Entity Component System Architecture for Scalable, Modular, and Power-Efficient IoT-Brokers

Franc Pouhela*, Dennis Krummacker* and Hans D. Schotten*†

*Intelligent Networks Research Group, German Research Center for Artificial Intelligence (DFKI GmbH), D-Kaiserslautern.
Email: {franc.pouhela | dennis.krummacker | hans_dieter.schotten}@dfki.de
†RPTU University of Kaiserslautern-Landau, D-67663 Kaiserslautern.
Email: {schotten}@rptu.de

***Abstract*—This paper proposes a modular and scalable approach for implementing an Internet of Things (IoT) broker using the Entity-Component-System (ECS) architecture. The broker is designed to handle numerous sessions and topics without performance degradation, utilizing a publish/subscribe messaging model similar to Message Queue Telemetry Transport (MQTT). The approach simplifies the implementation of privacy, security, and trust measures in data exchange and offers a notable improvement in energy efficiency over a conventional open-source implementation.**

***Index Terms*—Internet of Things, Scalability, Power-Efficiency, Entity Component System**

I. Introduction

Scalability is a major challenge for IoT-systems, not only in terms of the number of interconnected devices but also in terms of the associated costs related to performance and energy efficiency. Although significant progress is made on hardware to improve memory usage, cooling, etc., there is still much that can be done at the software level to ensure that IoT-brokers operate as efficiently as possible.

One approach to addressing this challenge is to carefully consider the design of the software. By adopting a well-structured, modular design that separates concerns and minimizes dependencies, it is possible to achieve greater scalability and performance. Another important consideration is the use of optimized algorithms and data structures, which can help to reduce computational overhead and improve response times.

ECS [1], [2] or Entity Component System, is a software architecture design pattern mostly utilized in the development of video games for the representation of in-game objects, which is different from the common Object-Oriented Programming (OOP) technique. Its main function is to separate data from behaviors to promote code reuse and cache-friendliness and enhance performance.

ECS provides a powerful and flexible way to manage data and device sessions by breaking down complex parts into modular components that can be easily assembled and reused. The proposed ECS approach provides several benefits, including flexibility, scalability, performance, and power efficiency. Additionally, it provides a highly performant architecture that can handle real-time analytics, which is critical for IoT systems.

In this paper, we showcase how we harnessed the versatility of ECS to successfully implement a customized publish/subscribe broker. Our approach yields substantial improvements when compared to a well-established open-source reference implementation. The paper begins by presenting some related works in Section II. Section III, provides a thorough overview of the ECS architecture and how we leveraged it in our study. Section IV presents some benchmark analysis to evaluate the effectiveness of our approach. Finally, Section V concludes the paper with a discussion of potential future research directions.

II. Related Work

One of the most popular approaches to scaling the communication between IoT devices is the use of Message-Oriented Middleware (MOM) using a publish/subscribe [3] messaging model for event-driven communication between IoT devices. Examples of MOM solutions include protocols such as: MQTT [4], Advanced Message Queuing Protocol (AMQP), and Extensible Messaging and Presence Protocol (XMPP).

Stony Brook University conducted a study to investigate a broker architecture that is both highly resilient and scalable [5]. The study proposed Nucleus, a container architecture that can support scaling to a large number of IoT devices, offers high resiliency against failures, and involves low overhead when it comes to data transfer between devices and the broker.

A study was undertaken to explore the potential of leveraging IoT brokers and the publish/subscribe messaging model in enabling a Context-Management (CoMa) architecture for facilitating decoupled acquisition and distribution of information in Next-Generation Mobile Networks. The study aimed to shed light on the potential benefits and challenges associated with adopting this approach, paving the way for advancements in the field of Next-Generation Mobile Networks. [6].

In [7], the authors proposed a novel ECS-based approach for managing the energy consumption of IoT devices in smart buildings. Their system used a distributed ECS architecture to dynamically allocate resources and optimize energy usage based on real-time occupancy data.

Finally, another approach similar to ECS known as Entity-Component-Attribute (ECA) was also proposed in [8] to keep the changeable nature of large-scale IoT applications such as smart cities maintainable. The research presents an automated mapping to a structure compliant with the Linked Data Platform W3C standard.

