Novo Diaz, Oscar

A Framework for Access Coordination in IoT

Published in:
2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)

DOI:
10.1109/SOCA.2018.00027

Published: 01/11/2018

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Please cite the original version:
Novo Diaz, O. (2018). A Framework for Access Coordination in IoT. In *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)* (pp. 129-136). (IEEE International Conference on Service-Oriented Computing and Applications). IEEE. https://doi.org/10.1109/SOCA.2018.00027

A Framework for Access Coordination in IoT

Oscar Novo
Ericsson Research, Finland
Aalto University, Finland

***Abstract*—IoT systems have typically been designed with the assumption that all their resources are available on a concurrent access basis. Conversely, some IoT scenarios only operate correctly if the access to certain resources is given in mutual exclusion fashion and limited to a confined number of nodes. For example, it might not be appropriate that a intrusion detection system prevents a fire suppression service to unlock the doors of a building.**

This article proposes a service-oriented, resource scheduling framework for IoT systems. The framework is based on the *Binary Floor Control Protocol*, a protocol designed for managing shared resources in conference systems. The article introduces the design model of the framework and presents an implementation of it. Based on this, the performance of the framework is evaluated and compared with the current state-of-the-art HTTP solutions for references. In addition, the framework is evaluated with respect to its suitability to be used in IoT. The results confirm that our approach is suitable for IoT environments and achieve good scalability.

***Index Terms*—IoT, Internet of Things, Shared Resources, BFCP**

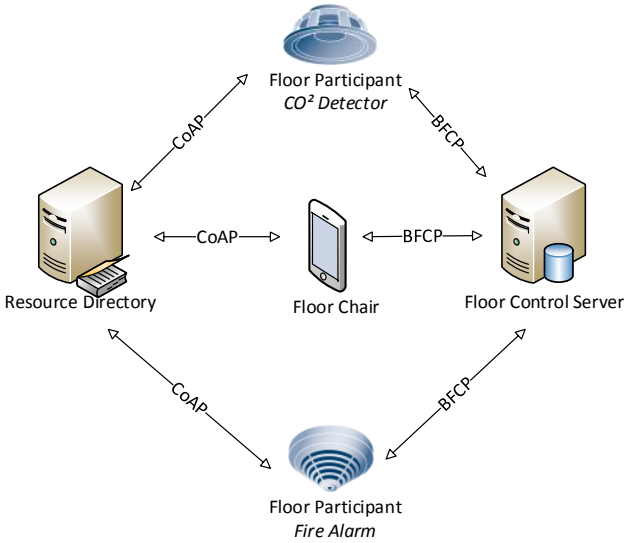


Fig. 1: Overview of the Logical Entities of the Framework

I. Introduction

Since its early days, one of the main goals of the Internet of Things (IoT) has been to provide connectivity to an entirely new set of devices [1] such as digitalized assets, equipment, vehicles, and processes in factories. As a result of this, home appliances, wearables, vehicles, and industrial electronics have become increasingly more connected to the Internet. Furthermore, the Internet of Things has been built on the assumption that the interaction and communication in IoT can always be done in a concurrent manner [2]. Contrarily to this claim, however, there are some specific IoT scenarios where the mutual exclusion of the information is essential for achieving valid outcomes.

For instance, it is of paramount importance that a fire alarm controls the access of a front door lock to provide an escape route for the residents of the building in the event of a fire. On the other hand, other IoT services, such as the intrusion detection system of the building should be prevented from simultaneously controling that front door lock.

Finding a solution to this problem is definitely not a simple matter. One possible solution could be to store a local policy in every IoT device. As a result, an IoT device would automatically check its local policy before granting each specific request. However, some types of IoT devices, in particular *class 0* and *class 1* devices, cannot implement their own local policies due to their limited capabilities. According to [3], *class 0* and *class 1* devices have very limited memory and processing power. They are, therefore, only capable of performing simple operations and communicating on the Internet using a constraint protocol stack. Despite that, the rest of the IoT devices (*class 2* and *class 3* devices) are less constrained and could implement their own local policy. However, many IoT scenarios are very dynamic making this possibility very unlikely. Basically, every time a new device is added or deleted from the system, the local policy of each IoT device belonging to that system has to be updated accordingly. In consequence, this option is no longer feasible for an IoT system with large number of devices.

This article presents a resource scheduling framework for IoT as a solution to this problem. The framework is based on the *Binary Floor Control Protocol* (BFCP) [4], a protocol to manage joint or exclusive access to the audio and video resources of a conference system [5]. Using the *Binary Floor Control Protocol*, participants of a conference can automatically request to use the audio and video of that conference reducing the switching delay between speakers and improving the interactivity.

