Real-time Emergency Response through Performant IoT Architectures

Claudio Arbib, Davide Arcelli, Julie Dugdale, Mahyar T Moghaddam, Henry Muccini

Real-time Emergency Response through Performant IoT Architectures

Claudio Arbib
University of L'Aquila
claudio.arbib@univaq.it

Davide Arcelli
University of L'Aquila
davide.arcelli@univaq.it

Julie Dugdale
University of Grenoble Alps
julie.dugdale@imag.fr

Mahyar T. Moghaddam
University of L'Aquila
mahtou@univaq.it *

Henry Muccini
University of L'Aquila
henry.muccini@univaq.it †

ABSTRACT

This paper describes the design of an Internet of Things (IoT) system for building evacuation. There are two main design decisions for such systems: *i)* specifying the platform on which the IoT intelligent components should be located; and *ii)* establishing the level of collaboration among the components. For safety-critical systems, such as evacuation, real-time performance and evacuation time are critical. The approach aims to minimize computational and evacuation delays and uses Queuing Network (QN) models. The approach was tested, by computer simulation, on a real exhibition venue in Alan Turing Building, Italy, that has 34 sets of IoT sensors and actuators. Experiments were performed that tested the effect of segmenting the physical space into different sized virtual cubes. Experiments were also conducted concerning the distribution of the software architecture. The results show that using centralized architectural pattern with a segmentation of the space into large cubes is the only practical solution.

Keywords

Emergency Evacuation, IoT, Software Architecture, Network Optimization, Queuing Network.

INTRODUCTION

Building evacuation plans are generally designed as static maps in which pre-defined routes are designed for people to follow when an emergency happens. However, such maps lack real-time awareness, e.g. emerging dangerous areas, congestion and obstacles. To overcome this the building may be equipped with IoT components. IoT is defined as the internal/external communication of intelligent components via the internet in order to improve the environment through providing smarter services (Muccini and Moghaddam 2018). An IoT infrastructure is generally composed of sensing, computation and actuation components that are distributed over the area. The way these components are related and combined together is specified by *software architectures*. IoT architectures can include logic rules for quicker and safer evacuation by tracking people in a building, detecting possible congestion, and updating safety paths. Consequently, the evacuation time under changing emergency conditions can be minimized.

Important questions are: *what kind of reasoning algorithm should an IoT-based emergency evacuation system use? How should it be embedded into an IoT software architecture?*. Regarding the logic rules, previous research (Arbib, Muccini, et al. 2018) (Arbib, Moghaddam, et al. 2019) showed how an IoT system could provide security staff with information to continuously monitor the shortest time required to evacuate people in a building, whilst also

*corresponding author.
†All authors contributed equally to this work.

showing the occupants the best evacuation path. Other research (Muccini, Arbib, et al. 2019) developed a network flow algorithm that can be used in a computer simulation for designing buildings, and also in real-time building evacuation. The authors (Muccini, Spalazzese, et al. 2018) (Muccini and Moghaddam 2018) further argued that the flow algorithm can be embedded into the core of an IoT architecture. The algorithm decomposes both the space (building plan) and the time dimension into finite elements: *unit cells* and *time slots*. The space element is monitored by IoT sensors, whose data are constantly feed into the algorithm. The algorithm can be run on one centralized component or on distributed collaborating controllers.

This paper extends previous works by assessing: *i)* how the the operational delay (evacuation time) and computational delay (CPU time), which are calculated by the algorithm, can be practically used used in an IoT system; *ii)* the feasibility of running the algorithm in distributed processing and storage (P&S) components; *iii)* the impact of changing the granularity of the space cells in the IoT system.

This paper makes the following contributions:

- We introduce a novel set of queuing network models that can be exploited for estimating the performance of IoT systems in order to support architectural decisions.
- We present the algorithm and assess its reaction to time- and space-decomposition.
- We present an optimization model for deciding on a good cell size.
- We evaluate our work by using the real case study of an exhibition venue, in the Alan Turing building, in Italy, with real data. Using various techniques and simulations we design the best internal building layout to handle emergency evacuations.

To the best of our knowledge, this is the first attempt to fill the gap between IoT software architectures, optimization algorithms for minimizing delays and performance modelling and analysis practices, in the context of emergency evacuation.

OVERVIEW

There is a large body of previous work in the three topics that shape our research: IoT software architecture, queuing networks, and optimization algorithms. However, their application to the emergency management has been rarely explored.

Related Work

QNs have been widely and successfully applied to the hw/sw performance assessment domain (Cortellessa et al. 2011; Petriu et al. 2012) and several implementations have been developed by providing editors and analysis environments with QN models. Many existing approaches use QNs as first-class entities for performance analysis (Arcelli and Cortellessa 2013; Trubiani et al. 2014; Altamimi et al. 2016; Arcelli, Cortellessa, and Leva 2016). Despite the wide adoption in the performance domain, QNs have started to be exploited for non-functional assessment in the context of IoT systems only in recent years.

El Kafhali et al (El Kafhali and Salah 2018) proposed an analytic model for a fog/cloud-based Medical IoT system showing how to reduce the cost of computing resources while guaranteeing performance constraints. They used the QN concept to predict the system response time and estimate the minimum required number of P&S resources to meet the service level agreement. However, they do not provide any kind of high or low level architectural model. Huang et al (Huang et al. 2018) propose a theoretical approach of performance evaluation for IoT services, which provides a mathematical prediction on performance metrics during design before system implementation. The authors formulate an atomic service by a queuing system in order to model IoT systems by a queuing network and obtain performance metrics. Whilst using QN, this paper does not address any modeling based on software architecture to be assessed by performance indices. Whilst few related works have been found on IoT systems modeling with QNs, we did not find any previous work on modeling emergency evacuation systems by QNs.

IoT software architectures

IoT architectures are generally composed of three main layers (Muccini and Moghaddam 2018) namely *Perception*, *Processing and Storage (P&S)*, and *Application*:

- The Perception layer represents the IoT physical sensors that collect information. For emergency management, this layer hosts a large number of different types of sensors, e.g. temperature, smoke and movement detectors.

- The Application layer determines the class of services provided by the IoT system. For emergency management, this layer hosts a large number of different types of actuators, e.g. dashboards, evacuation signs and alarms.
- The P&S layer is the central entity of an IoT system that stores and analyses data gathered by the perception components to be accessed by other entities for their applications. Based on the P&S design philosophy, this layer can be divided into various sub-layers to set up the *IoT patterns* as follows:

Centralized. In a centralized pattern, data coming from the perception layer are processed by a central component that makes decisions on actuation. This central component can either be a local controller or, for massive P&S requirements, the cloud. Based on this pattern, if a device wishes to use an IoT service, it must connect to the central P&S component. A centralized architecture simplifies things through a central implementation of analysis and planning algorithms.

Collaborative. In this pattern, data are processed and stored separately (locally and/or remotely) but with the potential collaboration of other local/remote P&S components of the IoT system. In this pattern, a network of local intelligent components can communicate in order to form and empower IoT services. The advantage is that, should a local P&S component fail, a service would still be provided.

Given the above, we designed a set of QNs for IoT architectures that are described in following sections.

