# Accelerating smart eHealth services execution at the fog computing infrastructure

Marisol García-Valls[a], Christian Calva-Urrego[b], Ana García-Fornes[c]

[a]*Departamento de Comunicaciones, Universitat Politècnica de València, Spain*
[b]*Departamento de Ingeniería Telemática, Universidad Carlos III de Madrid, Spain*
[c]*Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Spain*

**Abstract**

Fog computing improves the execution of computationally intensive services for remote client nodes, as part of the data processing is performed close to the location where the results will be delivered. As opposed to other services running on smart cities, a major challenge of eHealth services on the fog is that they typically span multiple computational activities performing big data processing on sensible data that must be protected. Using the capacities of current processors can improve the servicing of remote patient nodes. This paper presents the design and validation of a framework that improves the service time of selected activities at the fog servers; precisely, of those activities requested by remote patients. It exploits the capacities of current processors to parallelize specific activities that can be run on reserved cores, and it relies on the quality of service guarantees of data distribution platforms to improve communication and response times to mobile patients. The proposed approach is validated on a prototype implementation of simulated computationally intensive eHealth interactions, decreasing the response time by 4x when core reservation is activated.

*Keywords:* Fog computing, Resource management, Multicore, Distribution software, Quality of service, eHealth services, computation intensive services

## 1. Introduction

Running services on the cloud may result in latencies that are not suited for some application domains; precisely, providing real-time and predictable cloud computing [16] brings in great challenges that are still being researched.

Fog computing relies on the IoT (Internet of Things) geographically dispered nodes to bring computations closer to the physical devices and sensors. For this reason, fog computing enables improved provisioning of distributed services and applications as compared to the cloud paradigm [3]. The new paradigm of *Social Dispersed Computing* [17] embraces complex logic like augmented reality, voice translation or drone traffic control [14]; these complex applications find a more efficient model in the fog, as distributed services can be partly run closer to the delivery point, reducing the amount of data exchanged with the cloud.

The evolution of hardware is turning the prior resource constraint devices and sensors into medium capacity processing nodes; but still such nodes have reduced computation capacity if compared to, e.g., current single board computers. These nodes are typically complemented by proximity nodes in the form of, what we call, a *federation* or *cloudlet* [6]. Having federations provides autonomy and liberty to the participating nodes as mobility is facilitated; however, since they are under the supervision of an outstanding node (i.e., proximity node) in the federation, this provides a means to order the chaos of the fog.

The interaction among the physical devices and sensors and the proximity nodes requires distribution software to provide flexible, efficient, and quality of service (QoS) operation. Although QoS parameters are application dependant, it is the case that typically these refer to timing behavior and reliability characteristics of the systems, including timeliness, robustness, error limits for data transmission, data processing rates, etc.

The progressive increase of the computation power of the hardware has boosted the possibilities of distributed computation. More powerful hardware brings closer and closer to reality the desired real-time behavior that IoT devices and systems require [8]. It is now posible to provide mobile users with more computationally intensive services as these can offload some of the involved tasks in sorrounding cloudlet and fog servers, with multicore processors that can speed up their execution. Among these services, one may find eHealth and medical systems' services [13] that can be brought nearer the increasingly mobile patient residing far from the care centers; these patients can be monitored in real-time, from the distance, by embedding some of the most computationally expensive services that s/he may need in her/his sorrounding proximity infrastructures. Then, it is now easier that eHealth and medical systems can progressively adopt execution paradigms as the fog to provide improved services to patients, e.g., processor intensive data analysis from multiple environmental and health domains, to provide recommendations to patients in need of health monitoring.

This paper supports the cross-fertilization of the fog computing paradigm and the operating system techniques to control the execution in the nodes, and it achieves efficient remote servicing of patient nodes. On the one hand, current contributions on eHealth address service provisioning from an application perspective, and they marginally consider the performance side derived from the accurate control of the code functions on the nodes. On the the other hand, the recent scientific literature dealing with fog computing is crowed with contributions on how to enable the deployment of cloudlets and fog computing for supporting heavy computations in real-time. Most of the contributions address mobility by developing architectures for heterogeneous node interaction, and programming solutions to support interoperability and flexible service specification, publishing, and utilization across all participating nodes. However, these perspectives have not dealt with the objectives of this contribution that are to support timely servicing of mobile patients, exploiting the characteristics of the nodes' hardware and the QoS posibilities of the distribution software.