Even though the *Binary Floor Control Protocol* has been designed for conference systems, it is suitable for service-oriented IoT scenarios due to its low overhead. The binary encoding achieves a small message size, reducing the memory, CPU and battery consumption in constrained devices. Besides, it can run over UDP, incurring in fewer delays and retransmis-

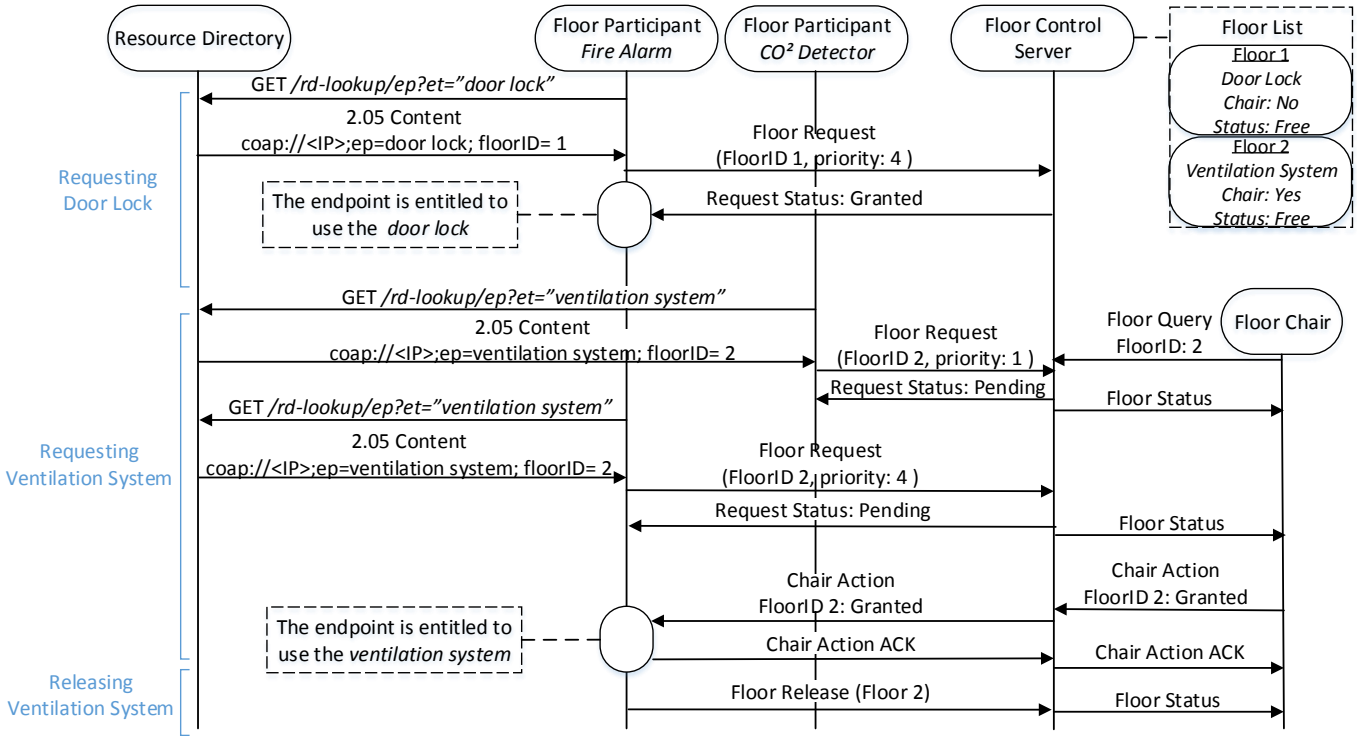


Fig. 2: Discovery, Retrieval, and Release of Floors

sion packets.

The article, additionally, describes the implementation of the framework. The last part of the article evaluates the performance of our implementation and the suitability to use the *Binary Floor Control Protocol* in our framework.

For convenience, a new term, *floor*, is introduced to describe a temporary permission to access or manipulate a specific shared resource or set of resources. The term has been adopted from the *Binary Floor Control Protocol* documentation [4], and it will be used throughout the article.

The rest of the article is organized as follows: the related work is briefly discussed in Section II, Section III describes in detail the different components of the framework and their interfaces. The implementation of the framework is described in Section IV, and Section V evaluates it. Finally, section VI consists on concluding remarks.

II. Related Work

Several research work have studied the management of complex IoT systems [6]–[9]. In particular, these approaches facilitate the configuration of complex smart environments for end users by combining services that are provided by IoT devices. Thus, an end user does not need to understand a complex system. The system can automatically deduce the different interactions between the different IoT devices. For instance, [10] and [11] respectively introduce a technique to facilitate the composition of heterogeneous services for end users in smart homes. Similarly, [12] presents a system for rapid reconfigurability in factory automation with the objective of evolving and adapting to mass customization.

