UNIVERSITÀ DEGLI STUDI DI CATANIA
Dipartimento di Matematica e Informatica
Corso di Laurea Triennale in Informatica

Carmelo Bertolami

Wireshark Dissector for the Matter Protocol

Final Project Report

Supervisor: Giampaolo Bella
Advisor: Marcello Maugeri

Academic Year 2023 - 2024

Abstract

The Internet of Things (IoT) has led to a heterogeneity of devices and protocols, creating significant challenges in the management and interoperability of systems. Despite this, with the development of Matter, a unified standard created by the Connectivity Standards Alliance (CSA), the possibility of simplifying and improving the IoT ecosystem has opened up.

The primary objective of this final project report is to examine the significance of network traffic analysis within Matter and extend its functionality, focusing on its applications in analytics and security. It also set itself the objective of implementing semantics for read interaction to interpret the attribute read request and response commands. The project presents tools such as Wireshark for traffic analysis and discusses the necessity for a Matter protocol-specific dissector for in-depth data understanding, which led to the creation of such a tool by a community of experts. This tool has been extended to achieve the intended goal.

The section on Matter provides a comprehensive examination of the protocol, including pivotal concepts such as clusters and commissioning, and an analysis of the chip-tool utilized in developing Matter devices. The implementation section addresses the challenges and solutions encountered when implementing the attribute read and request semantics in the Interaction Model dissector.

Finally, the demonstration section presents concrete evidence of the implementation through practical examples. These include the analysis of network traffic generated by the communication coming from the tools provided in the CSA public repository [1].

Contents

Chapter 1

Introduction

In the mid-1990s, the Internet expanded its reach worldwide, prompting researchers and technologists to explore new avenues for enhancing connections between humans and machines. In particular, in 1999, Kevin Ashton, a British technologist and co-founder of the Auto-ID Center at MIT, during a speech coined the famous term **Internet of Things** (IoT).
His idea was to connect devices to the Internet leveraging radio frequency identification (RFID) chips and other sensors. Over the next decade, there was a notable increase in public interest in the IoT, as companies began offering a wider variety of devices that could be connected to each other and to the Internet. This surge in public interest led to a wave of innovation from companies that have increased the production of small devices connected to the Internet. These devices have become an everyday part of people's lives and have transformed the way we interact with the world around us. However, this fast growth has also led to some challenges, including poor collaboration between companies. In fact, not all companies used the same technology in the creation of these devices.

Therefore, no one raised the issue of creating a standard to establish interoperability, compatibility, quality, security, innovation, and efficiency in the ecosystem. Ensuring that the technology landscape is not fragmented and chaotic creates problems for businesses and consumers who, before they buy a device, have to make sure it is compatible with their smart home ecosystem, and download the single app of the brand for the device they are buying.
The category of devices under discussion is vast and encompasses a multitude of products. It is clear that the devices in question vary from one house to another, but they all facilitate and assist people in their daily lives. To illustrate, there are a plethora of examples of devices that facilitate a multitude of tasks in people's daily lives. These include light bulbs [5], thermostats [6], refrigerators [7], smart sockets [8], gates, and many others. These devices can

be operated with a simple click of a button or a voice command from a voice assistant, which helps people to complete tasks more efficiently.

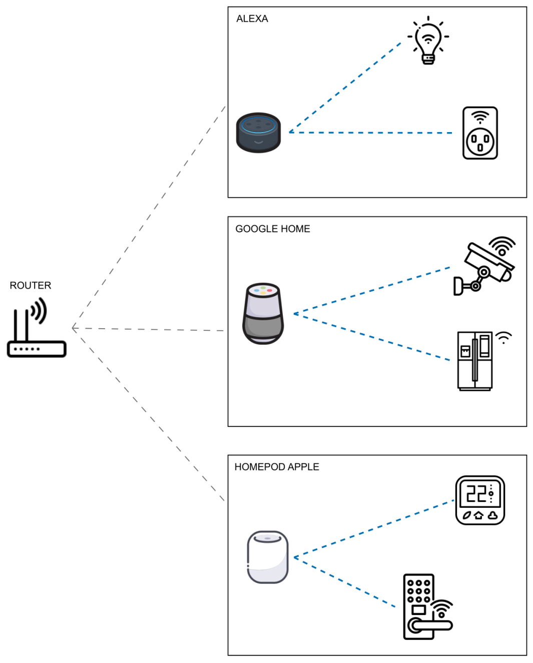


Figure 1.1: Before the advent of Matter

Numerous organizations have attempted to create their standards, including IEEE, W3C, and ISO/IEC [9]. However, it was not until 2019 that major companies such as *Amazon*, *Google*, *Apple*, *Samsung Smart-Things*, and many others (around 200 companies) [10], formed a significant collaboration known as CSA (Connectivity Standards Alliance) once known as Zigbee Alliance "founded 22 years ago". The main project in 2019 was Project CHIP (Connected Home over IP), which will only become the **Matter protocol** in 2021.

The objective of the project is to streamline the development process for smart home product brands and manufacturers while enhancing the interoperability of the products for consumers [11].

Furthermore, the issue of fragmentation in the smart device market, which currently complicates the integration of various technologies in IoT systems, will be addressed by looking for the Matter symbol. Devices bearing this symbol will ensure compatibility, thus eliminating the need to spend hours searching for products that work with one's voice assistant or home ecosystem.

The standard relies on the Internet Protocol (IP) [12] and relies on devices located within the network to manage all aspects of the system. This eliminates the need for proprietary hubs that are no longer required. Matter-certified products are designed to operate locally and do not depend on an internet connection for their core functions. Using **IPv6** addressing [13], the standard facilitates communication without the need to leave the network and connect to cloud services. Matter differs from protocols like Zigbee or Z-Wave and can theoretically operate on any IPv6-enabled network [11]. At present, official support is confined to Wi-Fi, ethernet, and Thread wireless mesh networking.

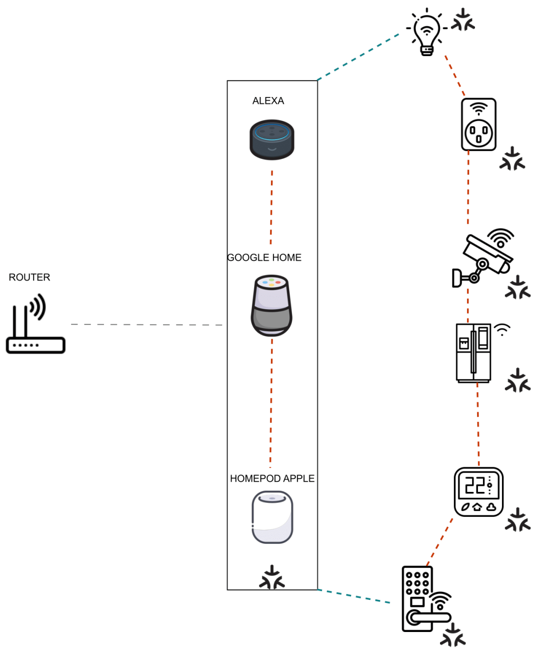


Figure 1.2: after the advent of Matter

As Matter is in its early stages of development, utilizing **developer mode** presents several constraints like limitations in clusters used, the need to implement libraries, command line usage, and difficulty in connections between nodes. This does not imply that it is not possible; however, it is necessary to refer to software that has been developed with this specific purpose in mind. In particular, an open-source tool has been released on GitHub called *chip-tool*, which is a **software development kit** (SDK). This enables users to create concrete examples on which to base their work, try new devices (also already present on the market), add new functionality in clusters, and gain an understanding of how to implement the associated protocol documentation. Furthermore, the (SDK) enables the addition of clusters, which are data model building blocks in Matter. A **cluster** represents a single functionality within a Matter device, such as turning a device on and off or controlling the thermostat temperature. Each **cluster** contains **attributes**, **commands**, and **events** that can be mandatory or optional. Attributes are stored in the device's memory, while commands can be used to modify or read the state of the device, including the cluster attributes.

During this intricate process, it is crucial to utilize a network traffic analyzer. By employing this type of device and, even more so, by developing one that adheres to the standards or by integrating code within these open-source projects with extensive communities, it is important to be able to discern the packages through which communication is conducted with and from the device, and especially how it interfaces and responds to the commands issued to it. A traffic analyzer is also used to assess the security of such devices, thereby preventing unauthorized access. The traffic analyzer runs on a PC and the latter is configured in **sniffing** mode, whereby the network card of the device under examination is placed in mixed mode. This enables the capture of all network traffic that traverses the network to which the device is connected.

It can be assumed that any individual could potentially read the network. However, an effective solution that is commonly employed by numerous protocols is to encrypt the single content of the packet that is forwarded. This ensures that only those who possess the decryption key can read its contents. In this particular instance, the traffic analyzer is employed to discern the transmissions and receptions of the chip-tool, as well as the manner in which the device establishes a connection to the network. Regrettably, given that this is a relatively nascent protocol, it is exceedingly challenging to identify a network analyzer that has incorporated a packet dissector, which is an extension enabling the examination and interpretation of packets specific to that protocol.
The network analyzer under consideration is Wireshark, one of the most prevalent free options on the market. The approach consists of utilizing a dissector developed by the GitHub community, namely “Matter Dissector”. This dissector is still in its preliminary stages and therefore does not include relevant features. It is therefore the objective of this work to develop in particular the part relating to the semantics of the interaction model, which comprises:

- Attribute Read
- Attribute Write

- Subscribe
- Events

The document is organized as follows. Chapter 2 describes the basics of the Matter protocol, including the interaction model layer. Chapter 3 describes the importance of traffic analysis and details the Wireshark tool. Chapter 4 presents the implementation of the read attribute dissection feature, including both the request and the response. Chapter 5 shows the developed features in action using the all-cluster application example provided by the CHIP project. Chapter 6 concludes the document and describes possible future directions, such as the implementation of missing features.

Chapter 2

Matter

One of **Matter's principal strengths** is its utilization of technologies that have already been established and are solely concerned with the implementation of communications at a higher level. Consequently, there is no necessity to employ new technologies that must be approved by certification bodies or that must gain the favour of consumers, as they have been approved for many years. **Interoperability** represents a fundamental strength because it allows all devices that comply with the standard to communicate transparently without the need for external means. **Security** represents a fundamental aspect of data management, particularly data or its transmission. This is why Matter is based on protocols that are consolidated and reliable, as well as aligned with rigorous standards.

Each **Matter device** undergoes a clearly defined life cycle, as outlined in the *specification format*. This facilitates **device management and maintenance**, even in the context of changes to the Specification. Matter also places particular attention on **cost reduction**. One of the key aspects of this approach is **backward compatibility**. Matter was designed so that even older devices that meet certain requirements can be used. While not all devices will be able to take advantage of this policy, there is a significant percentage of the market that will still be able to enjoy this backward compatibility. This feature not only extends the useful life of devices already in use but also allows users to adopt new technologies without the need to completely replace existing hardware. This facilitates a more economical and less demanding transition to *smarter home environments*.

The usability of connected devices has also improved since the use of Matter has the effect of reducing the need for users to configure such devices. The objective is to reach a state of **plug-and-play** functionality, whereby devices can be connected to a power supply and the network will automatically recognize and configure them for instant use. It provides the capacity to have a more intuitive and less intimidating smart home environment. The technology is designed to facilitate the daily life process without requiring a complex technical intervention from the user.
The appeal of this type of protocol has prompted numerous manufacturers to pledge their support for its adoption. Prominent companies with well-known brands, including Google Home, Alexa, and Apple HomeKit, are currently evaluating the potential integration of Matter into their respective ecosystems. Notably, although Matter originated from the ZigBee standard, from which it inherited several key features and concepts, it has demonstrated a more robust performance.

Despite the potential for interconnection offered by IoT technology, communication between devices remains limited. For instance, while it is possible to control the living room lights with a voice assistant like Siri or Google Assistant, it is not currently possible to allow two different ecosystem devices to communicate with each other without extra effort. Any device or network of devices that implements Wi-Fi, BLE, and 802.15.4-based communication interfaces is potentially a Matter device or network. Although some current versions of the earlier protocols are not directly compatible with the Matter standard, the current Matter specification includes guidelines for setting up a bridge device that can act as a translator between non-Matter devices and the standard [14].

2.1 Matter Protocol Stack

The Matter protocol stack architecture is comprised of four levels according to the TCP/IP stack, which are divided to help separate the different responsibilities and introduce a good level of encapsulation among the various pieces of the protocol stack. The first three levels, viewed from the bottom up, are already known and are linked to networking, transport, and link layers. Consider those that originate from the standard Internet, which allows packets to be transmitted from one device to another. The most recent developments can be observed at the highest level, which is the application level. This level contains all the implementations of the Matter protocol.

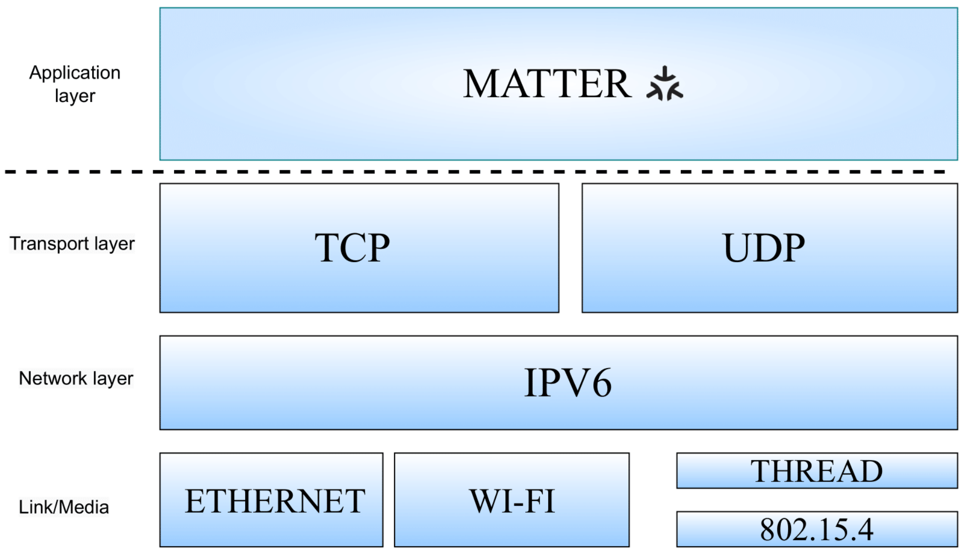


Figure 2.1: Protocol Stack

2.1.1 Application layer

The application layer is where the essence of Matter is located. It is here that the implementations are found, which allow the protocol to function following criteria that have been created and standardized previously. The Application Layer orchestrates high-level business logic, managing tasks such as turning on or off a smart home device[15]. Another example could be the thermostat, which can regulate the temperature of a room or zone within the house. Users can set desired temperature levels either manually or through automation rules; they can also adjust the temperature to make it cooler during hot summer days or warmer during cold winter nights.

2.1.2 Data model (DM)[2]

The Matter Data Model describes a hierarchical encapsulation of data elements within the Matter network. These include, but are not limited to, nodes, endpoints, clusters, and device types. The node is the highest-level data element within this model. A single node may represent a single physical Matter device, such as a light switch, door lock, or other fixture. A Matter fabric is an environment in which several Matter nodes communicate with each other. These nodes are linked by a shared root of trust. A node represents a physical Matter device on each distinct fabric. Each node on the fabric is individually identified by its Operational Node ID, which is a unique network address.

To illustrate, consider the Matter Lighting gadget on a Matter Switch device on a different Samsung Smart-Things Station fabric (red) in the same house has an operational node ID specific to the Smart-Things fabric, while an Apple HomePod fabric (blue) has an operational node ID exclusive to the HomePod fabric. Given that these identifiers exist on two distinct fabrics, they are independent of one another and may be identical or disparate. A Matter device may also be a component of multiple fabrics, in which case it is represented by a separate node on each fabric.

The gadget can be found on fabrics in the following figure. The operational functionality of the system is as follows: the two nodes that represent the device may have different operational node IDs or the same ID. Because the nodes are located on two distinct fabrics, the IDs are not related to each other.

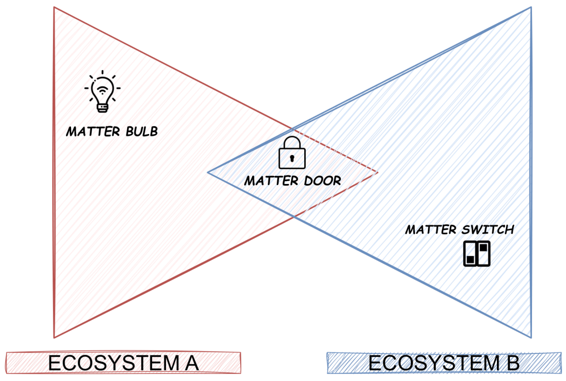


Figure 2.2: The overview of Matter ecosystem

2.1.2.1 Nodes

Each device is comprised of one or more nodes, which are complete implementations of the Matter application functionalities on a single stack. Each node is identifiable by a unique network address within a single network and is capable of direct communication with other nodes within the network. Each node contains the entirety of the application functionality for its device on a single stack. Consequently, nodes can communicate directly with other nodes on the network, negating the need for an intermediary. The nodes possess a set of related behaviors, which are collectively referred to as roles.
The following are the principal roles of nodes:

- **Commissioner**: The function of this entity is to add or commission new devices to a Matter network.
- **Controller:** is to oversee the functioning of the network and to manage the devices that comprise it. The role of the network manager is to supervise and regulate the functioning of one or more nodes within the network.
- **Controlee:** is used to describe a device that is under the control of another device. A device that can be controlled by one or more nodes within the network.
- **Over-the-Air (OTA) Provider:** is used to describe a system that is capable of providing software updates to devices via airborne transmission. The entity in question provides OTA software updates to devices upon request.
- **OTA Requestor:** is responsible for initiating requests for over-the-air (OTA) software updates from the OTA Provider. It initiates requests for OTA software updates from the OTA Provider.

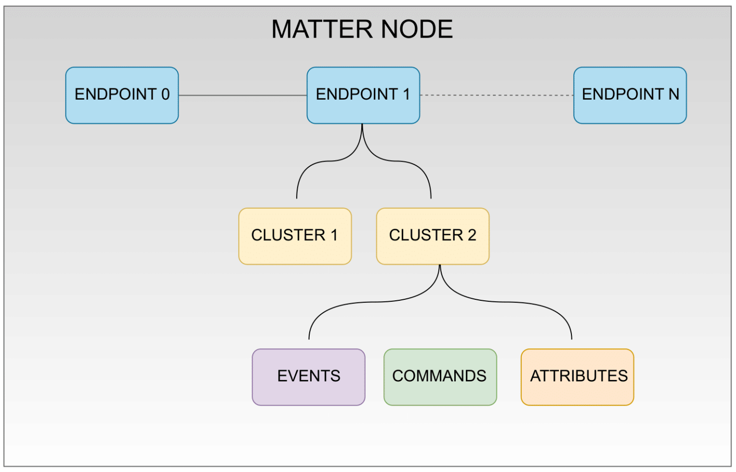


Figure 2.3: Matter Node schema

2.1.2.2 Endpoints

Endpoints serve as containers for specific components of a node's overall functionality, collectively embodying all the capabilities necessary for the node to operate effectively. For example, in a smart thermostat, two endpoints may be present: one managing temperature control and another overseeing temperature monitoring.

Each of these endpoints constitutes a feature set, which comprises clusters defining attributes, events, and commands pertinent to that endpoint's function.

- **Attributes** are data entities that represent a physical quantity or state. The aforementioned attributes are stored in the memory of the Matter device, although they can also be calculated dynamically on demand.

- **Commands** are instructions that can be employed to initiate a specific action on another device. To illustrate, in the case of a door lock device, the lock door command can be employed to initiate a physical action on the device in question.

- **Events** represent a specific type of attribute that is used to communicate changes in the state of the device. Furthermore, they can be regarded as historical data records of past events on the device.