Queuing networks

In order to estimate performance indicators and avoid the performance degradation issues associated with IoT architectural patterns, we rely on Queuing Network (QN) models. QNs have emerged as powerful instruments to model and estimate the performance of hardware and software systems. They ground on theoretical foundations based on an algebraic approach to computer system modelling proposed by Lazowska in 1984 (Lazowska et al. 1984), where the computer system is represented as a network of delay and/or queuing *stations* (i.e. topology of the QN). Different *classes of jobs* may flow through the QN, each representing different types of user requests (i.e. dynamics within the QN). While flowing through the QN, each task requires a certain amount of service, namely *service demand* (mentioned as *CPU time* in this paper) to each visited station, depending on the job class the task belongs to. It is worth mentioning that, service demands represent input parameters that must be specified during QN design. Beside service demands, *workload intensities* must be specified, that is the rate at which tasks of each job class enter the QN. For example, a request (of a certain job class) every 2.5 seconds.

Once a QN has been designed, it can be solved analytically or by simulation, carrying out performance indices of interest such as system/stations response time and throughput for both the overall system and single classes of jobs.

Algorithm

The building to be evacuated is represented as a graph $G = (V, A)$ with nodes corresponding to the unit cells i obtained by embedding the building into isometric square grids. Cell 0 conventionally represents the outside of the building, or in general a safe place. The arcs of G correspond to passages between adjacent cells: the passage has full capacity if cells share a boundary uninterrupted by walls, and a reduced capacity otherwise. With no loss of generality, arcs are supposed directed. Let us denote:

T = $\{0, 1, \ldots, \tau\}$, set of unit time slots;

y_i^t = state of cell $i \in V$ at time $t \in T$, that is, the number of persons that occupy i at t: this number is a known model parameter for $t = 0$ (in particular, $y_0^0 = 0$) and a decision variable for $t > 0$;

n_i = capacity of cell i: this is the maximum nominal number of people that i can host at any time (in particular, $n_0 \geq \sum_i y_i^0$); this amount depends on cell shape and size; if cells are assumed to be uniform one can set $n_i = n$ for all $i \in V, i \neq 0$.

x_{ij}^t = how many persons move from cell i to an adjacent cell j in $(t, t + 1]$: this gives the average speed at which the flow proceeds from i to j;

c_{ij} = c_{ji} = capacity of the passage between cell i and cell j: this is the maximum number of people that, independently of how many persons are in cell j, can traverse the passage in the time unit (independence of cell occupancy means that congestion is not taken into account: this will be considered later).

The flow model uses an acyclic digraph D with node set $V \times T$ and arc set

$$E = \{(i,t) \to (j,t+1) : ij \in A, t \in T\}$$

In other words, D models all the feasible transitions (moves between adjacent cells) that can occur in the building in the time horizon T. Transitions are associated with the x-variables defined above, whereas y-variables define the occupancy of each room (and of the building) over time. In real-time, these values can be periodically obtained by the previously described IoT infrastructure. The x- and y-variables are integers and subject to the following constraints:

$$y_j^t - y_j^{t-1} - \sum_{i:ij\in A} x_{ij}^{t-1} + \sum_{i:ji\in A} x_{ji}^{t-1} = 0 \qquad j \in V, t \in T, t > 0 \tag{1}$$

$$0 \le x_{ij}^t + x_{ji}^t \le c_{ij} \qquad t \in T, ij \in A \tag{2}$$

$$0 \le y_i^t \le n_i \qquad t \in T, i \in V \tag{3}$$

Equation (1) is just a flow conservation law: it expresses the occupancy of cell j at time t as the number y_j^{t-1} of persons present at time $t-1$, augmented by those during interval $(t-1,t]$ that move to j from another cell $i \neq j$, minus those that in the same interval leave cell j for another cell $i \neq j$. Box constraints (2), (3) reflect the limited hosting capability of the elements of G.

Maximizing outflow at a given time. To model the relation between time and people outflow, one can try to maximize the number of persons evacuated from the building within τ:

$$\max \quad y_0^\tau \tag{4}$$

To find the minimum total evacuation time, one can solve an Max Flow Problem for different τ, looking for the least value that yields a zero-valued optimal solution. To reduce computation time, this optimal τ can be computed by logarithmic search. The method can thus provide the decision maker with the Pareto-frontier of the conflicting objectives $\min\{\tau\}, \max\{y_0^\tau\}$. Linearizing arc capacities that is quite standard in applications can be find in our previous work (Muccini, Arbib, et al. 2019).

DESIGNING PERFORMANT ARCHITECTURES FOR EMERGENCY HANDLING

Software Architecture for Emergency

Figure 1 shows an example of an IoT-based environment for emergency response: CCTV cameras detect peoples position and movement that is used to feed the algorithm running in a P&S component. The algorithm decides on the actuation set based on the situation. In normal situations, the system shows, on a tablet, a 2D-representation of the monitored space and shows where crowds are located and how they move at any time. In this mode, the optimal flow algorithm is periodically run to estimate the minimum evacuation time required under current conditions. This value can be used to regulate visitor access to a venue in order to comply with safety conditions. If an emergency happens, in addition to the tablet map, alarm actuators are activated and evacuation signs in each area show the best evacuation routes based on the network model described above.

Figure 2 shows the corresponding software architecture. As depicted, additional sets of sensors can be embedded for emergency detection to further enable controllers to decide about normal or critical mode and activate a special set of actuators. As shown in the upper part of Figure 1, in addition to the computational delay of the P&S component, the sensors take some time to detect peoples position, transmit these data, and display the best evacuation routes. Reducing these delays to a minimum improves the system's functionality: since people can follow the given instructions more quickly and more individuals will be in a better evacuation position at the next monitoring time-spot. It is worth mentioning that reducing the aforementioned delays is a function of software architectural patterns, to be improved by properly relating the IoT components to one another. The following section presents a Queuing Network (QN) method that is designed on top of the software architecture, so as to facilitate assessing the performance of our IoT-based emergency handling system.

Queuing Networks for Emergency

Fig. 3 shows a QN representing the performance model that we will exploit later in the paper for our case study. The QN conforms to the architectural patterns of Fig. 2, in fact, from a topological perspective:

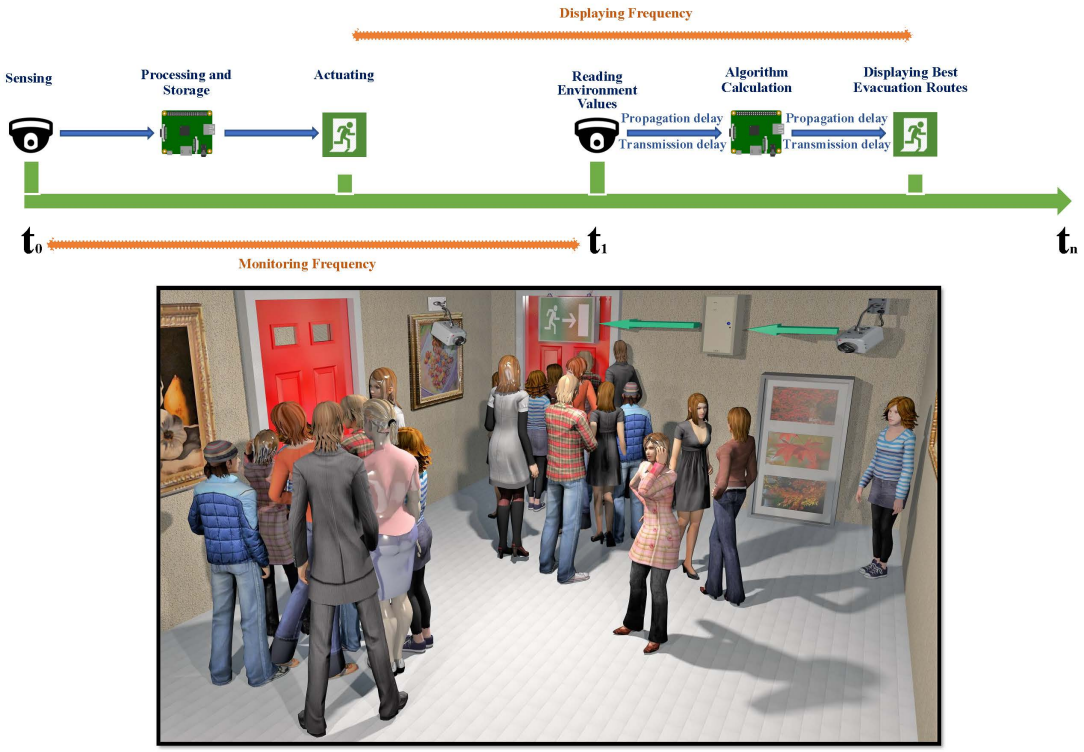

Figure 1. IoT Infrastructure for Emergency Handling.

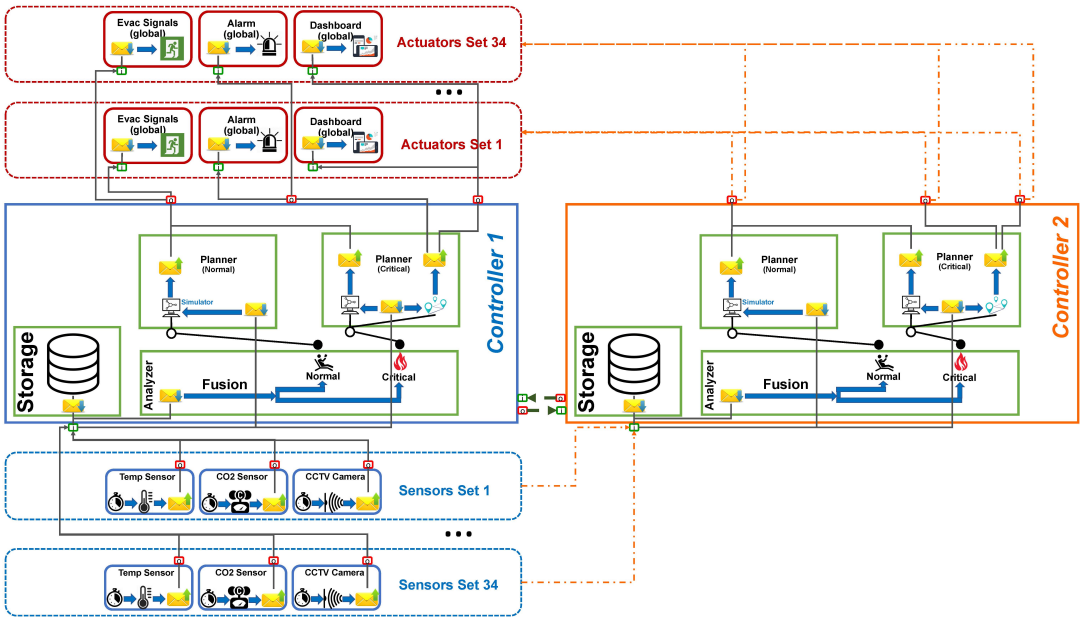

Figure 2. Architectural Patterns. Only Controller 1 Active: Centralized - Both Controllers Active: Collaborative.

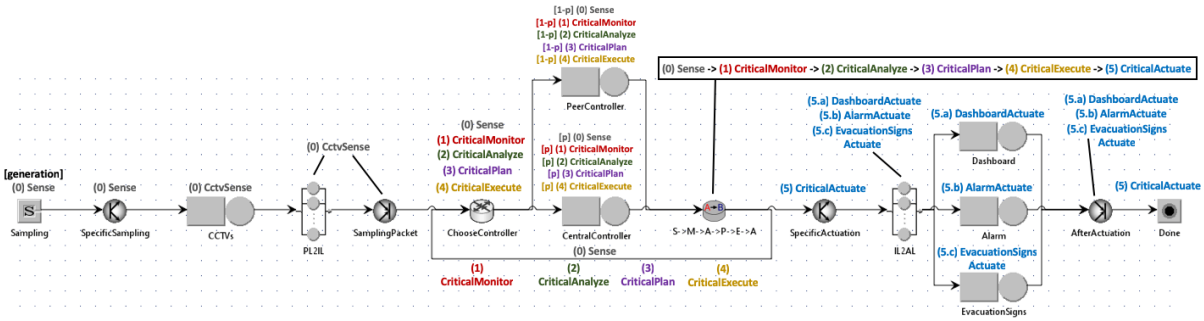

Figure 3. SMAPEA Queuing Network for the Alan Turing building case study.

- *CCTVs* corresponds to a specific kind of sensors, i.e. cameras.
- *Dashboard*, *Alarm* and *EvacuationSigns*, correspond to the three kinds of actuators.
- *CentralController* and *PeerController* correspond to the two P&S components.
- *PL2IL* and *IL2AL* represent networks between sensors and controllers and between controllers and actuators, respectively.
- The two QN constructs on the left and right of the figure, namely *Sampling* and *Done* represent the entry and exit points of the QN, respectively. In particular, *Sampling* indicates the point where data sampled by sensors is generated and *Done* represents the point where actuation ends.

Concerning dynamics, we devise the following control flow within IoT architectures: *i)* Data are sampled by sensors and forwarded through a network to controller(s); *ii)* A control layer aims at achieving the goal of evacuating people during emergency, through an actuation plan that is forwarded to actuators; *iii)* The actuation plan is implemented by actuators, thus possibly achieving the common goal.

The above control flow can be translated into QN language, by identifying a minimal set of 6 different kinds of tasks within a QN for IoT system performance (namely *SMAPEA* loop), sequentially executed as follows: *1.)* *Sense*: Raw data retrieval. *2.)*Monitor: Raw data aggregation and refinement for analysis at controller level. *3.)*Analyze: Interpretation of monitored data. *4.)*Plan: Building an actuation strategy. *5.)*Execute: Pre-processing the actuation strategy towards actuation. *6.)*Actuate: Practically undertaking the actuation by implementing the planned execution.

In order to implement the SMAPEA loop, a *class-switch* that transforms each SMAPEA task to a subsequent task is introduced (see the $S \rightarrow M \rightarrow A \rightarrow P \rightarrow E \rightarrow A$ element of Fig. 3). Moreover, routing probabilities for the switch must be properly defined. In particular, right after the switch, any SMAPEA task type is routed back to controllers through the *ChooseController* router (except *Actuate* tasks that are routed to *IL2AL* for actuation). By exploiting a probability-based routing strategy for *ChooseController* both Centralized and Collaborative patterns can be implemented, in fact the former may route any task to *CentralController*, whilst the latter may route tasks to *CentralController* or *PeerController* with 50% probability.