However, our work fundamentally differs from those studies in several aspects. Firstly, our research focus on managing the access to share resources in IoT systems to prevent different IoT services executing opposite actions in the same resource at the same time. Secondly, our approach enables arbitrary and independent IoT services to successfully operate between them. Lastly, our approach can be combined with the existing IoT composition systems, contributing to atomically control the access of the resources among different IoT services.

III. Proposed Architecture

In order to coordinate the access of different IoT resources in a network, we have implemented a new framework. The focus in this section is to describe this framework in more detail and highlight the different design decisions that have been taken during its development.

A. Components

The architecture can be divided into three logical entities: *floor participants*, *floor chairs*, and *floor control servers*. Figure 1 depicts an example that combines the different logical entities together. On a high level, the example contains the following components: a single *floor control server*, two *floor participants*, and a *floor chair*. These components communicate in a centralized fashion through the *Binary Floor Control Protocol* [4].

The *floor control server*, in addition to providing information about the different participants in the system, maintains the state of the floors, including which floors are in the IoT system and who holds them. *Floor participants* are IoT devices that request floors and information about the floors from the *floor control server*, while *floor chairs* are logical entities that manage the floors and grant, deny, or revoke them according to the situation. Although the example shows one single *floor chair*, many *floor chairs* managing several floors simultaneously can be included in the system.

Floor participants need to know which resources are associated in a network domain. They can obtain this information by using an entity called *Resource Directory*. A Resource Directory [13] is an IoT component used to discover IoT resources held on a network. Usually, sleeping IoT nodes and networks with inefficient multicast traffic prevent direct discovery of those resources. Thus, nodes can perform direct IP address lookups of those resources using the Resource Directory. The communication with the *Resource Directory* is done through the Constrained Application Protocol [14].

B. System Interactions

In continuing with our example, Figure 2 provides a detailed representation of the different iterations between the different components of the framework. In the figure, a fire alarm has presumably detected a fire in the building and, among other actions, needs to unlock the front door of the building to provide an escape route for the residents of the building and allow a quicker response from the firefighters. First, the fire alarm needs to know the IP address and port of the front door lock. We assume that the fire alarm does not know that information and needs to look it up in the Resource Directory. The Resource Directory responds with the information of the door lock and includes in the response message a new parameter, called *floorID*, that indicates if the access to that particular result is managed by our framework. Generally, not all IoT resources are necessarily managed by the framework. However, for simplicity, this example only focuses on a subset of managed IoT resources.

Once the necessary information of the front door lock is gathered, the fire alarm first contacts the *floor control server* to request permission to access the door lock. In addition, the fire alarm asks the *floor control server* to handle the floor request with the highest priority (priority: 4). Currently, the front door lock does not have any *floor chair* assigned to it, and it is up to the policy implemented in the *floor control server* to grant or deny the request. In this particular case, access to the door lock is immediately granted by the *floor control server*. In the hypothetical case that the front door lock was already granted to another node, the *floor control server* could revoke that access and give it to the fire alarm instead.

In the meantime, the CO^2 detector installed in the same building has detected high levels of carbon dioxide and requests the control of the ventilation system to automatically start the building's ventilation fans. As in the case of the fire alarm, the CO^2 detector first discovers the IP address and port

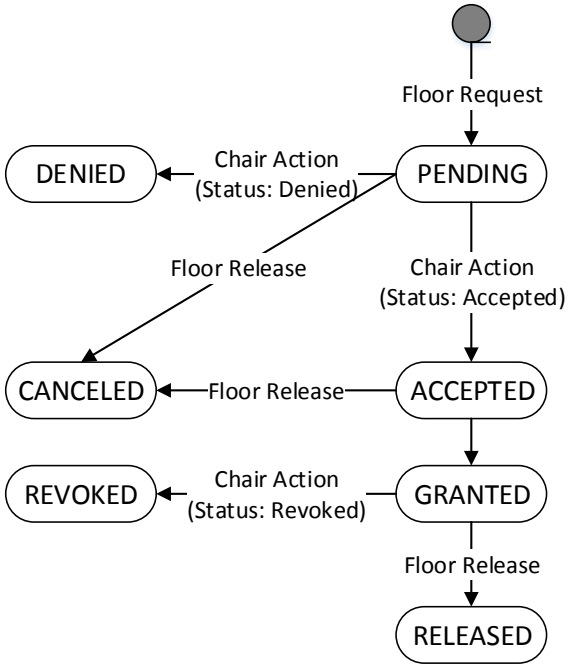


Fig. 3: Floor State Machine

of the ventilation system and requests permission to access it to the *floor control server*. In such case, the ventilation system is managed by a *floor chair*. Hence, the *floor control server* forwards the request to this specific *floor chair*. The action taken by the *floor control server* for pending requests is a matter of its local policy. A *floor control server* may perform its own decision automatically if a decision is not made by the *floor chair* after a certain period of time.