In the Matter framework, endpoints are classified into two primary categories: leaf endpoints and composed endpoints. Leaf endpoints (such as Endpoints

0, 11, and 14 in the illustration) function independently, without reliance on other endpoints. Conversely, composed endpoints (such as Endpoint 1 in the illustration) are dependent on the presence of other endpoints to fulfill their function. Connector lines visually represent the endpoints to which a given one has access. Nodes assign numerical identifiers to their endpoints, with the numbering starting from 0. Each endpoint encapsulates a unique feature set. Although endpoints with matching numbers across different nodes may encompass distinct feature sets, Endpoint 0 holds a special status, being reserved solely for Utility Clusters. These clusters facilitate the servicing of nodes, including discovery processes, addressing, diagnostics, and software updates.
Endpoints are individually assignable addresses, which facilitates the separate modification of their respective feature sets.

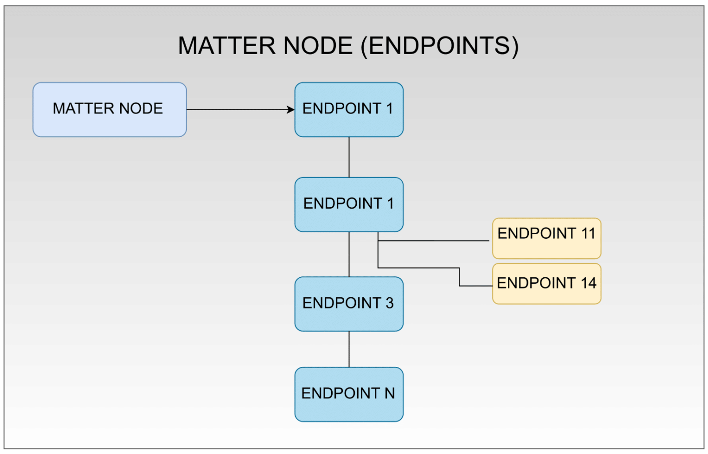


Figure 2.4: Matter Endpoints schema

2.1.2.3 Cluster

Clusters are data containers that aggregate attributes, events, and commands that relate to a given functionality. They represent a portion of a single functional unit within a given endpoint feature set. In other words, they are interfaces or object classes that exist in a device's data model as the smallest independent functional unit. The existence of multiple clusters per endpoint allows for the separation of distinct functional units, thereby facilitating the management of individual units in isolation. **Attributes** pertain to the present state, configuration, or capacity of a node. Such attributes may be employed to indicate whether a light is on or off, or whether a switch is in the up or down position. Fabric-scoped attributes are available to devices within a given fabric and may have one of the following data types: `unit8`, `string`, or `array`. **Commands** are analogous to verbs in that they relate to the actions that a cluster is capable of performing. It is always the case that there is a directionality to these communications, either from the client to the server or vice versa. The target may respond with a request, such as switching the ON/OFF cluster of the server, which represents a light. Alternatively, the target may respond with a response that might be a success or failure. **Events** represent the records of previous state transitions of the node. Such data may include timestamps, the priority of the change, and a counter tracking the number of state changes. They are employed for the capture of state transitions and the recording of historical data that is not stored by the attributes. **Clusters** can be categorized into two distinct groups: server clusters, which are stateful and comprise attribute, event, and command data, and client clusters, which are stateless and communicate with server clusters by reading/writing attributes, reading remote events, or invoking methods. Any cluster is capable of providing both server and client functionality. To illustrate, consider the case of a light and a switch. To illustrate, consider a client cluster in the light sending a command to a server cluster in the light to toggle the on/off feature of the light.

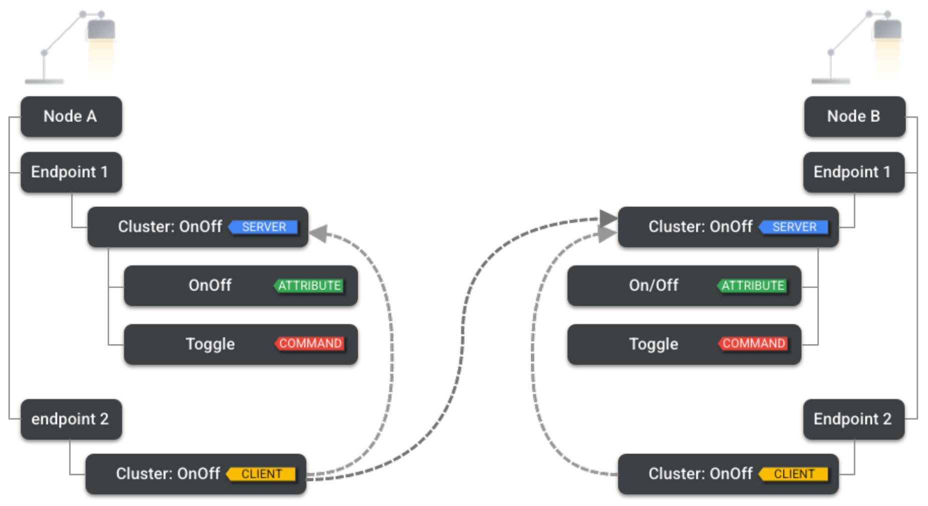


Figure 2.5: Example of Clusters [16]

2.1.3 Interaction Model (IM) [3]

The Matter Device Interaction Model (IM) delineates the methodologies employed for the exchange of information between nodes, functioning as the universal language for the transmission of data between nodes. Nodes communicate with each other through interactions that are defined as a sequence of transactions, which in turn are a sequence of actions.

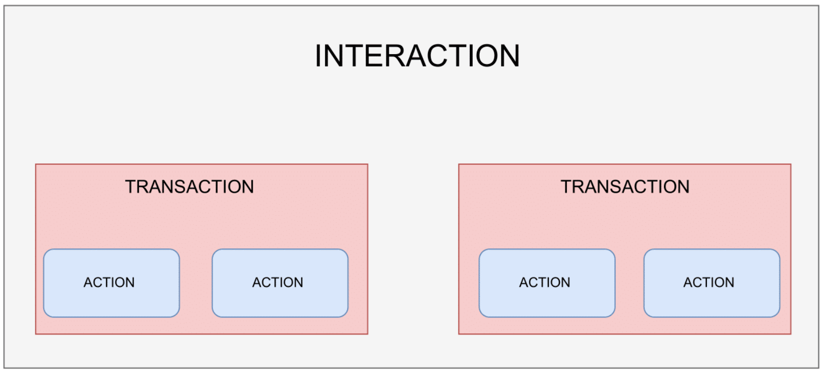


Figure 2.6: Matter interaction schema

For example, in a read interaction, a client cluster may initiate a read transaction, whereby the client may request to read an attribute, and a server cluster may respond by reporting the attribute. The client request and server response are distinct actions, yet they are both part of the same read transaction, which is encompassed by the read interaction.
The Interaction Model supports four types of interactions:

- Read
- Write
- Invoke
- Subscribe

All interaction types except Subscribe consist of one transaction. The Interaction Model supports five types of transactions:

- **Read:** Get attributes and/or events from a server.
- **Write:** Modify attribute values.
- **Invoke:** Invoke cluster commands.

- **Subscribe:** Create a subscription for clients to receive periodic updates from servers automatically.
- **Report:** Maintain the subscription for the Subscribe Interaction.

The following concepts are important for understanding transactions.

- **Initiators and Targets:** Interactions happen between an initiator node and target node(s). The initiator starts the transaction, and the target responds to the initiator's action. More specifically, the transaction is usually between a client cluster on the initiator node and a server cluster on the target node.
- **Transaction ID:** The transaction ID field must be present in all actions that are part of a transaction to indicate the logical grouping of the actions as part of one transaction. All actions that are part of the same transaction must have the same transaction ID.
- **Groups:** Groups of devices allow an initiator to send an action to multiple targets. This group-level communication is known as a groupcast, which leverages Ipv6 multicast messages.
- **Paths:** Paths are the location of the attribute, event, or command an interaction seeks to access. Examples of path assembly:

```
    <path> = <node> <endpoint> <cluster> <attribute /
    ↪  event / command>
    <path> = <group ID> <attribute / event / command>
```

When group casting, a path may include the group or a wildcard operator to address several nodes simultaneously, decreasing the number of actions required and thus decreasing the response time of interaction. Without groupcasting, humans may perceive latency between multiple devices reacting to an interaction. For example, when turning off a strip of lights, a path would include the group containing all the lights instead of turning off each light individually [3].

2.1.3.1 Read interaction

An initiator initiates a read interaction with the objective of determining the value of one or more attributes or events associated with a target node. The following steps are then carried out:

- **The Read Request Action:** The request is for a list of the target's attributes and/or events, along with the paths to each.
- **Report data action** has been generated in response to the read request action. This data was generated in response to the Read Request Action. The target then responds with the requested list of attributes and/or events, a suppress response, and a subscription ID.
  - **Suppress response** is defined as a flag provided to indicate whether the status response should be sent or withheld.
  - **Subscription ID:** An integer that serves to identify the subscription transaction, which is included only in the event that the report is part of a subscription transaction.
- **Status response action:** The system generates a status response by default; however, this is not sent if the suppress response flag is set. The transaction is concluded when the initiator transmits the Status Response or receives a Report Data with the suppress flag set.

Read transactions are permitted only in unicast mode. This implies that the Read Request and Report Data actions are unable to target groups of nodes, whereas the Status Response Action cannot be generated as a response to a groupcast.

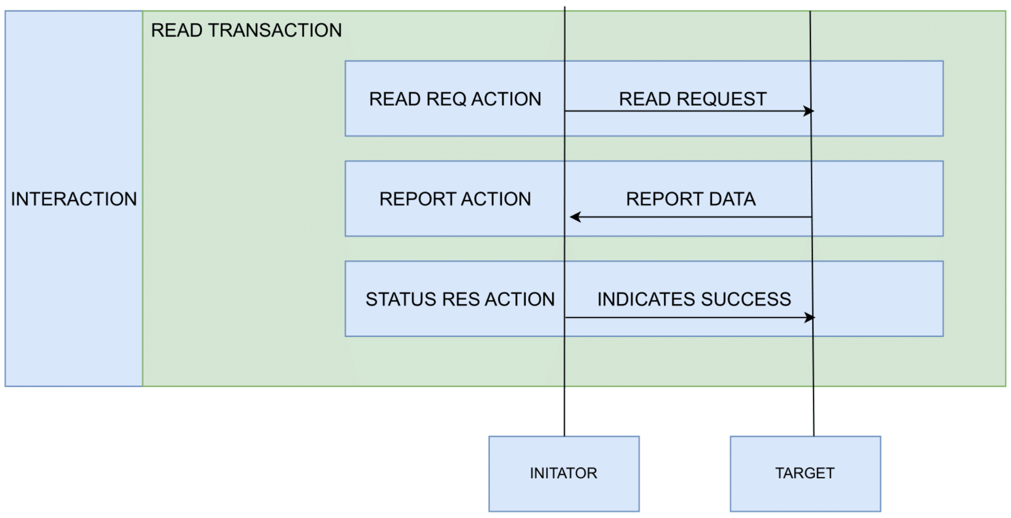


Figure 2.7: Read Interaction schema

2.1.3.2 Write Interaction

An initiator modifies a target's attributes through a Write Interaction, which is comprised of either a Timed or Untimed Write Transaction. An Untimed Transaction remains open to the receiver for an indefinite period, whereas a Timed Transaction establishes a maximum period (usually a few seconds) to receive a return action.

Timed transactions and security: play a pivotal role in the security of systems such as door locks, particularly in light of the potential for interception attacks. In order to fully comprehend the efficacy of timed transactions, it is essential to have a clear understanding of the mechanics of interception attacks. The typical sequence of events is as follows:

- Initiation: The initiator node transmits a message to the target node.
- The interception phase is initiated when the initiator node transmits a message to the target node. An adversary intercepts the message, preventing its delivery to the intended recipient.
- In the event that the message is not delivered, the initiator may elect to resend it. In the absence of a response, the initiator transmits another message.
- In the event of manipulation, the initiator may attempt to alter the message in some way, either by adding or removing content or by modifying the format or structure of the message. The assailant intercepts the second message and transmits the first message to the intended recipient, retaining the second message for future exploitation.
- In the event that the initiator's message is not acknowledged, the attacker may respond in a deceptive manner. The target, unaware of the manipulation, responds to the initial message, thereby confirming receipt to both the initiator and the attacker.

The crux of the problem lies in the second message. Since the target never receives the second message, the attacker gains a valid message that they can exploit at a later point in time. This could result in the execution of actions such as "unlock" or "open door", which could potentially compromise the security of the system. Timed transactions serve to mitigate this risk by imposing a deadline on the validity of messages. This implies that intercepted messages become ineffectual after a specified period, thwarting the attacker's endeavours.

Timed Write Transactions: A Timed Write Transaction is defined as a sequence of actions comprising the following:

- **Timed Request Action:** This action establishes the time interval for the transmission of a Write Request Action.
- **The Status Response Action** is as follows: This action serves to confirm the transaction and the specified time interval.
- **The Write Request Action** is as follows: The request comprises three items.
  - A list of tuples, each of which is called a write request, containing the path and data to be modified.
  - The flag indicating whether the transaction is timed.
  - It is recommended that the response flag be suppressed.

  In the event that a transaction is timed and a timed request flag is set, the initiator is additionally required to transmit a timeout. This is defined as the number of milliseconds during which the transaction remains open, during which the next action to be received is still valid.
- **Write Response Action**(Optional): A list of all paths or error codes pertaining to each write request. Similarly, a write response is not transmitted in the event that the suppress response flag is set.

Untimed Write Transaction: is defined as a transaction that does not require a specific time interval to be set or confirmed. It is comprised solely of the Write Request Action and the Write Response Action.

Write Transaction Restriction: It should be noted that there are differences between untimed and timed write transactions in terms of their restrictions. In contrast to the unicast-only nature of all actions in timed transactions, the multicast option is available for untimed write request actions, provided that the Suppress Response flag is set to prevent the network from flooding with status responses.

2.1.3.3 Invoke Interaction

When an initiator initiates actions on a target's clusters through Invoke Interactions, it has the option of selecting either a Timed or Untimed Invoke Transaction. This parallels the differentiation between Timed and Untimed Write Transactions.

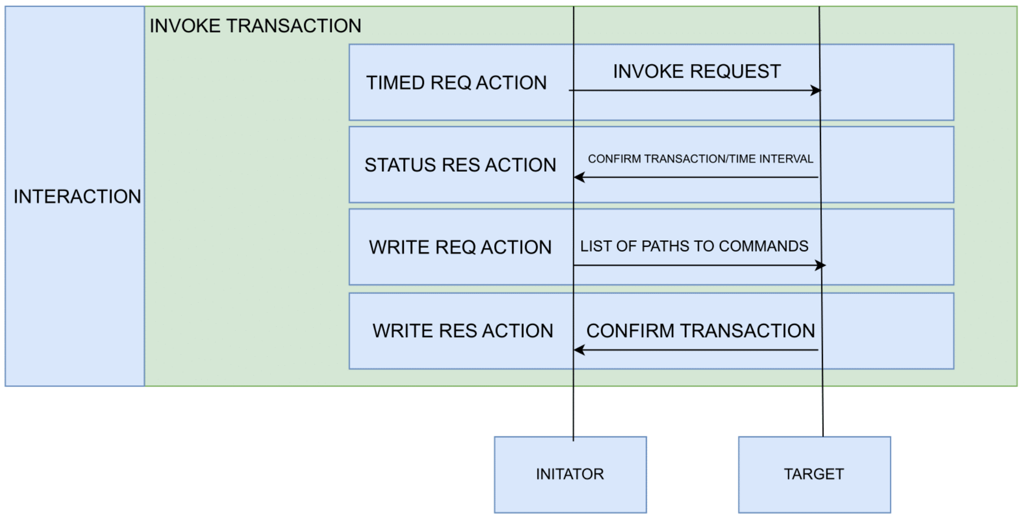


Figure 2.8: Invoke Interaction schema

The following section provides a detailed analysis of a Timed Invoke Transaction, which is analogous to a Timed Write Transaction.

- **A "timed request action"** is defined as follows: This setting determines the time interval for sending a Write Request Action.
- **The Status Response Action** is as follows: This action serves to confirm the transaction and its time interval.
- **The Invoke Request Action** is initiated by requesting the following: The transaction is initiated by requesting the following:
  - A list of paths to cluster commands is to be provided, each item of which contains an invoke command, potentially with arguments.
  - The timed request flag is a designation that indicates a request that is to be executed at a specific time.
  - It is recommended that the response flag be suppressed.

  - Interaction ID: An integer is used to correlate the Invoke Request with its corresponding Invoke Response.

    In order for an Invoke Request to initiate a timed Invoke Transaction, it must also include a timeout, which is analogous to a timed Write Transaction.

- The Invoke Response is an optional element. The target then responds by sending back the interaction ID and a list of invoked responses. These include command responses and statuses for each invoke request. In a manner analogous to a Write Response, an Invoke Response is omitted in the event that the suppress response flag is set.

Untimed and timed Invoke Transactions differ from each other in a manner analogous to the distinction between untimed and timed Write Transactions. This distinction is evident in both the actions performed and the restrictions placed on unicast or multicast.

2.1.3.4 Subscribe Interaction

An initiator employs a subscription interaction to automatically receive periodic report data transactions from the target. This establishes a relationship between the initiator and target, which are respectively referred to as the subscriber and publisher following the subscription.

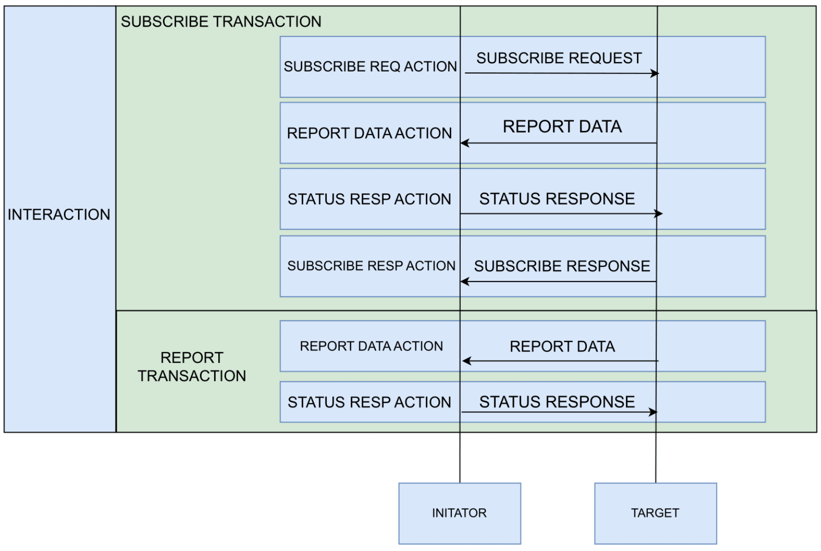


Figure 2.9: Subscribe Interaction schema

A subscription interaction comprises two distinct transaction types. A subscription interaction comprises a subscribe transaction and a report transaction.

The Subscribe Transaction is as follows:

- **Subscribe Request action:**The Subscribe Request Action requests three items.
  - The minimum interval between data reports is defined as the minimum interval floor.
  - The maximum interval ceiling (i.e., the maximum interval between data reports) is also a relevant parameter.
  - A request is made for the attributes and/or events to be reported.
- **Status Response Action:**The Subscribe Request Action is as follows: A Report Data Action containing the initial data set, designated as the Primed Published Data.
- **Status Response Action:**The status response action is as follows: This action acknowledges the Report Data Action.
- **Subscribe Response Action:**The Subscribe Response Action is as follows: The finalization of the subscription ID (an integer that serves as an identifier for the subscription) and the minimum and maximum interval floors and ceilings is completed. This indicates that a successful subscription has been established between the subscriber and the publisher.

2.1.3.5 Report Transaction

Subsequent to the successful conclusion of a subscription, a report of the transactions is transmitted to the subscriber. There are two distinct types of report transactions: those that are not empty and those that are empty.

- **Non-Empty:** the action is to If the SuppressResponse flag is set to FALSE, reports of data and/or events may be submitted. The status response is as follows: This indicator signifies the successful completion of a report or the occurrence of an error, the latter of which terminates the interaction.
- **Empty:** A report that contains no data or events with the SuppressResponse flag set to TRUE, indicating the absence of a Status Response.

Subsequently, following a successful subscription, report transactions are transmitted to the subscriber. There are two types of report transactions: those that are not empty and those that are empty.

