Tiny oneM2M C Language API

Rafael Pereira[a]

[a] *Computer Science and Communications Research Centre, School of Technology and Management, Polytechnic of Leiria, 2411-901 Leiria, Portugal*

ARTICLE INFO

Keywords:
quadrupole exciton
polariton
WGM
BEC

ABSTRACT

This template helps you to create a properly formatted LaTeX manuscript.
`\beginabstract ... \endabstract` and `\begin{keyword} ... \end{keyword}` which contain the abstract and keywords respectively.
Each keyword shall be separated by a `\sep` command.

1. Introduction

adicionar footnode com o link do standard OneM2M

The Internet of Things (IoT) has experienced tremendous growth in recent years, with an ever-increasing number of devices being connected and integrated into various applications, such as smart cities, industrial automation, and environmental monitoring. One key aspect enabling this widespread adoption is Machine Type Communication (MTC), which facilitates seamless communication between machines and devices without human intervention. MTC allows for efficient data exchange among heterogeneous devices, leading to better coordination and improved overall system performance.

In this context, interoperability plays a vital role in ensuring the seamless functioning of IoT ecosystems. As the number and diversity of connected devices continue to grow, so does the need for a robust mechanism to manage communication between devices employing different communication protocols, data formats, and standards. MTC gateways serve as an essential bridge between these devices and the network infrastructure, enabling effective data routing and ensuring smooth communication among heterogeneous devices.

Existing MTC gateway implementations predominantly rely on high-level programming languages, which offer the advantages of rapid development cycles and ease of use. However, these high-level languages often come with trade-offs, such as limited control over hardware resources, increased memory overhead, and suboptimal performance. These limitations can pose significant challenges when deploying large-scale IoT systems, where efficient resource utilization, interoperability, and high-throughput data transmission are essential.

In this paper, we present a novel MTC gateway implementation using a low-level programming language to address the limitations of high-level language-based implementations. Our approach is designed to deliver superior performance, reduced memory overhead, and increased control over hardware resources, while maintaining compatibility with a wide range of MTC protocols. By focusing on key challenges associated with MTC gateways, such as protocol interoperability, scalability, and latency, our implementation aims to provide a robust and efficient solution for the evolving IoT landscape.

Paper structure The remainder of this paper is organized as follows: Section 2 provides a comprehensive review of the related work in MTC gateway implementations, discussing various programming languages and architectural choices, as well as the importance of interoperability in IoT systems. Section 3 delves into the specific challenges associated with MTC protocol integration and outlines the design principles of our novel low-level language-based gateway. Section 5 presents the experimental setup and performance evaluation of our proposed solution. Finally, Section 8 concludes the paper and highlights potential future research directions.

2. Related work

MTC has become an increasingly critical aspect of modern communication systems, driving the transformation of various applications in the ever-evolving IoT ecosystem. MTC enables the seamless interaction between a vast array of devices, ranging from wearables and smartphones to industrial equipment and smart home appliances, facilitating the data exchange required for intelligent and automated decision-making processes. With the exponential growth of IoT devices and their stringent requirements for low latency, high reliability, and energy efficiency, the implementation of MTC in low-level languages has emerged as a promising solution to address these challenges.

In this section, we will provide an overview of the current state-of-the-art in MTC gateway implementations, highlighting the advantages and disadvantages of various programming languages and architectural choices. Additionally, we will explore the specific challenges associated with MTC protocol integration and discuss how our novel approach addresses these issues to provide a comprehensive solution for the next generation of IoT deployments.

Rubi et al. [1] propose an Internet of Medical Things (IoMT) platform with a cloud-based electronic health system that collects data from various sources and relays it through gateways or fog servers. The platform enables data management, visualization of electronic records, and knowledge extraction through big data processing, machine learning, and online analytics processing. Additionally, it offers data sharing services for third-party applications, improving health-

`rafael.m.pereira@ipleiria.pt` (R. Pereira)
ORCID(s): `0000-0001-8313-7253` (R. Pereira)
`https://www.linkedin.com/profile/view?id=rafaelmendespereira` (R. Pereira)

care services and interoperability.

The authors in [2] conducted Internet of Things Device Management (IoTDM) service testing for smart city management, focusing on the ecological situation in Saint-Petersburg's central district, using an Software-Defined Networking (SDN) network infrastructure. They developed a hierarchical model and IoT traffic generator in compliance with oneM2M specifications and implemented Hypertext Transfer Protocol (HTTP), Constrained Application Protocol (CoAP), and Message Queuing Telemetry Transport (MQTT) protocols. Their findings suggested that HTTP is the most suitable protocol for registering IoT devices to the IoTDM service, while MQTT is optimal for transmitting sensor data. The authors concluded that the SDN approach significantly reduces the Round Trip Time (RTT) parameter, demonstrating its potential in managing IoT traffic in smart city environments.

Soumya et al. [3] explore Fog Computing as a consumer-centric IoT service deployment platform, particularly for connected vehicles and intelligent transportation systems requiring low latency, high mobility, real-time data analytics, and wide geographic coverage. They present an IoT architecture for connected vehicles, integrating Fog Computing platforms into oneM2M standard architecture. The architecture comprises virtual sensing zones, Access Points (AP), Road-side Units (RSUs), Machine To Machine (M2M) gateways, and cloud systems. Fog Computing enables consumer-centric services such as M2M data analytics with semantics, discovery of IoT services, and management of connected vehicles. The paper highlights the advantages of Fog Computing, including reduced latency, improved Quality of Service (QoS), real-time data analysis, and actuation for superior user experience and consumer-centric IoT products.

This paper [4] presents a Cluster-based Congestion-mitigating Access Scheme (CCAS) that addresses the issue of collision and access efficiency in M2M communications. The authors propose a modified spectral clustering algorithm to form clusters based on the location and service requirements of Machine-type Communication Devices (MTCDs), ensuring that devices with similar requirements are grouped together. They also develop an Machine-type Communication Gateway (MTCG) selection algorithm that considers the delay requirements of MTCDs and selects an MTCG with the appropriate forwarding threshold. The proposed CCAS divides the data transmission process into an Machine-type Communication Device (MTCD)-MTCG stage and an MTCG-Base Station (BS) stage to reduce collision probability and increase the number of successfully received packets, without causing additional average access delays for MTCDs. The paper includes a theoretical analysis of the MTCG aggregation and forwarding process, as well as performance evaluations through simulations.

The paper [5] proposes a resource-efficient secure implementation of OneM2M based on C programming language for IoT, while also performing an analysis of the OneM2M architecture and protocols. Furthermore, the authors highlight the relevance of a lightweight implementation using optimized data structures and algorithms for IoT devices with limited resources and propose a solution that can reduce computational costs and processing time. Secure communication protocols such as Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS), as well as authentication and encryption mechanisms, were used to ensure security. The authors also suggested that the cryptography algorithms chosen must be lightweight to reduce the computational burden on IoT devices. The evaluation of this implementation meant using a set of benchmarks while comparing the developed OneM2M implementation with existing ones. Results showed that, in terms of memory usage and processing time, the author's proposed implementation of OneM2M using C language is more resource-efficient.

The authors in [6] present a secure authentication and access control scheme for OneM2M-based IoT systems. Moreover, propose an approach to address the challenge of identity and access management in large-scale IoT systems. The proposed scheme is based on the role-based access control model, where system administrators can define access policies based on the roles of the users. Their scheme entails a secure authentication and authorization mechanism based on certificates, which are issued by a trusted Certificate Authority (CA). The evaluation process used a prototype implementation of the OneM2M standard and compared it with existing access control schemes. Therefore concluding that their scheme achieves better performance and scalability while providing a flexible and fine-grained access control mechanism. However, the proposed scheme contains security and privacy issues for which the authors presented several countermeasures to mitigate them.

The paper [7] discusses the design, implementation and evaluation of an interoperable platform for the Internet of Things IoT using the OneM2M standard, where the authors focused on implementing the standard using C language. Moreover, the authors discussed how their C implementation handles registration, discovery and subscription-resource management tasks while proposing optimizations to improve performance. Experiments were designed to assess the scalability, reliability, and processing time with different loads and data sizes. Those experiments concluded that the C implementation of OneM2M achieved better performance than other implementations regarding response time and memory usage. Regarding security, the paper also details oneM2M's security features, namely, authentication, authorization, and confidentiality, how the C implementation handles these features and how to improve the platform's security. Interoperability features, such as standardized interfaces and protocols, and their C implementation were also detailed in the paper since one of oneM2M's main objects must be the interoperability between different IoT platforms and devices.