Similarly, the fire alarm also needs to have access to the ventilation system of the building to control the smoke created by the fire. The fire alarm repeats the same steps as before and first discovers the location of the ventilation system and, later, requests the permission to the *floor control server*. However, since the floor request of the fire alarm has higher priority (priority: 4) than the request of the CO^2 detector (priority: 1), the *floor control server* prioritizes the floor request of the fire detector and informs the *floor chair* about that. Afterwards, the *floor chair* decides to grant the access permission to the fire alarm. Later, once done with it, the fire alarm releases the floor of the ventilation system. In this particular case, the CO^2 detector controling the ventilation system may be dangerous, since injecting more oxygen to the building might cause the fire to spread quickly. Therefore, the *floor chair* could prevent it by denying the floor of the ventilation system to the CO^2 detector (not shown in Figure 2).

C. Message Processing in the Floor Control Server

As explained before, *floor control servers* are logical entities that maintain the state of the floors in an IoT system.

When a *floor participant* requests a floor using the *floor request* operation, the *floor control server* stores this informa-

tion in the system and assigns the *pending* state to the floor. Pending requests are the requests that have not been authorized by a *floor chair*. When the *floor chair* of a floor authorizes a request, the floor is moved from the *pending* to the *accepted* state. In case the *floor chair* denies this floor's access to the *floor participant*, the floor is removed from the system. If there is no chair assigned to a floor, this floor can be moved directly to the *accepted* state or deleted from the system based on the *floor control server*'s local policy. A floor in the *accepted* state will eventually move to a *granted* state.

A floor is also removed from the system when the *floor participant* assigned to it releases the floor or a *floor chair* revokes it. In addition, when a floor is released from a particular request, all the floors assigned to that request are also released.

Figure 3 describes the different states (squares) of a floor and the transitions between them (arrows). A floor has seven states: *pending*, *accepted*, *granted*, *denied*, *canceled*, *revoked*, and *released*. The state of a floor changes based on the inputs it receives.

Only the reception of a new *floor request* message in the *floor control server* can create a floor. On the other hand, a floor is removed from the system when it is in the *denied*, *canceled*, *revoked*, or *released* state. During its natural life cycle a floor is created in the *pending* state, moved to the *accepted* state at some point of its life cycle, and destroyed after the *granted* state has been achieved.

IV. Implementation

Based on our study, a prototype of the framework was implemented. In order to easy the implementation of the prototype, the solution did not use a *Resource Directory* to discover the resources of the IoT network. By contrast, the information of every resource was hard-coded separately in each *floor participant*. This design decision did not affect the functionality of the prototype. The rest of the components in the framework were implemented in the C language.

In addition, we measured the source lines of code (SLOC) of the *floor participant* and *floor control server* components using the *SLOCCount* tool[1]. The result was 4,288 and 7,492 lines of code respectively. The *Binary Floor Control Protocol* was implemented following the standards [4].

A. Handling Floors in the Floor Control Server

Different approaches can be used to manage floors inside the *floor control servers*. In this respect, a *queue* data type seems to be a good choice of design, since the floors, as well as the queues, are handled in order of addition.

Queue data types are linear structures where each element of the queue is inserted and removed according to the first-in first-out (FIFO) principle. In the FIFO principle, the first element added to the data structure will be the first one to be removed. Queue data types are normally used when the elements in a system have to be processed in the order they have been added. The design model chosen in the implementation of the *floor control server* is depicted in Figure 4.

[1] https://dwheeler.com/sloccount/

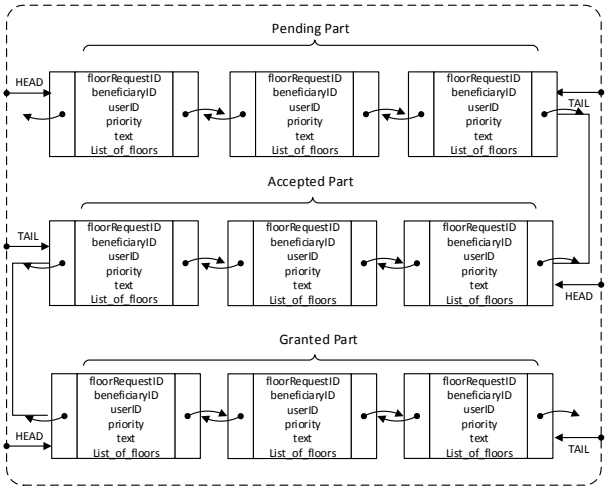


Fig. 4: Floor Management Design

The chosen design defines a global list for each floor request. This queue stores the information of the *pending*, *accepted*, and *granted* states of the floor state machine. The global queue structure is divided into three parts: *pending part*, *accepted part*, and *granted part*. Every element of the queue contains the floor information related to a specific request of a *floor participant*. Each of these states are implemented as a doubly-linked-list (also known as two-way linked list, or symmetrically linked list). This list behaves as a normal queue data structure but with the advantage that element insertions and deletions can be done from any position in the list and the list can be traversed in both directions.