III. ECS ARCHITECTURE

As stated in section I, ECS or Entity Component System is a software design pattern generally used for building complex and scalable software systems, particularly in the field of game development. It promotes code reuse by separating data and behavior. Performance is benefited from the frequent use of cache-friendly data storage techniques. It has gained popularity in game development due to its ability to handle large and complex data sets, its flexibility in handling different types of entities and components, and its support for parallel processing.

Flecs [9], EnTT [10], and EntityX [11] are prominent open-source C/C++ ECS libraries widely recognized and freely available. For our case study, we chose EnTT as our preferred ECS library. EnTT is a header-only, lightweight library developed using modern C++, offering excellent performance.

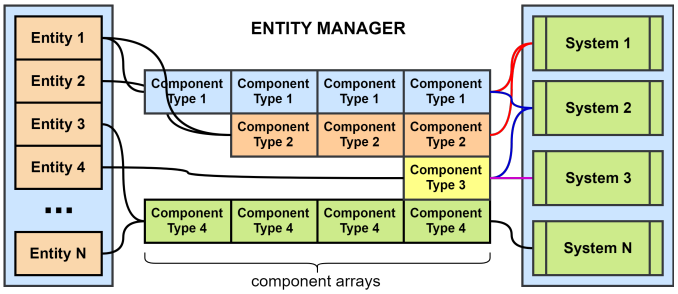

Figure 1. High-level architecture of ECS

A. Characteristics of ECS

Figure 1 illustrates the architecture of ECS, which can be characterized as follows:

1) It has Entities, Components, and Systems: It has Entities. An entity stands in for a generic object. Every single object is represented as an entity. Typically, it only consists of a unique identifier, a simple 32-bit or 64-bit integer. This id helps to identify the owner of specific components.

Components model an entity's behavior by storing necessary information. For example, in IoT, each client needs a connection. Components are simple structures with data fields and no inherent logic.

Entities possess the capability to be composed of one or more components. This feature is of paramount importance as it allows for the generation of an infinite array of behaviors through the addition or removal of specific components to and from entities.

Entities possess the capability to change their components at runtime, which is of particular significance in real-time communication contexts, such as that of the IoT.

Behavior is implemented through systems that target specific entities with defined components. Separating systems is advantageous because it allows focusing on relevant entities and facilitates multi-threading.

2) Cache-Friendly: The ECS architecture is designed to be cache-friendly, meaning that it can take advantage of the cache hierarchy of modern computers to improve performance. This is achieved by ensuring that frequently accessed data is stored in a way that allows the cache to be utilized effectively.

3) Provides Composition: The concept of composition, as applied in ECS, involves the aggregation of multiple components to define the properties and behavior of an entity, as opposed to utilizing inheritance from a parent object. The use of composition allows for increased versatility in the creation of entities, as new features can be added through the simple addition of components, without the need for modifications to a hierarchical structure. This approach also serves to mitigate the potential for complex class structures, as entities only encompass the components deemed necessary.

The code snippet in Listing 1 shows how an entity with multiple components can be created using EnTT as well as how to iterate over entities with a specific set of components.

```
struct Component1 {
    float Data;
};

struct Component2 {
    int Data;
};

EntityID entity1 = registry.create();
registry.emplace<Component1>(entity);
registry.emplace<Component2>(entity);

registry.view<Component1>([] (auto entity) {
    auto& comp1 = registry.get<Component1>(entity);
    // perform logic on component
});

registry.view<Component1, Component2>([] (auto entity) {
    auto& comp2 = registry.get<Component2>(entity);
});
```

Listing 1. ECS code snippet

B. Summary of Advantages and Disadvantages of ECS

Understanding both the benefits and the limitations of an approach is crucial for making informed decisions about its integration. Following are some relevant advantages and Disadvantages of ECS:

+ ECS helps create shorter, and less complicated code and enables a clean design that employs decoupling, modularization, and reusability of components.
+ Has highly flexible and scalable behavior and fits well with unit testing and mocking.
+ Good for dynamic, real-time systems and fits well with parallel processing.
- Hard to implement: To implement a custom ECS framework is quite tedious. It requires a good understanding of computer internal operations. Open-source implementation can alleviate this weakness.
- Lack of familiarity: ECS is not as widely recognized and adopted as other software development paradigms because it was mainly created for the gaming industry.
- Limited use cases: ECS may not be suitable for all types of projects or applications and may not be the best fit for systems that require a more traditional object-oriented design.

