Tiny oneM2M C Language API

Rafael Pereira[a], Carla Mendes[a] and Ana Cruz[a]

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

ARTICLE INFO

ABSTRACT

The Internet of Things (IoT) has revolutionized the way we interact with the environment by connecting everyday objects to the digital world, and Machine To Machine (M2M) communication is a key aspect of the IoT, enables devices to exchange data and collaborate autonomously, leading to intelligent and automated systems. In this paper, we present TinyOneM2M: an implementation of the OneM2M standard using a low-level programming language-based approach, to address the limitations associated with high-level language implementations. Our implementation aims to enhance performance and reduce memory overhead. Through different tests, we evaluate the usability, scalability, and reliability of the solution in various scenarios. The results demonstrate that our TinyOneM2M solution offers seamless interoperability and scalability to IoT and general applications on several Linux-based operating systems, even in resource-constrained scenarios.

1. Introdution

The IoT encompasses a vast network of diverse entities, including tags, sensors, embedded devices, handheld devices, and backend servers. It enables the development of novel services and applications across various domains such as civil transportation, electric power grids, medical treatment, and industrial and home automation [1].

M2M denotes technologies that enable communication and networking of different devices and is a part of the field of IoT. While the IoT focuses on the interconnection of physical objects and their interaction with humans, the M2M system focus towards seamless connectivity. It involves automated systems where devices autonomously gather data (e.g., temperature, humidity, speed, position, heartbeat) from remote sources, exchange information, and execute actions based on control messages. These operations generally occur without or with minimal human intervention, utilizing public network infrastructures. Consequently, the M2M system plays a crucial role in realizing the full potential of the IoT [1], [2], [3].

M2M systems are typically characterized by their heterogeneity, lack of structure, and loose integration, achieved through standardized communication technologies. These technologies include WiFi, Zigbee, Ethernet, and power line communications, among others. By leveraging standardized technologies, M2M systems enable seamless interoperability among devices and simplify mass-produced, standards-compliant equipment. This approach contributes to cost-effective implementations, reducing expenses associated with system development and deployment [2], [4].

The rapid expansion of M2M technologies has compelled

✉ rafael.m.pereira@ipleiria.pt (R. Pereira);
carla.c.mendes@ipleiria.pt (C. Mendes); ana.v.cruz@ipleiria.pt (A. Cruz)

ORCID(s): 0000-0001-8313-7253 (R. Pereira); 0000-0001-7138-7124 (C. Mendes); 0009-0002-4169-7644 (A. Cruz)

🔗 https://www.linkedin.com/profile/view?id=rafaelmendespereira (R. Pereira), https://www.linkedin.com/profile/view?id=carla-mendes-5b3586233 (C. Mendes), https://www.linkedin.com/profile/view?id=ana-cassia-vasconcelos-cruz10 (A. Cruz)

various standardization bodies, such as the European Telecommunications Standards Institute (ETSI), to establish M2M standards, resulting in the emergence of multiple standards (e.g ETSI M2M and OneM2M). Consequently, the OneM2M initiative was established to develop a unified global specification and standard for M2M communication [3].

In 2008, ETSI initiated the standardization process for M2M by examining requirements and identifying gaps in existing standards. ETSI began defining an M2M platform that could cater to various sectors such as smart metering, e-health, transportation, manufacturing, facility management, and more. In February 2015, OneM2M introduced its first global standard. OneM2M shared the same objective as ETSI M2M but aimed to achieve it on a global level [3].

Therefore, the oneM2M [1] standard aims to establish uniformity in the communication between devices, servers, and applications by offering a standardized M2M service. It defines a platform centered around the concept of resources and Representational state transfer (REST) Application Programming Interfaces (API). Moreover, it encompasses over a dozen Common Service Functions such as registration, subscription, discovery, and data management, among others [5].

In this paper, we present a novel implementation of the OneM2M standard using a low-level programming language to address the limitations of high-level language-based implementations. Our approach is purposefully designed to be lightweight, catering to weak computational systems with limited processing power and memory capacity, reducing memory overhead and increasing control over hardware resources. By focusing on key challenges associated with Machine Type Communication (MTC) gateways, such as protocol interoperability, scalability, and latency, our implementation aims to provide a robust and efficient solution for the evolving IoT landscape.

The remainder of this paper is organized as follows: Section 2 provides a comprehensive review of the related work in current M2M implementations, discussing various pro-

---

[1]OneM2M official page: https://www.onem2m.org/

gramming languages. In Section 3, we delve into the system architecture of our TinyOneM2M software, discussing the specific modules that constitute it where we present three scenarios: Home Gateway, Single Device (Peer-to-Peer (P2P)), and Cloud, highlighting how the architecture adapts to each scenario. In Section 4, we provide a comprehensive overview of our implementation of the OneM2M standard version 5.1, detailing the system resources that form the foundation of our solution. The testing mechanisms employed to examine the TinyOneM2M solution under various scenarios are detailed in Section 5. Section 6 encompasses the acquired knowledge and reflective insights gained throughout the system's development. Finally, in Section 7, we conclude the paper by summarizing the key findings and contributions, and we highlight potential future research directions.