Notice that, in the QN of Fig. 3, fork/join nodes have been introduced, namely *SpecificSampling* (fork), *SamplingPacket* (join), *SpecificActuation* (fork), *AfterActuation* (join), aimed at modelling the fact that *Sense* and *Actuate* tasks involve the specific sensor and actuator sets, respectively. For example, each *CriticalActuate* task is split into three new tasks, namely *DashboardActuate*, *AlarmActuate* and *EvacuationSignsActuate*, because in case of emergency *i)* the best evacuation paths must be displayed on dashboard, *ii)* an acoustic alarm must be triggered and *iii)* evacuation signs must be properly turned on.

Algorithm Settings for Emergency

Cell Size Setting and its Impact on Architecture.

The cell size has an obvious effect on the resulting spatial patterns, and consequently on both the computational efficiency and model accuracy: the larger the cell, the fewer vertices in G and the lower the refresh frequency at which people's positions are updated. Given the speed at which people move and the data that are acquired, relatively low refresh frequencies are not an issue. Instead, partitioning each room into identical cells may result in a huge network with consequently high CPU time. This issue has a direct impact on software architectural patterns since an operation which requires a huge amount of processing time on a low capacity machine is not suitable for real-time applications. Therefore it should be processed on a more powerful (potentially remote) P&S component.

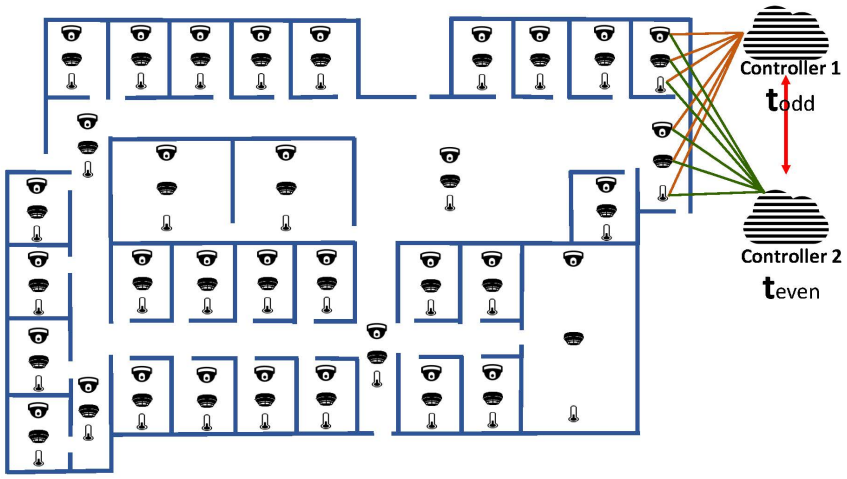

Figure 4. Alan Turing Building IoT Infrastructure.

In general, one can approximate the diverse room shapes by $a \times b$ rectangles as large as possible, while still minimizing any consequent error. Various ways can be adopted to measure approximation error: the most natural is the difference between real and approximated room area, in which case, for room k of size $p_k \times q_k$, the error is given by

$$e_k(a,b) = q_k[p_k \bmod(a)] + p_k[q_k \bmod(b)] - [p_k \bmod(a)][(q_k \bmod(b)]$$

As we need isometric cells and look for a uniform approximation, we set $a = b$ and find a minimizing $\max_k\{e_k(a,a)\}$, meanwhile limiting the total number of approximating cells to some predefined m: hence we choose among the values of a that fulfill $\sum_k \lfloor p_k/a \rfloor \lfloor q_k/a \rfloor \leq m$. A brief description of the method implementation is presented in the Application section.

Time and Space Decomposition Setting and its Impact on Architecture. In order to run the algorithm in distributed P&S components (collaborative pattern), we assessed the space- and time- decomposition feasibility. In the former case, we can give, for instance, two controllers the responsibility of two distinct areas, and let them share border information. Since the global objective cannot generally be satisfied by summing up two local objectives (that is running the algorithm in two local controllers instead of a central one), this method mostly leads to a non-optimal solution obtained by "gluing" together the two distinct areas. In the latter case, we conjectured that the optimal flow obtained at time t can be extended to $t + 1$ and stay optimal. This means that, for instance, one P&S element is in charge of solving the algorithm for t_{even}, and for t_{odd} (Figure 6) whilst sharing a level of data. Assessing this decomposition and taking into account our main requirement of optimally is described in the Application section. Here we can observe the ability of the algorithm to be run in a distributed way or centralized.

APPLICATION

Our proposed model has been applied to the evacuation of the Alan Turing building, in Italy, which is sometimes used for exhibitions. The considered building consists of 29 rooms, 4 main corridors and 34 sets of IoT sensors and actuators (Figure 4). In order to investigate our approach we address four research questions. The first two questions are centered around the algorithm and its levels of granularity and distribution:

- **RQ1**: what are the best cells sizes to divide the building surface, and how does the size affect the evacuation times and computational delays?
- **RQ2**: does running the algorithm in a time-decomposed way increase or decrease the computational and operational delays? What are the designed software architectures corresponding to these results?

The next two research questions focus on software system delays (using Queuing Networks):

- **RQ3**: what level of delay is associated with each QN model?
- **RQ4**: which software architectural design decisions facilitate real-time applications?

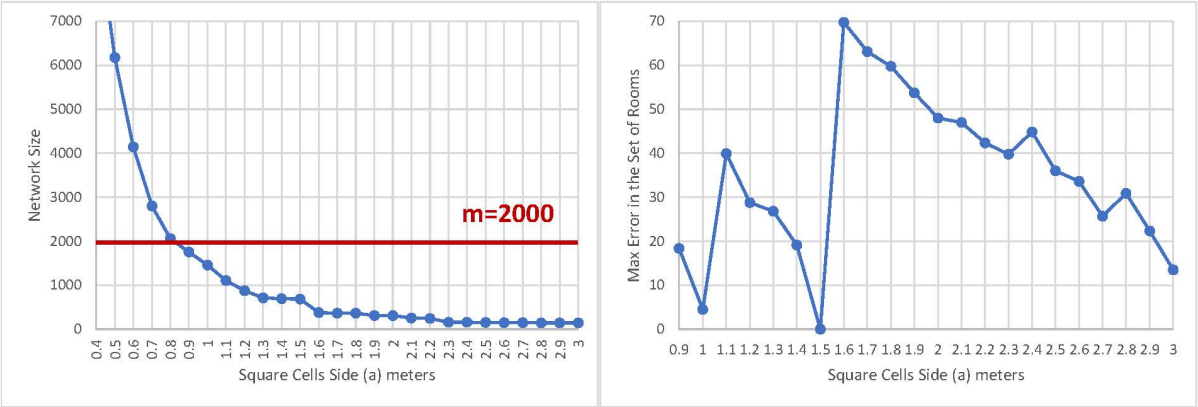

Figure 5. Optimal cell size: maximum error (right) and network size as a function of cell size (left).