2.1.3.6 Subscription Interaction Restrictions

It should be noted that there are a few restrictions associated with Subscription Interactions.

- it should be noted that the Subscribe Request and Subscribe Response actions are unicast-only, which means that an initiator cannot subscribe to more than one target simultaneously.
- it is necessary that Report Data Actions within the same Subscription Interaction have the same subscription ID.
- a subscription may be terminated if the subscriber responds to a Report Data Action with an *"INACTIVE-SUBSCRIPTION"* status or if the subscriber fails to receive a Report Data Action within the maximum interval ceiling. This implies that the publisher may terminate a subscription by failing to transmit Report Data Actions.

Chapter 3

Traffic Analysis and Wireshark

Traffic analysis is a method of examining and analyzing messages for valuable information. This information is derived from the set of communication patterns involved. The analysis of such data represents a crucial aspect of computer security. Over time, traffic analysis has assumed a pivotal role in cryptanalysis, particularly in instances where the decryption of encrypted messages hinges on a known plain-text attack. This frequently necessitates the formulation of informed hypotheses based on contextual cues in operational settings that may potentially unveil pivotal patterns or vulnerabilities. Moreover, has proven to be a valuable tool for understanding communication dynamics and, more importantly, for code-breaking and threat detection in a range of security and intelligence domains. This process can be carried out even when the messages are encrypted. In general, the greater the number of messages observed, the greater the information that can be inferred [17]. A traffic analysis method can be employed to breach the anonymity of anonymous networks. Two distinct methods of traffic analysis attack have been identified: passive and active

- In passive traffic analysis, the attacker extracts features from the traffic of a specific flow on one side of the network and searches for those features on the other side of the network.

- In the active traffic analysis method, the attacker modifies the timings of packets belonging to a specific flow following a predefined pattern, subsequently searching for this pattern on the other side of the network. This allows the attacker to link the flows on one side of the network to those on the other side, thereby compromising the anonymity of the network. Despite the addition of timing noise to the packets, it has been demonstrated that active traffic analysis methods remain robust against such noise.

In addition, traffic analysis is an important concern in the field of computer security. An attacker can gain valuable information by monitoring the frequency and timing of network packets.

3.1 Sniffer Software

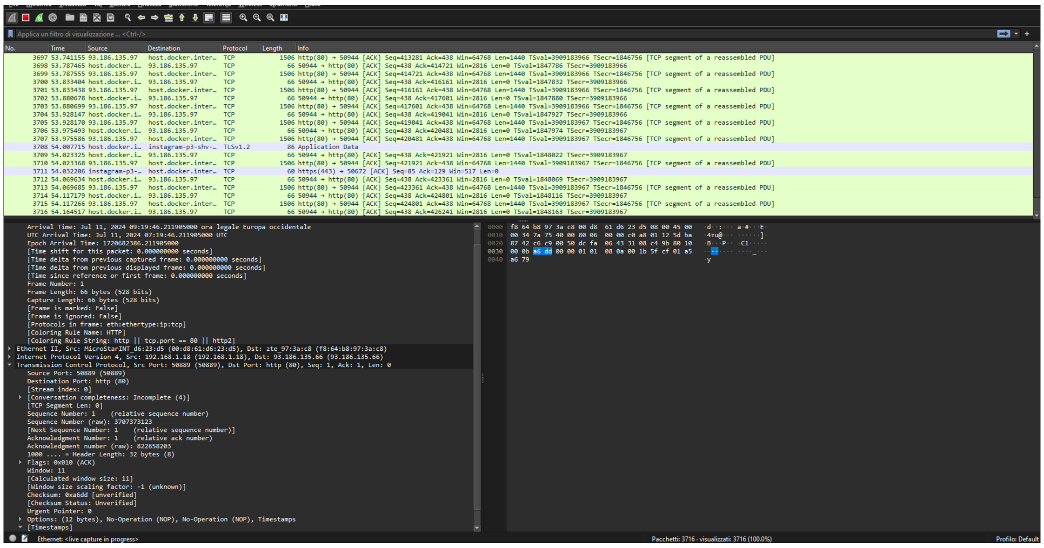
In order to conduct a traffic analysis, it is necessary to utilize specialized software, commonly referred to as sniffers. A packet sniffer is a program that runs on a network-attached device. Its function is to passively receive all data link layer frames that pass through the device's network adapter. It is also known as a network or protocol analyzer or an Ethernet sniffer. The packet sniffer captures data addressed to other machines, which it then saves for later analysis. It can be employed by a network or system administrator for the purposes of monitoring and troubleshooting network traffic in a legitimate manner. The information captured by the packet sniffer can be utilized by an administrator to identify erroneous packets and to identify and resolve bottlenecks in network data transmission. It is important to note that packet sniffers were never designed to be used for the purpose of hacking or stealing information. Their objective was distinctively different, namely to ensure the security of the system [18].

3.2 Pcap File

In computer network administration, pcap is an application programming interface (API) for capturing network traffic [19]. The pcap API is written in C, so other languages such as Java,.NET languages, and scripting languages usually use a wrapper; no such wrappers are provided by libpcap or WinPcap itself. C++ programs can link directly to the C API or use an object-oriented wrapper. A pcap file includes an exact copy of every byte of every packet as seen on the network, including OSI layers 2-7.

3.3 How Wireshark work[4]

GUI: Wireshark is a traffic analysis tool that includes a graphical user interface (GUI) that facilitates all user interactions, including creating and managing windows and dialogues. The application ensures that users can interact with it graphically, input data, and obtain output in a highly intuitive and accessible manner.



```
Applica un filtro di visualizzazione ... <Ctrl-/>

No.     Time      Source          Destination          Protocol  Length Info
   3697 53.741155 93.186.135.97   host.docker.inter…   TCP         1506 http(80) → 50944 [ACK] Seq=413281 Ack=438 Win=64768 Len=1440 TSval=3909183966 TSecr=1846756 [TCP segment of a reassembled PDU]
   3698 53.787465 host.docker.i…  93.186.135.97        TCP           66 50944 → http(80) [ACK] Seq=438 Ack=414721 Win=2816 Len=0 TSval=1847786 TSecr=3909183966
   3699 53.787555 93.186.135.97   host.docker.inter…   TCP         1506 http(80) → 50944 [ACK] Seq=414721 Ack=438 Win=64768 Len=1440 TSval=3909183966 TSecr=1846756 [TCP segment of a reassembled PDU]
   3700 53.833404 host.docker.i…  93.186.135.97        TCP           66 50944 → http(80) [ACK] Seq=438 Ack=416161 Win=2816 Len=0 TSval=1847832 TSecr=3909183966
   3701 53.833438 93.186.135.97   host.docker.inter…   TCP         1506 http(80) → 50944 [ACK] Seq=416161 Ack=438 Win=64768 Len=1440 TSval=3909183966 TSecr=1846756 [TCP segment of a reassembled PDU]
   3702 53.880678 host.docker.i…  93.186.135.97        TCP           66 50944 → http(80) [ACK] Seq=438 Ack=417601 Win=2816 Len=0 TSval=1847880 TSecr=3909183966
   3703 53.880699 93.186.135.97   host.docker.inter…   TCP         1506 http(80) → 50944 [ACK] Seq=417601 Ack=438 Win=64768 Len=1440 TSval=3909183966 TSecr=1846756 [TCP segment of a reassembled PDU]
   3704 53.928147 host.docker.i…  93.186.135.97        TCP           66 50944 → http(80) [ACK] Seq=438 Ack=419041 Win=2816 Len=0 TSval=1847927 TSecr=3909183966
   3705 53.928170 93.186.135.97   host.docker.inter…   TCP         1506 http(80) → 50944 [ACK] Seq=419041 Ack=438 Win=64768 Len=1440 TSval=3909183967 TSecr=1846756 [TCP segment of a reassembled PDU]
   3706 53.975493 host.docker.i…  93.186.135.97        TCP           66 50944 → http(80) [ACK] Seq=438 Ack=420481 Win=2816 Len=0 TSval=1847974 TSecr=3909183967
   3707 53.975586 93.186.135.97   host.docker.inter…   TCP         1506 http(80) → 50944 [ACK] Seq=420481 Ack=438 Win=64768 Len=1440 TSval=3909183967 TSecr=1846756 [TCP segment of a reassembled PDU]
   3708 54.007715 host.docker.i…  instagram-p3-shv-…   TLSv1.2       86 Application Data
   3709 54.023325 host.docker.i…  93.186.135.97        TCP           66 50944 → http(80) [ACK] Seq=438 Ack=421921 Win=2816 Len=0 TSval=1848022 TSecr=3909183967
   3710 54.023368 93.186.135.97   host.docker.inter…   TCP         1506 http(80) → 50944 [ACK] Seq=421921 Ack=438 Win=64768 Len=1440 TSval=3909183967 TSecr=1846756 [TCP segment of a reassembled PDU]
   3711 54.032206 instagram-p3-…  host.docker.inter…   TCP           60 https(443) → 50672 [ACK] Seq=85 Ack=129 Win=517 Len=0
   3712 54.069634 host.docker.i…  93.186.135.97        TCP           66 50944 → http(80) [ACK] Seq=438 Ack=423361 Win=2816 Len=0 TSval=1848069 TSecr=3909183967
   3713 54.069685 93.186.135.97   host.docker.inter…   TCP         1506 http(80) → 50944 [ACK] Seq=423361 Ack=438 Win=64768 Len=1440 TSval=3909183967 TSecr=1846756 [TCP segment of a reassembled PDU]
   3714 54.117179 host.docker.i…  93.186.135.97        TCP           66 50944 → http(80) [ACK] Seq=438 Ack=424801 Win=2816 Len=0 TSval=1848116 TSecr=3909183967
   3715 54.117266 93.186.135.97   host.docker.inter…   TCP         1506 http(80) → 50944 [ACK] Seq=424801 Ack=438 Win=64768 Len=1440 TSval=3909183967 TSecr=1846756 [TCP segment of a reassembled PDU]
   3716 54.164517 host.docker.i…  93.186.135.97        TCP           66 50944 → http(80) [ACK] Seq=438 Ack=426241 Win=2816 Len=0 TSval=1848163 TSecr=3909183967
```

```
    Arrival Time: Jul 11, 2024 09:19:46.211905000 ora legale Europa occidentale
    UTC Arrival Time: Jul 11, 2024 07:19:46.211905000 UTC
    Epoch Arrival Time: 1720682386.211905000
    [Time shift for this packet: 0.000000000 seconds]
    [Time delta from previous captured frame: 0.000000000 seconds]
    [Time delta from previous displayed frame: 0.000000000 seconds]
    [Time since reference or first frame: 0.000000000 seconds]
    Frame Number: 1
    Frame Length: 66 bytes (528 bits)
    Capture Length: 66 bytes (528 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ethertype:ip:tcp]
    [Coloring Rule Name: HTTP]
    [Coloring Rule String: http || tcp.port == 80 || http2]
› Ethernet II, Src: MicroStarINT_d6:23:d5 (00:d8:61:d6:23:d5), Dst: zte_97:3a:c8 (f8:64:b8:97:3a:c8)
› Internet Protocol Version 4, Src: 192.168.1.18 (192.168.1.18), Dst: 93.186.135.66 (93.186.135.66)
˅ Transmission Control Protocol, Src Port: 50889 (50889), Dst Port: http (80), Seq: 1, Ack: 1, Len: 0
    Source Port: 50889 (50889)
    Destination Port: http (80)
    [Stream index: 0]
  › [Conversation completeness: Incomplete (4)]
    [TCP Segment Len: 0]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 3707373123
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 822658203
    1000 .... = Header Length: 32 bytes (8)
  › Flags: 0x010 (ACK)
    Window: 11
    [Calculated window size: 11]
    [Window size scaling factor: -1 (unknown)]
    Checksum: 0xa6dd [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  › Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  ˅ [Timestamps]
```

```
0000  f8 64 b8 97 3a c8 00 d8  61 d6 23 d5 08 00 45 00   ·d··:···a·#···E·
0010  00 34 7a 75 40 00 80 06  00 00 c0 a8 01 12 5d ba   ·4zu@·········]·
0020  87 42 c6 c9 00 50 dc fa  06 43 31 08 c4 9b 80 10   ·B···P···C1·····
0030  00 0b a6 dd 00 00 01 01  08 0a 00 1b 5f cf 01 a5   ············_···
0040  a6 79                                              ·y
```

```
Ethernet: <live capture in progress>        Pacchetti: 3716 · visualizzati: 3716 (100.0%)        Profilo: Default
```

Figure 3.1: Wireshark's GUI

Wireshark is composed of the following components:

Core: The core represents the central component of the application, functioning as the primary "glue code" that unifies and integrates the various functionalities of the other components. The core's primary function is to integrate and coordinate the functionality of the various modules, thereby facilitating communication and data exchange between them. The source code for the core can be found within the root directory of the project.

EPAN (Enhanced Packet Analyzer): Epan is the engine that performs packet analysis. The software provides a number of key application programming interfaces (APIs) that facilitate comprehensive packet analysis.

- **Protocol Tree:** is a method of displaying the interrelationships between packets under various protocols. This is an application programming interface (API) that facilitates the decomposition of individual packets, enabling the data to be disaggregated into fundamental structures that are readily comprehensible.
- **Dissectors:** are utilized to delineate the data structures of disparate network protocols. The source of the dissectors is located within the Epan/dissectors directory.
- **Dissector Plugins:** This would permit the creation of dissectors as discrete modules, thereby obviating the necessity for modifying the

fundamental code base when introducing new protocols to Wireshark. The source code for dissector plugins is typically located within the plugins directory.

- **Display Filters:** constitute a component of the directory epan/dfilter, which serves as an engine for filtering and displaying packets based on specified criteria.

Wiretap: The Wiretap library is capable of reading and writing capture files in a variety of formats, including libpcap, pcapng, and several others. It can be reasonably assumed that the library will support compatibility with a variety of file formats utilized for network packet captures. The source code for the Wiretap will be accessible via the wiretap directory.

Capture: The capture component serves as the interface between the system and the capture engine, assuming responsibility for the management of the packet capture process. It ensures the efficient and effective collection of data from the network. The source code of the capture interface is located in the root directory.

Dumpcap: Dumpcap is a fundamental engine for the capture of network packets. Given that it operates in an elevated-privilege mode, it is imperative to directly access network interfaces. This module ensures the reliable and secure capture of network data. The source code of Dumpcap is also located in the root directory.

Npcap and libpcap: Npcap and libpcap are third-party libraries that facilitate low-level packet capturing and filtering on a range of systems. Capture filters are therefore intrinsic to the Npcap/libpcap framework and operate at a significantly more fundamental level of abstraction than the display filters utilized by Wireshark. As a consequence of this disparity in the level of abstraction, there are different syntaxes for display and capture filters. This system is highly comprehensive, comprising a multitude of specialized components that collectively facilitate robust packet capturing and analysis. The graphical user interface (GUI) enables user interaction, while the core component serves to integrate and orchestrate the system's diverse elements. Epan provides advanced packet analysis, Wiretap ensures compatibility with capture file formats from disparate vendors, the capture component oversees packet collection management, Dumpcap executes the capture under requisite privileges, and finally, Npcap and libpcap handle the low-level capture and filtering.

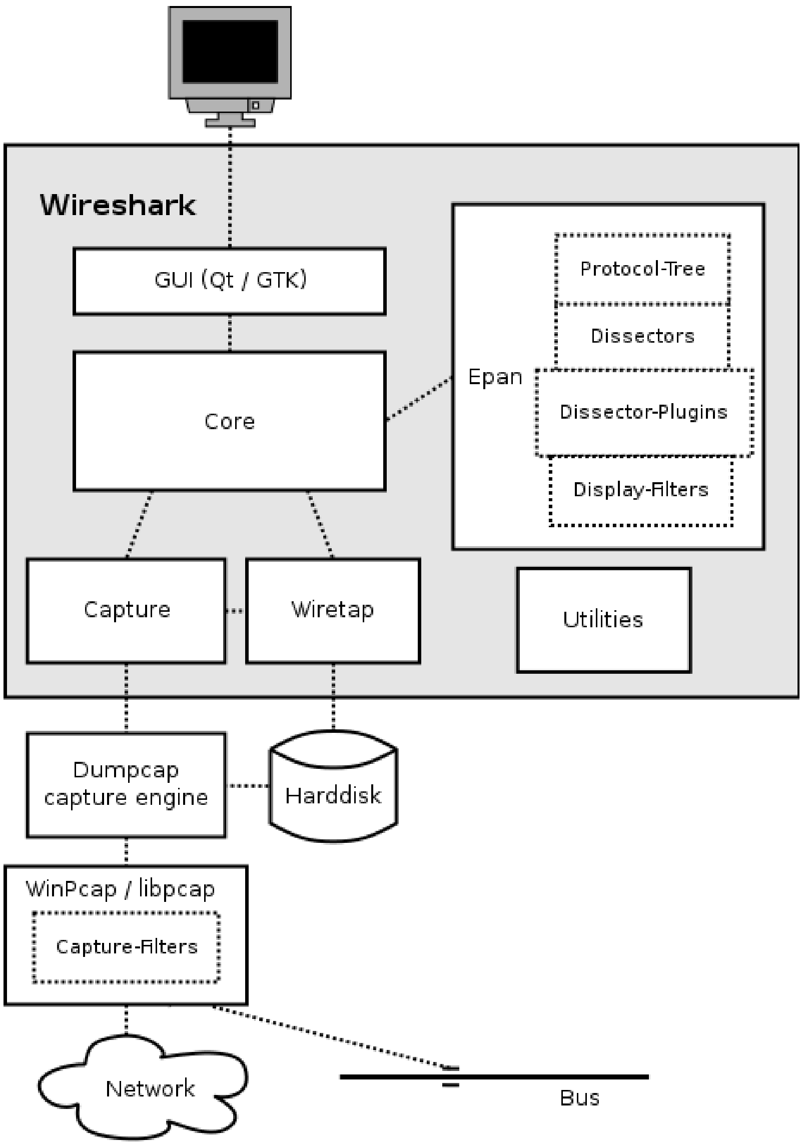


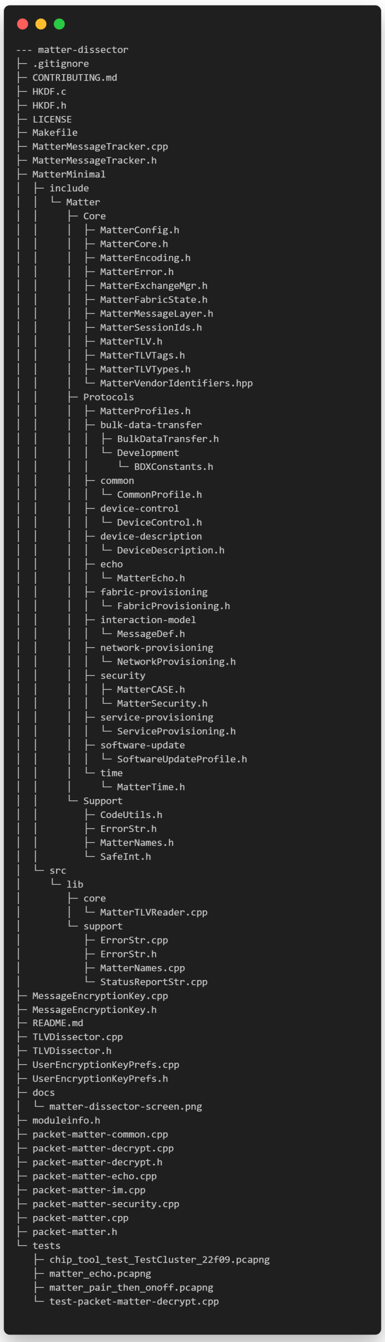
Figure 3.2: Wireshark function blocks [4]

Chapter 4

Implementation

This chapter is designed to demonstrate the writing of C++ code that is necessary to implement some of the previously described modules of the Wireshark dissector dedicated to the Matter protocol and also to explain how the tests were done to ensure that the dissector worked. A comprehensive account of the principal data structures, functions, and methodologies used for the decoding of protocol messages and the presentation of the results in a format that is easily comprehensible to the user will be provided. Conversely, the implementation of the dissector allows users to view and analyze Matter network packets in Wireshark, following the message exchanges. It also enables the diagnosis and resolution of communication failures that can occur between IoT devices while checking and testing security. The practical samples and technical explanations are accompanied by a detailed analysis of the code lines, which reveals the difficulties and solutions that have been adopted. This will be accomplished in a manner that provides a comprehensive understanding of the considerations involved in ensuring that dissectors perform effectively.