2. Related work

OneM2M provides a standardized framework and protocols to enable interoperability between diverse IoT devices. It can be regarded as an overarching framework that facilitates the implementation of M2M solutions, allowing for seamless communication and integration of various M2M interactions within an IoT ecosystem [5]. Therefore, in this section, we will provide an overview of the current state-of-the-art in M2M implementations, highlighting the advantages and disadvantages of various programming languages and architectural choices.

Rubi et al. [6] 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 (ML), and online analytics processing. Additionally, it offers data sharing services for third-party applications, improving healthcare services and interoperability.

The authors in [7] 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. [8] explore Fog Computing (FC) 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 FC platforms into oneM2M standard architecture. The architecture comprises virtual sensing zones, Access Points (AP), Roadside Units (RSUs), M2M gateways, and cloud systems. FC 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 FC, including reduced latency, improved Quality of Service (QoS), real-time data analysis, and actuation for superior user experience and consumer-centric IoT products.

The paper [9] 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 [10] 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 [11] 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 Author-

ity (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 [12] discusses the design, implementation and evaluation of an interoperable platform for the 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.
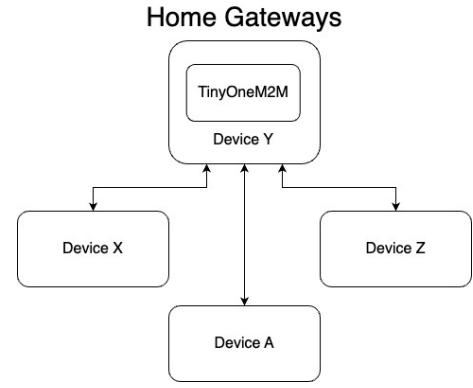
3. Architecture

The architecture of TinyOneM2M, a compact, cross-platform, and easy-to-use implementation of the OneM2M standard [5], 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.
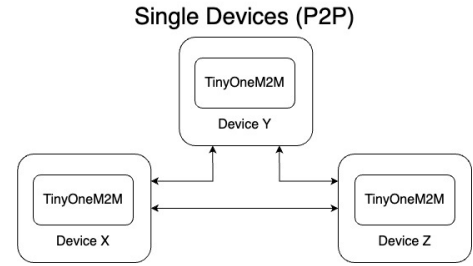
3.1. System Architecture

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 (P2P), 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, 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.

(a) Home Gateway Scenario.

(b) Single Devices Scenario (P2P).
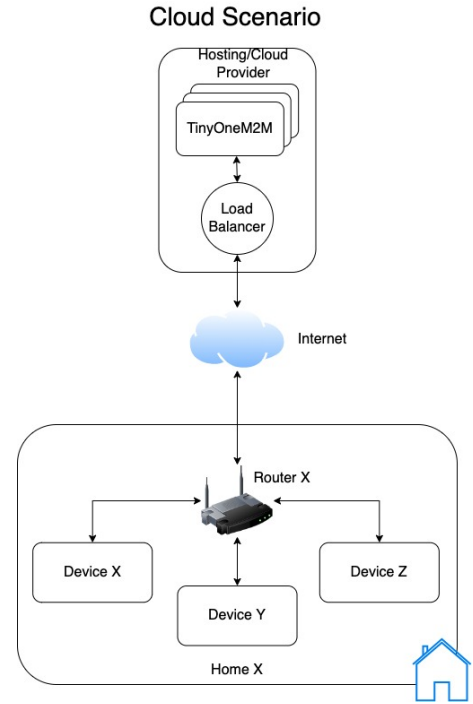
(c) Cloud Scenario.

Figure 1: TinyOneM2M System Architecture.