3. Architecture

The architecture of TinyOneM2M, a compact, cross-platform, and easy-to-use implementation of the OneM2M standard, is meticulously designed to adapt to various scenarios. The flexible architecture supports different modes of operation, allowing it to integrate seamlessly into existing infrastructure. This section details the system architecture and the specific modules that constitute the TinyOneM2M software.

3.1. System Architecture

The system architecture of TinyOneM2M is developed considering the diverse range of scenarios that could arise in IoT settings. Three primary scenarios have been identified where TinyOneM2M can be effectively employed, namely, the Home Gateway Scenario, the Single Devices Scenario (Peer-to-Peer), and the Cloud Scenario.

In the Home Gateway Scenario (Figure 1a), TinyOneM2M serves as a central gateway within a home network, facilitating communication between different devices in the network. This centralized approach allows for efficient data transfer and coordination between devices.

The Single Devices Scenario (Figure 1b) represents a decentralized, Peer-to-Peer (P2P) approach where each device contains its instance of TinyOneM2M. This approach allows devices to communicate directly with each other, enhancing scalability and data transfer efficiency by eliminating the need for a central point of communication.

In the Cloud Scenario (Figure 1c), TinyOneM2M is hosted on a cloud provider and can be distributed in several machines/instances and be managed by a load balancer, allowing devices to connect and communicate via the internet. This scenario facilitates large-scale, distributed IoT applications, providing the advantages of cloud computing such as high availability, scalability, and efficient data management.

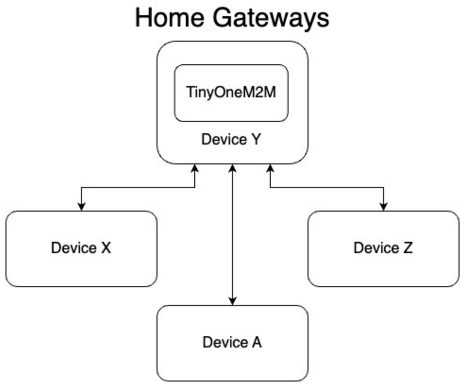
3.2. TinyOneM2M Architecture

TinyOneM2M is architected in a way that distinctly separates responsibilities across various modules, providing a robust and efficient system. Its architecture is underpinned by a seamless flow of operations that handles and processes incoming HTTP REST requests, ultimately leading to action being taken, such as data persistence or triggering notifications. An overview of the architecture is depicted in Figure 2 and will be described in the following sections dividing each module by section.

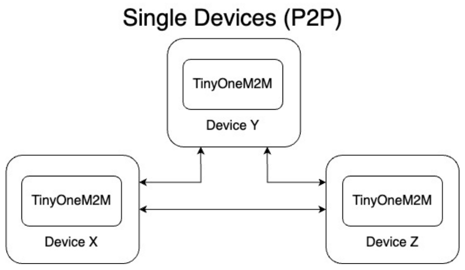
3.2.1. Overview of Resource Oriented Architecture

A crucial aspect of TinyOneM2M's system architecture, as depicted in Figure 3, is the interaction between third-party applications and TinyOneM2M, and between TinyOneM2M and external MQTT brokers or HTTP web servers. This figure outlines these interactions and the hierarchy of resources (described in Section 3.2.5) within the TinyOneM2M system.

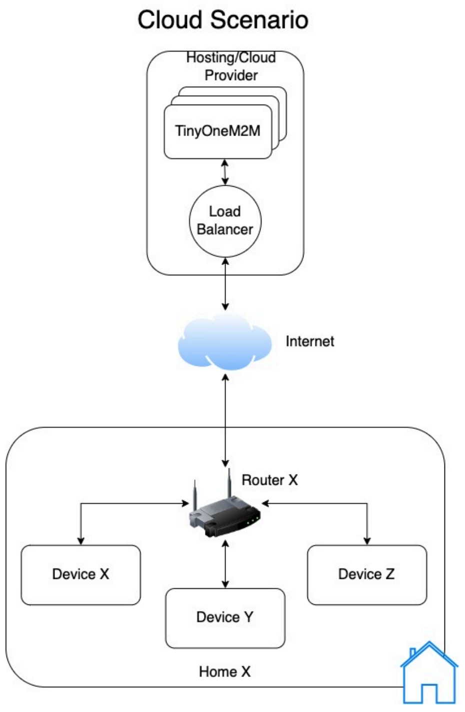
Third-party applications communicate with TinyOneM2M to create or retrieve resources. They send HTTP REST requests to the REST Webservice module in TinyOneM2M, which acts as an intermediary, accepting and processing various types of requests, including POST, GET, PUT, and DELETE. Depending on the type of request, the application can create a new resource, retrieve details of an existing resource, update a resource, or remove a resource.



(a) Home Gateway Scenario.



(b) Single Devices Scenario (Peer-to-Peer).



(c) Cloud Scenario.

Figure 1: TinyOneM2M System Architecture.

TinyOneM2M also communicates with MQTT brokers or HTTP WebServers, primarily for notifying about events based on subscriptions inserted into the TinyOneM2M system. If a resource marked as notifiable changes - for instance, if a new data point is added to a monitored sensor - TinyOneM2M's Notification module creates an MQTT or HTTP client to send a notification to the relevant MQTT broker or

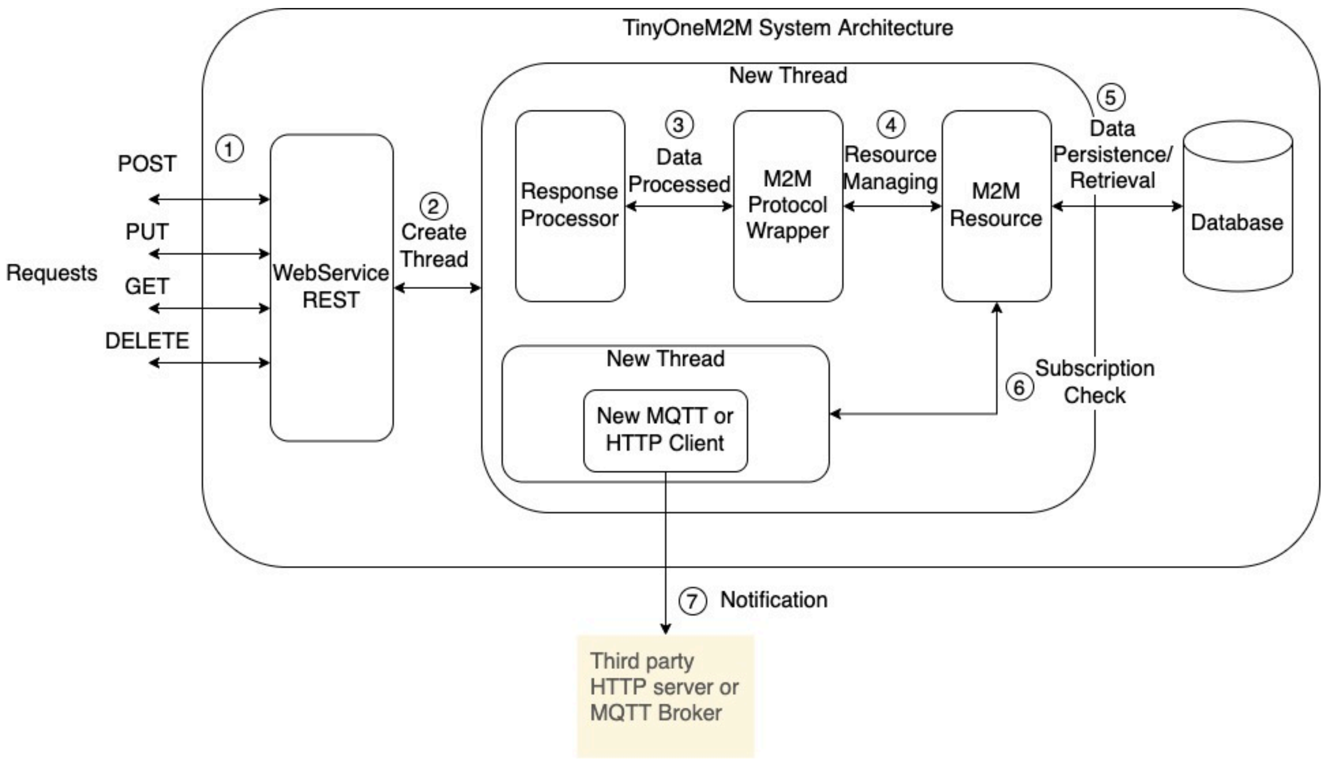


Figure 2: TinyOneM2M System Architecture.