When a *floor participant* requests a floor using a *floor request* message, a new element in the queue is created and attached to the *pending part* of the global queue. Once the chair accepts the floor request for this *floor participant*, the *floor control server* moves the node from the *pending part* to the *accepted part* of the queue. If the chair does not indicate any priority, the node is inserted in the last position of the *accepted part* of the queue. In case the chair does not accept this particular request for this *floor participant*, the node is removed from the *pending list*. When a *floor participant* is allowed by the *floor control server* to start using a floor, the node holding the request information is moved from the *accepted part* of the queue to the *granted part*. Once the *floor participant* releases the floor, the *floor control server* removes the node from the *granted part*.

An example of the design implementation is depicted in Figure 5. It shows a *floor request* identified by FloorRequestID 1 requesting floor 1 and another *floor request* identified by FloorRequestID 2 requesting simultaneously floors 1 and 2. In this example, the *floor control server* gives more priority to floor 1 than floor 2. Therefore, while floor 1 is already granted in the *floor request* identified by FloorRequestID 1, floor 2 remains ungranted in the other *floor request* waiting

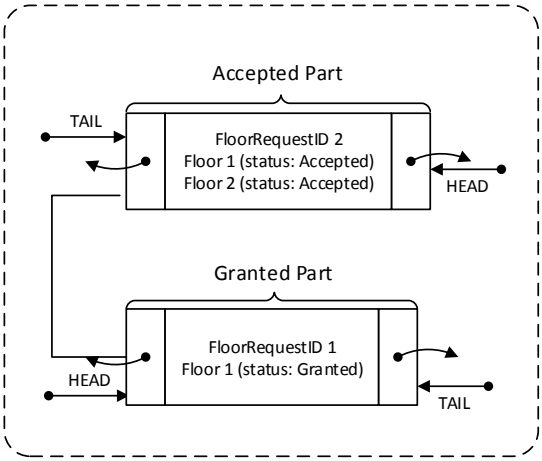


Fig. 5: Example of the Design Implementation

first for floor 1 (which has more priority) to be released.

Granting the floors in order of priority is a method to prevent deadlocks in the IoT services. A deadlock is a persistent and circular-wait condition, where each process involved in a deadlock waits indefinitely for the resources held by other processes while holding the resources needed by others. In our system, a deadlock is when a set of floors is stopped waiting for the resources held by other floors to be released, leading to a situation that makes it impossible for any of them to continue.

In order to break this condition, floors are acquired in order. This method is called Linear Ordering of Resources, and it enforces to first accept the high priority floors before the low priority ones.

Given that $\mathcal{P}$ is the priority to request a floor and $\mathcal{P} \in \mathbb{Z}$, $\mathcal{F}$ is the set of floors $\mathcal{F}^p$ where $p \in \mathcal{P}$, an IoT service holding a resource F_1^h can request a resource F_2^k if it meets the following condition:

$$Request(F_1^h, F_2^k) = \begin{cases} 1, & \forall h, k \in P : h \leq k \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

V. Evaluation

This section evaluates two important aspects of our architecture. First, it evaluates the performance of the *floor control server* regarding its throughput, latency, and scalability. In addition, the results obtained from the performance evaluation of the *floor control server* are compared to the state-of-the-art HTTP solutions. Two of the most popular HTTP servers, Apache[2] and NGINX[3], were included for references. Apache and NGINX are responsible of serving over 50% of the traffic on the Internet. Apache has been the most popular server on the Internet since the early 90s. On the other hand, NGINX has grown in popularity since its release in 2002 due to its ability to scale easily.

Additionally, this section provides a qualitative analysis of the *Binary Floor Control Protocol* and its feasibility to be used in the IoT area.

A. Experiment Setup

The experiment setup consists of two Ubuntu-16.04 laptops. Both laptops are directly connected via an Ethernet cable establishing an IPv4 connection between them. The *floor control server* and the state-of-the-art HTTP servers are installed in a laptop with an Intel®Core™i7-950M processor at 3.07 GHz and 16GB of RAM. Rather than having all the server instances running on the laptop at the same time, only one server is active at a time while the others are inactive to prevent unnecessary consumption of resources.