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.
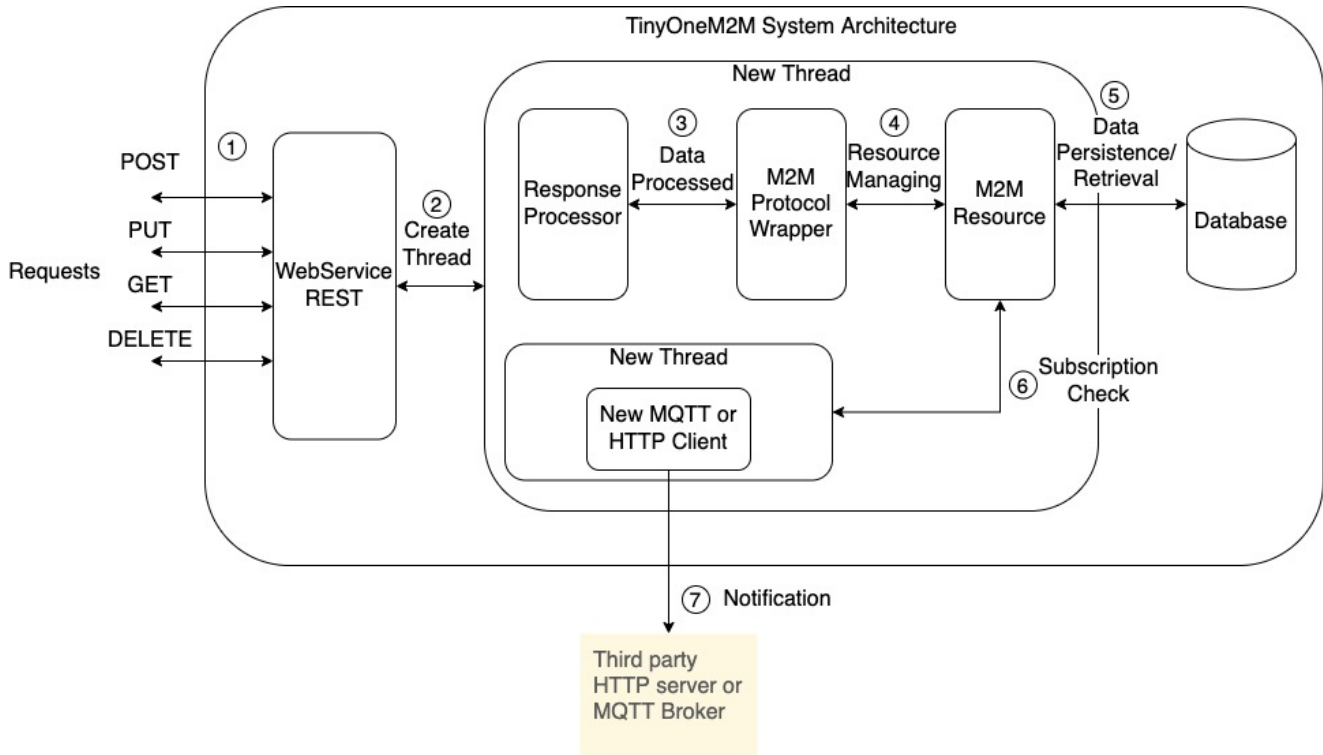
Figure 2: TinyOneM2M System Architecture.

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.

TinyOneM2M also communicates with MQTT brokers or HTTP WebServers, primarily for notifying about events based on subscriptions inserted into the TinyOneM2M sys-
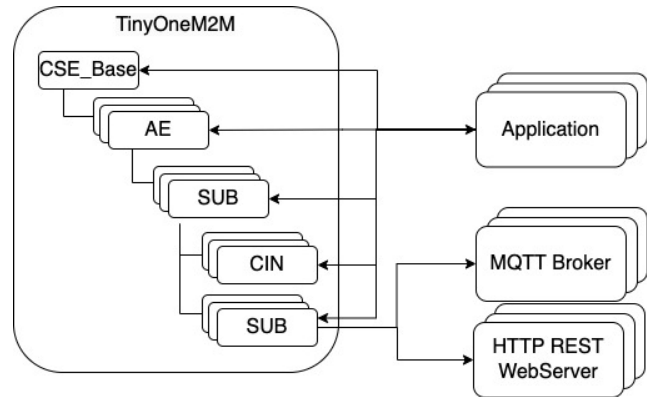
Figure 3: System Architecture.

tem. If a resource is 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 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 re-

quest is received, this module generates a new thread to manage the request, which is then passed onto the next layer for further processing.

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

For Application Entities (AEs), users can perform different actions on AEs using specific HTTP requests. They 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 issuing a DELETE request. These requests are sent to a specific URL structure, targeting the Common Services Entity (CSE_Base) and the AE_ResourceName.

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

Similarly, 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 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 resource sits at the root of this hierarchy, providing an entry point for all other resources in the system.

The AE is an application-level entity that provides services to end-users. It can encapsulate CNT and SUB 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, 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.
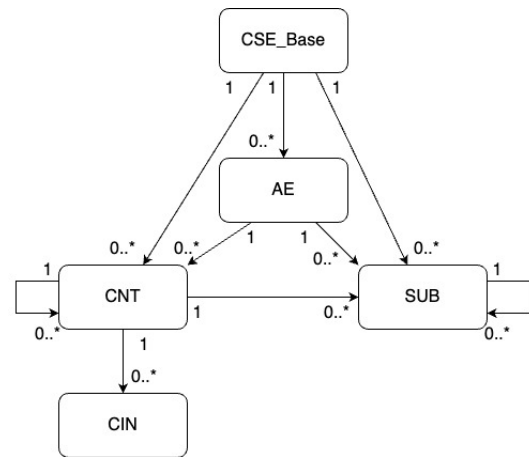

Figure 4: TinyOneM2M supported resources hierarchy.