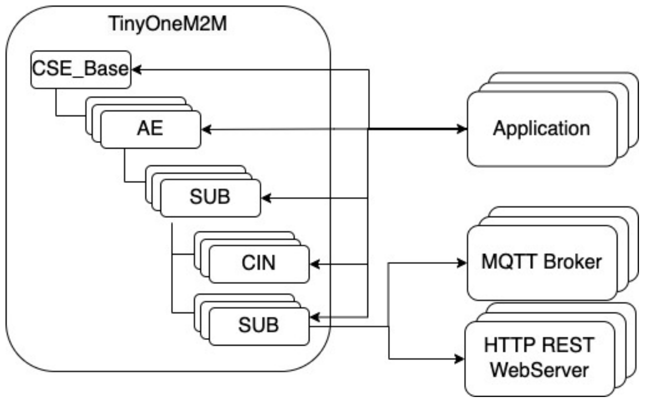


Figure 3: System Architecture.

HTTP server at the moment the changes happen. This notification mechanism allows applications to react promptly to changes in monitored resources, making it an integral part of many IoT scenarios, from anomaly detection in industrial settings to updates in smart home environments.

3.2.2. WebService REST

The REST Webservice module in TinyOneM2M functions as the system's primary interface, accepting and processing various types of HTTP REST requests. When a request is received, this module generates a non-blocking thread to manage the request, which is then passed onto the next layer for further processing.

The types of HTTP requests that are supported by TinyOneM2M are primarily structured around CRUD (Create, Retrieve, Update, Delete) operations for various resources. These operations pertain to different resources identified by OneM2M standards, including Application Entity (AE), Content Instance (CIN), and Subscription (SUB) resources.

For Application Entities (AEs), users can create a new AE by sending a POST request, retrieve the details of an AE using a GET request, update an AE using a PUT request, and remove an AE by means of a DELETE request. These requests are sent to a specific URL structure, targeting the CSE_Base and the `AE_ResourceName`.

For Content Instances (CINs), the operations mirror those for AEs but are extended with an additional URL parameter to specify the CIN. The operations thus allow users to create, retrieve, and delete CINs within a specific AE and Container (CNT) resource.

Similarly, Subscription (SUB) resources can be managed with POST, GET, PUT, and DELETE requests. These requests target a specific `SUB_ResourceName` within a specified CSE_Base, `AE_ResourceName`, and `CNT_ResourceName`.

The system also provides functionality to retrieve the root resource information with a specific GET request and perform discovery operations. The discovery operation uses a GET request with query parameters to filter resources based on their type and labels. These operations can be used to retrieve general or specific information about the resources available in the system.

3.2.3. *Response Processor*

The instantiated thread uses the Response Processor module to validate the request, determine the HTTP method, and conduct preliminary validations. The processed request is then forwarded to the next module for further action.

3.2.4. *M2M Protocol Wrapper*

The M2M Protocol Wrapper is responsible for processing and validating the incoming request according to the OneM2M standard. It acts as a bridge between the protocol-specific details of the request and the M2M resource module, ensuring that all processed requests comply with the OneM2M standard.

3.2.5. *M2M Resources*

The M2M Resources module in TinyOneM2M is a crucial component that handles data persistence and retrieval operations. Acting as an intermediary between the incoming requests and the database, this module either persists data to the database or retrieves data from it, based on the action specified in the processed request. Additionally, it checks if the parent resource associated with the incoming request is notifiable. If it is, the module triggers a new non-blocking thread for the Notification module, thus effectively bridging the communication between the client and the notification mechanisms.

This module supports different types of resources, following a hierarchical structure as prescribed by the OneM2M standard and as shown in Figure 4. The CSE_Base (Common Services Entity Base) resource sits at the root of this hierarchy, providing an entry point for all other resources in the system.

The AE (Application Entity) is an application-level entity that provides services to end-users. It can encapsulate CNT (Container) and SUB (Subscription) resources, which provide data organization and condition-based notification services, respectively.

The CNT resource represents a container for organizing data within AE and other CNT resources. It can contain other CNT, CIN (Content Instance), and SUB resources, allowing an organized and hierarchical data structure.

The CIN resource, residing under a CNT, encapsulates the actual data or content in the system. The SUB resource, which can be nested under CSE_Base, AE, and CNT resources, is responsible for managing subscriptions. This enables applications to receive notifications about changes in resources they're interested in.

3.2.6. *Embedded Single File Database*

The TinyOneM2M system relies on an embedded single-file database for optimized data storage and retrieval. This format offers high efficiency due to its low overhead and quick access time. The system employs a denormalized structure in certain areas, an approach commonly used in database design to improve performance, even at the expense of data redundancy.

In this setup, there is one main table named `mtc`. This table in 1 is a combination of multiple entity types in the TinyOneM2M hierarchy, such as Application Entity (AE), Content Instance (CIN), and Subscription (SUB). The attributes of these entities are represented as columns in the `mtc` table.

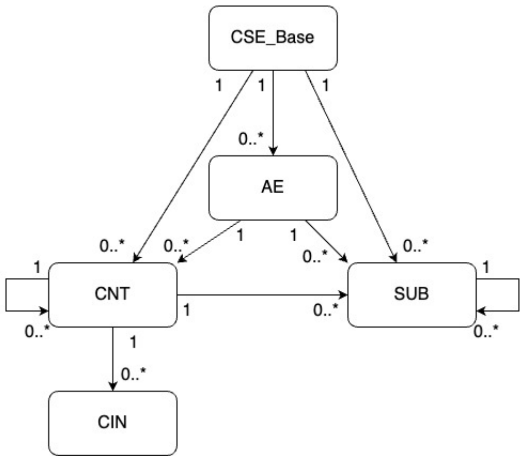


Figure 4: TinyOneM2M supported resources hierarchy.

Looking deeper into the table structure, the `ri` column serves as the primary key, providing unique identification for each record. It establishes relations with the pi column (parent identifier), which refers to the `ri` of the parent record. This sets up a hierarchical relationship between records, which mirrors the resource hierarchy in the TinyOneM2M system.

The table also includes specific columns for different resource attributes. For example, `ty` for the resource type, `rn` for the resource name, and et for the expiration time. It also stores specific attributes for AE like `aei`, for SUB like nu and `enc`, and for `CIN` like `con` (content) and `cnf` (content info).

Some columns like `cbs` and `mni` are used for handling quotas and limits in the TinyOneM2M system. For instance, `cbs` (current byte size) and `mbs` (maximum byte size) are related to the storage size occupied by CIN resource inside a CNT and its maximum allowed size, respectively.

The `mtc` table also includes columns like `url`, `lbl`, `acpi`, `daci`, and `poa`, which store various types of information required for OneM2M standard operations.

Finally, to optimize performance, especially for read operations, the database employs indexes on selected columns. This setup allows faster data retrieval and processing, a key requirement for real-time operations in IoT environments. For instance, the `sqlite_autoindex_mtc_1` index on the `ri` column helps in quickly locating records based on their unique identifiers.

3.2.7. *Notifications*

The Notification module is responsible for sending notifications based on subscription resources. When a resource (e.g. Container) contains a subscription directly inside and other resource (e.g. Content Instance) is created directly inside the resource (e.g. Container) an MQTT or HTTP client is asynchronous created to send a notification to the respective third-party MQTT or HTTP broker.

Table 1
Database structure.