The second laptop is an Intel®Core™2 Duo T9600 at 2.80 GHz with 4GB of RAM. This laptop runs the benchmark tools used to evaluate the different implementations. The performance of Apache and NGINX are measured with the ApacheBench tool. ApacheBench is a benchmarking tool designed to test the performance of the HTTP servers.

To the best of our knowledge, there is no benchmark tool available for the *Binary Floor Control Protocol*. Consequently, we developed our own benchmark tool in Python 3[4] scripting language. The tool allows us to run in parallel various *floor participants* during a certain time and allows us to specify the number of requests sent to the *floor control server*. The tool uses a basic congestion control where each *floor participant* sends a *hello* message and waits for the response before issuing a new request. After 10 seconds, the request times out and the loss is recorded in a different counter.

A *hello* message is used by the *floor participants* to check the liveliness of the *floor control servers*. A *floor control server* confirms that is alive on reception of a *hello* message by sending an acknowledge message back to the *floor participant* indicating which primitives and attributes supports.

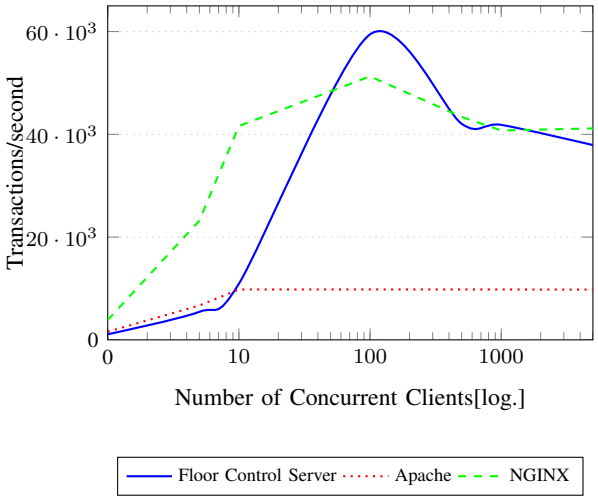
Each *floor participant* in the benchmark tool re-uses a single TCP connection for all its requests. Similarly, in order to better compare the performance of the *floor control server* against the HTTP servers, the keep-alive option of HTTP/1.1 is applied through all the ApacheBench experiments. The keep alive option allows an HTTP client to re-use a single TCP connection for all its consecutive requests.

B. Performance Evaluation

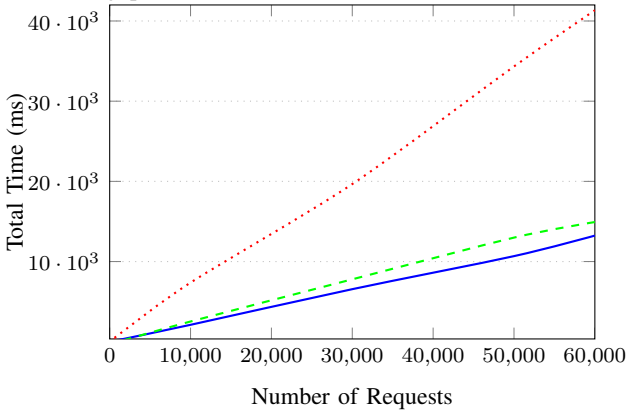
For this experiment, we evaluate the throughput, latency, and scalability of the *floor control server* and compare the results with the Apache and NGINX servers. The throughput is evaluated connecting a set of *floor participants* to the *floor control server* through the BFCP benchmark tool. The *floor participants* stress the *floor control server* by continuously sending requests to it for 60 seconds. The concurrency factor is increased from 1 to 5,000 simultaneous participants. To avoid bias, we repeated all the experiments five times.