Answers to RQ1: cellular approximation of physical space

We split each room in unit cells, each behaving as a (virtual) square room that can be traversed in a unit time slot. In practice, we embedded the building plan into a square grid as shown in Figures 6 and 8. To decide the cell size, we look at both the error introduced by room approximation and the number of nodes in the resulting graph G. The latter is in an inverse proportion of cell size (left diagram in Figure 5); the former varies irregularly with cell size (right diagram of Figure 5). We considered square cells up to 3×3 meters (the short edge of the smallest room) and allowed no more than 2000 nodes of G; then we selected the size that minimizes the largest error for all rooms. The reason of considering such a big maximum network size is to assess the impact of increasing the number of nodes on CPU time and consequently, on the software architectural pattern. As shown in Figure 5, 1.5×1.5 cells (Figure 8) give the best approximation (no error) and involve 687 graph nodes (Figure 9). With 3×3 cells (Figure 6) the error rises to 13.5 but G contains only 144 nodes (Figure 7). Summarizing, 3×3 leads to larger error but less CPU time; conversely, 1.5×1.5 causes larger CPU time but no error. We tested scenarios with both cell sizes in order to find the best efficiency/accuracy compromise.

Simulation. Simulations were first run for both cell sizes. The simulation code was written in the OPL language and problems were solved by CPLEX version 12.8.0. All experiments were run on a Core i7 2.7GHz computer with 16Gb of RAM memory under Windows 10 pro 64-bits. In all tests, we computed the minimum time required for 264 persons, randomly distributed in the building rooms, to reach a safe place. This datum comes from an experiment performed in the Alan Turing during the *the Researchers Night* event, when the IoT system recorded the simultaneous presence of 264 people in the building as a peak value. We solved problem (1-4) for $\tau = 1, 2, \ldots$ until a solution of value 264 was found.

To get a reliable model, some more parameters such as walking velocity under various conditions, door entrance capacities and room capacities must be set to numbers that reflect reality. We set these model parameters based on a literature review (Table 1).

Table 1. Evacuation Model Parameters

Model Parameter	Assigned Value
Walking Velocity	1.2 m/s (Ye et al. 2008)
Door Capacity	1.2 $p/m/s$ (Daamen and Hoogendoorn 2012)
Cell Capacity	1.25 p/m^2 (Uk Fire Safety)

Table 2 reports the number of evacuees at each τ and the computation time of each solution step. Computation is done for both *low-* and *high-resolution* networks (respectively, 3×3 and 1.5×1.5 cells). With the low-resolution network, we get the evacuation and CPU times shown in Table 2 left: in terms of evacuation, everyone has reached a safe place in 55 seconds; on the other hand, computation requires 2.33 seconds in the worst case, and is therefore totally compliant with real-time applications. Evacuation and CPU times for the high-resolution network are reported in Table 2 right. We see that everyone has reached a safe place in 98'75". CPU time is now much larger (382.24 seconds in the worst case) and the model appears inappropriate for real-time use, unless additional computational resources are deployed. Hence we can conclude that sufficient accuracy is obtained using the low resolution network.

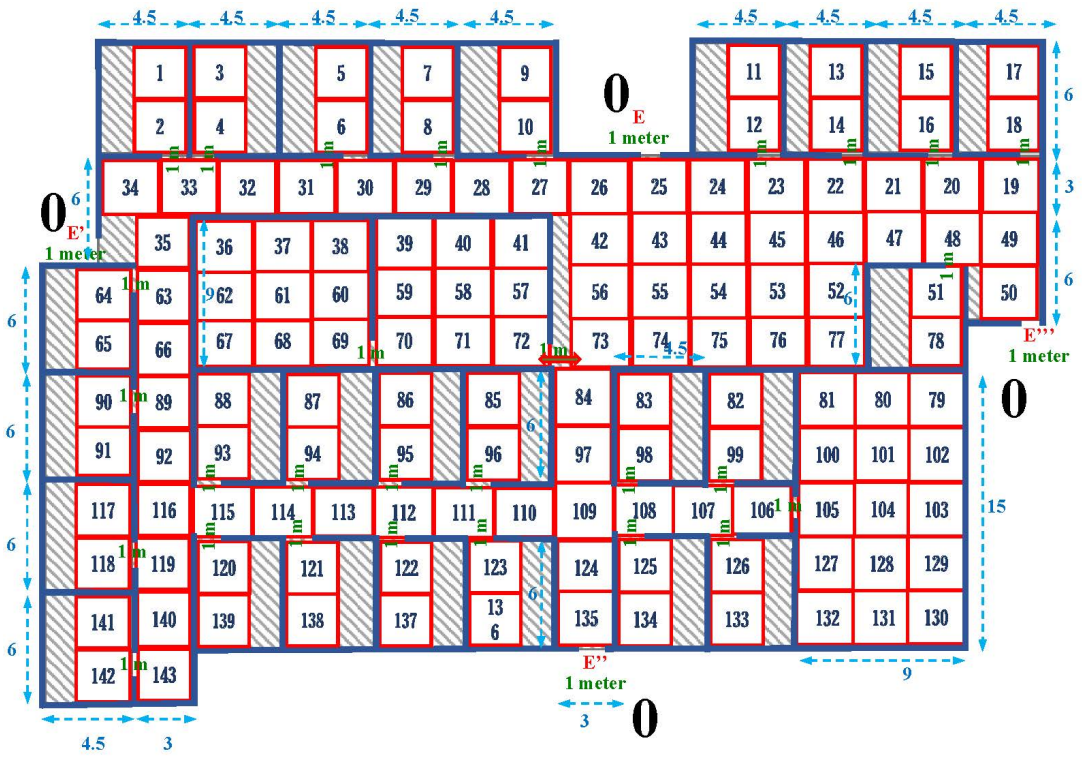

Figure 6. Plan embedding the Alan Turing building into square grids with a low resolution: 3×3 cells. The area that is not covered by cells (error) is shown in gray.

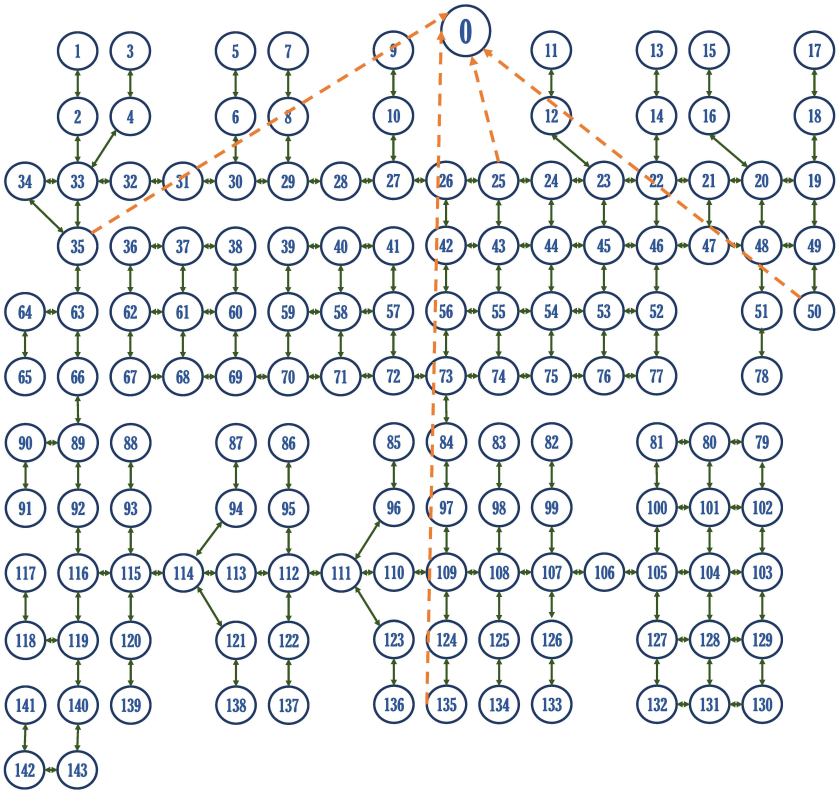

Figure 7. Network associated with the plan of Figure 6.