Column Name	Type	Reference
ty	INTEGER	-
ri	TEXT	PRIMARY KEY
rn	TEXT	-
pi	TEXT	REFERENCES mtc(ri) ON DELETE CASCADE
aei	TEXT	-
csi	TEXT	-
cst	INTEGER	-
api	TEXT	-
rr	TEXT	-
et	DATETIME	-
ct	DATETIME	-
lt	DATETIME	-
url	TEXT	-
lbl	TEXT	-
acpi	TEXT	-
daci	TEXT	-
poa	TEXT	-
srt	TEXT	-
blob	TEXT	-
cbs	INTEGER	-
cni	INTEGER	-
mbs	INTEGER	-
mni	INTEGER	-
st	INTEGER	-
cnf	TEXT	-
cs	INTEGER	-
con	TEXT	-
nu	TEXT	-
enc	TEXT	-

4. Implementation of TinyOneM2M Solution

Our solution adheres to the OneM2M standard version 5.1 [1], implementing a set of key resources integral to our intended functionalities. We opted for the implementation of the resources CSE_Base, AE, CNT, CIN, and SUB. The implementation of TinyOneM2M is designed with a strong focus on the system architecture presented in Figure 2 in the Architecture section. Each component of the architecture has a corresponding implementation detail which we aim to elucidate throughout this section.

The development of the TinyOneM2M solution was carried out using the C language. There are several reasons for this choice. Firstly, C is highly portable and efficient, making it ideal for IoT applications where devices can have varied processing capabilities. It also offers low-level access to memory, which allows fine-tuned control over system resources. However, C also has some disadvantages. It lacks the built-in object-oriented features found in languages like C++ or Python, which can lead to more complex code structures for large systems. Moreover, the developer is responsible for memory management, which can introduce potential issues if not handled carefully.

To gauge our level of compliance with the OneM2M standard, we quantified the degree of our implementation based on each resource. We calculated the percentage of implementation by comparing the number of implemented attributes to the total number of attributes suggested by the standard, all multiplied by one hundred.

Based on this calculation, we estimate that our solution covers approximately 57% of the standard on average. A detailed breakdown of this calculation, per resource, will be further elucidated in the section 4.4.

Our implementation is designed to ensure compatibility with services utilizing the OpenMTC [2] SDK, an open-source framework that provides an OneM2M-based version 1.1 platform. This interoperability is possible because the REST requests in our implementation have the same structure as those in the OpenMTC implementation and return similar error codes.

Throughout the development process, we've leveraged a variety of external and non-native libraries. The specific libraries used, along with their corresponding versions, will be duly referenced in the following sections. We believe this level of transparency is crucial to understand the depth and scope of our work.

In the subsequent sections, we will delve into greater detail about our implementation, covering different aspects of our TinyOneM2M solution, and providing a comprehensive view of what was achieved.

4.1. WebService REST

The WebService REST module, as discussed in Section 3.2.2, plays a vital role in our TinyOneM2M solution. It functions as the primary communication hub, receiving HTTP requests and handling client connections. This module is pivotal to our architecture, acting as the gateway through which third-party systems and applications interact with our TinyOneM2M solution.

The implementation of this module is carried out with the pthread library for thread creation, and the native socket library for setting up and managing the REST Webserver. When the webserver detects new clients, it accepts them and launches a new thread. This thread is responsible for reading the content of the request and executing the complete request processing workflow.

To manage webserver routes, a double-sorted list was employed. This list uses the alphabetical order of the resource path as the key for sorting and simple fields to facilitate resource queries. The resource path, which is a composite of the `resource name` of the resources in the resource hierarchy, is immutable.

Upon the application's start, the database is scanned to construct the ordered double list, which is subsequently stored in Random Access Memory (RAM). This strategy ensures enhanced query efficiency whenever the existence of a request's resource path needs to be verified. When a resource is inserted or deleted, the list is updated accordingly.

However, the use of pthread and the native socket library present a few limitations. While efficient, these libraries may not offer optimal compatibility for systems like Windows, thus limiting the platform independence of our solution. A potential enhancement could be the use of a cross-platform library, such as Drogon[3] for C++, which could offer non-

[1] https://www.onem2m.org/technical/published-specifications#listofreleases

[2] https://github.com/OpenMTC/OpenMTC

[3] https://github.com/drogonframework/drogon

blocking operations for the HTTP server.

It's important to note that the algorithm for managing the double-sorted list was custom implemented by the authors. While this allowed for a tailored solution to our specific needs, a well-tested library could have been utilized for this purpose, potentially offering more robustness and efficiency. Nonetheless, our current implementation is situated within the main process and has shown competent performance in managing routes for the webserver.

4.2. Response Processor

As introduced in Section 3.2.3, the Response Processor module is central to handling the incoming content from the solution's clients. It undertakes critical tasks such as reading the content of the clients' connections, validating the routes based on the sorted double list, and processing the HTTP request types.

The module employs a built-in hash table to efficiently map the resources affected by the requests. The hash table provides swift access to key values, making it highly suitable for this task. It inspects and interprets the resource types encapsulated in the HTTP requests, facilitating the subsequent processing steps.

One important aspect to note is the static nature of the hash table. Stored in memory, the hash table remains constant for the entire lifetime of the service. This means it is built only once during the service initialization and does not change over time.

However, similar to the WebService REST module, the algorithm to manage the hash table was also a custom implementation by the authors. Although it is designed to meet the specific needs of this application, a pre-existing, well-tested library could offer enhanced reliability and potentially better performance.

4.3. M2M Protocol Wrapper

As established in the Architecture section, the `M2M Protocol Wrapper` is instrumental for the operation of the entire system. Its functions range from initial protocol initialization, resource discovery, and building values, to defining default values and facilitating the insertion of resources into the database.

Upon protocol initialization, the database is created and the resource CSE_Base is populated if this has not been done previously. The resource discovery feature is 28% implemented according to the standard, implementing 9 of the suggested 32 filters.

Table 2 describes each implemented discovery attribute:

Table 2
Description of implemented discovery attributes

Filter Name	Cardinality	Description
resourceType	0..1	The `resourceType` attribute of the matched resource is the same as the specified value. It also allows `differentiating` between normal and announced resources.
labels	0..1	The `labels` attribute of the matched resource matches the specified value.
createdBefore	0..1	The `creationTime` attribute of the matched resource is chronologically before the specified value.
createdAfter	0..1	The `creationTime` attribute of the matched resource is chronologically after the specified value.
modifiedSince	0..1	The `lastModifiedTime` attribute of the matched resource is chronologically after the specified value.
unmodifiedSince	0..1	The `lastModifiedTime` attribute of the matched resource is chronologically before the specified value.
expireBefore	0..1	The `expirationTime` attribute of the matched resource is chronologically before the specified value.
expireAfter	0..1	The `expirationTime` attribute of the matched resource is chronologically after the specified value.
filterOperation	0..1	Indicates the logical operation (AND/OR) to be used for different condition tags. The default value is logical AND.
limit	0..1	The maximum number of resources to be included in the filtering result.

The module performs minor database queries to build values before their use. It also validates the attributes of the resources and defines default values, such as creating the automatic "Resource Name" when necessary. Furthermore, it builds a JSON structure that gets mapped to a native structure for later insertion into the database by the resource module.

The CJson[4] library (version 1.7.15) is utilized for handling Json content. Note that this library is used statically, as the source code was imported into this project.

[4] `https://github.com/DaveGamble/cJSON`

4.4. M2M Resources

Our TinyOneM2M solution provides a comprehensive implementation of several critical resources as per the OneM2M standard. These resources – CSE_Base, AE, CNT, CIN, and SUB – serve as the backbone of our application, enabling diverse functionalities and providing a robust foundation upon which our solution operates.

The CSE_Base resource acts as a central hub, organizing and managing the other resources and services of our solution. The AE resource encapsulates application-specific information, allowing our solution to be adaptable across multiple use cases and applications. The CNT and CIN resources work in tandem to handle and manage the information table and its records, enabling efficient and effective data management. Lastly, the SUB resource provides the functionality to 'watch' other resources, a crucial feature that allows for real-time monitoring and quick response to changes.

As shown in Table 3, we have been successful in achieving substantial coverage of the OneM2M standard. Our implementation covers 52% of the standard for the CSE_Base resource, 57% for AE, 68% for CNT, 70% for CIN, and 40% for SUB. These percentages reflect our commitment to aligning our solution as closely as possible with the OneM2M standard, ensuring compatibility and interoperability, while also adapting the solution to meet the specific needs and constraints of our use case.

The CSE_Base resource can be seen as a system's set of databases, the AE resource as a database containing an application's information, the CNT resource as an information

Table 3
Resource Implementation Percentage

Resource Type	Implemented Attributes	Attributes Suggested by the Standard	Percentage (%)
CSE_Base	12	23	52%
AE	20	35	57%
CNT	17	25	68%
CIN	14	20	70%
SUB	12	30	40%

table, and the CIN resource as the table's rows. Subscriptions (SUB) can be thought of as watchers that can be added to the resources CSE_Base, AE, CNT, or even within another subscription.