[2] https://apache.org/ [3] https://www.nginx.com/

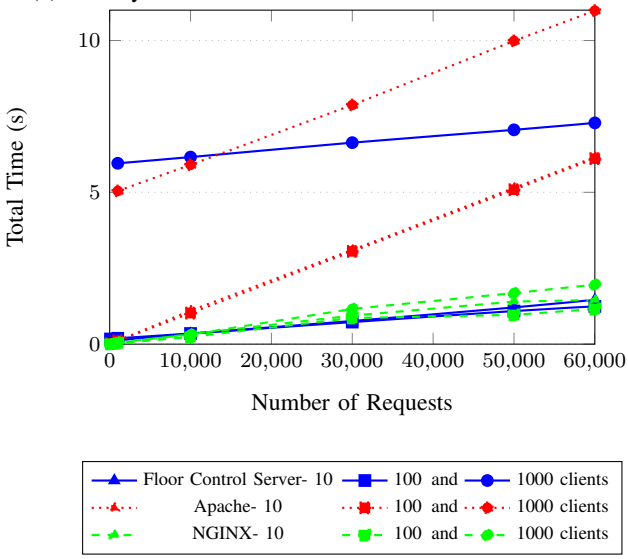
[4] https://www.python.org



(a) Throughput of the Floor Control Server and the HTTP Servers



(b) Latency of the Floor Control Server and the HTTP Servers



(c) Scalability of the Floor Control Server and the HTTP Servers

Fig. 6: Performance Results of the Floor Control Server

As a baseline, we show how Apache and NGINX servers scale under the same scenario using the ApacheBench tool. Figure 6a depicts the results of our experiment. The *floor control server* behaves similarly to Apache up to 10 concurrent clients. Beyond that point, it increases steadily its throughput to 60,000 request per second for hundred concurrent clients, achieving a higher peak performance than NGINX. After that, the performance of both servers decrease significantly to 40,000 request per seconds for 5,000 concurrent clients. Primarily, our implementation exhibits a better performance than Apache and is comparable to NGINX, achieving a better peak performance than both HTTP servers.

Our next experiment consists on evaluating the latency of the *floor control server*. Figure 6b illustrates the total response time, in milliseconds, of the *floor control server* to server a specific number of requests from a single *floor participant*. We gradually increase the number of requests generated by the *floor participant* from 1 to 60,000. Similar experiments were performed with Apache and NGINX. Overall, the performance of the *floor control server* is comparable to NGINX. Even our system achieves slightly better response times than NGINX and performs 3 times better than Apache.

The last experiment involves evaluating the scalability of the *floor control server*. In this experiment a specific number of requests ranging from 0 to 60,000 are sent to the *floor control server* through 10, 100, and 1,000 concurrent clients. The graphs shows the total time required for the *floor control server* to complete all the requests. Similar experiments are done for the HTTP servers using the ApacheBench tool. According to Figure 6c, the *floor control server* have a similar performance than NGINX for 10 and 100 concurrent clients. However, the response times rise to around 6 seconds for 1,000 concurrent clients. Even though NGINX performs better than our solution for highly concurrent scenarios, our solution outperforms Apache in terms of scalability.

As a baseline, the performance of the *floor control server* is overall acceptable. However, we run the experiments in a reasonably powerful computer. Thus, we believe that the hardware capabilities will be a much bigger limiting factor than the software for running the *floor control server*.

C. The Binary Floor Control Protocol

The messages in the *Binary Floor Control Protocol* are encoded in binary format. The message format starts with a fixed 12-bytes header followed by a set of attributes whose minimum length is 1 byte. Figure 7 provides an overview of the different message sizes of the protocol. The Figure shows the sizes of five of the most commonly used operations: *Floor Request*, *Floor Release*, *Chair Action*, *Floor Query*, and *Floor Status*. The size of each message is calculated in four different ways to provide a better understanding of the protocol overhead. In the first two categories only one floor is carried out in each message. The difference between these two categories lies in the number of attributes; the first category includes only the mandatory attributes, while the second category includes all the possible attributes in each message.

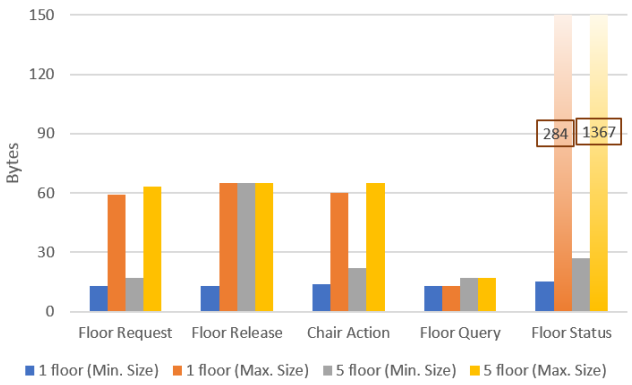


Fig. 7: Binary Floor Control Protocol Message Size

The last two categories carry the information of five floors. In the same way, the third category includes only the mandatory attributes in the message while the last category includes them all. Besides that, we assumed for each calculation that the size of every string value in the messages is 40 (UTF-8-encoded) bytes. Figure 7 shows that almost all the messages are considerable small in size with the exception of the second and fourth category of the *Floor Status* message. Those two categories include all the attributes in the message. However, it is very unlikely that a *Floor Status* message includes all those attributes. Attributes that describe the reason why a participant requested a floor or provides information about the requester of the floor are seldom used in the majority of the scenarios. Nevertheless, all the messages in Figure 7 could still fit within a single IP packet avoiding fragmentation. According to these results, the small size overhead of the *Binary Floor Control Protocol* is comparable to CoAP [14]. Additionally, the *Binary Floor Control Protocol* handles fragmentation, keeping large messages to be fragmented at the IP layer.