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, which Table 2 provides a representation of the MTC table's structure, in 2 is a combination of multiple entity types in the TinyOneM2M hierarchy, such as AE, CIN and SUB. The attributes of these entities are represented as columns in the mtc table using their abbreviations.

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,

Table 1
Atributes designations.

Attribute Abbreviation	Attribute Designation
ty	Resource type
ri	Resource identifier
rn	Resource name
pi	Parent identifier
aei	AE identifier
csi	CSE_Base identifier
cst	CSE_Base type
api	App identifier
rr	Request reachability
et	Expiration time
ct	Creation time
lt	Last modified time
url	Url
lbl	Labels
acpi	Access control policy IDs
daci	Dynamic authorization consultation IDs
poa	Point of access
srt	Serialization time
blob	Blob
cbs	Current byte size
cni	Current number of instances
mbs	Maximum byte size
mni	Current number of instances
st	State tag
cnf	Content info
cs	Content size
con	Content
nu	Notification URI
enc	Event notification criteria

Table 2
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	-

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, rn, and et. It also stores specific attributes for AE like aei, for SUB like nu and enc, and for CIN like con and cnf.

Some columns like cbs and mni are used for handling quotas and limits in the TinyOneM2M system. For instance, cbs and mbs 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 (point of access), which store various types of information required for OneM2M standard operations.

To optimize read operations and streamline data retrieval, we have introduced a blob column that stores the JavaScript Object Notation (JSON) object associated with each resource. This approach eliminates the need to select all the columns from the table and map them to the JSON object, as we can now retrieve the complete JSON object directly from the blob column. By doing so, we significantly improve performance and simplify the data retrieval process. Additionally, the database employs indexes on selected columns which 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. CNT) contains a SUB resource directly inside, and another resource (e.g. CIN) is created directly inside the resource (e.g. CNT) an MQTT or HTTP client is asynchronously created to send a notification to the respective third-party MQTT or HTTP broker.

4. Implementation of TinyOneM2M Solution

Our solution adheres to the OneM2M standard version 5.1 [2], 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 Section 3. 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 C language places the burden of memory management on the developer, which, if not executed with caution, can give rise to potential issues and challenges [13].

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 [3] 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 [4] for C++, which could offer non-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. 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 Section 3, 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 3 describes each implemented discovery attribute:

The module performs minor database queries to build values before their use. It also validates the attributes of the

---

[3]OpenMTC official page: `https://github.com/OpenMTC/OpenMTC`

[4]Drogon official page: `https://github.com/drogonframework/drogon`

Table 3
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.

Table 4
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%

ing 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 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 `et` attribute, thus preventing access to any resource whose expiration time is less than the current datetime. For the CNT resource, the attributes `mbs`, `mni`, `cbs`, and `cni` 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.

resources and defines default values, such as creating the automatic `rn` 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[5] 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.

4.4. 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 serves as a central hub, managing and organizing various resources and services within our solution. It acts like a system's set of databases, ensuring efficient data management. The AE resource encapsulates application-specific information, allowing our solution to adapt to different use cases and applications. The CNT and CIN resources work together to handle and manage information tables and their records, ensuring effective data management. Lastly, the SUB resource provides real-time monitoring and quick response to changes by acting as watchers that can be added to any of the mentioned resources, including CSE_Base, AE, CNT, or even within another SUB.

As shown in Table 4, we have been successful in achiev-

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 [6], 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.

---

[5]CJson library official page: `https://github.com/DaveGamble/cJSON`

[6]SQlite3 official page: `https://www.sqlite.org/cintro.html`

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.

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 [7]. However, this approach can pose challenges with memory blocking in a multi-threaded environment and requires careful handling [14].

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 [15].

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 [8] (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.

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 3.2. 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 date time. 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.

---
[7] https://www.sqlite.org/inmemorydb.html
[8] Mongoose library official page: https://github.com/cesanta/mongoose/tree/master

Table 5
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	Raspbian Lite 32x
Product Version	11
Kernel Version	6.1.21
Compiler	
GCC Version	Raspbian 10.2.1

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 5. 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 5 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 5. 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 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 6. For the insertion operation, the denormalized structure was slightly more time-efficient, taking an average of 6.96 microseconds per depth layer to insert a record, while the normalized structure ranged between 6.82 and 7.23 microseconds 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 4500 microseconds) than the normalized structure. However, with deep querying, the normalized structure showed an increase in operation time (up to 3300 microseconds) due to the complexity of join operations.