This paper proposes a software framework to support QoS aware interaction across devices that relies on the use of the data distribution system standard in order to use a de facto standard for QoS properties. One of the major contributions relies on the efficient usage of the general purpose multicore processors at the fog servers in order to provide shorter service times to remote patient requests. The proposed software framework is aware of the underlying hardware structure in those fog nodes that have a multicore processor that can be used to parallelize some offloaded processing tasks. The paper considers the importance of security mechanisms in the fog; it complements the traditional security schemes by providing a model that can be used to accelerate the response time of the security algorithms through core reservation.

## *1.1. Objectives*

This section summarizes the paper contributions that are the following:

- Provisioning a low level execution model that considers the fog elements for improving the capacity to support computationally intensive eHealth-like services with differentiated services to prioritized remote patient nodes.

- Leveraging the processing capacity of the fog servers and their multicore nature to accelarate the response time of services by parallelizing their operations and to prioritize particular activities.
- Providing an extended data distribution layer as the communication backbone across participating nodes that intercepts offloading requests from mobile patient devices and decides whether these are serviced depending on the current available spare capacity.
- Integration of parallelization infrastructures and operating system fine tunning mechanisms at the fog server side to speed up execution and providing real-time response to patients.
- Achieving a robust architecture that tolerates execution on noisy fog servers, preserving the normal operation.

### *1.2. Paper structure*

The rest of the paper is structured as follows. Section 3 describes the baseline technologies for this work. Section 2 describes a selection of the previous work that is most related to the present contribution. Section 4 presents the problems in execution of computation intensive functions on fog servers. Section 5 describes the proposed architecture. Section 6 reports the validation of the proposed model and highlights the obtained results. Section 7 concludes the paper and draws future research lines as continuation work.

## 2. Related work

As technology improves, increasingly complex smart systems are made posible. Over the last decade, light weight and highly available communication infrastructures that support the interaction across the participant nodes of the smart domains have backed up their realization. Now, it appears that fog computing will be the next key technological step that will support a major progress in the performance and effective realization of smart services. However, fog computing technologies are still very immature and there is no actual fog computing infrastructure that fully implements its promising vision.

Merging fog computing and eHealth/medical systems can be a major advance for society as it can enable remote real-time monitoring, recommendation, and guidance to *mobile patients*, increasing their autonomy, confidence, and also reducing the on site costs. Patients will be able to preserve their normal life activity, through personalized health care solutions such as [30] wherever they are; low cost mobile devices and sensors can provide a low cost infrastructure to exchange data with the environment and remote servers and collect rich information from the daily conditions of the patient or from her/his sorroundings such as weather conditions, city information on traffic [5], pollution, routes, etc.

Collecting all this information will result in the generation of big data that will be fed to eHealth services capable of analysing the data applying machine learning techniques [4], combining them with historical records and data from other medical sources. Such data cross-processing and analytics requires high computation power from either the cloud (that will result in increased latencies) or from the sorrounding fog infrastructure to which part of the eHealth services can be offloaded.

The fog brings in a number of challenges to proper interaction across nodes. Also, the security of data is challenged as extensively described in [26]; for this, a number of solutions have to be put in place such as data forensics [34, 7]; or security models for medical data transmission in IoT healthcare [12].

Contributions on smart eHealth have mostly dealt with the specific computation algorithms to derive statistical information of certain illness such as diabetes and the relation to information searches over the Internet (e.g. [23]); cloud-based execution of eHealth services such as [13] that yield increased latencies for the user; or strategies for efficient access to data bases for big data processing such as [20]. Also, eHealth solutions have addressed the application level functional requirements such as image encryption techniques for privacy preservation [21, 32, 33]. A number of platforms for dealing with the big data generated in eHealth have also been presented such as [11]; whereas other contributions such as [25] provide a platform for IoT big data handling in health services, but strictly from a security-centric perspective.