4.1 Structure of the code



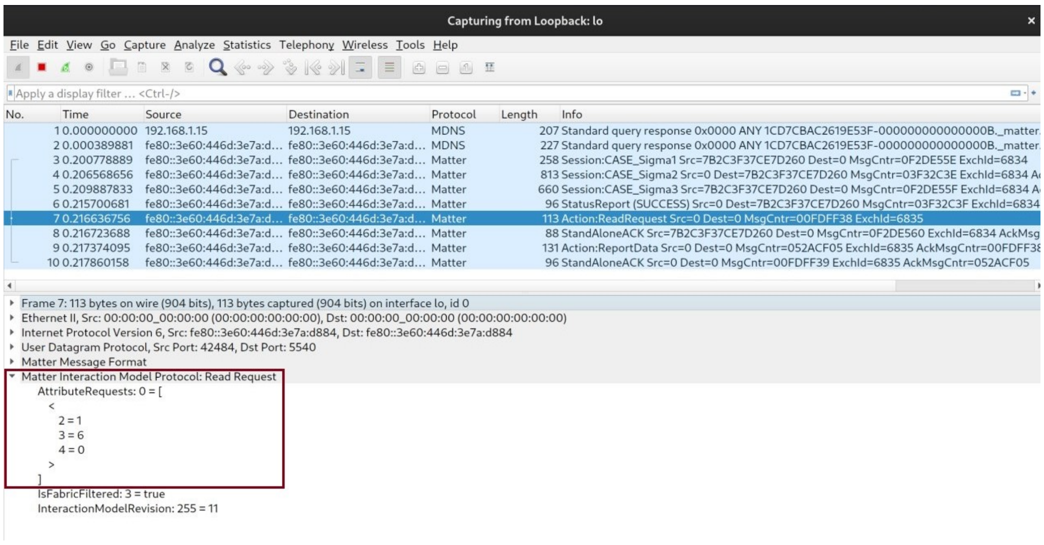
```
--- matter-dissector
├─ .gitignore
├─ CONTRIBUTING.md
├─ HKDF.c
├─ HKDF.h
├─ LICENSE
├─ Makefile
├─ MatterMessageTracker.cpp
├─ MatterMessageTracker.h
├─ MatterMinimal
│  ├─ include
│  │  └─ Matter
│  │     ├─ Core
│  │     │  ├─ MatterConfig.h
│  │     │  ├─ MatterCore.h
│  │     │  ├─ MatterEncoding.h
│  │     │  ├─ MatterError.h
│  │     │  ├─ MatterExchangeMgr.h
│  │     │  ├─ MatterFabricState.h
│  │     │  ├─ MatterMessageLayer.h
│  │     │  ├─ MatterSessionIds.h
│  │     │  ├─ MatterTLV.h
│  │     │  ├─ MatterTLVTags.h
│  │     │  ├─ MatterTLVTypes.h
│  │     │  └─ MatterVendorIdentifiers.hpp
│  │     ├─ Protocols
│  │     │  ├─ MatterProfiles.h
│  │     │  ├─ bulk-data-transfer
│  │     │  │  ├─ BulkDataTransfer.h
│  │     │  │  └─ Development
│  │     │  │     └─ BDXConstants.h
│  │     │  ├─ common
│  │     │  │  └─ CommonProfile.h
│  │     │  ├─ device-control
│  │     │  │  └─ DeviceControl.h
│  │     │  ├─ device-description
│  │     │  │  └─ DeviceDescription.h
│  │     │  ├─ echo
│  │     │  │  └─ MatterEcho.h
│  │     │  ├─ fabric-provisioning
│  │     │  │  └─ FabricProvisioning.h
│  │     │  ├─ interaction-model
│  │     │  │  └─ MessageDef.h
│  │     │  ├─ network-provisioning
│  │     │  │  └─ NetworkProvisioning.h
│  │     │  ├─ security
│  │     │  │  ├─ MatterCASE.h
│  │     │  │  └─ MatterSecurity.h
│  │     │  ├─ service-provisioning
│  │     │  │  └─ ServiceProvisioning.h
│  │     │  ├─ software-update
│  │     │  │  └─ SoftwareUpdateProfile.h
│  │     │  └─ time
│  │     │     └─ MatterTime.h
│  │     └─ Support
│  │        ├─ CodeUtils.h
│  │        ├─ ErrorStr.h
│  │        ├─ MatterNames.h
│  │        └─ SafeInt.h
│  └─ src
│     └─ lib
│        ├─ core
│        │  └─ MatterTLVReader.cpp
│        └─ support
│           ├─ ErrorStr.cpp
│           ├─ ErrorStr.h
│           ├─ MatterNames.cpp
│           └─ StatusReportStr.cpp
├─ MessageEncryptionKey.cpp
├─ MessageEncryptionKey.h
├─ README.md
├─ TLVDissector.cpp
├─ TLVDissector.h
├─ UserEncryptionKeyPrefs.cpp
├─ UserEncryptionKeyPrefs.h
├─ docs
│  └─ matter-dissector-screen.png
├─ moduleinfo.h
├─ packet-matter-common.cpp
├─ packet-matter-decrypt.cpp
├─ packet-matter-decrypt.h
├─ packet-matter-echo.cpp
├─ packet-matter-im.cpp
├─ packet-matter-security.cpp
├─ packet-matter.cpp
├─ packet-matter.h
└─ tests
   ├─ chip_tool_test_TestCluster_22f09.pcapng
   ├─ matter_echo.pcapng
   ├─ matter_pair_then_onoff.pcapng
   └─ test-packet-matter-decrypt.cpp
```

Figure 4.1: Codebase

The most pertinent files for the development of the requisite code are as follows:

- TLVDissector.cpp
- MessageDef.h
- Packet-Matter-im.cpp

The files coming from the official connected-home-ip repo contain the fundamental elements of the Matter Interaction Model and numerous functions for the analysis and display of packet content in Wireshark.
The use of the `chip-tool` and `all-cluster-app` enables the observation of the communication between the two tools with Wireshark listening. This implies that, based on the request sent via the `chip-tool` terminal, a response will be received from the `all-cluster-app`. In the context of Wireshark, analysis of individual packets exchanged during communication is possible from the loopback interface, given that the environment is local.
Although the dissector has a discrete implementation, it was deemed necessary to supplement the documentation with additional details to assist developers and testers in comprehending the message exchange without having to continually refer to the specification. As illustrated in Figure 4.2, decontextualized values do not provide a clear and comprehensive representation of the underlying phenomenon. Therefore, it is frequently necessary to consult the specification to obtain the semantic translation of the values. The addition of more detail to the dissector serves to facilitate data interpretation and reduce reliance on documentation.

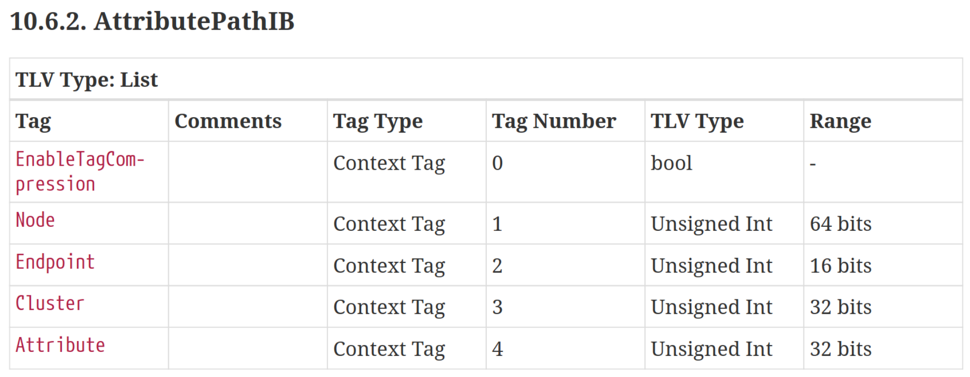

Capturing from Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.   Time         Source                 Destination            Protocol  Length  Info
   1 0.000000000   192.168.1.15           192.168.1.15           MDNS       207 Standard query response 0x0000 ANY 1CD7CBAC2619E53F-000000000000000B._matter.
   2 0.000389881   fe80::3e60:446d:3e7a:d... fe80::3e60:446d:3e7a:d... MDNS    227 Standard query response 0x0000 ANY 1CD7CBAC2619E53F-000000000000000B._matter.
   3 0.200778889   fe80::3e60:446d:3e7a:d... fe80::3e60:446d:3e7a:d... Matter  258 Session:CASE_Sigma1 Src=7B2C3F37CE7D260 Dest=0 MsgCntr=0F2DE55E ExchId=6834
   4 0.206568656   fe80::3e60:446d:3e7a:d... fe80::3e60:446d:3e7a:d... Matter  813 Session:CASE_Sigma2 Src=0 Dest=7B2C3F37CE7D260 MsgCntr=03F32C3E ExchId=6834 A
   5 0.209887833   fe80::3e60:446d:3e7a:d... fe80::3e60:446d:3e7a:d... Matter  660 Session:CASE_Sigma3 Src=7B2C3F37CE7D260 Dest=0 MsgCntr=0F2DE55F ExchId=6834 A
   6 0.215700681   fe80::3e60:446d:3e7a:d... fe80::3e60:446d:3e7a:d... Matter   96 StatusReport (SUCCESS) Src=0 Dest=7B2C3F37CE7D260 MsgCntr=03F32C3F ExchId=6834
   7 0.216636756   fe80::3e60:446d:3e7a:d... fe80::3e60:446d:3e7a:d... Matter  113 Action:ReadRequest Src=0 Dest=0 MsgCntr=00FDFF38 ExchId=6835
   8 0.216723688   fe80::3e60:446d:3e7a:d... fe80::3e60:446d:3e7a:d... Matter   88 StandAloneACK Src=7B2C3F37CE7D260 Dest=0 MsgCntr=0F2DE560 ExchId=6834 AckMsg
   9 0.217374095   fe80::3e60:446d:3e7a:d... fe80::3e60:446d:3e7a:d... Matter  131 Action:ReportData Src=0 Dest=0 MsgCntr=052ACF05 ExchId=6835 AckMsgCntr=00FDFF3
  10 0.217860158   fe80::3e60:446d:3e7a:d... fe80::3e60:446d:3e7a:d... Matter   96 StandAloneACK Src=0 Dest=0 MsgCntr=00FDFF39 ExchId=6835 AckMsgCntr=052ACF05

▸ Frame 7: 113 bytes on wire (904 bits), 113 bytes captured (904 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 6, Src: fe80::3e60:446d:3e7a:d884, Dst: fe80::3e60:446d:3e7a:d884
▸ User Datagram Protocol, Src Port: 42484, Dst Port: 5540
▸ Matter Message Format
▾ Matter Interaction Model Protocol: Read Request
    AttributeRequests: 0 = [
      <
        2 = 1
        3 = 6
        4 = 0
      >
    ]
    IsFabricFiltered: 3 = true
    InteractionModelRevision: 255 = 11
```

Figure 4.2: Wireshark window without semantic number

It is easy to think that these numbers do not have a well-defined meaning at first glance, but by reading the specification, it is possible to see that in a read-type packet that is exchanged inside, there will be values that indicate certain fields/attributes.



10.6.2. AttributePathIB

TLV Type: List

Tag	Comments	Tag Type	Tag Number	TLV Type	Range
EnableTagCom-pression		Context Tag	0	bool	-
Node		Context Tag	1	Unsigned Int	64 bits
Endpoint		Context Tag	2	Unsigned Int	16 bits
Cluster		Context Tag	3	Unsigned Int	32 bits
Attribute		Context Tag	4	Unsigned Int	32 bits

Figure 4.3: page 634 Matter specific version:1.3

As illustrated, the values to the left of the equal sign will be interpreted following the specification. For example, consider the number 2, which represents a specific node within our network. This node is equivalent to 1, signifying the network endpoint. This is because the send packet was forwarded to endpoint 1.

4.2 TLVDissector

The `TLVdissector.cpp` file is a critical component within Wireshark that is responsible for dissecting messages encoded in the `TLV (Tag-Length-Value)` format. This file plays a pivotal role in ensuring the accurate analysis and interpretation of data packets that conform to the TLV structure. The following are some of the most important parts of this file, along with an accompanying explanation. A TLV code are already implemented in order to execute the TLV encoding and decoding methodologies, as outlined in the appendix of the Matter specification. TLV, or Tag-Length-Value, is a process of encoding tagged values and structures in a compact binary format. The majority of Matter messages are encoded using TLV, which makes this implementation crucial for the effective handling of these messages within Wireshark. The TLV encoding format is highly efficient and allows for the structured representation of data. Each TLV item consists of:

- Tag: an identifier that indicates the type of the value.
- Length: of the value field.
- Value: The actual data being encoded.

This structure enables the parsing and comprehension of intricate data structures in a manner that is both space-efficient and straightforward to navigate programmatically.
All the code was already present in the file and therefore no changes were introduced

4.2.1 Error Handling

TLVdissector.cpp incorporates comprehensive error handlers to cope with any possible issues that may occur throughout the decode process: this may pertain to the handling of malformed TLV items, unexpected end-of-data scenarios, and mismatched tag-length-value triplets. Proper error handling makes the dissector stable and reliable in the sense that it guarantees correctness and reliability of data interpretation in the presence of corrupted or incomplete data packets.

4.2.2 Integration into Wireshark

TLVdissector.cpp has been integrated transparently into the existing framework of Wireshark. Thus, such a dissector can automatically take advantage of very powerful packet analysis and visualization facilities available in Wireshark to present, at the end, a detailed view of decoded TLV messages to an end user. The dissector will then register itself with the protocol dissectors available in Wireshark to be able to automatically recognize and treat TLV-encoded packets as part of packet capture and analysis sessions.

4.2.3 TLV constructor and destructor

```
using namespace Matter::TLV;
using namespace Matter::Profiles::InteractionModel;

TLVDissector::TLVDissector()
{
    mElemStart = mElemLen = 0;
    mLastItem = NULL;
    mContainerStack = NULL;
    mContainerStackSize = 0;
    mContainerDepth = 0;
    mSourceBufBaseOffset = 0;
}

TLVDissector::~TLVDissector()
{
    if (mContainerStack != NULL)
        free(mContainerStack);
}
```

This code defines a class TLVDissector within the namespaces Matter::TLV and Matter::Profiles::InteractionModel.

- **Namespace Usage:** The using namespace statements bring symbols from Matter::TLV and Matter::Profiles::InteractionModel into scope.
- **Constructor (TLVDissector::TLVDissector()):** Initializes member variables mElemStart, mElemLen, mLastItem, mContainerStack, mContainerStackSize, mContainerDepth, and mSourceBufBaseOffset to specific values.
- **Destructor (TLVDissector:: TLVDissector()):** Checks if mContainerStack is not NULL and frees the memory allocated to it using free(), ensuring proper cleanup of dynamically allocated resources.

4.2.4 TLV dissector Next

```
Matter_ERROR TLVDissector::Next(void)
{
    Matter_ERROR err;

    mLastItem = NULL;

    err = Skip();
    SuccessOrExit(err);

    err = MarkElemStart();
    SuccessOrExit(err);

    err = ((TLVReader *)this)->Next();
    SuccessOrExit(err);

    err = MarkElemLength();
    SuccessOrExit(err);

exit:
    return err;
}
```

- **Initialization:** It initializes ‘mLastItem‘ to ‘NULL‘.
- **Marking Element Start:** It calls ‘MarkElemStart()‘ to mark the start of an element, likely for further processing or analysis.
- **Calling Base Class Method:** It invokes the ‘Next()‘ method from the ‘TLVReader‘ base class or interface (cast to ‘(TLVReader *)this‘), which reads the next TLV item.
- **Marking Element Length:** It calls ‘MarkElemLength()‘ to determine or mark the length of the current TLV element.
- **Error Handling:** The ‘SuccessOrExit(err);‘ macro or function is used after each operation to check if ‘err‘ indicates success; if not, it exits the function and returns ‘err‘.

4.2.5 TLV dissector Enter Container

```
Matter_ERROR TLVDissector::EnterContainer()
{
    Matter_ERROR err;
    TLVType containerType;

    err = ((TLVReader *)this)->EnterContainer(containerType);
    SuccessOrExit(err);

    err = PushContainer(containerType);
    SuccessOrExit(err);

exit:
    return err;
}

Matter_ERROR TLVDissector::ExitContainer()
{
    Matter_ERROR err;
    TLVType containerType;

    containerType = PopContainer();

    err = ((TLVReader *)this)->ExitContainer(containerType);
    SuccessOrExit(err);

exit:
    return err;
}
```

- **EnterContainer():**
  - Calls the ‘EnterContainer()’ method from the base class ‘TLVReader’, passing ‘containerType’ as a parameter.
  - Checks for success (‘SuccessOrExit(err)’).
  - Calls ‘PushContainer(containerType)’ to push ‘containerType’ onto a container stack.
  - Uses ‘exit:‘ label for error handling and returns ‘err‘.

- **ExitContainer():**
  - Pops the ‘containerType’ from a container stack using ‘PopContainer()’.
  - Calls the ‘ExitContainer()’ method from the base class ‘TLVReader’, passing ‘containerType’ as a parameter.
  - Checks for success (‘SuccessOrExit(err)’).
  - Uses ‘exit:‘ label for error handling and returns ‘err‘.

4.2.6 Append IM Path

```
Matter_ERROR
AppendIMPath(TLVReader& reader, char *& strBuf, size_t&
↪ strBufSize)
{
    Matter_ERROR err;
    TLVType pathContainer, instanceLocatorContainer;
    bool addSep = false;

    VerifyOrExit(reader.GetType() == kTLVType_Path, err =
    ↪ Matter_ERROR_UNEXPECTED_TLV_ELEMENT);

    err = reader.EnterContainer(pathContainer);
    SuccessOrExit(err);
    AppendStringF(strBuf, strBufSize, "//");

    while (true) {
        err = reader.Next();
        if (err == Matter_END_OF_TLV)
            break;
        SuccessOrExit(err);

        TLVType type = reader.GetType();
        uint64_t tag = reader.GetTag();
        unsigned tagNum = TagNumFromTag(tag);

        VerifyOrExit(IsContextTag(tag), err =
        ↪ Matter_ERROR_UNEXPECTED_TLV_ELEMENT);

        if (addSep) {
            AppendStringF(strBuf, strBufSize, ",");
        }

        switch (TagNumFromTag(tag)) {
        case CommandPathIB::kTag_Endpoint:
        case CommandPathIB::kTag_Cluster:
        case CommandPathIB::kTag_Command:
        {
            VerifyOrExit(type == kTLVType_UnsignedInteger, err =
            ↪ Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
```

```
                uint32_t pathElement;
                err = reader.Get(pathElement);
                SuccessOrExit(err);

                char *pathElementName;
                switch (tagNum) {
                    case CommandPathIB::kTag_Endpoint:
                     ↪  pathElementName = "Endpoint"; break;
                    case CommandPathIB::kTag_Cluster:
                     ↪  pathElementName = "Cluster"; break;
                    case CommandPathIB::kTag_Command:
                     ↪  pathElementName = "Command"; break;
                    default:
                        pathElementName = "Unknown";
                }
                AppendStringF(strBuf, strBufSize, "%s=0x%08" PRIX32,
                 ↪  pathElementName, pathElement);
                break;
            }

            default:
                ExitNow(err = Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
            }

            addSep = true;
        }

    AppendStringF(strBuf, strBufSize, "/");
    addSep = false;

    while (true) {
        err = reader.Next();
        if (err == Matter_END_OF_TLV)
            break;
        SuccessOrExit(err);

        TLVType type = reader.GetType();
        uint64_t tag = reader.GetTag();

        VerifyOrExit(IsContextTag(tag), err =
         ↪  Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        VerifyOrExit(type == kTLVType_Null, err = Matter_ERROR_UNEXPECTED_TLV_ELEMENT);