In the case of the update operation, the normalized structure was slightly quicker, taking about 6.77 to 7.32 microsec-
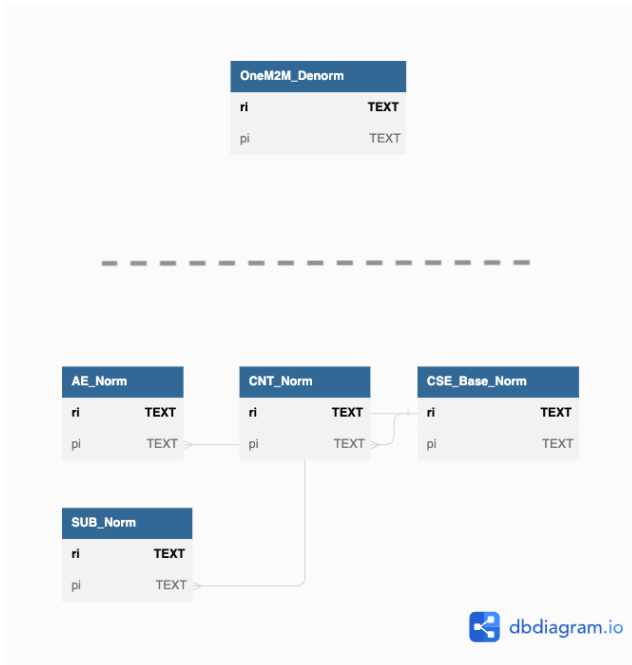
Figure 5: Diagram of Database Structure used for Comparison of the Normalized Strategy and Denormalized Strategy.

Table 6

Average time per depth layer for CRUD operations on the databases

Operation	Denormalized Structure (μs)	Normalized Structure (μs)
Insert	6.96	6.82 to 7.23
Select	4500	1.33 to 3300
Update	6.90	6.77 to 7.32
Delete	3.60	3.67 to 4.01

onds per depth layer compared to 6.90 microseconds in the denormalized structure. Similar to the update operation, the delete operation was slightly faster in the normalized structure, taking between 3.67 and 4.01 microseconds per depth layer, while the denormalized structure took about 3.60 microseconds.

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.

5.2. 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 shown in Table 5, 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 of 'expiration time' attribute can impact the implementation, even when the attribute is indexed. We understand that there are many other ways to optimize the database statements with the presence of 'expiration time'. For instance, one could consider the use of partitioning or sharding based on expiration times [16], where data is logically divided into subsets based on the 'expiration time' value, thus potentially reducing the query scope and improving performance. Another could be techniques such as caching frequently accessed or recently accessed data could also be employed to improve performance [17, 18]. It is, however, important to note that these strategies come with their own trade-offs and complexities and should be carefully considered based on the specific requirements and constraints of the system.

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 6, `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.

Our results shown in Table 7 indicates that the inclusion of the 'expiration time' attribute and its indexing can degrade

Table 7
Results from the 'Expiration Time' test

Scenario	CREATE Avg Time (μs)	READ Avg Time (μs)	UPDATE Avg Time (μs)	DELETE Avg Time (μs)
Original Codebase without Expiration Time Condition	2.37	0.976	2.39	0.726
Non-Indexed Expiration Time Condition	2.48	1.29	529 000	0.946
Indexed Expiration Time Condition	2.69	1.35	3 050 000	1.09
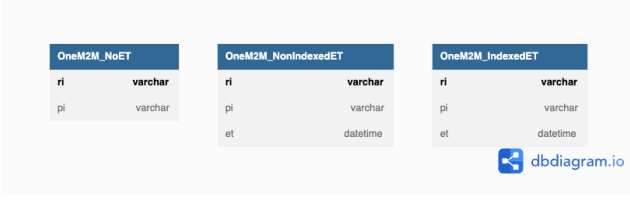
Figure 6: Efficiency of Meeting Expiration Time Attribute Test Database Tables.

Table 8
Comparative Analysis of time measure for CRUD Operations in TinyOneM2M and OpenMTC.

	TinyOneM2M		OpenMTC	
Operation	Min (ms)	Max (ms)	Min (ms)	Max (ms)
POST	0.123773	1.140625	0.011947	15.606682
GET	0.011656	0.115155	0.008696	1.075310
PUT	0.116248	0.592979	0.012049	0.136800
DELETE	0.122092	0.544047	0.008289	0.053867

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.

5.3. 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, the both standard implementations will be running on Machine 2, shown in Table 5, and the requests will be done using Machine 1. The test will be conducted using a shell script that sends 500 requests affecting Content Instances, as we considered the most created resource. Because of Content Instances does not support update operations we will be doing 500 Update operations to Containers resources.

The shell script represented by Algorithm 2 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 and resources (CIN, and CNT), 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.