To the best of our knowledge there are no contributions that aim at integrating the different required levels (i.e., from data distribution and resource management) with the logic to achieve control over the assignment of the processing resources of the available multicore processors to the application level eHealth activities. Although there are efficient distribution software designs such as [18] that has been used for real-time video transmission over dynamic distributed service oriented systems, it is unaware of the underlying structure of multicore processors. To exploit the potential of the availble IoT infrastructures and deliver timely services to patients, it is needed to consider the computation capacity and possibilities of the execution nodes. In the context of cyber physical systems, rigurous design techniques will have to be applied such as [19] that follows a pragmatic approach based on parametric models that guarantee the temporal properties in the presence of uncertainty.

## 3. Background technologies

This section describes the basic techniques to enable the control of the execution in the hardware by providing an overview of different technologies for parallelization and scheduling control. Precisely, OpenMP (Open Multi-Processing) [2] as a libraty for parallelization with shared memory and the Message Passing Interface (MPI) [1] are introduced. The former is a state of the art library for parallelization with shared memory, whereas the latter is an interface specification for message passing.

### *3.1. Parallel processing infrastructures*

OpenMP [2] is a parallelization infrastructure that provides an application programming interface (API) consisting on a set of compiler/preprocessor directives, library routines, and environment variables for shared memory parallel programming [10], which constitutes a language for multi-threaded applications. It uses Pthreads on some operating systems, which favors the development of portable solutions.

An advantage of OpenMP over other parallel programming paradigms is that it can be easily integrated on existing code and some studies suggest that using OpenMP over threads with Pthreads increases the robustness without sacrificing performance [24]. The spawning strategy is typically based on a fork-join model for parallelizing tasks such as used in OpenMP. In the specific target system, the parallelizable activities are parts of the eHealth services that are run on the fog servers.

In multicore processors, multi-threading may increase significantly the performance of an application as several computations are performed at the same time on different processors. Threads are like processes for the system in terms of execution, they have their own stack and program counter, but they have access to the same virtual memory address space of its parent process. A process can be either single- or multi-threaded. The proposed system uses threads as they result in more efficient execution given their reduced context switch cost. As threads of a same parent process have a shared memory space, they can communicate through shared memory (global variables); this comes at the cost of needing synchronization to avoid race conditions.

The most attractive feature of OpenMP for our eHealth processing framework is the magic loop parallelization. Element-wise operations on arrays can be performed simultaneously on different processors. Code 1 shows the basics of loop parallelization with OpenMP. In the example, all the variables are shared, except the iteration index `i`. The *reduction* directive indicates that each processor maintains a private copy of the shared variable `x`, and that these private copies are combined with the indicated operation at the end of the parallel execution area.

Code 1: Parallelized loop with OpenMP

```
#pragma omp parallel for reduction(+:x)
for (i=0; i < n; i++) {
  c[i] = a[i] + b[i];
  x += a[i];
}
```

The other main infrastructure used for parallelization is MPI [1], that is a language independent specification for interprocess *message passing*. MPI is designed for systems with distributed memory, cluster environments of single processor machines, but since version 3 it incorporates an extension for shared memory processing. It has several implementations that comply with the specification, like IntelMPI [22] and OpenMPI [15].

Although the current version of MPI is suitable for shared memory applications, OpenMP has a simpler syntax, being easier to use and to debug for shared memory systems like current symmetric multiprocessor computers.

Hybrid systems with different memory spaces, with any of them associated with several processors, are a challenge to application design. Existing applications using either OpenMP or MPI may implement the extensions of their current infrastructure to scale to the new scenario. Nevertheless, both platforms are compatible and can be used together carefully to increase the performance [31].

## *3.2. Execution control and instrumentation*

The fundamental functions for controlling the processor are provided by the operating systems and they are: process scheduling and memory management. The default scheduling policy for some operating systems (e.g. Linux) can be customized to suit different applications targets such as real-time. The following are some selected kernel functions to control the execution of activities and their assignment to individual cores. Table 1 shows the main facilities used for controlling the distributed eHealth processing.

Table 1: Activity processing support functions at the fog server

| **Function** | **Purpose** |
|---|---|
| Scheduling, process management | Prioritize eHealth activity execution |
| Threading and concurrency | Prevent shared data race conditions |
| Priorities and processor affinity | Core assignment to eHealth activities |
| Signals | Handle communication interrupts |
| File I/O | Log of eHealth activities'operation |
| Transport level transmission | Data communication |
| Real-time clocks | Precision delay measurements |
| Memory management | Efficient consumption avoiding leaks |

The proposed framework uses standard runtime support functions by means of POSIX threads (Pthreads) that is also used in the core distribution software to handle the concurrency accessing the received packets queue and through OpenMP for parallelizing the computations.