IV. Implementation Evaluation

In this section, we present how the use of ECS as the underlying architecture has impacted the implementation of our custom IoT broker application. We also evaluate the performance of our approach compared to a well-established open-source implementation reference.

As part of our study, we undertook the development of a publish/subscribe-based IoT broker application called: 'NetMQ-Broker'. the application was implemented using C++17 and was carefully architected using EnTT as ECS framework. To handle I/O operations with utmost efficiency, we integrated the standalone version of Asio [12], a highly performant C++ networking library.

Our study aimed to explore the capabilities and potential of this implementation approach. Therefore, we conducted thorough evaluations and measurements to assess its performance, scalability, and reliability under different scenarios, providing valuable insights into its suitability for real-world deployment in various contexts within the IoT domain.

A. Architecture and Communication

IoT-brokers generally consist of four main layers. The first layer, device connectivity, enables the connection of IoT devices to the broker and facilitates communication between them. The data processing layer filters, processes, and transforms data received from IoT devices. Processed data is stored in the data storage layer for later use by applications. Finally, the application interface layer exposes interfaces for applications to interact with the broker and access data. While our implementation does not include all functionalities of each layer described above, our study primarily focused on real-time communication between devices with regard to scalability and energy efficiency.

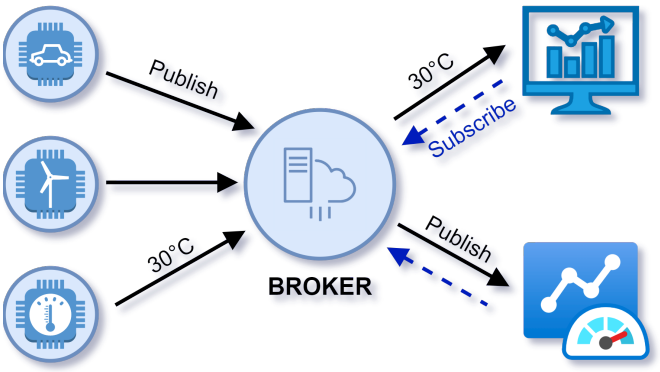

Figure 2. publish/subscribe messaging model

Figure 2 describes the publish/subscribe messaging model used in our implementation to enable data exchange between multiple clients. This model is similar to the one used in MQTT brokers. It ensures that the communication between the devices is scalable and decoupled. A device only needs to know the broker's address and the topic of interest to subscribe to, while the broker takes care of managing the communication between the devices. This architecture makes it easy to add or remove devices from the network, making it a suitable model for IoT systems with large numbers of devices.

The sequence diagram in Figure 3 describes the data flow between a publisher and a subscriber using the MQTT protocol. A client connected as a publisher to the broker can publish a message on a topic. The broker receives the data from the client and stores it in its message queue because there is no subscriber to forward it to. Another connected client then subscribes to that topic to receive messages. The broker then sends the last message published on that topic to the new subscriber. Going forward, the broker will forward any new message published on that topic to the subscriber automatically. The queuing of the message is optional.

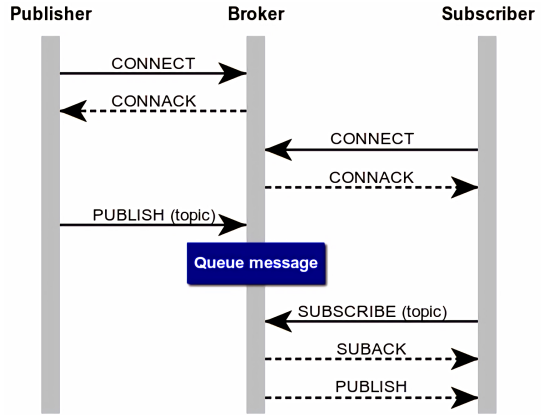

Figure 3. MQTT sequence diagram

B. Evaluating the Impact of using ECS

This paper does not aim to suggest that the ECS design pattern is a panacea that solves all challenges faced in IoT management. However, it endeavors to demonstrate that using ECS as the underlying architecture to develop IoT broker has the potential to significantly enhance the performance and reduce the overall cost of IoT management.