Algorithm 2 Pseudocode for Time Measurement

1: Initialize $count, sum, min, max, sq\_diff\_sum \leftarrow 0$
2: $min \leftarrow$ static 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 $standard\_deviation \leftarrow \sqrt{\frac{sq\_diff\_sum}{500}}$

In a comparative analysis of latency for CRUD operations between TinyOneM2M and OpenMTC shown in Table 8, significant differences were noted. TinyOneM2M displayed a lower maximum latency for POST operations (1.140625ms) compared to OpenMTC's higher maximum (15.606682ms), suggesting more stable performance. Conversely, OpenMTC demonstrated a quicker response under optimal conditions, outperforming TinyOneM2M in terms of minimum latency for all CRUD operations, for instance, the minimum latency for GET operations was 0.008696ms for OpenMTC com-

pared to 0.011656ms for TinyOneM2M. Further, TinyOneM2M showed less variation in response times across operations, indicating more predictable and stable performance, which could be advantageous in real-world scenarios. However, occasional latency spikes might be an issue with OpenMTC, as suggested by its high maximum latency, potentially affecting overall performance.

5.4. Cross-Platform Evaluation: Installation and Compilation on Ubuntu, macOS, and Raspbian Operating Systems

This section expands our cross-platform evaluation of the proposed solution to include a third operating system, Raspbian. The objective is to ensure seamless integration across diverse operating environments, underlining the solution's versatility and adaptability. This cross-platform evaluation, integral in computer science, tests the portability and interoperability of the solution across Ubuntu, macOS, and now Raspbian, catering to a broader range of users and applications.

The test case involves the installation of necessary dependencies, compilation of the source code, and execution of the proposed solution on all three operating systems, with a focus on identifying platform-specific issues. The insights gained from this evaluation will help refine the solution, ensuring its robustness and broad applicability in the context of the oneM2M standard.

The source code was imported to all machines, that use one of each operating system addressed in this section, shown as Table 5, and the solution was successfully installed and compiled on all machines, leveraging the pre-installed GCC compiler. The Makefile facilitated the compilation, resulting in the creation of the executable without complications. An interesting observation is that the resultant executable file across all the operating systems maintained a consistent size of 1.4Mb, which indicates a uniform and efficient code compilation and linking process. Dependencies of the compiled executable were examined using the `ldd` and `otool` commands, revealing an exclusive reliance on system libraries.

The solution's dependence on system libraries alone, such as `libdl.so.2`, `libpthread.so.0`, and `libc.so.6` for Ubuntu and Raspian, and only `libSystem.B.dylib` for macOS simplifies the installation process and enhances portability and maintainability across platforms. It also benefits from the stability and performance optimizations provided by the operating system's native libraries.

This successful installation, compilation, and dependency identification demonstrates the adaptability of our solution to different operating environments. This adaptability is crucial in ensuring the software's portability and interoperability across different platforms, broadening its potential user base.

Our solution exhibits impressive cross-platform compatibility, having been successfully installed and compiled on Ubuntu, macOS, and Raspbian. The primary reliance on system libraries boosts its portability and maintainability, simplifying the process across various platforms. This re-

Table 9

Resource Utilization for Different Numbers of Content Instance Resources on Raspberry Pi 0 W.

Instances	Database Size (KB)	RAM Usage (KB)
10 000	9 136	5 608
50 000	46 336	9 696
100 000	92 984	14 772

inforces the potential for broad adoption of the solution and the importance of designing versatile software across diverse platforms.

However, while we remain confident about the straightforward installation of the solution on most Linux-based operating systems, additional steps may be required to support Windows, such as an abstraction layer for the pthread library. This highlights the continual need for refinement and adaptation to ensure comprehensive cross-platform compatibility.

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

In this section, we present a case study using the Machine 3 microcontroller, as shown in Table 5. Our objective is to test the compatibility of our proposed solution with resource-constrained environments. This test will be conducted using minimal hardware, an essential consideration in computer science, as many real-world applications require efficient and lightweight solutions capable of running on devices with limited processing power, memory, and energy resources.

The hardware selected for this test is a Raspberry Pi 0 W, a cost-effective and popular microcontroller with integrated Wi-Fi and Bluetooth capabilities. For storage, we used a Kingston 64GB SD Card with a read speed of 100MB/s, a suitable choice for our test scenario.

Our experiment was designed to create SQLite3 CIN resources in quantities of 10 000, 50 000, and 100 000, storing them in the database located on the SD card. We measured the size of the database on the disk and the RAM usage when all resources were loaded by our server application. The shell script we used executed SQLite3 commands to create the desired number of resources and measured the database size, while the top command measured RAM usage, targeting our server application process. The results of these measurements are tabulated in Table 9.

The outcomes of this test case provide significant insights into real-world scenarios, especially those where resources are limited. The efficiency of a solution in such circumstances is determined by its ability to perform effectively under these constraints. These tests can help us understand the suitability of our solution in such situations, and also provide a basis for further optimization.