        AppendStringF(strBuf, strBufSize, "%s%" PRId32, (addSep) ? "/" : "", TagNumFromTag(tag));

        addSep = true;
    }

    err = reader.ExitContainer(pathContainer);
    SuccessOrExit(err);

exit:
    if (err != Matter_NO_ERROR) {
        if (strBuf != NULL) {
            free(strBuf);
            strBuf = NULL;
        }
    }
    return err;
}
```

The function `AppendIMPath`, already present in the repo, appends an Interaction Model (IM) path, read from a `TLVReader`, to a string buffer, resizing the buffer if necessary.

- **Initialization:**
- **Enter Path Container:**
  - Enter the TLV container representing the path.
  - Append "//" to the string buffer to indicate the start of the path.
- **Iterate Through Path Elements:**
  - Loop through the elements of the path container.
  - For each element, determine its type and tag.
  - Verify that the tag is a context tag and the type matches the expected type.
  - Append the path element to the string buffer in the format "ElementName=0xElementValue".

  - Set the `addSep` flag to add a comma between elements.

- **Handle Separator and Instance Locators:**
  - Append "/" to the string buffer to separate path elements from instance locators.
  - Loop through instance locator elements in a similar manner as path elements.
  - Append each instance locator to the string buffer, prefixed with a "/" if necessary.

- **Exit Containers:**
  - Exit the path container.

4.2.7 TLV dissector add generic TLV item

```
    Matter_ERROR TLVDissector::AddGenericTLVItem(proto_tree
    ↪ *tree, int hfindex, tvbuff_t *tvb, bool suppressTag)
{
    return AddGenericTLVItem(tree, hfindex, tvb, suppressTag,
    ↪ false, 0);
}
```

The function `AddGenericTLVItem` is a helper function that calls another overloaded version of `AddGenericTLVItem`, providing default values for some of its parameters.

- **Function Signature:**
  - The function is defined to take four parameters: a pointer to a protocol tree (`proto_tree* tree`), a header field index (`int hfindex`), a pointer to a tvbuff (`tvbuff_t* tvb`), and a boolean flag to suppress tags (`bool suppressTag`).

- **Call to Overloaded Function:**
  - The function calls another version of `AddGenericTLVItem`, passing `tree`, `hfindex`, `tvb`, and `suppressTag` as arguments.
  - It also provides default values for two additional parameters: `false` for `someBoolFlag` and `0` for `someIntValue`.

- **Return Value:**
  - The function returns the result of the call to the overloaded `AddGenericTLVItem` function.

4.2.8 TLV dissector Add list item

```
    Matter_ERROR TLVDissector::AddListItem(proto_tree* tree, int
    ↪  hfindex, int ett, tvbuff_t* tvb, ListElemHandler
    ↪  elemHandler)
{
    Matter_ERROR err;
    proto_tree *listTree;
    proto_item *listItem;
    uint32_t elemCount = 0;

    VerifyOrExit(GetType() == kTLVType_Array || GetType() ==
    ↪  kTLVType_Path, err = Matter_ERROR_INCORRECT_STATE);

    err = AddSubTreeItem(tree, hfindex, ett, tvb, listTree);
    SuccessOrExit(err);

    listItem = LastItem();

    err = EnterContainer();
    SuccessOrExit(err);

    while (true) {
        err = Next();
        if (err == Matter_END_OF_TLV)
            break;
        SuccessOrExit(err);

        err = elemHandler(*this, listTree, tvb);
        SuccessOrExit(err);

        elemCount++;
    }

    err = ExitContainer();
    SuccessOrExit(err);

    if (listItem != NULL) {
        proto_item_append_text(listItem, (" (len = %" PRIu32
        ↪  ")"), elemCount);
    }

exit:
    return err;
}
```

The function `AddListItem` processes a list of TLV elements, adding them to a protocol tree. Here's a concise breakdown of its functionality:

- **Function Signature:**
  - The function takes five parameters:
    - `proto_tree* tree`: Pointer to the protocol tree.
    - `int hfindex`: Header field index.
    - `int ett`: Ett index.
    - `tvbuff_t* tvb`: Pointer to the tvbuff.
    - `ListElemHandler elemHandler`: Function pointer to handle individual list elements.
- **Initialization:**
  - Declare variables for error handling (`err`), the list tree (`listTree`), the list item (`listItem`), and the element count (`elemCount`).
- **Type Verification:**
  - Verify that the current TLV type is either `kTLVType_Array` or `kTLVType_Path`. Exit with an error if not.
- **Add SubTree Item:**
  - Call `AddSubTreeItem` to add a subtree item to the protocol tree.
  - Retrieve the last item added to the tree (`listItem`).
- **Enter Container:**

- Enter the TLV container.

- **Process Elements:**
  - Loop through the elements in the container.
  - Call `Next` to move to the next element. Break the loop if the end of the TLV is reached.
  - Use the `elemHandler` function to process each element and add it to the `listTree`.
  - Increment the element count (`elemCount`) for each processed element.

- **Append Element Count:**
  - If `listItem` is not `NULL`, append the element count to the list item text.

4.2.9 TLV dissector add generic TLV item

```
Matter_ERROR TLVDissector::AddGenericTLVItem(proto_tree *tree,
↪ int hfindex, tvbuff_t *tvb, bool suppressTag, bool
↪ suppressHeader, int indentLevel)
{
    Matter_ERROR err = Matter_NO_ERROR;
    TLVType type = GetType();
    int64_t tag = GetTag();
    char *strBuf = NULL;
    size_t strBufSize = 0;
    char closeChar = ' ';

    if (suppressHeader) {
        AppendRepeated("    ", indentLevel, strBuf, strBufSize);
    }

    if (!suppressTag) {
        if (IsContextTag(tag)) {
            uint32_t contextTagNum = TagNumFromTag(tag);
            AppendStringF(strBuf, strBufSize, ("%" PRId32 " =
            ↪ "), contextTagNum);
        }
        else if (IsProfileTag(tag)) {
            uint32_t profileId = ProfileIdFromTag(tag);
            uint32_t tagNum = TagNumFromTag(tag);
            AppendStringF(strBuf, strBufSize, ("%08" PRIX32 ":"
             ↪  PRId32 " = "), profileId, tagNum);
        }
    }

    switch (type) {
    case kTLVType_SignedInteger:
    {
        int64_t val;
        err = Get(val);
        SuccessOrExit(err);
        AppendStringF(strBuf, strBufSize, ("%" PRId64), val);
        break;
    }
    case kTLVType_UnsignedInteger:
    {
        .....
    }
    case kTLVType_Boolean:
    {
        .....
    }
    case kTLVType_FloatingPointNumber:
    {
        .....
    }
    case kTLVType_UTF8String:
    {
        char *val;
        err = DupString(val);
        SuccessOrExit(err);
        uint32_t escapedLen = escape_string_len(val);
        char *escapedVal = (char *)malloc(escapedLen + 1);
        escape_string(escapedVal, val);
        AppendStringF(strBuf, strBufSize, "%s", escapedVal);
        free(val);
        free(escapedVal);
```

```
            break;
        }
        case kTLVType_ByteString:
        {
            const uint8_t *data;
            uint32_t dataLen = GetLength();
            err = GetDataPtr(data);
            SuccessOrExit(err);
            for (uint32_t i = 0; i < dataLen; i++, data++) {
                AppendStringF(strBuf, strBufSize, " %02x", *data);
            }
            break;
        }
        case kTLVType_Null:
            AppendStringF(strBuf, strBufSize, "null");
            break;
        case kTLVType_Structure:
            AppendStringF(strBuf, strBufSize, "{");
            closeChar = '}';
            break;
        case kTLVType_Array:
            AppendStringF(strBuf, strBufSize, "[");
            closeChar = ']';
            break;
        case kTLVType_Path:
            AppendStringF(strBuf, strBufSize, "<");
            closeChar = '>';
            break;
        default:
            ExitNow(err = Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        }

        if (!suppressHeader) {
            err = AddStringItemF(tree, hfindex, tvb, "%s", strBuf);
            SuccessOrExit(err);
        }

        else {
            err = AddStringItemWithHeaderF(tree, hfindex, tvb, "%s",
             ↪  strBuf);
```

```c
        SuccessOrExit(err);
    }

    // Reset the string.
    strBuf[0] = 0;

    if (type == kTLVType_Structure || type == kTLVType_Array || type == kTLVType_Path) {

        err = EnterContainer();
        SuccessOrExit(err);

        while (true) {
            err = Next();
            if (err == Matter_END_OF_TLV)
                break;
            SuccessOrExit(err);
            err = AddGenericTLVItem(tree, hfindex, tvb, false, true, indentLevel + 1);

            SuccessOrExit(err);
        }

        err = ExitContainer();
        SuccessOrExit(err);

        AppendRepeated("    ", indentLevel, strBuf, strBufSize);
        AppendStringF(strBuf, strBufSize, "%c", closeChar);

        err = AddStringItemWithHeaderF(tree, hfindex, tvb, "%s", strBuf);
        SuccessOrExit(err);
    }

exit:
    if (strBuf != NULL)
        free(strBuf);
    return err;
}
```

The function `AddGenericTLVItem` processes various TLV (Tag-Length-Value) elements and adds them to a protocol tree.

- **Function Signature:**
  - The function takes six parameters:
    - `proto_tree *tree`: Pointer to the protocol tree.
    - `int hfindex`: Header field index.
    - `tvbuff_t *tvb`: Pointer to the tvbuff.
    - `bool suppressTag`: Flag to suppress the tag in the output.
    - `bool suppressHeader`: Flag to suppress the header in the output.
    - `int indentLevel`: The indentation level for formatting the output.
- **Initialization:**
  - Declare variables for error handling (`err`), TLV type (`type`), tag (`tag`), string buffer (`strBuf`), string buffer size (`strBufSize`), and closing character (`closeChar`).
- **String Buffer Handling:**
  - Append indentation to the string buffer if the header is suppressed.
  - Append tag information to the string buffer if the tag is not suppressed.
- **TLV Type Handling:**
  - Based on the TLV type, extract the value and append it to the string buffer. Handle different TLV types such as signed integer, unsigned integer, boolean, floating point number, UTF-8 string, byte string, null, structure, array, and path.
- **Adding to Protocol Tree:**
  - Add the string buffer content to the protocol tree using either `AddStringItemF` or `AddStringItemWithHeaderF`, depending on whether the header is suppressed.

4.2.10 TLV dissector Add sub tree item

```
    Matter_ERROR TLVDissector::AddSubTreeItem(proto_tree* tree,
     ↪  int hfindex, int ett, tvbuff_t* tvb, proto_tree*&
     ↪  subTree)
{
    Matter_ERROR err = Matter_NO_ERROR;

    VerifyOrExit(GetType() != kTLVType_NotSpecified, err =
     ↪  Matter_ERROR_INCORRECT_STATE);

    mLastItem = proto_tree_add_item(tree, hfindex, tvb,
     ↪  mElemStart + mSourceBufBaseOffset, mElemLen, ENC_NA);

    subTree = proto_item_add_subtree(mLastItem, ett);

exit:
    return err;
}
```

The function `AddSubTreeItem` adds a subtree item to a protocol tree in the TLV (Tag-Length-Value) dissector. Here's a detailed breakdown of its functionality:

- **Function Signature:**
  - The function takes five parameters:
    - `proto_tree *tree`: Pointer to the main protocol tree.
    - `int hfindex`: Header field index.
    - `int ett`: Ett (Expert Information Tree) index.
    - `tvbuff_t *tvb`: Pointer to the tvbuff.
    - `proto_tree *&subTree`: Reference to a pointer for the subtree.
- **Initialization:**
  - Declare an error variable (`err`) initialized to `Matter_NO_ERROR`.
- **State Verification:**
  - Verify that the current TLV type is not `kTLVType_NotSpecified`.

  - If the type is
    `kTLVType_NotSpecified`,
    set the error to `Matter_ERROR_INCORRECT_STATE` and exit.

- **Adding Item to Protocol Tree:**
  - Add the current item to the protocol tree using
    `proto_tree_add_item`.
  - Store the result in `mLastItem`.

- **Adding Subtree:**
  - Add a subtree to `mLastItem` using `proto_item_add_subtree`.
  - Store the result in the reference `subTree`.

The file contains a number of functions, some of which are listed here. These functions are also invoked in the `Packet-Matter-im.cpp` file, where additional functions have been added to achieve the intended purpose.

4.3 MessageDef.h

To correctly manage and interpret these values, they have been inserted into a special file, called *MessageDef.h*. This file contains the value mappings and the functions necessary for their decoding.

```
#pragma once

namespace Matter {
namespace Profiles {
namespace InteractionModel {
```

These namespaces encapsulate code within a structured hierarchy, which serves to organize and prevent name conflicts. The code is part of the Matter protocol, specifically in the context of the interaction model (InteractionModel).

```
/**
 * Interaction Model Opcodes
 * See 10.2.1. Protocol OpCodes
 */
enum
{
    kMsgType_StatusResponse              = 0x01,
    kMsgType_ReadRequest                 = 0x02,
    kMsgType_SubscribeRequest            = 0x03,
    kMsgType_SubscribeResponse           = 0x04,
    kMsgType_ReportData                  = 0x05,
    kMsgType_WriteRequest                = 0x06,
    kMsgType_WriteResponse               = 0x07,
    kMsgType_InvokeRequest               = 0x08,
    kMsgType_InvokeResponse              = 0x09,
    kMsgType_TimedRequest                = 0x0A,
};


```

The specification in section 10.2.1 reports that: "Each Action in the IM specification SHALL be mapped to a message with a unique Protocol Opcode, namespaced under the PROTOCOL_ID_INTERACTION_MODEL Protocol ID." [20]

4.3.1 Common Action Info

```
namespace CommonActionInfo
{
    enum {
        kTag_InteractionModelRevision  = 0xFF,
    };
};
```

The constant kTag_InteractionModelRevision, for example, represents the revision of the interaction model and is of critical importance in ensuring that messages are interpreted correctly by the specific version of the protocol. The inclusion of this element in action messages guarantees that data exchange will occur in a manner that is both compliant and predictable.

4.3.2 Implementing Read Transaction

The code added to implement the new features will then be shown in this section. All of this always refers to the MesageDef.h file. In particular, it is necessary to underline that this code introduced is faithful to the specification for semanticizing the **reading interaction.**

4.3.3 Attribute Path IB

```
namespace AttributePathIB {
    enum{
        kTag_enableTagCompression    = 0,    // boolean
        kTag_node                    = 1,    // uint64
        kTag_endpoint                = 2,    // uint16
        kTag_cluster                 = 3,    // uint32
        kTag_attribute               = 4,    // uint32
        kTag_listIndex               = 5,    // uint16
        kTag_WildcardPathFlags       = 6,    // uint32
    };};
```

Due to the absence of the requisite data in the currently available version of the repository managed by the CSA, it was necessary to supplement this with other fields at a later stage to achieve the objective of the project. The following sections will present and elucidate the additional fields.

4.3.3.1 EnableTagCompression

This tag allows two different interpretations when tags `Node`, `Endpoint`, `Cluster`, `Attribute` are omitted:

- **False or not present**: The omission of tags (except `Node`) indicates wildcard semantics.
- **True**: Uses a tag compression scheme. The value for any omitted tag is set to the value from the last `AttributePathIB` seen in the same context.

4.3.3.2 Node

- If the `Group` field is present, the group ID is encoded in the DST field of the message header and omitted here.

- The `Node` tag may be omitted if the path's target node matches the `NodeID` of the involved server.

4.3.3.3 Endpoint, Cluster

- These tags can be omitted. Their omission depends on the value of `EnableTagCompression`.

4.3.3.4 Attribute, ListIndex

- **EnableTagCompression false or not present**:
  - `Attribute omitted, ListIndex omitted`: Selects all attributes within the specified `Node`, `Endpoint`, `Cluster`.
  - `Attribute present, ListIndex omitted`: Selects a specific attribute within the specified `Node`, `Endpoint`, `Cluster`.
  - `Attribute present, ListIndex present`: Selects a specific list item within a top-level attribute of type list.
- The `ListIndex` tag is nullable. A null value indicates a list append operation.

4.3.3.5 WildcardPathFlags

- The flags present in this tag are defined in `WildcardPathFlagsBitmap`.
- The absence of the `WildcardPathFlags` tag indicates that all flags are false.
- If the value of `WildcardPathFlags` is zero, it should be omitted by clients.
- The effect of `WildcardPathFlags` on the processing of `AttributePathIB` is described in Section 8.2.1.7.1, *Attribute Wildcard Path Flags* [20].

4.3.4 Attribute IB

```
namespace AttributeReportIB {
    enum
    {
        kTag_AttributeStatus          = 0,    //
         ↪  AttributeStatusIB
        kTag_AttributeData            = 1,    // AttributeDataIB
    };
}

namespace AttributeDataIB {
    enum
    {
        kTag_DataVersion              = 0,    // uint32
        kTag_Path                     = 1,    // AttributePathIB
        kTag_Data                     = 2,    // variable
    };
}

namespace AttributeStatusIB {
    enum
    {
        kTag_Path                     = 0,    // AttributePathIB
        kTag_Status                   = 1,    // StatusIB
    };
}
```

4.3.4.1 AttributeReportIB:

This namespace contains tags used to represent an attribute report, which can contain information about both the attribute status and the attribute data itself.

- `kTag_AttributeStatus` (0):
  - This tag represents an attribute status field. The value associated with this tag is an `AttributeStatusIB` object, providing information on the status or result of an operation on the attribute (e.g., whether it was read successfully or encountered an error).

- `kTag_AttributeData` (1):
  - This tag represents an attribute data field. The value associated with this tag is an `AttributeDataIB` object, containing the actual data of the attribute (e.g., the value of the attribute read from a device).

4.3.4.2 AttributeDataIB

This namespace contains tags used to represent the actual data of an attribute in a report.

- `kTag_DataVersion` (0):
  - This tag represents the version of the attribute data. It is a 32-bit unsigned integer (`uint32`). The data version is important for consistency management and to determine if the data is up-to-date.
- `kTag_Path` (1):
  - This tag represents the path of the attribute within the data structure. It is an `AttributePathIB` object, uniquely identifying the specific attribute in question (e.g., which device, endpoint, cluster, and attribute it refers to).
- `kTag_Data` (2):
  - This tag represents the actual value of the attribute. The type of this value is variable (`variable`), as it can be of various types depending on the specific attribute (e.g., integers, strings, booleans).

4.3.4.3 AttributeStatusIB

This namespace contains tags used to represent the status of an attribute in a report.

- `kTag_Path` (0):
  - This tag represents the path of the attribute. It is an `AttributePathIB` object and specifies which attribute is referred to in the report.
- `kTag_Status` (1):

  – This tag represents the status of the attribute. It is a `StatusIB` object, providing information on the attribute's status (e.g., whether the operation on the attribute was completed successfully or if an error occurred).

4.4 Packet-Matter-im.cpp

Inside the Packet-Matter-im.cpp file, we have the operating logic of the interaction model. The functions were modified or introduced from scratch to allow for the semanticization of the read interaction.