1) Session Management: Effective session management is crucial for the design and implementation of IoT-brokers, which need to facilitate connectivity between a wide range of devices, applications, infrastructures, and other entities. The ECS approach enables each client to be represented as an entity with a dedicated session component.

```
struct SessionComponent {
    //...
    Flags Flags;
    Socket Socket;
};

void OnConnect(Socket socket) {
    EntityID client = registry.create();
    registry.emplace<SessionComponent>(client, socket);
}
```

Listing 2. Adding new Connection

Entities are highly versatile objects that can be modeled dynamically, allowing for different types of sessions to be created and managed, even at runtime. This level of flexibility

makes it possible to support multiple sessions for a single client, without compromising the overall system performance. By prioritizing session management and leveraging the power of ECS, IoT-brokers can better handle the complex demands of modern interconnected environments. All it takes to extend the behavior of an entity is to determine which components are needed and how a system should interact with them. This is depicted in the code snippet in Listing 2.

2) Topic Management: A topic is a string used to identify the subject of a message that is being published. The broker is responsible for maintaining a registry of all topics and subtopics. Managing topics presents challenges in maintaining a scalable and organized system, such as maintaining a clear and consistent naming convention, managing subscriptions, and ensuring optimized broker performance.

To overcome these challenges, it is crucial to design a robust messaging architecture with efficient topic management. MQTT is currently the most popular protocol in the IoT industry. It uses a hierarchical naming structure for topics. Topics are organized into a tree-like structure, where each level of the hierarchy is separated by a forward slash ("/") e.g., *'building1/sensor10'*.

```
// topic data
struct TopicComponent {
    std::string Name;
};

// subtopic data
struct SubtopicComponent {
    std::string FullName;
    EntityID Parent;
}

// topic suscribers list
using Subscribers = std::set<EntityID>;

//Add new topic entity
EntityID topic1 = registry.create();
registry.emplace<Subscribers>(topic1);
registry.emplace<TopicComponent>(topic1, "top1");

//Add a subtopic entity
EntityID subtop1 = registry.create();
registry.emplace<Subscribers>(subtop1);
registry.emplace<TopicComponent>(subtop1, "subtop");
registry.emplace<SubtopicComponent>(subtop1, topic1);

// Callback when a new message is read
void OnMessage(EntityID from, Message&& msg) {
    // get topic' subscribers list
    auto& subview = registry.get<Subscribers>(msg.Topic()
        );
    // forward message to subscribers
    for(auto subscriber : subview) { Write(subscriber,
        msg); }
}
```

Listing 3. Adding new Topics

One major downside of this approach is the fact that every message sent by the client or the broker must contain the full topic string as part of the header. The topic string is then parsed by the broker before the message is forwarded to subscribers. A recent solution proposed to address this issue is to assign an alias to a topic, which would be a 32-bit integer identifier, that can be used while sending messages. However, the topic alias can only be negotiated during connection establishment and there are some rules to follow when requesting a topic alias from the broker which adds up the requirement in terms of implementation. Additionally, the topic alias is not used by all clients, as only the clients who requested the alias can use it to send and receive messages.

In contrast, our ECS approach represents topics and subtopics as entities, each with a specific set of components. To know if a topic is a subtopic, we can simply check if the entity has a *'SubtopicComponent'* as shown in the code snippet in Listing 3. Each subtopic knows about its parent identifier and each topic also has another component that stores a list of its subscribers. Since topics are just entities with a 4-byte unique identifier, we can just leverage the entity identifier as the topic alias as part of the message header. This approach enables our brokers to manage topics in a flexible and scalable manner by taking away the need to parse the topic name from messages because the size of topic names can be quite large.

The sequence diagram in figure 4 is a depiction of how topic identifiers are provided to clients as they request it from the broker during connection establishment or through a topic request. The Publisher initiates the communication by sending a connection request (CONREQ) to the Broker, specifying the topic "topic1" it wants to publish to. The Broker responds with a connection acknowledgment (CONACK), providing a topic identifier of 5. Then, the Subscriber does the same but without a topic specified in the (CONREQ) packet. Next, the Subscriber requests to subscribe to "topic1" by sending a topic request (TOPREQ), and the Broker responds with a topic acknowledgment (TOPACK) and assigns the topic identifier 5 to the subscription. Finally, the Publisher sends a publish request (PUBREQ) to the Broker, indicating the topic identifier 5, and the Broker forwards the publish request to the Subscriber.

This process ensures that the topic name will not be needed when publishing messages, but only the topic identifier which is always 4 bytes.