It's important to note that the recorded RAM usage and disk space are the results of operating in a controlled environment. Actual values can vary depending on a number of other factors, including the specifics of the implementation, the operating conditions, and the additional overhead imposed by other running processes and OS services. Nev-

ertheless, this experiment serves as a valuable indicator of the resource efficiency of our solution.

6. Lessons Learned

In the pursuit of technological innovation, it is not uncommon to traverse uncharted territory and learn through trial and error. This document, reflecting on our journey with the implementation of the TinyOneM2M project, endeavors to impart the lessons we have gleaned, offering readers a distilled version of our experiences and the wisdom we have accrued. This section will broadly focus on key areas such as our choice of database technology, database structure, the use of libraries, and the implementation of the OneM2M standard, each accompanied by the lessons learned and the strategies we adopted for overcoming challenges.

During the development stage, we opted for a single file database over a server-based database, believing it would offer superior performance. However, as we progressed, we discovered that this assumption might not hold. The performance gains we expected did not materialize, prompting us to reevaluate our choice of database technology and explore alternative options, such as a MySQL database because of concurrency, and high write rate, on the other hand, would increase the size of the executable because of the size package size, and the database size occupied.

Following established database best practices, we initially pursued a normalization approach for our database structure which involved creating two tables: `mtc` and `multivalue`. The introduction of the `multivalue` table played a vital role in our database structure since it facilitated the mapping of intricate dependency relationships between each M2M resource. This approach proved particularly valuable in handling the dynamic and complex nature of JSON relationships, enabling us to effectively represent the infinite-like associations that can exist within JSON values. While this approach adhered to standard guidelines, it downgrades performance due to the time-consuming processing required to map a JSON structure in C into corresponding database entries significantly impacting the overall performance of our system. Consequently, we recognized that denormalization would be a more suitable strategy for our project, allowing for improved performance and streamlined data retrieval. Therefore, this ultimately became the final approach we chose to implement our database.

Another of the lessons we learned relates to using static C libraries. While these libraries offer certain advantages, such as ease of distribution and reduced dependency management, they pose challenges when keeping up-to-date with new versions. On the positive side, static libraries provide a stable and predictable development environment. However, the downside is that they do not benefit from bug fixes, security patches, or new features introduced in subsequent releases. This trade-off necessitates careful consideration and evaluation of the specific project requirements when choosing between static and dynamic libraries.

Implementing the OneM2M standard involved incorpo-

rating various attributes, including `et`, `mbs`, and `mni`, among others. While these attributes align with the standard, their inclusion increases implementation complexity and hinders performance. Moreover, in practical scenarios, these fields often have little relevance or are not frequently utilized. Therefore, the attributes to implement must be extensively evaluated to strike a balance between adherence and practicality.

7. Conclusion and Future Work

In this section, we will discuss the conclusions drawn from our C implementation of OneM2M and outline the potential areas for future work and improvement.

To overcome the limitations associated with high-level language-based implementations, we present a novel approach by leveraging a low-level programming language for the implementation of the OneM2M standard. Our methodology aims to address the challenges posed by the evolving IoT landscape, with a particular focus on MTC gateways. By emphasizing crucial aspects like protocol interoperability, scalability, and latency, our implementation offers improved performance, reduced memory overhead, and enhanced control over hardware resources. Through comprehensive testing, we thoroughly assessed the efficacy, usability, scalability, and reliability of the TinyOneM2M solution in various scenarios. Our findings validate its suitability as a robust and efficient solution for meeting the demands of the IoT ecosystem.

Future endeavours encompass various areas where our implementation of OneM2M can be further refined and enhanced. Currently, there is no validation to ensure that the content of the request body is a JSON, therefore potentially leading to erroneous or inconsistent data in the requests. To fix this issue it's mandatory to ensure that the request contains a JSON object as the request body type.

Secondly, the current implementation doesn't fully guarantee the integrity and accuracy of the received data since only a few attributes of the standard's resources are being validated (e.g. `et`, `uri` and `rn`). Therefore, future work consists of performing attribute validation for all the resource's attributes passed in the JSON body of the requests.

During the implementation process, ensuring compatibility with Windows systems posed a persistent challenge, primarily due to the absence of the pthread library, which is specific to Unix systems. As a result, alternative libraries, such as pthreads-w32, would need to be employed when working with Windows systems. Consequently, a crucial future task is to develop an abstraction layer that enables cross-platform compatibility. By abstracting the thread implementation, we can enhance its portability and accessibility and ensure the execution of our implementation on diverse operating systems.