The required high precision execution time measurement is performed through `clock_gettime(clockid_t clk_id, struct timespec *tp)` method, selecting a high resolution clock (real time, wall time, or process time) with nanoseconds precision (typically $1ns$ resolution) to obtain the requested timestamp (`timespec`).

## *3.3. Communication backbone*

The data distribution service (DDS) [27] implementations can integrate with the standard operating system interface, Posix. Data distribution service provides publish-subscribe (P/S) interaction relying on the concept of a global data space where entities exchange messages based on their type and content. Types are based on the concept of *topics* that are constructions that enable the actual data exchange. Topics are identified by a unique name, a data type and a set of *quality of service* (QoS) policies. Topics can use *keys* that enable the existence of different instances of a topic so that the receiving entities can differentiate the data source.

One of the most successful elements of the data distribution system standard is the set of quality of service parameters that it defines, namely *QoS policies*. Some of these policies are related to the temporal behavior of the communication; others provide different guarantees over the data transmission. In the proposed distribution model, the policies that influence the communication time and the overhead of the system are indicated below:

- *Deadline* is the maximum expected elapsed time between arriving data samples (unkeyed data) for the data readers; and it is the maximum committed time to publish data samples or instances for the data writter.
- *History* stores the sent or received data in cache. It affects Reliability and Durability (receive samples sent prior to joining) QoS policies.
- *Resource limits* is the limit to the allocated memory (message queues for history, etc.). It limits the queue size for History when the Reliability protocol is used.
- *Latency budget* is an indication on how to handle data that requires low latency. It provides a maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.
- *Timed based filter* limits the number of data samples sent for each instance per a given time period.
- *Transport priority* establishes a priority for the data sent by a writer. The accomplishment of the priority indicated by transport priority is actually dependent on the transport characteristics and also supported only by some operating systems.
- *Reliability* is a global policy that specifies whether or not data will be delivered reliably. It can be configured on a per data writer or data reader basis.

### *3.4. Security concerns at the fog*

Current research challenges in fog computing are many and highly diverse such as data breaching. In general, the risk of malicious insider attacks and other privacy problems increases in fog environments. Here, it is presented an overview of the security issues in the fog infrastructure in relation to the execution of eHealth services.

Security can be viewed from two different perspectives: (1) design of algorithms for identification, non-repudiation, authorization, data integrity, encryption, etc.; and (2) providing low-cost execution infrastructures that improve the execution speed and response times of the former algorithms. This contribution deals with (2), but below, we acknowledge the major challenges and their implications for eHealth:

- *Malicious fog nodes* can cause data breach, that call for efficient and secured end-to-end schemes for IoT devices to perform data storage and access. For this, encryption (or proxy-encryption) and low cost end-to-end schemes need be designed.

- *Malicious insider attacks* may steal private key and access user data. A possible solution is to deploy decoy technology to reduce the amount of stolen data; but it is needed to select the right location of this technology placement in the fog.

- *Fog forensics* [34] provides evidence by reconstructing the past events in the fog environment. However, this brings in an additional problem such as how to retrieve log data from the large number of possible fog nodes. There are some contributions such as [7] that identifies how fog and cloud forensics differ, requiring international rules for addressing cross-border challenges.

- *Privacy preservation* is concerned with guaranteeing the undisclosure of information such as the identity or location of a node (i.e., that can be attached to a physical object such as a human user or a unmanned aerial vehicle, etc.). There are a number of techniques to solve this such as pseudonym modification, group signatures (for disclosure selection to specific nodes), etc. Challenges such as integration of such nodes needing high privacy preservation is needed with low cost mechanisms that reduce the costs of storage, computation complexity, communication overhead, or additional delays.

Controlling the execution of the activities inside the processor can contribute to preserving data security. Confining the execution of certain code functions to a specific core, decreases the exposure of data to other code functions. On the first place, this decreases the execution time of selected code functions (e.g., those serving a priority patient) as cache invalidations due to tranfer of the function to another core are avoided. On the second place, if a priority function runs on a given core and other functions (possibly malicious) are not allowed to run there, the data cache will be protected.