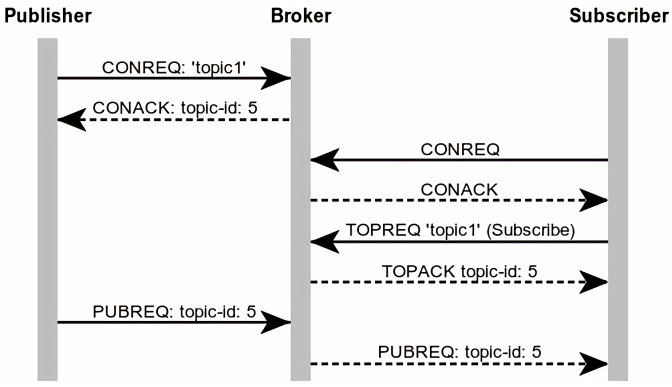

Figure 4. Topic management and message forwarding

3) Bandwidth Usage: The utilization of topic identifiers, instead of strings, is a highly effective approach that can significantly reduce the amount of metadata transmitted in messages. This approach also eliminates the need for parsing to extract the topic name for each transaction, thereby reducing processing time. By assigning each topic a unique number, each

message contains a fixed header, providing a more structured and streamlined data transmission process.

This approach is particularly useful in high-frequency data exchange scenarios such as Vehicle-to-everything (V2X) [13], [14] communication, where thousands of devices, such as cars and traffic lights, are connected, and deep topic structures for layers and geolocation are needed. By using topic identifiers instead of strings, bandwidth usage can be drastically reduced.

For instance, a complex topic string such as *'Europe/Germany/Kaiserslautern/City-Center/Bismarck-Street/Traffic-Light2'* requires **72 bytes**, while a topic identifier such as "2324" as an integer requires only **4 bytes**. Around 68 bytes of extra bandwidth is used to send a message.

Considering 10,000 devices exchanging messages at a rate of 10Hz, the total extra bandwidth usage for a topic string of that length would be approximately 6 MB/s, equivalent to around 16 terabytes per month per device. This calculation does not even take into account message forwarding, which means the broker would still need to publish the messages to all subscribers. This can lead to an exponential increase in bandwidth usage.

4) Security and Trust: Security and trust will always be crucial concerns when it comes to networking. One of the extra benefits besides the performance and the modularity of our ECS approach is the fact that its inherent composition nature simplifies the implementation of security and trust measures within the broker. For example, as shown in Listing 4 we can add an optional *'AccessComponent'* to topics, to ensure that only clients with the appropriate signature can publish and/or subscribe to them. This could help to detect potential threats from untrustworthy clients.

```
// topic access data
struct AccessComponent {
    std::string Key;
    NetFlags Level;
}

// client trust score
struct TrustComponent {
    uint32_t Score;
}

// Callback to subcribe to topic
void Subscribe(EntityID client, std::string& topicName) {
    EntityID topic = FindTopic(topicName);

    // retrieve topic access data
    auto& access = registry.get<AccessComponent>(topic);

    // retrieve client signanture
    auto& sig = registry.get<ClientSignature>(client);

    // check if client has rights
    if(CanSubscribe(sig, access)) {
        //subscribe client to topic
    }
    else {
        // update client score
        registry.get<TrustComponent>(topic).Score++;
    }
}
```

Listing 4. Topic access safety

We have defined three distinct access levels for topics. At **Access Level 0**, all clients are granted unrestricted access, allowing them to publish and subscribe to the topic without any limitations. Moving up to **Access Level 1**, clients are required to provide a key for authentication when attempting to publish or subscribe. This key is then compared by the broker with the topic's password to ensure proper authorization. Lastly, **Access Level 2** introduces an additional layer of security. Clients must provide a password to the broker, which is subsequently forwarded to the topic's owner. The owner then verifies the password to authenticate the client before granting access to publish or subscribe to the topic. These access levels enable varying degrees of control and security in managing topic-based interactions within the system.

Furthermore, we could assign a trust score component to clients that would maintain a score reflecting their behavior throughout their session lifespan. If, for instance, a client attempts to send a message using a topic identifier to which it is not authorized, the broker will increment the score based on a defined policy and, if necessary, terminate the client session.