Figure 8. Plan embedding the Alan Turing building into square grids with a high resolution: 1.5×1.5 **cells.**

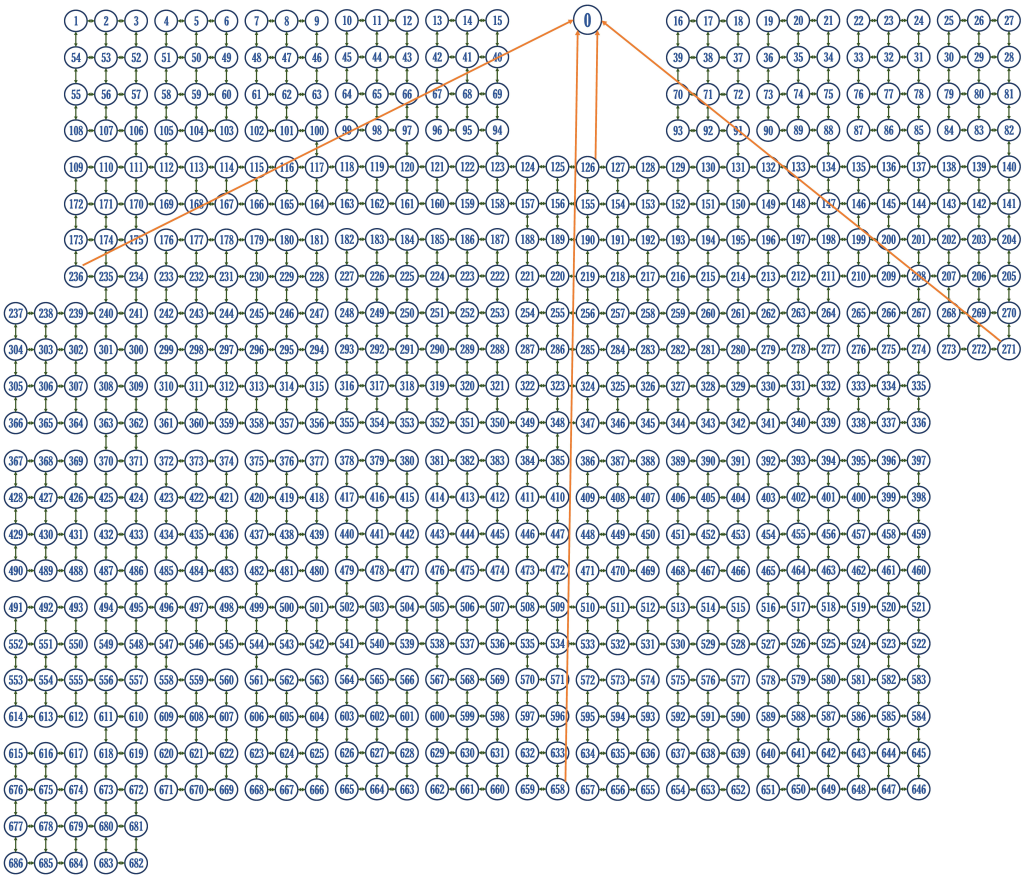

Figure 9. Network associated with the plan of Figure 8.

Table 2. Evacuation and computation time: *a*) 3×3 **cells (time slots of 2.5 seconds);** *b*) 1.5×1.5 **cells (time slots of 1.25 seconds).**

$\tau(a)$	evacuees (a)	CPU Time (a)	$\tau(b)$	evacuees (b)	CPU Time (b)	$\tau(b)$	evacuees (b)	CPU Time (b)
1	12	0.48 sec	1	6	1.85 sec	41	207	79.10 sec
2	24	0.45 sec	2	12	1.99 sec	42	209	84.37 sec
3	36	0.50 sec	3	18	2.10 sec	43	210	87.07 sec
4	48	0.53 sec	4	24	2.35 sec	44	212	94.39 sec
5	60	0.59 sec	5	30	2.90 sec	45	213	95.04 sec
6	72	0.62 sec	6	36	3.21 sec	46	215	97.86 sec
7	84	0.66 sec	7	42	3.72 sec	47	216	98.71 sec
8	96	0.71 sec	8	48	4.18 sec	48	218	105.83 sec
9	108	0.79 sec	9	54	5.18 sec	49	219	120.60 sec
10	120	0.88 sec	10	60	5.53 sec	50	221	125.50 sec
11	132	0.97 sec	11	66	6.27 sec	51	222	130.07 sec
12	144	1.02 sec	12	72	6.76 sec	52	224	128.54 sec
13	156	1.11 sec	13	78	7.92 sec	53	225	138.54 sec
14	168	1.16 sec	14	84	8.76 sec	54	227	139.20 sec
15	180	1.42 sec	15	90	10.10 sec	55	228	154.73 sec
16	192	1.60 sec	16	96	10.54 sec	56	230	160.65 sec
17	204	1.76 sec	17	102	12.38 sec	57	231	161.58 sec
18	216	1.82 sec	18	108	14.19 sec	58	233	160.01 sec
19	228	1.86 sec	19	114	15.17 sec	59	234	171.88 sec
20	240	2.12 sec	20	120	17.36 sec	60	236	169.26 sec
21	252	2.29 sec	21	126	19.26 sec	61	237	178.87 sec
22	264	2.33 sec	22	132	21.55 sec	62	239	182.49 sec
			23	138	23.50 sec	63	240	201.91 sec
			24	144	24.98 sec	64	242	205.39 sec
			25	150	27.67 sec	65	243	240.93 sec
			26	156	30.12 sec	66	245	213.88 sec
			27	162	34.03 sec	67	246	216.48 sec
			28	168	36.00 sec	68	248	216.67 sec
			29	174	38.31 sec	69	249	230.26 sec
			30	180	40.06 sec	70	251	310.11 sec
			31	186	38.93 sec	71	252	256.64 sec
			32	192	43.35 sec	72	254	329.78 sec
			33	195	47.16 sec	73	255	324.49 sec
			34	197	51.70 sec	74	257	337.55 sec
			35	198	55.31 sec	75	258	348.49 sec
			36	200	63.31 sec	76	260	300.19 sec
			37	201	66.90 sec	77	261	381.83 sec
			38	203	67.27 sec	78	263	382.24 sec
			39	204	71.91 sec	79	264	323.34 sec
			40	206	75.90 sec			

Answers to RQ2: Time Decomposition

Table 3 gives the number of evacuees at each τ and the computation time of each solution step corresponding to time-decomposed networks (collaborative P&S). With a time-decomposed simulation of 3×3 cells, we obtain the evacuation and CPU times in Table 3 left: in terms of evacuation, everyone has reached a safe place in 75 seconds, however computation requires 0.58 seconds in the worst case, and is therefore compliant with real-time applications. Compared with continuous simulation (central P&S) of the same case that is presented in Table 2 left, whilst CPU time is now significantly reduced, the optimal evacuation time is increased by 36 percent. Looking at the time-decomposed simulation results for 1.5×1.5 cell (Table 3 right), CPU time is decreased by at least 123 percent (being 3.11 seconds in the worst case) and the evacuation time remained constant.