The implementation of each feature is based on the attributes suggested in Section 9.6.3 of the OneM2M standard documentation (version 5.1), excluding some less relevant attributes like `appName`, `AE-ID`, `CSE-ID`, etc.

Crucially, the implementation of certain attributes alters the system's normal functioning. All resources implement the expiration time attribute, thus preventing access to any resource whose expiration time is less than the current datetime. For the CNT resource, the attributes max byte size, max number of instances, current byte size, and current number of instances were implemented. For the SUB resource, the event notification criteria attribute was implemented to specify which HTTP request methods will trigger notifications.

The standard is broad, and some fields may not be relevant for all users. Hence, we have selected the most pertinent attributes for implementation and earmarked the remainder for potential future work.

4.5. Embedded Single File Database

The architecture of our TinyOneM2M solution employs an Embedded Single File Database module, serving as the central data repository. Tasked with persisting resource data, managing hierarchical relationships between resource entities, and enabling efficient data retrieval mechanisms, this module is instrumental in achieving a reliable, efficient, and manageable software system.

For our database management system, we selected SQLite3 [5], an established file-based database known for its performance and simplicity. We integrated the SQLite3 library, specifically version 3.41.1, as source code into our architecture, thereby ensuring broad compatibility across different operating systems. This choice aligns well with our commitment to a platform-independent solution.

SQLite3's incorporation offers numerous advantages, aligning perfectly with our solution's design principles. It is lightweight, efficient, and enables swift database operations with minimal resource usage. Its ease in executing queries contributes directly to our system's responsiveness and performance.

The initial architecture of our database comprised two tables: one for single-valued attributes and another for multi-valued attributes, such as labels and points of access. This design aimed to maintain the structure and initial order of these attributes. However, this approach posed performance challenges during database operations, prompting us to shift towards a denormalized, single-table design to enhance performance. Section 5.1 provides a detailed comparison of operations on normalized versus denormalized databases.

To optimize our database further, we strategically added indexes on attributes such as parent id, resource id, and URL (resource path). Despite not being a standard attribute, URL was implemented to facilitate query operations, particularly during route mapping.

While SQLite3 serves our current needs effectively, we are mindful of potential limitations. Alternative embedded database solutions like MySQL or PostgreSQL could prove beneficial under specific conditions such as high concurrency or heightened system requirements. One area for potential enhancement within SQLite3 is the use of cache mode, leveraging RAM for faster operations [6]. However, this approach can pose challenges with memory blocking in a multi-threaded environment and requires careful handling [].

For future development, a detailed analysis of SQLite3 database operations could identify potential optimizations at the attribute level, possibly using the PRAGMA operator. Transactional behavior could also be improved by exploring write-ahead logging (WAL) to enhance concurrency. Particularly in a multi-threaded environment, concurrent reads and writes can be significantly optimized with WAL, providing a valuable addition to our implementation [].

4.6. Notifications

The Notifications module plays an integral role in our TinyOneM2M solution, responsible for alerting subscribers about changes in resources they have subscribed to. Notifications, often generated through modifications in resources, are essential to ensure subscribers stay informed about the current state of resources they are interested in.

Our notification system is built around the subscription resource feature. Whenever a notification needs to be dispatched, a new thread is spun up to handle the delivery process. This thread-centric approach enables our system to maintain a high level of responsiveness and efficiency by preventing the blocking of main execution threads. A subscription in TinyOneM2M serves as a contractual agreement between a subscriber and a resource. Whenever a resource that a client has subscribed to undergoes any change, a notification is triggered. This subscription-driven approach ensures that notifications are contextually relevant and valuable to the subscriber.

To facilitate the creation of MQTT and/or HTTP clients within these notification threads, we have utilized the Mongoose library [7] (version 7.10). The use of this library greatly simplifies the process of setting up clients and managing connections, contributing to the overall efficiency and reliability of our notification system.

[5] `https://www.sqlite.org/cintro.html`

[6] `https://www.sqlite.org/inmemorydb.html`

[7] `https://github.com/cesanta/mongoose/tree/master`

The delivery addresses for notifications are defined within the `nu` attribute in the subscription. For example, the delivery addresses could be specified as `mqtt://0.0.0.0:1883`, `http://0.0.0.0:1400/monitor`. Similarly, the topic or resource path, such as `/onem2m/ae-test/cnttest/subtest`, denotes the address of the resource associated with the notification.

Further enhancing the specificity of our notification system, we've implemented an attribute called event notification criteria, as described in Section 4.4. This attribute allows specifying which HTTP request methods should trigger notifications. For instance, setting this attribute to `"GET, POST"` ensures that notifications are triggered only when HTTP GET or POST requests affecting direct child resources occur. This gives subscribers precise control over which actions result in notifications, thus providing a highly customizable notification experience.

For example, if a container resource contains a subscription, a POST request to create a content instance resource within this container will trigger a notification. However, a DELETE request will not lead to a notification, as the DELETE method is not included in the event notification criteria. This flexibility provides precise control over notification triggers, tailoring notifications based on subscribers' specific interests or system requirements.

5. Tests

In this crucial section, we delve into the testing mechanisms employed to examine the TinyOneM2M solution under various scenarios, highlighting its strengths and exposing areas for improvement.

Firstly, the Database Normalisation versus Database Denormalisation test will provide a performance comparison of operation execution time between normalized and denormalized databases. The purpose of this test is to unveil any potential performance ramifications of the chosen database structure.

Our second test is dedicated to Assessing the Efficiency of Meeting Expiration Time Attribute Objectives in Database Queries on Performance. Here, we focus on the expiration time attribute, which either hides or eliminates records with an expiration time value surpassing the current datetime. We aim to contrast the duration of database operations with and without the implementation of this attribute, focusing on the average time spent on each operation type.

A crucial part of our testing process is the Comparative Analysis of Latency for CRUD operations using both our proposed solution and OpenMTC. This test involves a direct performance comparison between our TinyOneM2M implementation, adhering to version 5.1, and OpenMTC, which complies with the standard version 1.1. We meticulously measure the average time expended on the Create, Read, Update, and Delete operations.

In the Cross-Platform Evaluation, we demonstrate the adaptability of the TinyOneM2M system by evaluating its ease of installation and compilation on two distinct operating systems: Ubuntu and macOS. The test showcases the platform-independent nature of our solution, adding to its versatility.

Lastly, our stress test puts the TinyOneM2M solution through rigorous trials on the Raspberry Pi 0 W microcontroller under the Minimal Hardware Stress Testing test. This test is intended to simulate high-pressure scenarios involving a large number of insertion requests to examine the resilience of our solution.

We've conducted these tests on three machines of varying specifications, each listed in Table 4. The diversity in our testing apparatus reiterates the solution's flexibility across an array of hardware and software configurations. Notably, the libraries deployed on each machine, along with their respective versions, remain static across the board as outlined in Section 4.

Table 4 delivers a comprehensive overview of the specific hardware, operating system, compiler, and library configurations for the machines used in our tests. This information is instrumental for anyone aiming to replicate the testing environment and offers valuable context to our results.

5.1. Database Normalization VS Database Denormalization

In this section, we present an investigation into the performance of database normalization versus denormalization. This analysis does not directly implement the OneM2M standard but instead provides a comparative assessment of these two database structuring strategies.

To execute the tests, we employed a Python script directly on the SQLite3 database. The script uses the SQLite3 library and was run on Machine 1 from Table 4. The version of Python used was 3.9.10 and the version of the SQLite3 library used was 3.37.0. The SQLite3 database was allocated in the RAM.

Algorithm 1 Pseudocode to compare SQLite Performance with Hierarchy Complexity

```
1:  procedure [INSERT/SELECT/UPDATE/DELETE]DATA(conn, table, depth)
2:      start time
3:      for $i = 1$ to depth do
4:          execute SQL to [Insert/Select/Update/Delete] data in table with hierarchy depth i
5:      end for
6:      end time
7:      calculate and print time difference divided by depth
8:  end procedure
9:
10: procedure COMPAREPERFORMANCE
11:     conn ← open database connection
12:     tables ← [OneM2M_Denorm, CSE_Base_Norm, AE_Norm, CNT_Norm, SUB_Norm]
13:     depth ← 15
14:     CreateDenormalizedTables(conn)
15:     CreateNormalizedTables(conn)
16:     for each table in tables do
17:         InsertData(conn, table, depth)
18:         SelectData(conn, table, depth)
19:         UpdateData(conn, table, depth)
20:         DeleteData(conn, table, depth)
21:     end for
22:     close database connection
23: end procedure
```

To execute the tests, we employed a Python script directly on the SQLite3 database. A pseudocode representation of the script used can be found in Algorithm 1. The

Table 4
Hardware, Operating System, Compiler, and Library Specifications for Test Bed Machines.