C. Benchmark Results

To provide context for the benchmark results, we conducted three measurement rounds, each with an increasing number of connected clients. In all rounds, each client subscribed and published a 1K payload every second on a common topic, allowing all clients to send and receive messages to and from one another.

The measurements were performed using an *11th Gen Intel Core i7-11800H @ 2.30GHz* processor, and the Intel Power Gadget [15] which is a software tool provided by Intel that monitors real-time power usage, temperature, frequency, and utilization of Intel processors was used to measure the CPU's power consumption. The throughput measurements were made using the system's built-in Resource Monitor on Windows 11.

The second broker (Eclipse Mosquitto) used for reference to our implementation (NetMQ) is an open-source (EPL/EDL licensed) message broker that implements the MQTT protocol versions 5.0, 3.1.1, and 3.1. It is implemented in C using a modular software design, with various components responsible for different functionalities. Mosquitto is lightweight and is suitable for use on all devices from low-power single-board computers to full servers. It also provides a C library for implementing MQTT clients.

The charts depicted in the figures 5, 6, 7 reveal insightful comparisons between the two IoT brokers. When examining the write speed, NetMQ consistently outperforms Mosquitto across all client numbers, exhibiting a significant superiority of approximately 112%. We find it crucial to point out the fact that the low write speed exhibited by Mosquitto is due to load balancing (message dropping). We didn't implement any message-dropping mechanism in NetMQ, in order to measure the cost of not having it compared to Mosquitto.

For read speed, both brokers demonstrate comparable performance, with slight variations observed between different client numbers. Moving on to power consumption, Mosquitto tends to consume more power than NetMQ, albeit the difference is not substantial, with a marginal variance of approximately 35%. The notable aspect of NetMQ outperforming Mosquitto in terms

of data sent while consuming less power can be attributed to several factors. Firstly, the underlying architecture and design choices. NetMQ uses ECS which as presented earlier has a lot to add when scaling is an important factor, enabling it to handle a higher volume of data transmission with improved efficiency. Secondly, NetMQ uses an optimized communication protocol to minimize redundant data transfers. By reducing unnecessary data duplication or optimizing packet sizes, NetMQ can achieve higher data throughput while utilizing fewer system resources. Furthermore, it's important to consider that power consumption is influenced by various factors beyond data transmission, such as the processing overhead of the broker itself.