## 4. Computation within the fog servers

This section presents the execution model for intelligent fog services from two different perspectives in a bottom-up fashion: low level execution control and the application level logic. Firstly, it is described the fog computing model that is applied to improve servicing of patients, basically by computation offloading to fog servers. Secondly, the model for accelerating the execution of eHealth services in fog servers is described.

### *4.1. Fog model*

The proposed framework is based on the vision of fog computing that is sketched in Figure 1.

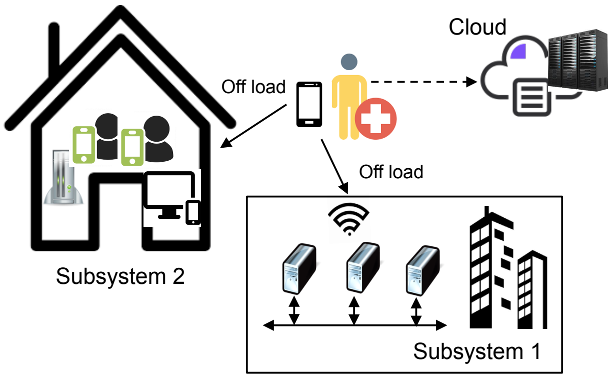

Figure 1: Computation offloading

In it, there are two different subsystems connected to the Internet and to the cloud. Subsystem 1 is configured as a LAN to which a number of heterogeneous nodes are connected, ranging from high-end servers to other IoT sensors and devices; this corresponds to some organization or especialized environment such as a hospital, care center, or possibly the smart building where the targeted patient lives. Subsystem 2 is private space also configured as a LAN, possibly with a partly adhoc infrastructure, with an overall limited processing capacity if compared to Subsystem 1 or the cloud; the latter corresponds posibly to the patients' home.

The basic principles of the fog computing model that is the basis for this proposal are are the following:

- *Computation offloading.* In a cloud centered design, the computationally complex services are run at the cloud. In a fog computing environment, part of these complex services can be run at nodes located close to the point of delivery of the results. Figure 2 shows this by presenting the software structure of two fog nodes: a fog server and a patient node. Selected processor intensive activities such as face recognition, voice interpretation, or health recommendations can be run at the fog nodes close to the patient. Computation offloading results in battery savings at the IoT nodes and specifically at mobile patient nodes as it implies that there is a drastic reduction in the amount of data transmitted to the cloud.

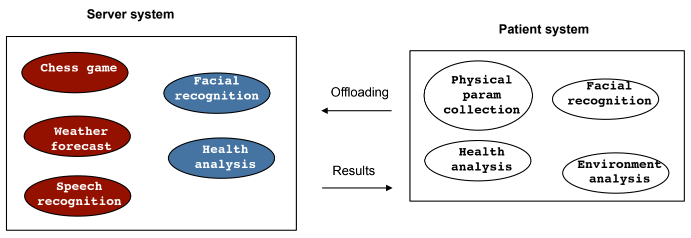

Figure 2: Using the fog servers for offloading computationally intensive eHealth services

- *Latency.* The networks used to connect to a cloud server typically exceed the LAN boundaries, and WAN latencies increase significantly. In fog computing, data is sent to the fog servers in the vicinity at LAN speeds, i.e., with very low latencies. If improved latency is required, the computation intensive activities of mobile patient clients will be offloaded to more powerful devices within the fog (see figure 2) that are the middle end servers.

- *Network bandwidth.* The cloud model has scalability issues in data intensive application domains. It incurs in high network bandwidth consumption with an associated that could be prohibitive. Exchanging data across the fog nodes, mostly in the LAN boundaries will impresively decrease network bandwidth and the derived costs.

- *Security.* As data can be partly stored at the fog nodes and need not circulate to the cloud, systems are less exposed to network security breaches. Nevertheless, other security problems arise such as those described in §3.4.

In this framework, more efficient execution in the fog servers is possible, providing lower service time to the offloaded activities by appropriately controlling the execution on them. As the fog servers are typically middle end nodes, they have multicore servers that can be exploited to accelerate the execution of the ehealth services.