Machine1	
Hardware	
Chip	Apple M1 Pro @ 3.2 GHz
Total Number of Cores	10 (8 performance and 2 efficiency)
RAM Memory	16 GB
Operating System	
Product Name	macOS
Product Version	12.6
Kernel Version	Darwin 21.6.0
Compiler	
GCC Version	Apple clang version 14.0.0 (clang-1400.0.29.102)
Machine2	
Hardware	
Chip	Intel(R) Core(TM)2 CPU 6600 @ 2.40GHz
Total Number of Cores	2
RAM Memory	8 GB
Operating System	
Product Name	Ubuntu
Product Version	20.04.5
Kernel Version	5.15.0-67-generic
Compiler	
GCC Version	Ubuntu 9.4.0-1ubuntu1 20.04.1
Machine3	
Hardware	
Chip	BCM2835 @ 1GHz ARMv6
Total Number of Cores	1
RAM Memory	512MB
Operating System	
Product Name	Debian
Product Version	12.6.3
Kernel Version	Darwin 21.6.0
Compiler	
GCC Version	Debian 8.3.0-6

script performs Insert, Select, Update, and Delete operations on tables of different hierarchical depths, calculating the average operation time per depth. The depth used for these operations was 15, and each operation was performed 1000 times for each database. The structures of the normalized and denormalized databases are depicted in Figure 5 mainly showing the hierarchy of resources.

To analyze the results, let's refer to Table 5. For the insertion operation, the denormalized structure was slightly more time-efficient, taking an average of 6.96×10^{-6} seconds per depth layer to insert a record, while the normalized structure ranged between 6.82×10^{-6} and 7.23×10^{-6} seconds per depth layer.

The selection operation showed a notable difference between the two database structures. Simple read operations on the denormalized structure consumed significantly more time (around 0.0045 seconds) than the normalized structure. However, with deep querying, the normalized structure showed an increase in operation time (up to 0.0033 seconds) due to

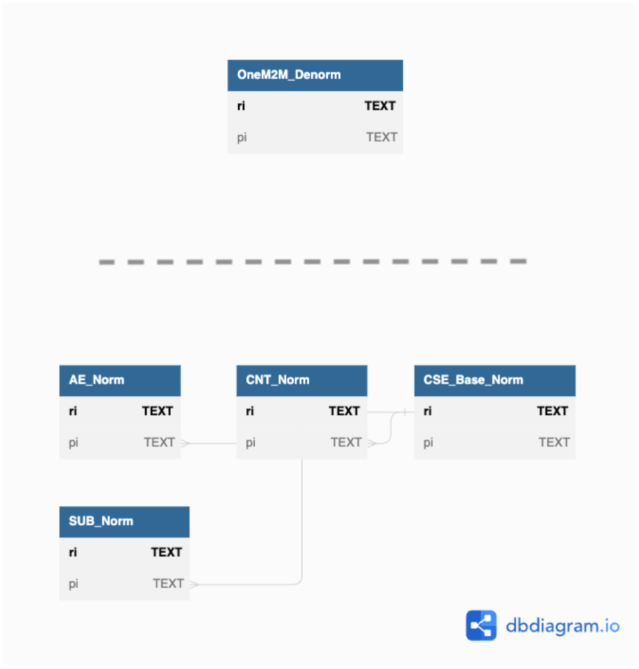


Figure 5: Diagram of Database Structure used for Comparison of the Normalized Strategy and Denormalized Strategy.

Table 5
Average time per depth layer for CRUD operations on the databases

Operation	Denormalized Structure Seconds	Normalized Structure Seconds
Insert	6.96×10^{-6} (s)	6.82×10^{-6} to 7.23×10^{-6} (s)
Select	0.0045 (s)	1.33×10^{-6} to 0.0033 (s)
Update	6.90×10^{-6} (s)	6.77×10^{-6} to 7.32×10^{-6} (s)
Delete	3.60×10^{-6} (s)	3.67×10^{-6} to 4.01×10^{-6} (s)

the complexity of join operations.

In the case of the update operation, the normalized structure was slightly quicker, taking about 6.77×10^{-6} to 7.32×10^{-6} seconds per depth layer compared to 6.90×10^{-6} seconds in the denormalized structure. Similar to the update operation, the delete operation was slightly faster in the normalized structure, taking between 3.67×10^{-6} and 4.01×10^{-6} seconds per depth layer, while the denormalized structure took about 3.60×10^{-6} seconds.

From these results, we observe that the denormalized structure was more efficient in terms of insert operations, a crucial factor for applications necessitating large volumes of data insertions. However, when it comes to deep querying in a hierarchical structure, the denormalized structure outperforms due to the overhead of join operations in normalized tables. While these results are based on a relatively small example and an embedded database system, it is expected that as the database scales, these differences could become more pronounced.

6. Assessing the Efficiency of Meeting Expiration Time Attribute Objectives in Database Queries on Performance

In this part of our study, we focus on assessing the efficiency of meeting the 'expiration time' attribute objectives in database queries on performance. This test was carried out on Machine 1, using Python version 3.9.10 and SQLite3 version 3.37.0.

The OneM2M standard can be strictly followed in certain environments, but this can sometimes lead to a degradation of system performance. A common cause for this is the inclusion of the 'expiration time' attribute and the associated logic. It's worth noting that our implementation does not delete records for which the expiration time has already passed. However, there are a couple of other possible approaches to this situation:

1. *On-demand Record Deletion:* All requests use the current context to search for records whose "expiration time" field has already passed and deletes them. This method adds complexity to each request, potentially impacting performance and increasing the likelihood of implementation errors.
2. *Periodic Expiration Check:* A thread is created that remains active indefinitely, checking every X time if there are fields with an exceeded "expiration time". This approach can lead to additional resource consumption and potential race conditions, making it less ideal than the first approach.

For our test, we considered three scenarios: no implementation of the 'expiration time' attribute, implementation of the attribute without indexing, and implementation of the attribute with indexing. We chose these scenarios to demonstrate how much the presence and indexing of the 'expiration time' attribute can impact the implementation.

A Python script was implemented to perform this test. The script created the following tables: `OneM2M_NoET`, `OneM2M_NonIndexedET`, and `OneM2M_IndexedET`. Each table underwent 10,000 operations of each type (Create, Read, Update, and Delete).

As shown in Figure X, `OneM2M_NoET` table does not implement the 'expiration time' attribute, `OneM2M_NonIndexedET` implements it without indexing, and `OneM2M_IndexedET` implements it with indexing. Indexing the 'expiration time' attribute can improve search times when looking for records with specific expiration times, but it also adds additional overhead when inserting or updating records, as the index needs to be updated as well.

Figure 6: Diagram of the tables

Our results are as follows:

Table 6
Results from the 'Expiration Time' test

Scenario	CREATE Avg T
Original Codebase without Expiration Time Condition	2.37e-06
Non-Indexed Expiration Time Condition	2.48e-06
Indexed Expiration Time Condition	2.69e-06

Our results show that the inclusion of the 'expiration time' attribute and its indexing can degrade performance in SQLite3 database operations. Notably, the UPDATE operation was significantly slower when we considered 'expiration time', especially for the Indexed scenario. This is likely due to the overhead of maintaining the index. Similarly, for both the CREATE and DELETE operations, the presence of 'expiration time' attribute, and more so its indexing, resulted in slower operation times. These results highlight the importance of understanding the trade-offs between adhering to standards like OneM2M and the performance requirements of specific applications.

