTI: NASA Human Exploration Telerobotis”. (http://www.omg.org/hot-topics/documents/dds/NASA_Telerobotics.pdf)

[37] D. C. Schmidt, A. Corsaro and H. van’t Hag. “Addressing the Challenges of Tactical Information Management in Net-Centric Systems With DDS”, 2008.

[38] RTI: “Why is DDS the Right Technology for the Industrial Internet?” (http://omgwiki.org/dds/sites/default/files/RTILunchAddressOMG2014v2.pdf)

[39] RTI community – DataWriters/Publishers and DataReaders/Subscribers (https://community.rti.com/static/documentation/connext-dds/5.2.3/doc/manuals/connext_dds/html_files/RTI_ConnextDDS_CoreLibraries_UsersManual/Content/UsersManual/DataWriters_Publishers_and_DataReaders.htm)

[40] OMG – “The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification”, 2008.

[41] V. Veselý, O. Ryšavý and M. Švéda. “Protocol Indepentent Multicast in OMNeT++”, 2014.

[42] B. Cain, S. Deering, I. Kouvelas, B. Fenner and A. Thyagarajan. “RFC 3376: Internet Group Management Protocol, Version 3.” (https://tools.ietf.org/html/rfc3376)

[43] U. Goyal, G. Bhatti and P. Singh. “A Novel Framework for Mitigating DDoS Attacks”, 2013.

[44] Prismtech – OpenSplice DDS. (http://www.prismtech.com/sites/default/files/documents/OpenSplice_DDS_ATC_ATM_Overview.pdf)

[45] R. A. Sahner adn K. S. Trivedi (IEEE). “Performance and Reliability Analysis Using Directed Acyclic Graphs”, 1987.

[46] David G. Sullivan (Harvard Extension School). “Binary Trees and Huffman Encoding Binary Search Trees”, 2012.

[47] C. T. Silva, J. S. B. Mitchell and P. L. Williams. “An Extract Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes”.

[48] Memcpy(3) Linux Programmer's Manual. (http://man7.org/linux/man-pages/man3/memcpy.3.html) Retrieved: April 13th 2017.

[49] J. Triplett, P. E. McKenney and J. Walpole. "Resizable, Scalable, Concurrent Hash Tables".

[50] N. CANADA. "Av Canada: Media – the Canadian Automated Air Traffic System" (http://www.navcanada.ca/EN/media/Pages/publicationscorporate-direct-route-story-1.aspx.)

[51] G. Csmak. "Euroscope – Power of Control" (http://www.euroscope.hu.)

[52] Siam Hasan and T. R. Dean. "A constraint-based IDS". (*Pending of revision for publication*)