Hence, for a high resolution time-decomposed network: whilst the operational delay remains constant, the computational delay improves significantly. However total response time should be observed since the quicker sampling rate tied up with high resolution may cause a negative impact. For a low resolution time-decomposed network: whilst the computational delay decreases, the evacuation delay (that has a priority over all other delays) increases: so that running the algorithm in a collaborative architecture is not recommended, regardless of melioration/deterioration of total response time.

Taking into account the result of this subsection and keeping evacuation time and CPU time as inputs, the following subsection practically assesses, in terms of total response time (delay), the quality of proposed architectural patterns with respect to four different scenarios, resulting from the different combinations of architectural patterns designed to handle emergency situations:

1. *Centralized with High Resolution (Centralized-HR)*: the critical situation is handled by a continuous simulation of the algorithm, (on a single controller) and physical space is divided into 1.5×1.5 cells.
2. *Centralized with Low Resolution (Centralized-LR)*: the critical situation is handled by a continuous simulation of the algorithm, (on a single controller) and physical space is divided into 3×3 cells. Therefore, again all tasks are routed to a central controller.

Table 3. Evacuation and computation time for time-decomposed scenarios: *a*) 3×3 **cells (time slots of 2.5 seconds);** *b*) 1.5×1.5 **cells (time slots of 1.25 seconds).**

$\tau(a)$	evacuees (a)	CPU Time (a)	$\tau(b)$	evacuees (b)	CPU Time (b)	$\tau(b)$	evacuees (b)	CPU Time (b)
1	12	0.48 sec	1	6	2.43 sec	41	205	1.89 sec
2	24	0.56 sec	2	12	3.11 sec	42	208	1.88 sec
3	36	0.45 sec	3	18	2.45 sec	43	210	1.89 sec
4	48	0.54 sec	4	24	2.42 sec	44	212	1.89 sec
5	60	0.56 sec	5	30	2.31 sec	45	213	1.83 sec
6	72	0.50 sec	6	36	2.39 sec	46	215	1.85 sec
7	84	0.51 sec	7	42	2.32 sec	47	216	1.86 sec
8	96	0.52 sec	8	48	2.53 sec	48	218	1.82 sec
9	108	0.49 sec	9	54	2.45 sec	49	219	1.89 sec
10	120	0.51 sec	10	60	2.54 sec	50	221	1.84 sec
11	132	0.48 sec	11	66	2.39 sec	51	222	1.90 sec
12	144	0.55 sec	12	72	2.42 sec	52	224	1.84 sec
13	156	0.50 sec	13	78	2.30 sec	53	225	1.90 sec
14	168	0.58 sec	14	84	2.48 sec	54	227	1.85 sec
15	180	0.56 sec	15	90	2.48 sec	55	228	1.92 sec
16	192	0.51 sec	16	96	2.89 sec	56	230	1.93 sec
17	204	0.53 sec	17	102	2.24 sec	57	231	1.84 sec
18	211	0.52 sec	18	108	1.97 sec	58	233	1.85 sec
19	217	0.41 sec	19	114	2.05 sec	59	234	1.82 sec
20	223	0.43 sec	20	120	1.90 sec	60	236	1.87 sec
21	229	0.43 sec	21	126	1.98 sec	61	237	1.83 sec
22	235	0.52 sec	22	132	1.95 sec	62	239	1.85 sec
23	241	0.43 sec	23	138	2.04 sec	63	240	1.87 sec
24	246	0.50 sec	24	144	1.99 sec	64	242	1.91 sec
25	249	0.39 sec	25	150	1.87 sec	65	243	1.95 sec
26	252	0.55 sec	26	156	1.86 sec	66	245	1.89 sec
27	255	0.50 sec	27	162	187 sec	67	246	1.81 sec
28	258	0.51 sec	28	166	1.93 sec	68	248	1.89 sec
29	261	0.53 sec	29	169	1.93 sec	69	249	1.85 sec
30	264	0.50 sec	30	172	1.85 sec	70	251	1.95 sec
			31	175	1.87 sec	71	252	1.84 sec
			32	178	1.87 sec	72	254	1.85 sec
			33	181	1.89 sec	73	255	1.92 sec
			34	184	1.86 sec	74	257	1.90 sec
			35	187	1.90 sec	75	258	1.81 sec
			36	190	1.96 sec	76	260	1.90 sec
			37	193	1.83 sec	77	261	1.94 sec
			38	196	1.89 sec	78	263	1.81 sec
			39	199	1.82 sec	79	264	1.98 sec
			40	202	1.96 sec			

3. *Collaborative with High Resolution (Collaborative-HR)*: the situation is handled by a time-decomposed simulation of the algorithm, (on collaborative controllers) and physical space is divided into 1.5×1.5 cells. This means that the two available controllers are intermittently chosen as the destination of routed tasks.
4. *Collaborative with Low Resolution (Collaborative-LR)*: the emergency situation is handled by a time-decomposed simulation of the algorithm, (in a collaborative way) and physical space is divided into 3×3 cells. Thus, one of the controllers handles the situation for *odd* and the other for *even* time slots.

Answers to RQ3: Total Delay Assessment using Queuing Networks Parameterization

In order to realize the software architectures resulting from algorithm simulations, we exploit the QN of Fig. 3 that we have previously introduced, with the four different sets of input parameters needed to model the software architectures resulting from algorithm simulations.

Sense tasks (by CCTVs) are generated at a certain rate (i.e. every 2.5 seconds for high resolution and 1.25 seconds for low resolution networks). These rates are the monitoring frequencies, and is the time taken for an individual to cross a single cell in our crowd monitoring algorithm. Such time intervals represent a key-value, due to their impact on the overall evacuation delay.

Exponential distributions are used to define CPU times. Table 4 reports the means of such distributions for our case study, which have been estimated in several ways. Our focus is on evacuation, i.e. *CriticalPlan-HR* and *CriticalPlan-LR*, which have been estimated by formulating the evacuation handling problem within CPLEX. Other parameters are set as follows:

- Service time distribution means for sensors, networks and actuators, have been obtained by modelling the IoT system for the Alan Turing building with CAPS, our simulation framework (Muccini and Sharaf 2017). CPU times for sensors, actuators and networks, have been calculated in terms of transmission and propagation delays (*td* and *pd*, respectively).

- Service time distribution means for controllers refer to SMAPEA task types. *Sense* tasks have zero CPU time since they do not introduce additional computation for controllers, hence they just need to be transformed into *Monitor* tasks.
- As shown in Table 4, *Monitor* and *Execute* tasks are equal and have the same order of magnitude of *Analyze*. To avoid further complexity, we ignore formulating optimization models for such task types, by setting arbitrary values as follows: we assume that aggregating raw data (i.e. *Monitor*) and building a list of atomic actions to execute (i.e. *Execute*) are less demanding than interpreting monitored data (i.e. *Analyze*).