In this section, we focus on a specific test case that aims to assess the efficiency of meeting expiration time field objectives in database queries on performance, using codebase version 1.1. The database contains 2000 records in the "mtc" table and 41,000 in the "multivalue" table, adding difficulty to the SELECT queries. The "expiration time" field's purpose is to ensure that resources are deleted or become unused after their expiration. The test will be performed on a "discovery" request with five parameters: ty (resource type), lbl (labels), expireAfter using the "expiration time" field, expireBefore using the "expiration time" field, and filterOperation. It is important to note that the "labels" attribute is an array and its search requires more effort because it searches in the "multivalue" table. By default, the "discovery" request will have a limit of 50 rows listed, which can be adjusted as a parameter, but for this test, the default value will be used. As the limit value increases, the query's latency may grow exponentially, making it crucial to optimize the database and query structure. Three distinct approaches were considered, with the first one being the primary focus of this study:

- Selective Query Filtering - This approach filters out expired records by adding a condition to consultation, update, and removal queries to apply only to records with a valid "expiration time" attribute. A REST route is created to delete invalid records. This method is advantageous because it minimizes the overhead associated with constantly checking and deleting expired records while still maintaining data integrity. The performance of this approach will be compared with two other scenarios in terms of query time.

- On-demand Record Deletion - In this approach, all requests use the current context to search for records whose "expiration time" field has already passed and, if so, deletes them. This method adds complexity to each request, potentially impacting performance and increasing the likelihood of implementation errors.

- Periodic Expiration Check - A thread is created that remains active indefinitely, checking every X time if there are fields with an exceeded "expiration time". However, there is a chance that expired records would still be present when queries are made on the main threads. This approach can lead to additional resource consumption and potential race conditions, making it less ideal than the first approach.

The test case under examination focuses on the first approach, "Selective Query Filtering," and examines its performance in three different scenarios using codebase version 1.1: the codebase with the "expiration time" field as a condition in database queries, the codebase with the "expiration time" field as a condition in database queries when it is indexed in the database, and the original codebase without the "expiration time" field as a condition. The test will be performed by executing 500 "discovery" requests with five parameters in each of the three scenarios on "Machine 1", as shown in Table 4. By evaluating the efficiency of these scenarios in meeting the objectives of the "expiration time" field, we can identify the most effective method for ensuring proper resource management.

The first approach, "Selective Query Filtering," has been chosen as the primary focus of this study due to its potential for minimizing overhead while maintaining data integrity. The comparative analysis of the three scenarios will provide valuable insights into the impact of different handling techniques for the "expiration time" field on system performance using codebase version 1.1. This assessment will help identify potential optimizations and guide the selection of the most suitable approach for managing resources with an expiration time, ultimately enhancing the overall efficiency of the system.

To assess the performance of the three scenarios in the test case, we have designed a simple algorithm that sends a total of 500 requests, each retrieving data through SELECT queries to the target route for each scenario, considering the five parameters mentioned above. This algorithm measures the time taken for each request in seconds and accumulates the minimum, maximum, and total time. Additionally, it calculates the squared difference of each request's time and the average time to determine the standard deviation. By using this algorithm, we can consistently evaluate the time efficiency of the "Selective Query Filtering" approach in comparison to the other scenarios. The pseudocode for this algorithm is shown in Algorithm 2.

With the algorithm in place, we can proceed to the actual testing phase. For each of the three scenarios, we will run the algorithm on "Machine 1" and record the minimum, maximum, average, and standard deviation of the request times. This data will allow us to thoroughly analyze the performance impact of each approach and draw conclusions about their efficiency in meeting the "expiration time" field objectives. The results of the tests are presented in Table 7.

Upon examining the results in Table 7, we can derive several insights into the efficiency of the "Selective Query Filtering" approach in managing resources with an expiration time. These insights will be valuable in guiding future optimizations and ensuring proper resource management within the system.

Algorithm 2 Pseudocode for Time Measurement

```
1: Initialize $count, sum, min, max, sq\_diff\_sum \leftarrow 0$
2: $min \leftarrow$ large value
3: for $i \leftarrow 1$ to 500 do
4:     Send a request to the target route
5:     Measure the time taken for the request ($time\_ms$)
6:     if $time\_ms < min$ then
7:         $min \leftarrow time\_ms$
8:     end if
9:     if $time\_ms > max$ then
10:        $max \leftarrow time\_ms$
11:    end if
12:    $sum \leftarrow sum + time\_ms$
13: end for
14: Calculate $average \leftarrow \frac{sum}{500}$
15: for $i \leftarrow 1$ to 500 do
16:    $diff \leftarrow time\ taken\ for\ request\ i - average$
17:    $sq\_diff \leftarrow diff \times diff$
18:    $sq\_diff\_sum \leftarrow sq\_diff\_sum + sq\_diff$
19: end for
20: Calculate $standard\_deviation \leftarrow \sqrt{\frac{sq\_diff\_sum}{500}}$
```

It is important to highlight the trade-offs when comparing the three scenarios. First, the original codebase without the expiration time condition exhibits slightly better average performance compared to the indexed expiration time condition scenario. This result is expected, as the original codebase does not include any additional checks for expired resources, which results in reduced complexity and faster query times. However, the difference in average time between the original codebase and the non-indexed expiration time condition scenario is minimal, indicating that the added complexity of checking for expiration does not significantly impact the overall performance.

When comparing the non-indexed expiration time condition scenario with the indexed expiration time condition scenario, we observe a slight increase in the average time for the indexed case, which might seem counterintuitive as indexing usually improves query performance. However, it is essential to consider that the standard deviation for the indexed expiration time condition scenario is considerably higher, indicating more varied response times. This variability could be attributed to the database's specific structure, the distribution of data within the tables, or the types of queries executed during the test. Despite the higher average time, the indexed expiration time condition scenario offers a more efficient approach to managing resources with an expiration time. The increased standard deviation highlights the variability in response times, which can be attributed to various factors such as the database's structure or the nature of the queries executed during the test. Consequently, the indexed scenario demonstrates greater scalability and adaptability in handling larger volumes of data, making it the preferred choice for implementation.

As the system evolves, the indexed expiration time condition scenario will be the one that prevails and will be part of the solution implementation. It is recommended to further investigate the factors contributing to the variability in response times for this scenario and explore the performance

Table 7
Performance comparison of the three scenarios

Scenario	Min (s)	Max (s)	Avg (s)	Std Dev (s)
Original Codebase without Expiration Time Condition	0.614603	0.683479	0.626082	0.007875
Non-Indexed Expiration Time Condition	0.614021	0.672247	0.622916	0.007017
Indexed Expiration Time Condition	0.629939	1.446011	0.640483	0.045949

impact of different limit values. This analysis will help guide future optimizations and enhance the system's overall efficiency while meeting the expiration time field objectives.

Furthermore, it is essential to note that the "discovery" request route being tested involves five parameters, which add complexity to the evaluation of each scenario's performance. As the limit value of 50 rows listed increases, the query's latency may grow exponentially. Therefore, the current test with a default limit of 50 rows serves as a starting point, and it is crucial to evaluate the performance of these scenarios with varying limit values to understand their impact on the overall system efficiency.

In conclusion, the "Selective Query Filtering" approach has shown promise in terms of meeting the expiration time field objectives with a minimal impact on database query performance. The small performance difference between the original codebase without the expiration time condition and the non-indexed expiration time condition scenario indicates that checking for expiration does not drastically impact performance. It is recommended to investigate the indexed expiration time condition scenario further to understand the variability in its response times and to explore the performance impact of different limit values. This analysis will guide future optimizations and help select the most suitable approach for managing resources with an expiration time, ultimately enhancing the system's overall efficiency.

6.1. Comparative Analysis of Latency for CRUD Operations Using the Proposed Solution and OpenMTC

In this section, we delve into a detailed test case that focuses on the comparative analysis of latency for CRUD (Create, Read, Update, and Delete) operations using the proposed solution and OpenMTC. The objective of this test case is to evaluate the performance of our solution in managing resources against the established OpenMTC standard, providing a benchmark for assessing the efficiency and effectiveness of our approach.

The CRUD operations will be performed through REST requests over the network, rather than on a local machine, as the primary goal of the oneM2M gateway is to facilitate the exchange of information across a distributed network infrastructure. This setup simulates real-world conditions where gateway systems must efficiently handle remote requests, thereby providing a more accurate representation of the solution's performance in practical applications.

To carry out this test case, we will use "Machine 2", as shown in Table 4 as described in the previous test case, with the following specifications: Apple M1 Pro chip, 10 cores, 16 GB RAM, and macOS version 12.6. The codebase version for the proposed solution is 1.2. The test will be conducted using a shell script that sends 500 Application Entity (AE) resource CRUD operations for both the proposed solution and the OpenMTC platform.