```
...
static int ett_ReadRequest_AttributeRequests = -1;
...
static int ett_AttributeStatusIB = -1;
static int ett_AttributeDataIB = -1;
...
static int hf_ReadAttributeRequest_enableTagCompression = -1;
static int hf_ReadAttributeRequest_node = -1;
static int hf_ReadAttributeRequest_endpoint = -1;
static int hf_ReadAttributeRequest_cluster = -1;
static int hf_ReadAttributeRequest_attribute = -1;
static int hf_ReadAttributeRequest_listIndex = -1;
static int hf_ReadAttributeRequest_WildcardPathFlags = -1;

static int hf_AttributeDataIB = -1;
static int hf_AttributeDataIB_DataVersion = -1;
static int hf_AttributePathIB = -1;

static int hf_AttributeReportIB = -1;
static int hf_AttributeReportIB_AttributeStatus = -1;
static int hf_AttributeReportIB_AttributeData = -1;

static int hf_StatusIB_Status = -1;
static int hf_StatusIB_ClusterStatus = -1;
...
static int hf_StatusIB = -1;
...
```

This is the initialization of the new useful attributes. In the above code, a series of static variables of the integer data type are declared. These variables are utilized to represent "Entity Trees" (ett) and "Header Fields" (hf) within the context of network packet analysis. These variables are complementary with the namespace in the Messagedef.h defined previously.

4.4.1 Add Status IB

```
static Matter_ERROR
AddStatusIB(TLVDissector& tlvDissector, proto_tree *tree,
↪  tvbuff_t* tvb)
{
    Matter_ERROR err;
    proto_tree *dataElemTree;
    int hf_entry;

    err = tlvDissector.AddSubTreeItem(tree, hf_StatusIB,
    ↪  ett_DataElem, tvb, dataElemTree);
    SuccessOrExit(err);

    err = tlvDissector.EnterContainer();
    SuccessOrExit(err);

    while (true) {

        err = tlvDissector.Next();
        if (err == Matter_END_OF_TLV)
            break;
        SuccessOrExit(err);

        uint64_t tag = tlvDissector.GetTag();
        //TLVType type = tlvDissector.GetType();
        VerifyOrExit(IsContextTag(tag), err =
        ↪  Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        tag = TagNumFromTag(tag);
        switch (tag) {
            case StatusIB::kTag_Status:
                hf_entry = hf_StatusIB_Status;
                err = tlvDissector.AddTypedItem(dataElemTree,
                ↪  hf_entry, tvb);
                break;
            case AttributeStatusIB::kTag_Status:
                hf_entry = hf_StatusIB_ClusterStatus;
                err = tlvDissector.AddTypedItem(dataElemTree,
                 ↪  hf_entry, tvb);
                break;
            default:

                ExitNow(err =
                 ↪  Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        }
        SuccessOrExit(err);
    }

    err = tlvDissector.ExitContainer();
    SuccessOrExit(err);

exit:
    return err;
}
```

Function Parameters

- `TLVDissector& tlvDissector`: Reference to an object that manages TLV dissection.
- `proto_tree *tree`: Pointer to the tree structure where the decoded data will be added.
- `tvbuff_t* tvb`: Pointer to the buffer containing the data to be analyzed.

Local Variables

- `Matter_ERROR err`: Variable to handle potential errors.
- `proto_tree *dataElemTree`: Pointer to a subtree for data elements.
- `int hf_entry`: Temporary variable to store the current header field.

Function Operation

1. **Adding a Subtree**
   The function adds a subtree (`dataElemTree`) to the main tree (`tree`), using `hf_StatusIB` as the header field and `ett_DataElem` as the entity tree. If an error occurs, the function exits:

2. **Entering the TLV Container**
   The function enters the TLV container to start reading the data:

3. **Iterating Over TLV Tags**
   The function iterates over each TLV element within the container. For each element:

   The function:

   - Calls `Next()` to read the next TLV element.
   - Verifies that the tag is a context tag.
   - Converts the tag to get the tag number.
   - Adds the element to the subtree based on the identified tag:
     - `StatusIB::kTag_Status`
     - `AttributeStatusIB::kTag_Status`
   - If the tag is not recognized, the function exits with an error.

4. **Exiting the TLV Container**
   The function exits the TLV container after completing the data reading:

5. **Error Handling**
   The function returns the error code `err`:

4.4.2 Add Attribute Path IB

```cpp
static Matter_ERROR
AddAttributePathIB(TLVDissector& tlvDissector, proto_tree *tree, tvbuff_t* tvb)
{
    Matter_ERROR err;
    proto_tree *dataElemTree;
    int hf_entry;

    err = tlvDissector.AddSubTreeItem(tree, hf_AttributePathIB, ett_DataElem, tvb, dataElemTree);
    SuccessOrExit(err);

    err = tlvDissector.EnterContainer();
    SuccessOrExit(err);

    while (true) {

        err = tlvDissector.Next();
        if (err == Matter_END_OF_TLV)
            break;
        SuccessOrExit(err);

        uint64_t tag = tlvDissector.GetTag();
        //TLVType type = tlvDissector.GetType();
        VerifyOrExit(IsContextTag(tag), err = Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        tag = TagNumFromTag(tag);
        switch (tag) {
            case AttributePathIB::kTag_enableTagCompression:
                hf_entry = hf_ReadAttributeRequest_enableTagCompression;
                break;
            case AttributePathIB::kTag_node:
                hf_entry = hf_ReadAttributeRequest_node;
                break;
            case AttributePathIB::kTag_endpoint:
                hf_entry = hf_ReadAttributeRequest_endpoint;
                break;
            case AttributePathIB::kTag_cluster:
                hf_entry = hf_ReadAttributeRequest_cluster;
                break;
            case AttributePathIB::kTag_attribute:
                hf_entry = hf_ReadAttributeRequest_attribute;
                break;
            case AttributePathIB::kTag_listIndex:
                hf_entry = hf_ReadAttributeRequest_listIndex;
                break;
            case AttributePathIB::kTag_WildcardPathFlags:
                hf_entry =
                 ↪  hf_ReadAttributeRequest_WildcardPathFlags;
                break;
            default:
                ExitNow(err =
                 ↪  Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        }
        SuccessOrExit(err =
        ↪  tlvDissector.AddGenericTLVItem(dataElemTree,
        ↪  hf_entry, tvb, false));
    }

    err = tlvDissector.ExitContainer();
    SuccessOrExit(err);

exit:
    return err;
}
```

Function Parameters

- `TLVDissector& tlvDissector`: Reference to an object that manages TLV dissection.
- `proto_tree *tree`: Pointer to the tree structure where the decoded data will be added.
- `tvbuff_t* tvb`: Pointer to the buffer containing the data to be analyzed.

Local Variables

- `Matter_ERROR err`: Variable to handle potential errors.
- `proto_tree *dataElemTree`: Pointer to a subtree for data elements.
- `int hf_entry`: Temporary variable to store the current header field.

Function Operation

1. **Adding a Subtree**
   The function adds a subtree (`dataElemTree`) to the main tree (`tree`), using `hf_AttributePathIB` as the header field and `ett_DataElem` as the entity tree. If an error occurs, the function exits.
2. **Entering the TLV Container**
   The function enters the TLV container to start reading the data.
3. **Iterating Over TLV Tags**
   The function iterates over each TLV element within the container. For each element:
   - Calls `Next()` to read the next TLV element.
   - Verifies that the tag is a context tag.
   - Converts the tag to get the tag number.
   - Adds the element to the subtree based on the identified tag:
     - `AttributePathIB::kTag_enableTagCompression`
     - `AttributePathIB::kTag_node`
     - `AttributePathIB::kTag_endpoint`
     - `AttributePathIB::kTag_cluster`
     - `AttributePathIB::kTag_attribute`
     - `AttributePathIB::kTag_listIndex`
     - `AttributePathIB::kTag_WildcardPathFlags`
   - If the tag is not recognized, the function exits with an error.
4. **Exiting the TLV Container**
   The function exits the TLV container after completing the data reading.
5. **Error Handling**
   The function returns the error code `err`.

4.4.3 Add Attribute Data IB

```
static Matter_ERROR
AddAttributeDataIB(TLVDissector& tlvDissector, proto_tree *tree,
 ↪  tvbuff_t* tvb)
{
    Matter_ERROR err;
    proto_tree *dataElemTree;
    int hf_entry;

    err = tlvDissector.AddSubTreeItem(tree, hf_AttributeDataIB,
     ↪  ett_AttributeDataIB, tvb, dataElemTree);
    SuccessOrExit(err);

    err = tlvDissector.EnterContainer();
    SuccessOrExit(err);

    while (true) {

        err = tlvDissector.Next();
        if (err == Matter_END_OF_TLV)
            break;
        SuccessOrExit(err);

        uint64_t tag = tlvDissector.GetTag();
        //TLVType type = tlvDissector.GetType();
        VerifyOrExit(IsContextTag(tag), err =
         ↪  Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        tag = TagNumFromTag(tag);
        switch (tag) {
            case AttributeDataIB::kTag_DataVersion:
                hf_entry = hf_AttributeDataIB_DataVersion;
                err = tlvDissector.AddTypedItem(dataElemTree,
                 ↪  hf_entry, tvb);
                break;
            case AttributeDataIB::kTag_Path:
                err = AddAttributePathIB(tlvDissector,
                 ↪  dataElemTree, tvb);
            case AttributeDataIB::kTag_Data:
                //err =
                 ↪  tlvDissector.AddGenericTLVItem(dataElemTree,
                 ↪  hf_DataElem_PropertyData, tvb, false);
                err = tlvDissector.AddIMPathItem(dataElemTree,
                 ↪  hf_DataElem_PropertyPath, tvb);
            default:
                ExitNow(err =
                 ↪  Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        }
        SuccessOrExit(err);
    }

    err = tlvDissector.ExitContainer();
    SuccessOrExit(err);

exit:
    return err;
}
```

Function Parameters

- `TLVDissector& tlvDissector`: Reference to an object that manages TLV dissection. This object provides methods to navigate and interpret TLV data structures.
- `proto_tree *tree`: Pointer to the main tree structure where the decoded data will be added. This tree represents the hierarchical organization of the decoded protocol fields.
- `tvbuff_t* tvb`: Pointer to the buffer containing the data to be analyzed. This buffer holds the raw TLV-encoded data that needs to be dissected.

Local Variables

- `Matter_ERROR err`: Variable to handle potential errors. This ensures proper error handling and reporting during the dissection process.
- `proto_tree *dataElemTree`: Pointer to a subtree for data elements. This subtree is used to organize and display individual TLV elements within the main tree.
- `int hf_entry`: Temporary variable to store the current header field. This field identifier is used to associate specific decoded elements with their corresponding tree nodes.

Function Operation

1. **Adding a Subtree**
   The function begins by adding a subtree (`dataElemTree`) to the main tree (`tree`), using `hf_AttributeDataIB` as the header field and `ett_AttributeDataIB` as the entity tree. If an error occurs, the function exits.

2. **Entering the TLV Container**
   The function then enters the TLV container to start reading the data. This sets up the context for interpreting the nested TLV elements.

3. **Iterating Over TLV Tags**
   The function iterates over each TLV element within the container. For each element:

   - It calls `Next()` to read the next TLV element.
   - It verifies that the tag is a context tag, ensuring that the data structure follows the expected format.
   - It converts the tag to get the tag number, which identifies the specific type of data within the TLV structure.
   - Based on the identified tag, it assigns the appropriate header field (`hf_entry`) to map the element to the correct tree node:
     - `AttributeDataIB::kTag_DataVersion` is mapped to `hf_AttributeDataIB_DataVersion`.
     - `AttributeDataIB::kTag_Path` triggers a call to `AddAttributePathIB` to handle nested path data.
     - `AttributeDataIB::kTag_Data` adds a generic TLV item or a specific path item.
   - If the tag is not recognized, the function exits with an error.
   - It adds the element to the subtree based on the identified tag.

4. **Exiting the TLV Container**
   After iterating over all TLV elements, the function exits the TLV container, finalizing the reading process.

5. **Error Handling**
   Finally, the function returns the error code `err`. This ensures that any issues encountered during the dissection process are properly reported.

4.4.4 Add Attribute Status IB

```
static Matter_ERROR
AddAttributeStatusIB(TLVDissector& tlvDissector, proto_tree
↪  *tree, tvbuff_t* tvb)
{
    Matter_ERROR err;
    proto_tree *dataElemTree;
    //int hf_entry;

    err = tlvDissector.AddSubTreeItem(tree,
     ↪  hf_AttributeReportIB_AttributeStatus,
     ↪  ett_AttributeStatusIB, tvb, dataElemTree);
    SuccessOrExit(err);

    err = tlvDissector.EnterContainer();
    SuccessOrExit(err);

    while (true) {

        err = tlvDissector.Next();
        if (err == Matter_END_OF_TLV)
            break;
        SuccessOrExit(err);

        uint64_t tag = tlvDissector.GetTag();
        //TLVType type = tlvDissector.GetType();
        VerifyOrExit(IsContextTag(tag), err =
         ↪  Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        tag = TagNumFromTag(tag);
        switch (tag) {
            case AttributeStatusIB::kTag_Path:
                err = AddAttributeDataIB(tlvDissector,
                 ↪  dataElemTree, tvb);
                break;
            case AttributeStatusIB::kTag_Status:
                err = AddStatusIB(tlvDissector, dataElemTree,
                 ↪  tvb);
                break;
            default:
                ExitNow(err =
                 ↪  Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        }
        SuccessOrExit(err);
    }

    err = tlvDissector.ExitContainer();
    SuccessOrExit(err);

exit:
    return err;
}
```

Function Purpose and Operation

The main objective of the `AddAttributeStatusIB` function is to decode TLV elements within a provided data buffer (`tvbuff_t`) and represent them within a hierarchical tree structure (`proto_tree`). This structured representation aids in the analysis and debugging of network protocols by presenting data in a clear and organized manner.

Function Parameters

- `TLVDissector& tlvDissector`: Reference to an object managing TLV dissection. Provides methods to navigate and interpret TLV data structures.
- `proto_tree *tree`: Pointer to the main tree structure where decoded data is added. Represents the hierarchical organization of protocol fields.
- `tvbuff_t* tvb`: Pointer to the buffer containing data to be analyzed. Holds raw TLV-encoded data for dissection.

Local Variables

- `Matter_ERROR err`: Variable for handling potential errors. Ensures proper error reporting during the dissection process.
- `proto_tree *dataElemTree`: Pointer to a subtree for data elements. Organizes individual TLV elements within the main tree.

Function Operation

1. **Adding a Subtree**
   The function starts by adding a subtree (`dataElemTree`) to the main tree (`tree`), using `hf_AttributeReportIB_AttributeStatus` as the header field and `ett_AttributeStatusIB` as the entity tree. If an error occurs, the function exits.

2. **Entering the TLV Container**
   Next, the function enters the TLV container to begin reading the data. This prepares for interpreting nested TLV elements.

3. **Iterating Over TLV Tags**
   The function iterates through each TLV element within the container. For each element:

   - It calls `Next()` to read the next TLV element.
   - Verifies that the tag is a context tag, ensuring the data structure conforms to expectations.
   - Converts the tag to get the tag number, identifying the specific type of data within the TLV structure.
   - Based on the tag, performs the following actions:
     - `AttributeStatusIB::kTag_Path`:
       Calls `AddAttributeDataIB` to handle nested data paths.
     - `AttributeStatusIB::kTag_Status`:
       Calls `AddStatusIB` to handle status information.
     - If the tag is unrecognized, the function exits with an error.
   - It adds the decoded element to the subtree (`dataElemTree`) based on the identified tag.

4. **Exiting the TLV Container**
   After processing all TLV elements, the function exits the TLV container, completing the data reading process.

5. **Error Handling**
   Finally, the function returns the error code `err`. This ensures any issues encountered during dissection are appropriately reported.

4.4.5 Add Attribute Report IB

```
static Matter_ERROR
AddAttributeReportIB(TLVDissector& tlvDissector, proto_tree *tree, tvbuff_t* tvb)
{
    Matter_ERROR err;
    proto_tree *dataElemTree;
    //int hf_entry;

    err = tlvDissector.AddSubTreeItem(tree, hf_AttributeReportIB, ett_DataElem, tvb, dataElemTree);
    SuccessOrExit(err);

    err = tlvDissector.EnterContainer();
    SuccessOrExit(err);

    while (true) {

        err = tlvDissector.Next();
        if (err == Matter_END_OF_TLV)
            break;
        SuccessOrExit(err);

        uint64_t tag = tlvDissector.GetTag();
        //TLVType type = tlvDissector.GetType();
        VerifyOrExit(IsContextTag(tag), err = Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        tag = TagNumFromTag(tag);
        switch (tag) {
            case AttributeReportIB::kTag_AttributeStatus:
                err = AddAttributeStatusIB(tlvDissector, dataElemTree, tvb);
                break;
            case AttributeReportIB::kTag_AttributeData:
                err = AddAttributeDataIB(tlvDissector, dataElemTree, tvb);
                break;
            default:
                ExitNow(err = Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
        }
        SuccessOrExit(err);
    }

    err = tlvDissector.ExitContainer();
    SuccessOrExit(err);

exit:
    return err;
}
...
```

Function Parameters

- `TLVDissector& tlvDissector`: Reference to an object managing TLV dissection. Provides methods to navigate and interpret TLV data structures.
- `proto_tree *tree`: Pointer to the main tree structure where decoded data is added. Represents the hierarchical organization of protocol fields.
- `tvbuff_t* tvb`: Pointer to the buffer containing data to be analyzed. Contains the raw TLV-encoded data for dissection.

Local Variables

- `Matter_ERROR err`: Variable for handling potential errors. Ensures proper error reporting during the dissection process.
- `proto_tree *dataElemTree`: Pointer to a subtree for data elements. Organizes individual TLV elements within the main tree.

Function Operation

1. **Adding a Subtree**
   The function begins by adding a subtree (`dataElemTree`) to the main tree (`tree`), using `hf_AttributeReportIB` as the header field and `ett_DataElem` as the entity tree. If an error occurs, the function exits.
2. **Entering the TLV Container**
   The function enters the TLV container to start reading the data. This prepares for interpreting nested TLV elements.

3. **Iterating Over TLV Tags**
   The function iterates through each TLV element within the container. For each element:

   - It calls `Next()` to read the next TLV element.
   - Verifies that the tag is a context tag, ensuring the data structure conforms to expectations.
   - Converts the tag to get the tag number, identifying the specific type of data within the TLV structure.
   - Based on the tag, performs the following actions:
     - `AttributeReportIB::kTag_AttributeStatus`: Calls `AddAttributeStatusIB` to handle nested attribute status data.
     - `AttributeReportIB::kTag_AttributeData`: Calls `AddAttributeDataIB` to handle nested attribute data.
     - If the tag is unrecognized, the function exits with an error.
   - It adds the decoded element to the subtree (`dataElemTree`) based on the identified tag.

4. **Exiting the TLV Container**
   After processing all TLV elements, the function exits the TLV container, completing the data reading process.

4.4.6 Dissect IM Read Request

```
static int
DissectIMReadRequest(tvbuff_t *tvb, packet_info *pinfo,
 ↪  proto_tree *tree _U_, const MatterMessageInfo& msgInfo)
{
        ...
        TLVType type = tlvDissector.GetType();
        ...
        tag = TagNumFromTag(tag);
        switch (tag) {
            case ReadRequest::kTag_AttributeRequests:
                hf_entry = hf_ReadRequest_AttributeRequests;
                VerifyOrExit(type == kTLVType_Array, err =
                 ↪  Matter_ERROR_UNEXPECTED_TLV_ELEMENT);
                err = tlvDissector.AddListItem(tree,
                 ↪  hf_ReadRequest_AttributeRequests,
                 ↪  ett_ReadRequest_AttributeRequests, tvb,
                 ↪  AddAttributePathIB);
                SuccessOrExit(err);
                continue;
...
```

In the DissectIMReadRequest function, the unique addition is `TLVType type = tlvDissector.GetType();` To store what types are inside *tlvdissector*.

The provided code snippet is part of a function responsible for dissecting and interpreting TLV (Tag-Length-Value) encoded data within a network packet buffer, likely in the context of the Matter (formerly Thread) protocol.

Tag Extraction and Processing

1. **Tag Extraction**:
   The variable `tag` is assigned the result of `TagNumFromTag(tag)`, which extracts the numerical tag value from the full tag representation. This numerical tag is pivotal in determining the type of TLV data encountered.

2. **Switch-Case Handling**:
   The function utilizes a switch-case statement based on the extracted `tag` value:

   - `case ReadRequest::kTag_AttributeRequests`:
     If the tag corresponds to `ReadRequest::kTag_AttributeRequests`, it sets `hf_entry` to `hf_ReadRequest_AttributeRequests`. Subsequently, it verifies that the TLV type (`type`) is `kTLVType_Array`. If this condition fails, it sets `err` to `Matter_ERROR_UNEXPECTED_TLV_ELEMENT`.
   - `continue;`:
     After successfully verifying the TLV type, the function continues to the next iteration of the loop to process additional TLV elements.

3. **TLV Dissection and Tree Construction**:
   Within each case statement:

   - It calls `tlvDissector.AddListItem` to add a list item to the protocol tree (`proto_tree *tree`) using

hf_ReadRequest_AttributeRequests.
The function AddAttributePathIB is likely responsible for further dissecting and adding details to this tree structure.

- SuccessOrExit(err);:
  This macro ensures that if an error (err) occurs during any operation, the function exits gracefully, handling the error condition appropriately.

Purpose
This code segment serves the crucial role of:

- Identifying specific TLV elements based on their tags within the packet buffer.
- Validating that the type of each TLV element meets expected criteria (kTLVType_Array in this case).
- Constructing and populating a hierarchical protocol tree (proto_tree) with dissected TLV data, facilitating structured analysis and debugging of network communication.

4.4.7 Dissect IM Report Data