Moreover, given the extensive scope of the OneM2M standard, numerous resources have not yet been developed. Currently, only a partial implementation of the CSE_Base, AE, CNT, CIN, and SUB resources has been accomplished. Therefore, a vital area for future work is to complete the im-

plementation of the standard, ensuring that all the resources outlined in the OneM2M standard are fully developed. This undertaking is crucial to provide a more comprehensive and feature-rich implementation of OneM2M, enabling a broader range of functionalities and accommodating a wider array of use cases.

Lastly, TinyOneM2M is currently at the margin of having concurrency problems in a multi-threaded environment. Therefore, it would be advantageous to improve the transactional behaviour with WAL or other techniques to enhance concurrency. Additionally, our system could also be optimized at a database attribute level by performing deep analysis and study of the SQLite3 database operations and available operators such as PRAGMA.

Overall, these future enhancements will further solidify our implementation and ensure its effectiveness in the IoT ecosystem.

The Home Gateway Scenario, the Single Devices Scenario (P2P), and the Cloud Scenario are the three main scenarios supported by TinyOneM2M's system architecture. TinyOneM2M functions as a central gateway within a home network in the Home Gateway Scenario, facilitating communication between various devices. The scenario for single devices is a decentralized method in which each device has its own instance of TinyOneM2M, allowing for direct connection between devices without the need for a centralized communication hub. In the cloud scenario, TinyOneM2M is hosted by a cloud provider, enabling large-scale dispersed IoT applications by enabling device connectivity and internet communication.

Our TinyOneM2M solution implementation follows to version 5.1 of the OneM2M standard and concentrates on important resources including CSE_Base, AE, CNT, CIN, and SUB. The C programming language, which offers portability and efficiency for IoT applications, was used to construct the solution. However, it requires careful memory management and lacks built-in object-oriented capabilities. Also, the fact of in media our implementation covers 57% of the standard is very promissory. As the embedded single file database, we used SQLite3, which offered dependable data persistence and effective retrieval techniques. Performance was enhanced by the denormalized, single-table design, although for particular circumstances, other database options like MySQL or PostgreSQL might be taken into consideration.

Our tests yielded several noteworthy outcomes. The implementation of the TinyOneM2M solution proved to be highly reliable, effectively addressing the interoperability and communication demands within an IoT ecosystem. Interacted with various platforms, protocols, and devices, facilitating efficient resource management and few effort data transmission.

The solution showcased impressive scalability, exhibiting consistent performance even with an increasing number of connected devices and growing data traffic. This scalability is vital for accommodating the expanding IoT landscape, ensuring the system's ability to handle large-scale deploy-

ments and accommodate future growth. This solution has potential for enterprises looking for effective management of their IoT ecosystems as IoT continues to develop and grow.

Overall, TinyOneM2M's architecture offers a reliable and effective mechanism for putting the OneM2M standard into IoT contexts. It allows for seamless communication and data management in IoT applications by providing flexibility, scalability, and ease of integration into diverse scenarios. The TinyOneM2M solution we implemented shows a strong foundation and significant advancement towards the OneM2M standard. Platform independence, performance, and resource coverage can all be further improved and enhanced.

References

[1] Jaewoo Kim, Jaiyong Lee, Jaeho Kim, and Jaeseok Yun. M2m service platforms: Survey, issues, and enabling technologies. *IEEE Communications Surveys & Tutorials*, 16(1):61–76, 2014.

[2] G. Lawton. Machine-to-machine technology gears up for growth. *Computer*, 37(9):12–15, September 2004. Conference Name: Computer.

[3] M. Pticek, V. Cackovic, M. Pavelic, M. Kusek, and G. Jezic. Architecture and functionality in m2m standards. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 413–418, Opatija, Croatia, May 2015. IEEE.

[4] Yang Cao, Tao Jiang, and Zhu Han. A survey of emerging m2m systems: Context, task, and objective. *IEEE Internet of Things Journal*, 3(6):1246–1258, December 2016.

[5] oneM2M Sets Standards For The Internet Of Things & M2M.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, October 2000. Conference Name: Computer.

[14] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against dead-

locks. *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, pages 295–308, 2019.

[15] Sungmin Kim, Minseok Kim, Dam Quang Tuan, and Youjip Won. Atomic multi-database transaction of wal journaling mode in sqlite. In *2017 19th International Conference on Advanced Communication Technology (ICACT)*, pages 874–878, Bongpyeong, 2017. IEEE.

[16] Jaideep Vaidya and Chris Clifton. Privacy preserving association rule mining in vertically partitioned data. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 639–644, 2002.

[17] Yandong Mao, Eddie Kohler, and Robert Morris. Cache craftiness for fast multicore key-value storage. *EuroSys'12 - Proceedings of the EuroSys 2012 Conference*, pages 183–196, 2012.

[18] Guoliang Xing, Tian Wang, Weijia Jia, and Minming Li. Rendezvous design algorithms for wireless sensor networks with a mobile base station. *Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 231–239, 2008.