Table 4. SMAPEA Task Types and CPU Times for case Study

SAMPEA-QN Layer	Service Center	Task Type		Mean CPU time (exp)		Notes
		ID	Name	Centralized	Collaborative	
Perception	CCTVs	1	CctvSense	0.01386		*By CAPS (td).*
Network	PL2IL	2	CctvSense	0.023480633		*By CAPS (td+pd).*
P&S	Controllers	3	Sense	0		*Sense just becomes xMonitor*
		4	CriticalMonitor	0.0045067	0.0045067	*Arbitrarily set.*
		5	CriticalAnalyze	0.00676	0.005735	*By CPLEX.*
		6	**CriticalPlan-HR**	110.1791139	2.0164557	*By CPLEX*
		7	**CriticalPlan-LR**	1.1668182	0.5016667	*By CPLEX*
		8	CriticalExecute	0.0045067	0.0045067	*Arbitrarily set.*
Network	IL2AL	9	DashboardActuate	0.013619641		*By CAPS (td+pd).*
		10	AlarmActuate	1.013619641		
		11	EvacuationSignsActuate	2.013619641		
Application	Dashboards	12	DashboardActuate	0.000921667		*By CAPS (td).*
	Alarms	13	AlarmActuate	0.000921667		
	EvacuationSigns	14	EvacuationSignsActuate	0.000921667		

Simulation. We assess the total delay corresponding to each IoT architectural pattern design based on the flow algorithm. The response time (delay) that is analyzed is the mean time spent from starting the sampling to the time that actuation ends.

Table 5 reports, for each scenario, the overall system response time (in seconds) in the second column and the architectural design decision in the third column.

Table 5. Experimental results.

Pattern	System response time (s)	Architectural design decision
Centralized with HR	*System saturation*	Violates real-time requirement
Centralized with LR	*1.5085*	Practical
Collaborative with HR	*26.5597*	Violates real-time requirement
Collaborative with LR	*0.5864*	Violates optimally requirement

Answers to RQ4: Architectural Design Decision. Experimental results show that, in LR, the collaborative pattern minimizes system response time. However, this pattern does not satisfy the precondition of optimal evacuation of people from the building. Thus, the centralized pattern may be more appropriate in a critical mode. The drawback is that, managing critical cases with the centralized architecture increases system response time by more than 200% with respect to the collaborative one. The solution could use the HR network. However, HR does not allow the system to fulfill the real-time requirement, due to two factors: *i)* working in HR requires much more CPU time; *ii)* the sampling rate doubles when the system performs in HR. Both factors contribute to a significant worsening of performance that might lead to system saturation (as for the Centralized-HR).

As a result of the considerations above, *only one pattern could be adopted, i.e. the Centralized-LR.*

It is worth noticing that, in order to address the sampling rate for HR (i.e. 1.25), while satisfying the precondition that minimizes operational delay, the CPU time of the controller in the Centralized pattern should have the same order of magnitude of the controllers in the Collaborative-LR. For example, with a CPU time of 0.5016667 (i.e. the same as the Collaborative-LR), the Centralized-HR would show a system response time of 0.6535 seconds

(i.e. 11.5% more than Collaborative-LR). As a second example, with a CPU time of 2.0164557 (i.e. the same as Collaborative-HR), the Centralized-HR would experience continuous saturation that might lead to a system response time in the order of hours.

CONCLUSION

This work uses the Queuing Network concept in order to model the IoT architectural patterns for emergency evacuation, and assess the patterns' corresponding delays. The architecture has a core computational component in the form of network flow, which supports the design decision by providing the model with expected operational and computational delays. Preliminary evaluations using data from a real case, and an *ad-hoc* IoT infrastructure showed the suitability of a centralized software architecture based on a low resolution division of the building surface.

REFERENCES

Altamimi, T., Zargari, M. H., and Petriu, D. C. (2016). "Performance analysis roundtrip: automatic generation of performance models and results feedback using cross-model trace links". In: *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering, CASCON 2016, Toronto, Ontario, Canada, October 31 - November 2, 2016*. IBM, pp. 208–217.

Arbib, C., Moghaddam, M. T., and Muccini, H. (2019). "IoT Flows: A Network Flow Model Application to Building Evacuation". In: *M. Gaudioso, M. Dell'Amico and G. Stecca (Eds.) A view of operations research applications in Italy.* Springer.

Arbib, C., Muccini, H., and Moghaddam, M. T. (2018). "Applying a network flow model to quick and safe evacuation of people from a building: a real case". In: *Proceedings of the GEOSAFE Workshop on Robust Solutions for Fire Fighting, RSFF 2018, L'Aquila, Italy, July 19-20, 2018.* Pp. 50–61.

Arcelli, D. and Cortellessa, V. (2013). "Software model refactoring based on performance analysis: better working on software or performance side?" In: *Proceedings of the 10th International Workshop on Formal Engineering Approaches to Software Components and Architectures, FESCA*. Vol. 108. EPTCS, pp. 33–47.

Arcelli, D., Cortellessa, V., and Leva, A. (2016). "A Library of Modeling Components for Adaptive Queuing Networks". In: *Computer Performance Engineering - 13th European Workshop, EPEW 2016, Chios, Greece, October 5-7, 2016, Proceedings*. Vol. 9951. Lecture Notes in Computer Science. Springer, pp. 204–219.

Cortellessa, V., Di Marco, A., and Inverardi, P. (2011). *Model-Based Software Performance Analysis*. Springer Berlin Heidelberg.

El Kafhali, S. and Salah, K. (2018). "Performance Modeling and Analysis of IoT-enabled Healthcare Monitoring Systems". In: *IET Networks*.

Huang, J., Li, S., Chen, Y., and Chen, J. (2018). "Performance modelling and analysis for IoT services". In: *International Journal of Web and Grid Services* 14, p. 146.

Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C. (1984). *Quantitative system performance - computer system analysis using queueing network models*. Prentice Hall.

Muccini, H., Arbib, C., Davidsson, P., and Tourchi Moghaddam, M. (2019). "An IoT Software Architecture for an Evacuable Building Architecture". In: *Proceedings of the 52nd Hawaii International Conference on System Sciences*, pp. 678–687.

Muccini, H. and Moghaddam, M. T. (2018). "IoT Architectural Styles". In: *European Conference on Software Architecture*. Springer, pp. 68–85.

Muccini, H. and Sharaf, M. (2017). "Caps: Architecture description of situational aware cyber physical systems". In: *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE, pp. 211–220.

Muccini, H., Spalazzese, R., Moghaddam, M. T., and Sharaf, M. (2018). "Self-adaptive IoT architectures: an emergency handling case study". In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. ACM, p. 19.

Petriu, D. C., Alhaj, M., and Tawhid, R. (2012). "Software Performance Modeling". In: *Formal Methods for Model-Driven Engineering - 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM*. Vol. 7320. Lecture Notes in Computer Science. Springer, pp. 219–262.

Trubiani, C., Marco, A. D., Cortellessa, V., Mani, N., and Petriu, D. C. (2014). "Exploring synergies between bottleneck analysis and performance antipatterns". In: *ACM/SPEC International Conference on Performance Engineering, ICPE'14*. ACM, pp. 75–86.