```
static int
DissectIMReportData(tvbuff_t *tvb, packet_info *pinfo,
↪  proto_tree *tree _U_, const MatterMessageInfo& msgInfo)
{
        ...
        TLVType type = tlvDissector.GetType();

            case ReportData::kTag_AttributeReports:
                err = tlvDissector.AddGenericTLVItem(tree,
                 ↪  hf_entry, tvb, false);
                break;

                ..
                VerifyOrExit(type == kTLVType_Array, err =
                ...
                err = tlvDissector.AddListItem(tree, hf_entry,
                 ↪  ett_DataElem, tvb, AddAttributeReportIB);
                ...

            case ReportData::kTag_EventReports:
                hf_entry = hf_ReportData_EventReports;
                err = tlvDissector.AddGenericTLVItem(tree,
                 ↪  hf_entry, tvb, false);
                break;

            case ReportData::kTag_MoreChunkedMessages:
                ...
                err = tlvDissector.AddGenericTLVItem(tree,
                 ↪  hf_entry, tvb, false);
                break;

            case ReportData::kTag_SuppressResponse:
                hf_entry = hf_ReportData_SuppressResponse;
                err = tlvDissector.AddGenericTLVItem(tree,
                 ↪  hf_entry, tvb, false);
                break;

            case
             ↪  CommonActionInfo::kTag_InteractionModelRevision:
                hf_entry = hf_ImCommon_Version;
                err = tlvDissector.AddGenericTLVItem(tree,
                 ↪  hf_entry, tvb, false);
                break;

            default:
                ...
                err = tlvDissector.AddGenericTLVItem(tree,
                 ↪  hf_entry, tvb, false);
                ...
        ...
```

Also here there are some cases, that permit the print of the right parameters.

4.4.8 Proto register Matter im

```
// ===== Read Attribute Request ====
        {
            &hf_ReadAttributeRequest_enableTagCompression,
            { "Enable Tag Compression",
             ↪  "im.read_attr_req.enable_tag_compression",
            FT_STRING, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },
        {
            &hf_ReadAttributeRequest_node,
            { "Node", "im.read_attr_req.node",
            FT_STRING, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },
        {
            &hf_ReadAttributeRequest_endpoint,
            { "Endpoint", "im.read_attr_req.endpoint",
            FT_STRING, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },
        {
            &hf_ReadAttributeRequest_cluster,
            { "Cluster", "im.read_attr_req.cluster",
            FT_STRING, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },
        {
            &hf_ReadAttributeRequest_attribute,
            { "Attribute", "im.read_attr_req.attribute",
            FT_STRING, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },
        {
            &hf_ReadAttributeRequest_listIndex,
            { "List Index", "im.read_attr_req.list_index",
            FT_STRING, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },
        {
            &hf_ReadAttributeRequest_WildcardPathFlags,
            { "Wildcard Path Flags",
             ↪  "im.read_attr_req.wildcard_path_flags",
            FT_STRING, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },

        // ===== AttributeReport =====
        {
            &hf_AttributeReportIB_AttributeStatus,
            { "AttributeStatusIB",
             ↪  "im.attribute_report.attribute_status",
            FT_STRING, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },
        {
            &hf_AttributeReportIB_AttributeData,
            { "AttributeDataIB",
             ↪  "im.attribute_report.attribute_data",
            FT_STRING, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },

        .....
        { &hf_StatusIB,
            { "StatusIB", "im.struct.CommandStatusIB",
            FT_NONE, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },
        {
            &hf_StatusIB_Status,
            { "Status", "im.struct.statusIB",
            FT_UINT16, BASE_DEC, NULL, 0x0, NULL, HFILL }

        },
        {
            &hf_StatusIB_ClusterStatus,
            { "Cluster Status", "im.struct.clusterStatus",
            FT_UINT16, BASE_DEC, NULL, 0x0, NULL, HFILL }

        },
        ......
        { &hf_AttributeDataIB,
            { "AttributeDataIB", "im.struct.AttributeDataIB",
            FT_NONE, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },
        { &hf_AttributePathIB,
            { "AttributePathIB", "im.struct.AttributePathIB",
            FT_NONE, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },
        { &hf_AttributeDataIB_DataVersion,
            { "DataVersion",
            ↪  "im.struct.AttributeDataIB.DataVersion",
            FT_UINT32, BASE_DEC, NULL, 0x0, NULL, HFILL }
        },
        { &hf_AttributeReportIB,
            { "AttributeReportIB",
            ↪  "im.struct.AttributeReportIB",
            FT_NONE, BASE_NONE, NULL, 0x0, NULL, HFILL }
        },

```

Structure

- Each field is defined using a pair consisting of a field identifier (&hf_...) and a table describing the field's properties.
- The tables contain information such as:
  - The field name (displayed in Wireshark)
  - Protocol name (used internally)
  - Field type (`FT_STRING`, `FT_UINT16`, `FT_UINT32`, etc.)
  - Base type (`BASE_DEC` indicates decimal representation)
  - Other properties like size and display format (`HFILL`)

Field Types

- `FT_STRING`: Indicates the field contains a string value.
- `FT_UINT16`, `FT_UINT32`: Indicates unsigned integer types of 16-bit and 32-bit sizes respectively that are explained better in the specific.
- `FT_NONE`: Indicates the field does not have a specific type or is not displayed.

Usage

These definitions allow Wireshark to parse and display packet data according to the defined fields. For example, when dissecting a packet, Wireshark can use these definitions to interpret and display values like node identifiers, endpoint numbers, attribute statuses, etc.

Organization

Fields are organized under different sections (**Read Attribute Request**, **AttributeReport**, **StatusIB**, etc.), likely corresponding to different parts of the protocol being analyzed.

Additionally, this code enables the association of a string that provides semantic meaning to a number within the Wireshark graphical user interface (GUI).

4.4.9 Expanded tree type defines

```
...
&ett_ReadRequest_AttributeRequests,
...
&ett_AttributeStatusIB,
&ett_AttributeDataIB
...
```

This code defines an array of integer pointers (`gint *ett[]`).
Which have been initialized previously

Chapter 5

Demonstration

This final chapter, as mentioned in the introduction 1, will be the one that will demonstrate the project created to extend the functionality of this Wireshark plugin for the interaction model. The demonstration will be conducted using the tools provided by the official software, and one of the simplest available clusters, implemented in both software. The cluster in question is the on/off cluster.

5.1 Setup Environment

5.1.1 Chip-Tool

To perform the tests, the official SDK of the Matter protocol was chosen. The *Connectedhomeip* GitHub repository contains a file called `chip-tool` within the *examples* subfolder. This is merely the implementation of the Matter protocol. In more detail, it can be stated that this implementation of the Matter controller enables the commissioning of a Matter device into the network and the communication with it using Matter messages, which may encode Data Model actions, such as cluster commands. In addition, the tool offers other utilities that are specific to Matter. These include parsing the setup payload and performing discovery actions. The item in question can be located in the *examples/chip-tool* directory.

5.1.1.1 Build And Run Chip Tool

To build and run the CHIP Tool:

1. Install all required packages for Matter and prepare the source code and the build system. Read the *Building Matter* guide for instructions.

2. Open a command prompt in the `connectedhomeip` directory.
3. Run the following command:

```
$./scripts/examples/gn_build_example.sh examples/chip-tool PATH
```

In this command, *PATH* specifies where the target binaries are to be placed.

5.1.1.2 Run the Chip-Tool

In order to ascertain the functionality of the Chip-Tool, it is necessary to execute the following command from the *PATH* directory:

```
$ ./chip-tool
```

Consequently, the Chip-Tool initiates operation in the default single-command mode, whereby all available commands are printed. In this context, these are referred to as **clusters**, although it should be noted that not all the commands listed correspond to the clusters in the data model. For example, the pairing and discover commands are not included in this list. However, it should be noted that each listed command can become the root of a more complex command by adding it with subcommands. A detailed description of the specific commands and their applications can be found in the section titled *Supported Commands and Options*.
The requisite context for the implementation of a process that allows for multiple tests to be executed in sequence without necessitating a system reboot is that of an *interactive mode*.

5.1.1.3 Starting the interactive mode

In this operational mode, a command will terminate with an error if it is not completed within the specified time-out period. However, the Chip-Tool will not terminate and will not terminate the processes that the previous commands have initiated. Furthermore, in the context of the interactive mode, the CHIP-Tool will establish a new *CASE* session only when there is no existing session available. In the case of the following commands, the existing session will be utilized.

```
$ ./chip-tool interactive start
```

In this mode, you can subscribe to events or attributes.
The official documentation contains a section entitled: *"Utilizing the CHIP Tool for Matter Device Testing."* In the context of the present paper, it is not necessary to perform the initial five steps. In point of fact, should one already possess a Matter device, it is advisable to proceed directly to the sixth item, which pertains to the commissioning process. Before initiating communication with the Matter device, it is necessary for the device to first join an existing IP network.
Matter devices can utilize a variety of commissioning channels.

- If a device is not yet connected to the target IP network, Bluetooth LE is used as the commissioning channel.
- Once a device has already joined an IP network, it is only necessary for it to use the IP protocol to commission to the Matter network.

5.1.1.4 Commissioning over Bluetooth LE

In this instance, the device in question can be integrated into an existing IP network via Bluetooth LE (Lower Energy), after which it can be commissioned into a Matter network. A variety of scenarios are available for Thread and Wi-Fi networks, as described in the following subsections. Upon establishing a connection between the device and the controller via Bluetooth LE, the latter will print the following log:

```
Secure Session to Device Established
```

This log message indicates that a PASE (Password-Authenticated Session Establishment) session has been established using the SPAKE2+ protocol. Subsequently, following the device's initial commissioning, the user may choose the method of connection to the device in order to access the functions within the network. The most common method of connection is through a Wi-Fi network, which is established through Bluetooth LE. To commission the device to the existing Wi-Fi network, use the following command pattern:

```
$ ./chip-tool pairing ble-wifi <node_id> <ssid> <password>
 ↪  <pin_code> <discriminator>
```

In this command:

- `<node_id>` is the user-defined ID of the node being commissioned.

- `<ssid>` and `<password>` are credentials determined in the step
- `<pin_code>` and `<discriminator>` are device-specific keys determined in the previous step.

5.1.1.5 Commissioning into a network over IP

This option is available when the Matter device is already present in an IP network, but has not been commissioned to a Matter network yet. Should one utilize the implementation of the simulated device that is already integrated within the network, one may employ the command:

```
$ ./chip-tool pairing code <node_id>
↪  <qrcode_payload-or-manual_code>
```

The command keeps trying devices until pairing with one of them succeeds or until it runs out of pairing possibilities. In this command:

- `<node_id>` is the user-defined ID of the node being commissioned.
- `<qr-code_payload-or-manual_code>` is the QR code payload ID, for example, MT:Y3.12OTB00KC0628G00.

5.1.1.6 Parsing the setup payload

The CHIP Tool provides a convenient utility for parsing the Matter onboarding setup payload and presenting it in a readable format. This is particularly useful when one wishes to display the payload on the device console during the boot process. In order to effectively parse the setup code, it is necessary to utilize the payload command in conjunction with the `parse-setup-payload` subcommand. The following command structure is designed to extract device data, including the PIN code and discriminator:

```
$ ./chip-tool payload parse-setup-payload <payload>
```

Nevertheless, there are numerous other commands that are not the subject of study but which can be found in the documentation [1].

5.1.2 All-cluster-app

To facilitate the ease of use and execution of various tests due to encryption in the physical device, a tool was utilized within the connected-home-ip

repository. The tool, named `all-cluster-app`, is specifically designed to support testing across multiple clusters rather than a single device.
One of the applications of `all-cluster-app` is to streamline testing by allowing the evaluation of various clusters on the same platform. This capability is particularly valuable for developers and testers responsible for ensuring the interoperability and performance of multiple clusters within a connected home environment. The tool enables comprehensive testing to validate the reliability and functionality of different components, which is crucial for effective testing procedures.

```
$ cd connectedhomeip/examples/all-clusters-app/linux
```

Once this directory has been entered from the terminal, the following command can be used to download the tool:

```
$ ./scripts/run_in_build_env.sh
↪  `./scripts/build/build_examples.py --target
↪  linux-x64-all-clusters-no-ble-asan-clang build`
```

This command initiates the build process for the `all-cluster-app`, creating the necessary binaries and dependencies required for its execution. The `build_example.py` script simplifies the setup by automating the configuration and compilation steps, ensuring that the tool is ready to test a wide array of clusters within the connected home-ip ecosystem.
To make this tool work, it must be used in conjunction with the CHIP tool. To run the all-cluster-app, the following command should be used:

```
$ cd /connectedhomeip/out/all-cluster-app/out/
$ ./all-cluster-app/chip-all-cluster-app
```

Finally, to connect the CHIP tool to the all-cluster-app, the following command should be used while the CHIP tool is in interactive mode:

```
$ pairing onnetwork <Node Id> 20202021
```

This command establishes a connection between the CHIP tool and `all-cluster-app`, enabling interaction with all clusters for testing within the connected-home-ip environment. It is crucial to specify these parameters within the command to ensure effective communication between devices, facilitating comprehensive testing and validation.

5.1.3 Example Read interaction

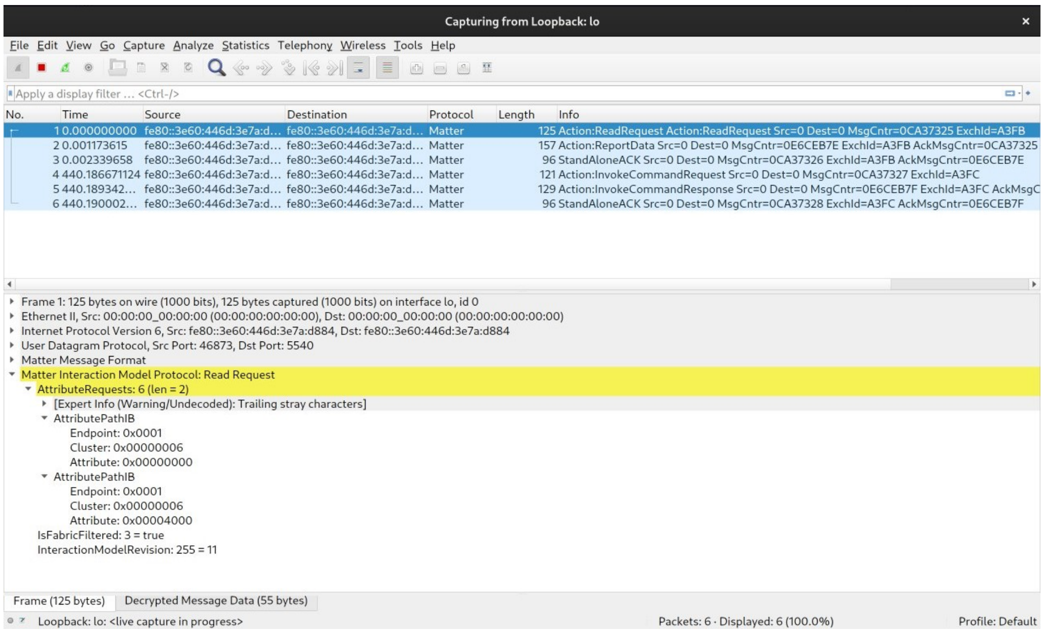
To assess the functionality of the system, a specific command will be employed that enables the simultaneous evaluation of multiple features.

```
    $ onoff read-by-id 0x0000,0x4000 11 1
```

This is a command to test read by id, which is part of the reading interaction:

- **onoff:** is the cluster name
- **read-by-id:** command name
- **0x0000,0x4000:**Attributes on which the request is made
  - OnOff
  - GlobalSceneControl
- **“11”:**Node
- **“1”:**Endpoint

5.1.4 Read Request



Capturing from Loopback: lo

```
No.  Time          Source                 Destination            Protocol  Length  Info
  1  0.000000000   fe80::3e60:446d:3e7a:d…  fe80::3e60:446d:3e7a:d…  Matter    125 Action:ReadRequest Action:ReadRequest Src=0 Dest=0 MsgCntr=0CA37325 ExchId=A3FB
  2  0.001173615   fe80::3e60:446d:3e7a:d…  fe80::3e60:446d:3e7a:d…  Matter    157 Action:ReportData Src=0 Dest=0 MsgCntr=0E6CEB7E ExchId=A3FB AckMsgCntr=0CA37325
  3  0.002339658   fe80::3e60:446d:3e7a:d…  fe80::3e60:446d:3e7a:d…  Matter     96 StandAloneACK Src=0 Dest=0 MsgCntr=0CA37326 ExchId=A3FB AckMsgCntr=0E6CEB7E
  4  440.186671124 fe80::3e60:446d:3e7a:d…  fe80::3e60:446d:3e7a:d…  Matter    121 Action:InvokeCommandRequest Src=0 Dest=0 MsgCntr=0CA37327 ExchId=A3FC
  5  440.189342…   fe80::3e60:446d:3e7a:d…  fe80::3e60:446d:3e7a:d…  Matter    129 Action:InvokeCommandResponse Src=0 Dest=0 MsgCntr=0E6CEB7F ExchId=A3FC AckMsgC
  6  440.190002…   fe80::3e60:446d:3e7a:d…  fe80::3e60:446d:3e7a:d…  Matter     96 StandAloneACK Src=0 Dest=0 MsgCntr=0CA37328 ExchId=A3FC AckMsgCntr=0E6CEB7F

▸ Frame 1: 125 bytes on wire (1000 bits), 125 bytes captured (1000 bits) on interface lo, id 0
▸ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▸ Internet Protocol Version 6, Src: fe80::3e60:446d:3e7a:d884, Dst: fe80::3e60:446d:3e7a:d884
▸ User Datagram Protocol, Src Port: 46873, Dst Port: 5540
▸ Matter Message Format
▾ Matter Interaction Model Protocol: Read Request
  ▾ AttributeRequests: 6 (len = 2)
    ▸ [Expert Info (Warning/Undecoded): Trailing stray characters]
    ▾ AttributePathIB
        Endpoint: 0x0001
        Cluster: 0x00000006
        Attribute: 0x00000000
    ▾ AttributePathIB
        Endpoint: 0x0001
        Cluster: 0x00000006
        Attribute: 0x00004000
    IsFabricFiltered: 3 = true
    InteractionModelRevision: 255 = 11