VI. Conclusion

This article introduces a novel service-oriented approach to managing exclusive access to shared resources in the IoT systems. Currently, the IoT systems assume that the IoT resources can be accessed concurrently and simultaneously at any time. However, such practices might incur in undesirable outcomes in some specific IoT scenarios. For instance, when an IoT service simultaneously receives opposite orders from different devices.

Concurrency is not a new topic in computer science. Concurrency has already been addressed by several studies proposing different standards. In this matter, rather than creating a new solution, our approach attempts to embrace the existing solutions and adapt them to the IoT area. In this regard, our solution adopts the *Binary Floor Control Protocol*, a protocol designed for managing shared resources in conference systems. Even though the *Binary Floor Control Protocol* [4] has been designed for conference systems, the evaluation study performed for this article shows its feasibility in IoT scenarios. Its relatively small overhead in terms of packet size makes it suitable for IoT.

Furthermore, we developed an implementation of our novel approach based on that study. However, part of this proposed architecture is still subject to ongoing research. Expanding the solution to a decentralized architecture, as well as security, will require further study.

Acknowledgment

The authors would like to thank Kristian Slavov, Nicklas Beijar, Roberto Morabito, Jimmy Kjällman, Alireza Ranjbar, and Mario Di Francesco for their valuable comments and suggestions.

References

[1] "Ericsson Mobility Report: On the pulse of the networked society," Ericsson, Tech. Rep., June 2018. [Online]. Available: https://www.ericsson.com/mobility-report

[2] Y. Liu and G. Zhou, "Key Technologies and Applications of Internet of Things," in *2012 Fifth International Conference on Intelligent Computation Technology and Automation*, Jan 2012, pp. 197–200.

[3] C. Bormann, M. Ersue, and A. Keranen, "Terminology for Constrained-Node Networks," May 2014, RFC7228. [Online]. Available: http://tools.ietf.org/rfc/rfc7228.txt

[4] G. Camarillo, K. Drage, T. Kristensen, J. Ott, and C. Eckel, "The Binary Floor Control Protocol (BFCP)," Internet Engineering Task Force, Internet-Draft draft-ietf-bfcpbis-rfc4582bis, Nov. 2015, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-bfcpbis-rfc4582bis

[5] S. P. Romano, "UMPIRE: a universal moderator for the participation in IETF remote events," *IEEE Communications Magazine*, vol. 53, no. 4, pp. 234–239, April 2015.

[6] V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadist, M. Autili, M. A. Gerosa, and A. B. Hamida, "Service-oriented middleware for the Future Internet: state of the art and research directions," *Journal of Internet Services and Applications*, vol. 2, no. 1, pp. 23–45, Jul 2011. [Online]. Available: https://doi.org/10.1007/s13174-011-0021-3

[7] B. Cheng, M. Wang, S. Zhao, Z. Zhai, D. Zhu, and J. Chen, "Situation-Aware Dynamic Service Coordination in an IoT Environment," *IEEE/ACM Transactions on Networking*, vol. 25, no. 4, pp. 2082–2095, Aug 2017.

[8] D. Hussein, E. Bertin, and V. Frey, "Access control in IoT: From requirements to a candidate vision," in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, March 2017, pp. 328–330.

[9] Y. B. Saied, A. Olivereau, D. Zeghlache, and M. Laurent, "Trust management system design for the Internet of Things: A context-aware and multi-service approach," *Computers and Security*, vol. 39, pp. 351 – 365, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167404813001302

[10] S. Mayer, R. Verborgh, M. Kovatsch, and F. Mattern, "Smart Configuration of Smart Environments," *IEEE Transactions on Automation Science and Engineering*, vol. 13, no. 3, pp. 1247–1255, July 2016.

[11] H. Singh, V. Pallagani, V. Khandelwal, and U. Venkanna, "IoT based smart home automation system using sensor node," in *2018 4th International Conference on Recent Advances in Information Technology (RAIT)*, March 2018, pp. 1–5.

[12] J. L. M. Lastra and M. Delamer, "Semantic web services in factory automation: fundamental insights and research roadmap," *IEEE Transactions on Industrial Informatics*, vol. 2, no. 1, pp. 1–11, Feb 2006.

[13] Z. Shelby, M. Koster, C. Bormann, P. V. der Stok, and C. Amsss, "CoRE Resource Directory," Internet Engineering Task Force, Internet-Draft draft-ietf-core-resource-directory, Oct. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-core-resource-directory

[14] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, Jun. 2014. [Online]. Available: https://rfc-editor.org/rfc/rfc7252.txt