The shell script operates as follows: It begins by initializing variables and CSV files for each CRUD operation (POST, PUT, GET, and DELETE). Then, it iterates 500 times, performing the following steps in each iteration: sending a POST, PUT, GET, and DELETE request to the respective servers, measuring the time taken for each request, and appending the time to the corresponding CSV file. After completing the 500 iterations, the script calculates the minimum, maximum, average, and standard deviation of the times for each CRUD operation from their respective CSV files. The results of these calculations provide insights into the latency of the proposed solution and OpenMTC for each CRUD operation.

By comparing the latency for 500 Application Entity (AE) resource CRUD operations on both platforms, we can highlight the strengths and areas for improvement of our proposed solution. This analysis will not only help us to understand the impact of our solution on system performance but also to identify any potential bottlenecks and areas that could benefit from optimization, particularly in the context of network latency and resource management.

The insights gained from this test case will be crucial in the ongoing development and refinement of our solution, ultimately contributing to a more robust and efficient system for managing resources in the context of the oneM2M standard while accounting for the network-related challenges inherent in distributed systems.

6.2. Cross-Platform Evaluation: Installation and Compilation on Ubuntu and macOS Systems

In this section, we introduce a test case aimed at evaluating the cross-platform compatibility of our proposed solution through installation and compilation on two widely-used operating systems, Ubuntu and macOS. The objective of this test case is to ensure that our solution can be seamlessly integrated into a variety of operating environments, thereby demonstrating its versatility and adaptability to different system configurations.

A cross-platform evaluation is crucial in the realm of computer science, as it verifies the portability and interoperability of software across different platforms. By validating our solution's compatibility with both Ubuntu and macOS, we demonstrate its ability to function effectively in diverse contexts, catering to a broader range of users and potential applications.

This test case will involve the installation of necessary dependencies, compilation of the source code, and execution of the proposed solution on both operating systems. The evaluation will not only focus on the successful completion of these tasks but also on identifying any platform-specific issues, discrepancies, or bottlenecks that may arise during the process.

The insights gained from this cross-platform evaluation will contribute to the ongoing refinement of our solution, ensuring that it remains robust and effective across various operating environments. This, in turn, enhances its potential for widespread adoption and application in the context of the oneM2M standard.

In this test case, we successfully installed and compiled the proposed solution on Ubuntu, one of the two operating systems being evaluated for cross-platform compatibility. The process began by importing the source code to "Machine 2", as shown in Table 4, using SFTP, since the operation system have already installed the GCC compiler theres no more dependencies to install and the system is ready to be compiled.

The proposed solution was successfully compiled using the provided Makefile, which facilitated the creation of the executable without any issues. A closer examination of the dependencies of the compiled executable was carried out using the `ldd` command. The command revealed that the proposed solution relies exclusively on system libraries such as 'libdl.so.2', 'libpthread.so.0', and 'libc.so.6', which provide essential functionality for dynamic loading, multi-threading, and standard C functions, respectively.

By relying solely on system libraries, the proposed solution demonstrates a significant advantage in terms of portability and maintainability. This approach eliminates the need for third-party dependencies, reducing potential compatibility issues and simplifying the installation process across various platforms. Furthermore, it ensures that the solution benefits from the stability and performance optimizations provided by the operating system's native libraries, which are generally well-maintained and frequently updated.

The successful installation, compilation, and identification of dependencies on Ubuntu demonstrate that the proposed solution is adaptable to various operating environments. This adaptability is essential in the realm of computer science as it ensures the portability and interoperability of the software across different platforms. By validating our solution's compatibility with Ubuntu, we have shown its ability to cater to a broader range of users and potential applications.

As the next step in the cross-platform evaluation, the proposed solution will be installed and compiled on macOS. This process will involve identifying and installing any necessary dependencies, compiling the source code, and executing the solution on this operating system.

In this test case, the proposed solution was successfully installed and compiled on the macOS operating system. The source code was imported to "Machine 1", and the necessary dependency, GCC, was found to be already installed on the system. The provided Makefile facilitated the compilation of the files, resulting in the generation of the executable without any issues.

The examination of the dependencies of the compiled executable revealed that the proposed solution relies exclusively on system libraries. The `otool` command showed that the solution depends on '/usr/lib/libSystem.B.dylib', which provides essential functionality for dynamic loading, memory allocation, and system calls. By relying solely on system libraries, the proposed solution demonstrates a significant advantage in terms of portability and maintainability, as it eliminates the need for third-party dependencies.

The use of system libraries highlights the proposed solution's adaptability to different system configurations, ultimately enhancing its potential for widespread adoption and application in the context of the oneM2M standard. This approach ensures that the solution benefits from the stability and performance optimizations provided by the operating system's native libraries, which are generally well-maintained and frequently updated.

By successfully installing and compiling the proposed solution on both Ubuntu and macOS, we have demonstrated its ability to function effectively in diverse contexts, catering to a broader range of users and potential applications. The insights gained from this cross-platform evaluation will inform any necessary adjustments or refinements to ensure seamless integration into various operating environments.

...Final Thoughts...

6.3. Minimal Hardware Compatibility Testing: Raspberry Pi 0 W Case Study

In this section, we present a test case focused on assessing the compatibility of our proposed solution with minimal hardware, specifically using the ESP32 microcontroller as a case study. The objective of this test case is to verify the suitability of our solution for resource-constrained environments, thereby broadening its potential applications and enabling its adoption in a wider range of use cases.

Minimal hardware compatibility is an essential aspect of computer science, as many real-world applications require efficient and lightweight solutions that can run on devices with limited processing power, memory, and energy resources. By evaluating the performance of our proposed solution on the ESP32, a popular and cost-effective microcontroller with integrated Wi-Fi and Bluetooth capabilities, we can demonstrate its ability to function effectively under such constraints.

This test case will involve the adaptation, deployment, and execution of our solution on the ESP32 platform. The evaluation will not only focus on the successful completion of these tasks but also on identifying any hardware-specific issues, discrepancies, or bottlenecks that may arise during the process. Additionally, we will assess the resource usage, including memory and processing overhead, to ensure that the solution remains viable and efficient in resource-constrained scenarios.

The insights gained from this minimal hardware compatibility testing will contribute to the ongoing refinement of our solution, ensuring that it remains both robust and versatile, capable of meeting the demands of a diverse range of applications in the context of the oneM2M standard.

7. Lessons Learned

Referir que as bibliotecas por serem estáticas não são atualizadas quando uma nova versão é lançada, porém isto trás X vantagens e Y desvantagens.

8. Conclusion and Future Work

CRediT authorship contribution statement

: Conceptualization of this study, Methodology, Software.

References

[1] Jesús N.S. Rubí and Paulo R.L. Gondim. Iomt platform for pervasive healthcare data aggregation, processing, and sharing based on onem2m and openehr. *Sensors 2019, Vol. 19, Page 4283*, 19:4283, 10 2019.

[2] Artem Volkov, Abdukodir Khakimov, Ammar Muthanna, Ruslan Kirichek, Andrei Vladyko, and Andrey Koucheryavy. Interaction of the iot traffic generated by a smart city segment with sdn core network. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10372 LNCS:115–126, 2017.

[3] Soumya Kanti Datta, Christian Bonnet, and Jerome Haerri. Fog computing architecture to enable consumer centric internet of things services. *Proceedings of the International Symposium on Consumer Electronics, ISCE*, 2015-August, 8 2015.

[4] Liang Liang, Lu Xu, Bin Cao, and Yunjian Jia. A cluster-based congestion-mitigating access scheme for massive m2m communications in internet of things. *IEEE Internet of Things Journal*, 5:2200–2211, 6 2018.

[5] Steffen Thielemans, Benjamin Sartori, An Braeken, and Kris Steenhaut. Integration of onem2m in inter-iot's platform of platforms. In *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–2, 2019.

[6] Benoygopal E. B, Satheesh G, Prakash Rosayyan, and Hemant Jeevan Magadum. Cosmic - common smart iot connectiv onem2m common service platform for intelligent transportation system. In *2021 2nd International Conference on Communication, Computing and Industry 4.0 (C2I4)*, pages 1–5, 2021.

[7] Jaeho Kim, Sung-Chan Choi, Jaeseok Yun, and Jang-Won Lee. Towards the oneM2M standards for building IoT ecosystem: Analysis, implementation and lessons. *Peer-to-Peer Networking and Applications*, 11(1):139–151, January 2018.