Frame (125 bytes)   Decrypted Message Data (55 bytes)
Loopback: lo: <live capture in progress>        Packets: 6 · Displayed: 6 (100.0%)        Profile: Default
```

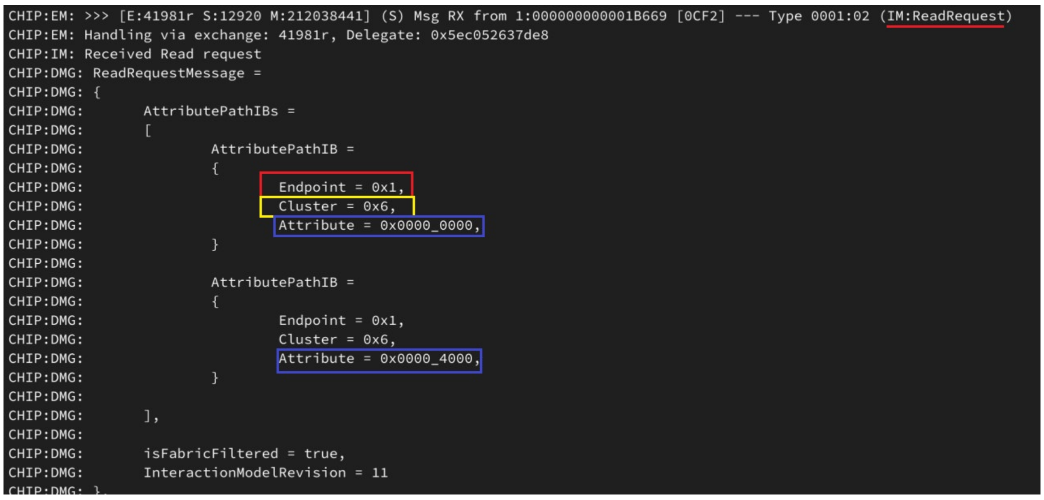
Figure 5.1: Read request

The Read Request action represents the initial action of a Read transaction (and interaction).

Action Field	**Type**	**Description**
AttributeRequests	list[AttributePathIB]	list of zero or more request paths to cluster attribute data
DataVersionFilters	list[DataVersionFilterIB]	list of zero or more cluster instance data versions
EventRequests	list[EventPathIB]	list of zero or more request paths to cluster events
EventFilters	list[EventFilterIB]	list of zero or more minimum event numbers per specific node
FabricFiltered	bool	limits data read within fabric-scoped lists to the accessing fabric

Table 5.1: Table of Read Request Action Information [20]

The table provides an illustrative overview of the data that will be presented in the Wireshark output. As illustrated in the figure 5.1 on the left side of the equation, it is possible to discern the designation of the corresponding value. In the tree of AttributeRequest, all the information that is anticipated is present. In this example, it is evident that in the subtree, called AttributePathIB, the value hex '0x0001' corresponds to the endpoint '1', as previously mentioned in the issued command. The difference between the two attributes is also evident.
Further visual evidence can be provided using the chip instrument, which summarizes the same information. It should be noted, however, that this tool has been designed for use with specific internal devices and is therefore not applicable to external devices, as it was originally developed for use with the dissector.

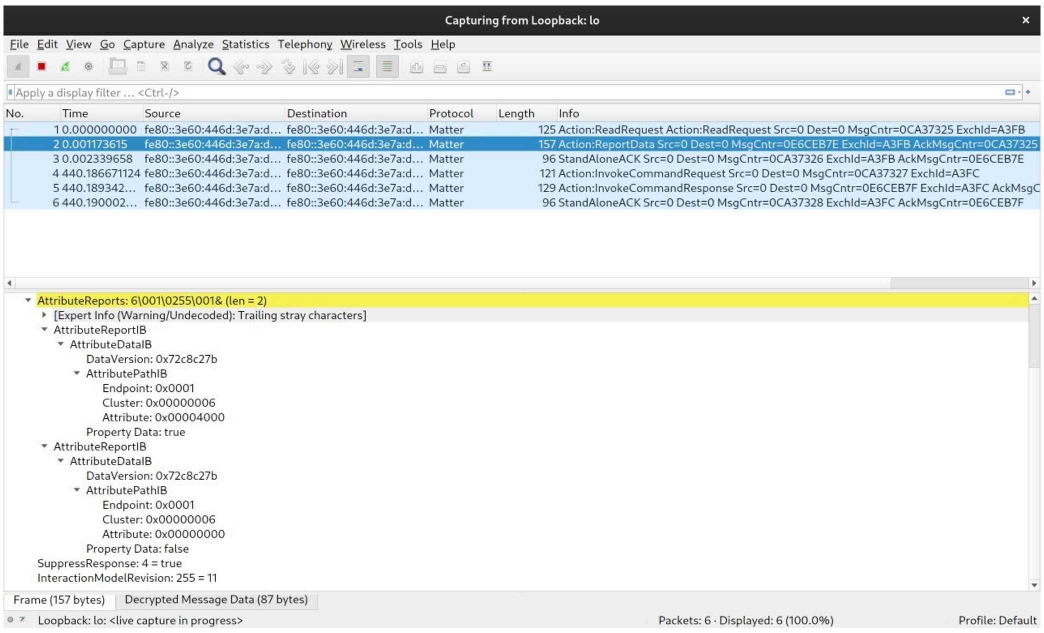


```
CHIP:EM: >>> [E:41981r S:12920 M:212038441] (S) Msg RX from 1:000000000001B669 [0CF2] --- Type 0001:02 (IM:ReadRequest)
CHIP:EM: Handling via exchange: 41981r, Delegate: 0x5ec052637de8
CHIP:IM: Received Read request
CHIP:DMG: ReadRequestMessage =
CHIP:DMG: {
CHIP:DMG:       AttributePathIBs =
CHIP:DMG:       [
CHIP:DMG:               AttributePathIB =
CHIP:DMG:               {
CHIP:DMG:                       Endpoint = 0x1,
CHIP:DMG:                       Cluster = 0x6,
CHIP:DMG:                       Attribute = 0x0000_0000,
CHIP:DMG:               }
CHIP:DMG:
CHIP:DMG:               AttributePathIB =
CHIP:DMG:               {
CHIP:DMG:                       Endpoint = 0x1,
CHIP:DMG:                       Cluster = 0x6,
CHIP:DMG:                       Attribute = 0x0000_4000,
CHIP:DMG:               }
CHIP:DMG:
CHIP:DMG:       ],
CHIP:DMG:
CHIP:DMG:       isFabricFiltered = true,
CHIP:DMG:       InteractionModelRevision = 11
CHIP:DMG: },
```

Figure 5.2: Chip-tool command executed

As illustrated in the figure displayed above, the red rectangle shows the same number as in the Wireshark GUI. The same applies to the other highlighted values.
This is because the format is the type we expect and coincides perfectly with what is requested. In order to obtain further information regarding the node, it is necessary to read the relevant attribute.

5.1.4.1 Report Data Action


Capturing from Loopback: lo
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

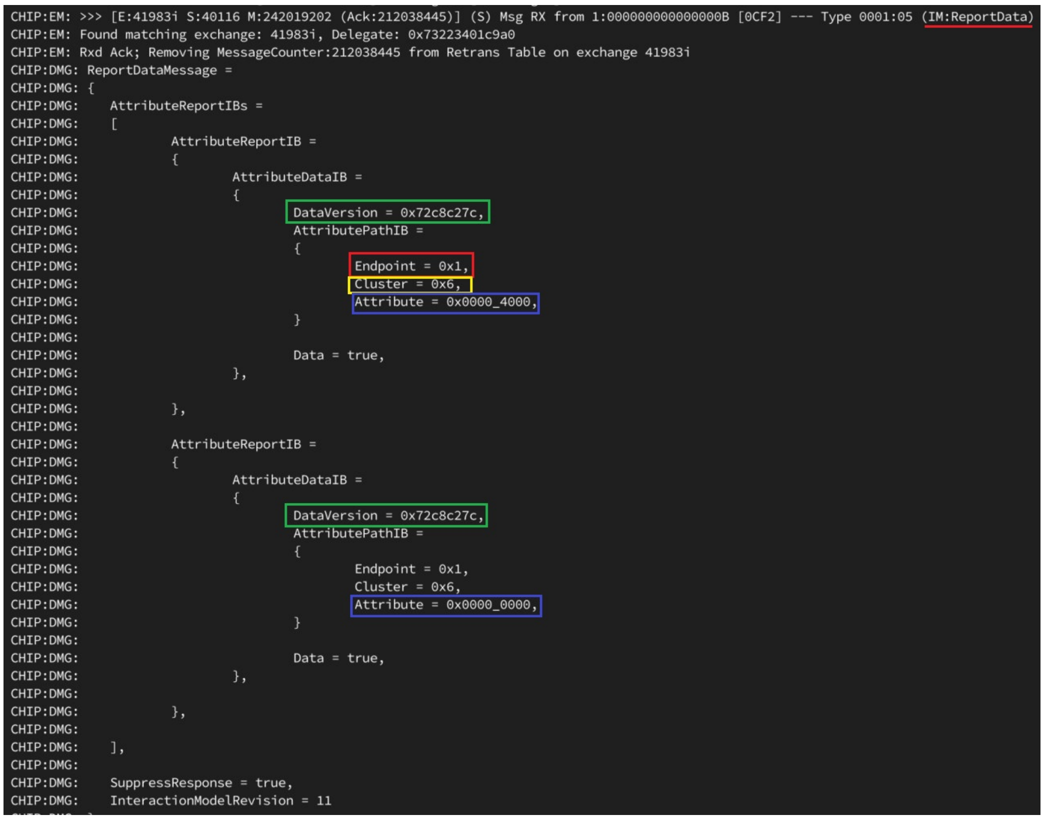
```
No.  Time          Source                 Destination            Protocol  Length  Info
  1 0.000000000    fe80::3e60:446d:3e7a:d… fe80::3e60:446d:3e7a:d… Matter   125 Action:ReadRequest Action:ReadRequest Src=0 Dest=0 MsgCntr=0CA37325 ExchId=A3FB
  2 0.001173615    fe80::3e60:446d:3e7a:d… fe80::3e60:446d:3e7a:d… Matter   157 Action:ReportData Src=0 Dest=0 MsgCntr=0E6CEB7E ExchId=A3FB AckMsgCntr=0CA37325
  3 0.002339658    fe80::3e60:446d:3e7a:d… fe80::3e60:446d:3e7a:d… Matter    96 StandAloneACK Src=0 Dest=0 MsgCntr=0CA37326 ExchId=A3FB AckMsgCntr=0E6CEB7E
  4 440.186671124  fe80::3e60:446d:3e7a:d… fe80::3e60:446d:3e7a:d… Matter   121 Action:InvokeCommandRequest Src=0 Dest=0 MsgCntr=0CA37327 ExchId=A3FC
  5 440.189342…    fe80::3e60:446d:3e7a:d… fe80::3e60:446d:3e7a:d… Matter   129 Action:InvokeCommandResponse Src=0 Dest=0 MsgCntr=0E6CEB7F ExchId=A3FC AckMsgC
  6 440.190002…    fe80::3e60:446d:3e7a:d… fe80::3e60:446d:3e7a:d… Matter    96 StandAloneACK Src=0 Dest=0 MsgCntr=0CA37328 ExchId=A3FC AckMsgCntr=0E6CEB7F

AttributeReports: 6\001\0255\001& (len = 2)
  [Expert Info (Warning/Undecoded): Trailing stray characters]
  AttributeReportIB
    AttributeDataIB
      DataVersion: 0x72c8c27b
      AttributePathIB
        Endpoint: 0x0001
        Cluster: 0x00000006
        Attribute: 0x00004000
      Property Data: true
  AttributeReportIB
    AttributeDataIB
      DataVersion: 0x72c8c27b
      AttributePathIB
        Endpoint: 0x0001
        Cluster: 0x00000006
        Attribute: 0x00000000
      Property Data: false
SuppressResponse: 4 = true
InteractionModelRevision: 255 = 11

Frame (157 bytes)   Decrypted Message Data (87 bytes)
Loopback: lo: <live capture in progress>        Packets: 6 · Displayed: 6 (100.0%)        Profile: Default
```

Figure 5.3: Chip-tool command executed

This action represents either the initial action within a report transaction (as part of a subscribe interaction) or a subsequent action following a read request or subscribe request action.
The data report action like the read request, presents the semantic interpretation of the value on the left side of the equal sign. It should be noted that all values and semantics presented here have been previously defined in the section on the explanation of the code 4. The report Data action is the response from the node, that communicates with the client.



```
CHIP:EM: >>> [E:41983i S:40116 M:242019202 (Ack:212038445)] (S) Msg RX from 1:000000000000000B [0CF2] --- Type 0001:05 (IM:ReportData)
CHIP:EM: Found matching exchange: 41983i, Delegate: 0x73223401c9a0
CHIP:EM: Rxd Ack; Removing MessageCounter:212038445 from Retrans Table on exchange 41983i
CHIP:DMG: ReportDataMessage =
CHIP:DMG: {
CHIP:DMG:     AttributeReportIBs =
CHIP:DMG:     [
CHIP:DMG:             AttributeReportIB =
CHIP:DMG:             {
CHIP:DMG:                     AttributeDataIB =
CHIP:DMG:                     {
CHIP:DMG:                             DataVersion = 0x72c8c27c,
CHIP:DMG:                             AttributePathIB =
CHIP:DMG:                             {
CHIP:DMG:                                     Endpoint = 0x1,
CHIP:DMG:                                     Cluster = 0x6,
CHIP:DMG:                                     Attribute = 0x0000_4000,
CHIP:DMG:                             }
CHIP:DMG:
CHIP:DMG:                             Data = true,
CHIP:DMG:                     },
CHIP:DMG:
CHIP:DMG:             },
CHIP:DMG:
CHIP:DMG:             AttributeReportIB =
CHIP:DMG:             {
CHIP:DMG:                     AttributeDataIB =
CHIP:DMG:                     {
CHIP:DMG:                             DataVersion = 0x72c8c27c,
CHIP:DMG:                             AttributePathIB =
CHIP:DMG:                             {
CHIP:DMG:                                     Endpoint = 0x1,
CHIP:DMG:                                     Cluster = 0x6,
CHIP:DMG:                                     Attribute = 0x0000_0000,
CHIP:DMG:                             }
CHIP:DMG:
CHIP:DMG:                             Data = true,
CHIP:DMG:                     },
CHIP:DMG:
CHIP:DMG:             },
CHIP:DMG:
CHIP:DMG:     ],
CHIP:DMG:
CHIP:DMG:     SuppressResponse = true,
CHIP:DMG:     InteractionModelRevision = 11
```

Figure 5.4: Chip-tool command executed

Also in this figure relating to the chip-tool it is possible to notice how the fields coincide exactly with what is shown on Wireshark. The unique field different from the read request is: **Data version**.
This field is used to identify instances where data within a cluster has undergone a change. Upon each update of a cluster's data, the value of the "Data Version" is incremented. This enables clients to ascertain whether data has been modified without having to peruse the entire data set, thereby enhancing communication efficiency. The "Data Version" feature enables clients to synchronize data between the device and their own system. In the event of a discrepancy between the "Data Version" of the device and that which is currently stored on the client, it can be inferred that modifications have been made which necessitate an update on the client's end. This is particularly advantageous in scenarios where the client is required to maintain a local copy of the data for offline operations or to reduce latency. Instead, the suppress Response field is a boolean used to indicate a negative response to the proposition of receiving a reply.

Conclusion

The ultimate goal of this project was to provide semantics for the numerical values represented in the Wireshark tool about the Matter protocol interaction model. To accomplish this goal, alterations were made to the dissector code, thereby enabling the definition of the requisite semantics following the protocol specification certified by the Connectivity Standards Alliance (CSA), accessible on the official website.

During the development process, the Matter software development kit (SDK) provided the necessary tools. The preliminary hypothesis was that an external device, such as a smart plug, would be a suitable means of testing the dissector. However, this approach proved to be ineffective, necessitating the use of the chip tool and the all-cluster app. The complexity of Matter, characterized by multiple layers of security, makes it extremely challenging to trace the key used to encrypt packets and establish a connection, even with modern tools. Indeed, it should be noted that Matter employs a hashing algorithm (SHA-256 Message Authentication HMAC-SHA-256), a public key (ECC Curve NIST P-256), and a message encryption method (AES-CCM with 128-bit keys).

Essentially, every device operates under a design that supports the security of the data exchanged within. This way, principles that preserve packet integrity and confidentiality are upheld. As such, these are cornerstones of cybersecurity to which the Matter protocol sticks with all possible stringency. End-to-end encryption is one of these security mechanisms, which ensures that only intended devices can ever possibly decode messages, thereby protecting the communications from eavesdropping and any other unauthorized access.

It was decided to adopt specialized tools to overcome inherent difficulties in packet decoding for the development of a Matter-compliant device. The encryption of the protocol would introduce limitations that could be bypassed, thus easing the analysis between the source and destination nodes. As will

be discussed in detail in Section 4, the modifications to the dissector with regard to the implementation of the extension on read interaction, proof, and effectiveness are already satisfactory in terms of their current state of development.
A comprehensive examination revealed that enhanced the dissector's ability to accurately interpret the data transmitted under the Matter protocol. The ability to ascribe exact semantics to numeric values within the Wireshark tool signifies a substantial leap forward in understanding the interactions between Matter-compliant IoT devices.

Further developments may be directed towards optimizing the capabilities of the dissector, thereby enhancing its efficiency with the introduction of other features regarding the interaction model, such as the semantics relating to the write interaction or even the subscription event. As an illustration, the incorporation of sophisticated debugging and analysis capabilities could be undertaken, thereby facilitating for developers the diagnosis of network issues and the enhancement of the functionality of IoT devices. Furthermore, integration with other analytical tools could enhance the dissector's utility, facilitating a more comprehensive view of communications in a Matter-based IoT network.

Finally, the work carried out partially in the project has laid very strong foundations for data semanticization in the context of the Matter protocol and for treating undesired problems befalling its intrinsic complexity and robust security. Modifications in dissector code and specific development tools have been used to open ways to decrypt the packets precisely, thus opening the path for future enhancements and practical applications in the area of security and management of IoT networks.

Bibliography

[1] Connectivity Standards Alliance. Working with the chip tool. https://github.com/project-chip/connectedhomeip/blob/master/docs/guides/chip_tool_guide.md. [Accessed 11-05-2024].

[2] Silicon Labs. The matter data model. https://docs.silabs.com/Matter/2.2.1/Matter-fundamentals-data-model/, 2024. Accessed: 2024-05-05.

[3] Silicon Labs. The matter interaction model. https://docs.silabs.com/Matter/2.2.1/Matter-fundamentals-interaction-model/, 2024. Accessed: 2024-05-05.

[4] Wireshark. Wireshark developer&x2019;s guide — wireshark.org. https://www.wireshark.org/docs/wsdg_html_chunked/index.html. [Accessed 10-07-2024].

[5] Emil Diaconu. Smart lighting system. *The Scientific Bulletin of Electrical Engineering Faculty*, 21:6–9, 04 2021.

[6] Levent Özgür, Vahid Khalilpour Akram, Moharram Challenger, and Orhan Dağdeviren. An iot based smart thermostat. In *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, pages 252–256, 2018.

[7] Haidawati Nasir, Wan Aziz, Fuead Ali, Kushsairy Kadir, and Sheroz Khan. The implementation of iot based smart refrigerator system, 07 2018.

[8] Khaled ELORBANY, Cüneyt Bayilmiş, and Seda Balta. A smart plug equipped with iot technologies for energy management of electrical appliances. *Sakarya University Journal of Computer and Information Sciences*, 4, 12 2021.

[9] Z. Mahmood. *Connected Environments for the Internet of Things: Challenges and Solutions*. Computer Communications and Networks. Springer International fPublishing, 2018.

[10] Connectivity Standards Alliance (CSA). Members of the connectivity standards alliance. `https://csa-iot.org/members/`, 2024. Accessed: 2024-04-30.

[11] Wikipedia contributors. Matter (standard) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Matter_(standard)&oldid=1220276850`, 2024. [Online; accessed 26-April-2024].

[12] Bob Hinden and Dr. Steve E. Deering. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, December 1998.

[13] Developer Center Google. Thread and ipv6. `https://developers.home.google.com/Matter/primer/thread-and-ipv6`, 2024. Accessed: 2024-04-30.

[14] Dimitri Belli, Paolo Barsocchi, and Filippo Palumbo. Connectivity standards alliance matter: State of the art and opportunities. *Internet of Things*, 25:101005, 2024.

[15] Marcello Maugeri. Fuzzing matter(s): A white paper for fuzzing the matter protocol. In *Proceedings of the 10th International Conference on Information Systems Security and Privacy - ICISSP*, pages 446–451. INSTICC, SciTePress, 2024.

[16] The device data model — matter — google home developers — developers.home.google.com. `https://developers.home.google.com/Matter/primer/device-data-model`. [Accessed 10-07-2024].

[17] Wikimedia Foundation. Traffic analysis. `https://en.wikipedia.org/wiki/Traffic_analysis`, 10 May 2024. [Accessed 11-05-2024].

[18] Mohammed Qadeer, Arshad Iqbal, Mohammad Zahid, and Misbahur Siddiqui. Network traffic analysis and intrusion detection using packet sniffer. *Communication Software and Networks, International Conference on*, 0:313–317, 01 2010.

[19] wikipedia. pcap. `https://en.wikipedia.org/wiki/Pcap`, 1 Apr 2024. [Accessed 11-05-2024].

[20] Connectivity Standards Alliance. 23-27349-005-matter-1.3-core-specification.pdf. https://csa-iot.org/wp-content/uploads/2024/05/Matter-1.3-Core-Specification.pdf. [Accessed 10-05-2024].