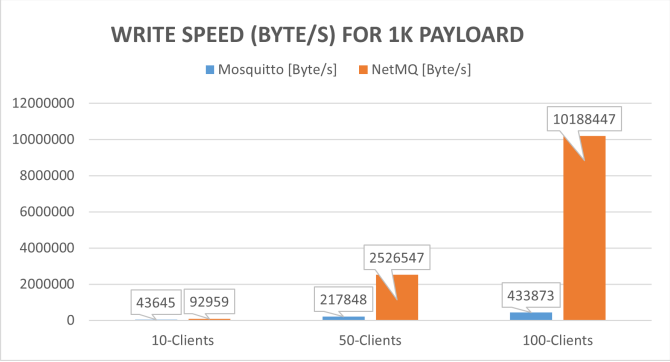

Figure 5. Comparison of write speed in byte/s

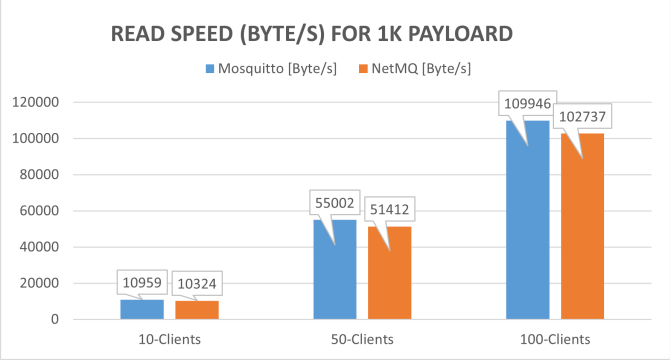

Figure 6. Comparison of read speed in byte/s

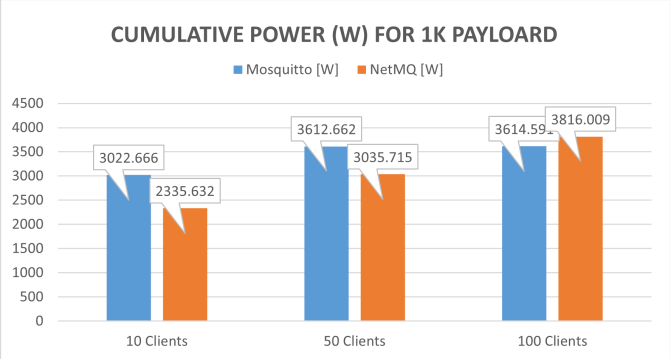

Figure 7. Comparison of power consumption in watt

V. Conclusion and Future Work

In conclusion, the ECS architecture has the potential to greatly improve the design and implementation of IoT-brokers. As we have seen, it can improve performance, bandwidth usage, power efficiency, topic management, security, and trust measure implementation by providing a flexible and scalable framework for managing data.

Future work should focus on conducting additional benchmarks to compare the performance and efficiency of our ECS-based implementation against other existing architectures. This can include tests for data processing and transmission, scalability, and security measures. The results of these benchmarks will provide valuable insights into the real-world capabilities of ECS in IoT-broker implementation and will help determine the feasibility and practicality of adopting ECS as a standard architecture for IoT-brokers.

Acknowledgment

The authors acknowledge the financial support by the German *Federal Ministry for Education and Research (BMBF)* within the project »Open6GHub« {16KISK003K}.

References

[1] P. Torgerson, "Entity-component-systems," in *Proceedings of the 2015 World Congress on Computer Science and Information Engineering*, vol. 2, 2015, pp. 508–512.

[2] D. Masiukiewicz, D. Masiukiewicz, and J. Smołka, "Research of an entity-component-system architectural pattern designed with using of data-oriented design technique," *Journal of Computer Sciences Institute*, vol. 13, pp. 349–353, Dec. 2019. [Online]. Available: https://ph.pollub.pl/index.php/jcsi/article/view/1331

[3] S. C. L. Hernandes, M. E. Pellenz, and A. Calsavara, "A study on publish-subscribe middlewares for selective notification delivery in smart cities," in *2019 XLV Latin American Computing Conference (CLEI)*, 2019, pp. 1–10.

[4] J. Roldán-Gómez, J. Carrillo-Mondéjar, J. M. Castelo Gómez, and S. Ruiz-Villafranca, "Security analysis of the mqtt-sn protocol for the internet of things," *Applied Sciences*, vol. 12, no. 21, p. 10991, 2022.

[5] S. Sen and A. Balasubramanian, "A highly resilient and scalable broker architecture for iot applications," in *2018 10th International Conference on Communication Systems & Networks (COMSNETS)*, 2018, pp. 336–341.

[6] F. Pouhela, D. Krummacker, and H. D. Schotten, "Towards 6G Networks," in *A Context Management Architecture for Decoupled Acquisition and Distribution of Information in Next-Generation Mobile Networks*, ser. ITG, vol. 157, VDE. IEEE, 5 2023.

[7] P. García, R. Alcarria, D. Sánchez-de Rivera, and T. Robles, "Improving energy efficiency in smart buildings with an entity-component-system approach," *Sensors*, vol. 19, no. 20, p. 4553, 2019.

[8] T. Spieldenner, R. Schubotz, and M. Guldner, "Eca2ld: Generating linked data from entity-component-attribute runtimes," in *2018 Global Internet of Things Summit (GIoTS)*, 2018, pp. 1–4.

[9] "https//github.com," https://github.com/SanderMertens/flecs Flecs.

[10] "https//github.com," https://github.com/skypjack/entt EnTT.

[11] "https//github.com," https://github.com/alecthomas/entityx EntityX.

[12] "Asio c++ library," https://think-async.com/Asio/ Asio.

[13] S. Gyawali, S. Xu, Y. Qian, and R. Q. Hu, "Challenges and solutions for cellular based v2x communications," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 1, pp. 222–255, 2021.

[14] K. Abboud, H. A. Omar, and W. Zhuang, "Interworking of dsrc and cellular network technologies for v2x communications: A survey," *IEEE Transactions on Vehicular Technology*, vol. 65, no. 12, pp. 9457–9470, 2016.

[15] B. Prieto, J. J. Escobar, J. C. Gómez-López, A. F. Díaz, and T. Lampert, "Energy efficiency of personal computers: A comparative analysis," *Sustainability*, vol. 14, no. 19, p. 12829, 2022.