## *4.2. Low level control: Execution acceleration*

Current computers, ranging from high-end servers to low cost single board computers, have multicore processors that support parallel execution. This characteristic can be exploited to increase the computation power offered to the offloaded eHealth services, providing them the following benefits:

- To prioritize those activities which are of higher urgency that will allow to offer differentiated service to higher priority patients;

- To decrease the response time of the servicing so that interactivity is improved for the requesting patients;

- To improve data security due to the isolation of the execution of some activities which may manipulate sensible data.

Using the general purpose scheduling facilities of the operating system does not well address the priorization nor the decreased response-time arguments. The default scheduling of highly efficient operating systems such as Linux aims at lowering the average service times of all the activities which are run; this is not suitable in situations where there is an outstanding preferred activity that should always be run in the first place.

Figure 3 exemplifies the two possible execution situations when a user requests to offload some computationally intensive activities ($A_{in}$, $A_{1,1}$, $A_{1,2}$, $A_{2,1}$, $A_{2,2}$, $A_{out}$) on the fog. Default scheduling targets at avoiding data dependency effects by intensively using a single core; in this situation spliting activities to other cores does not take place unless the current core is near 100% of its capacity.

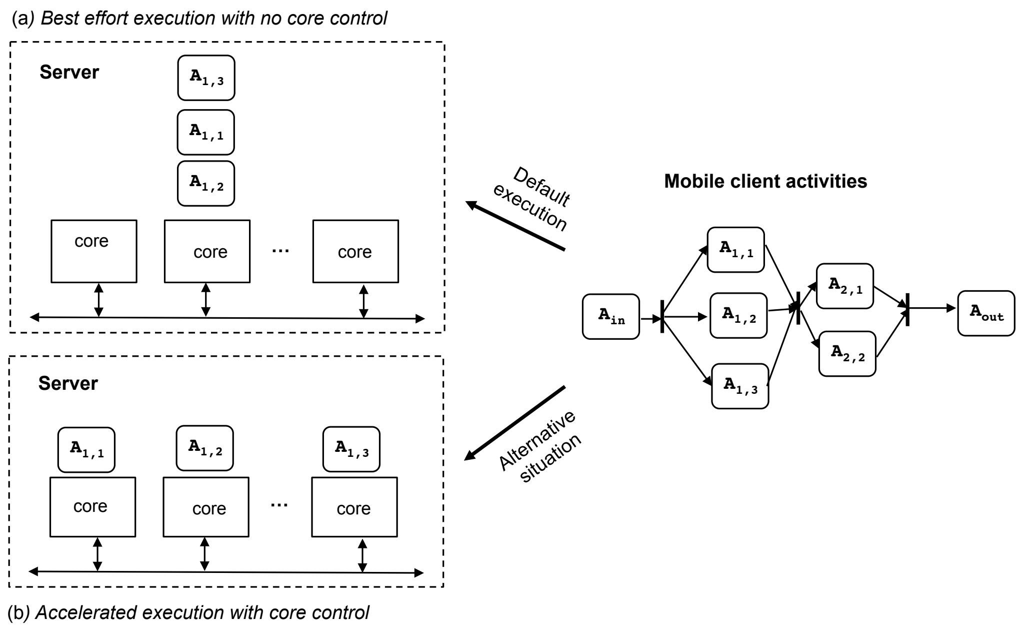

Figure 3: Execution possibilities on a fog server.

Consequently, figure 3(a) shows the typical execution of three of the activities of the mobile client; although they are parallelizable, they are run sequentially in one of the cores. This will typically yield acceptable response times; however, it is not desirable in the event of some unexpected situation in which the offloading system is heavily loaded or in the event that some activity of the mobile client has higher priority over the rest where the execution alternative of figure 3(b) would be required. In (b), it is possible to assign specific parallelizable tasks to cores and even reserve some cores for some specific task.

## 5. Fog computing architecture with hardware awareness

### *5.1. System model*

A server node of a fog computing environment can run services on behalf of requesting mobile nodes. A service $s_k$ (e.g. a face recognition functionality) has, or is implemented by, a set of activities $a_i$. Then, $s_k = \{a_i\} \quad \forall i = 1, ..., na(s_k)$, where $na(s_k)$ is the number of activities of a service, in this case of service $s_k$. Following, it is summarized the notation used in the framework:

- $\mathcal{N}$: Set of all fog server nodes.
- $\mathcal{S}$: Set of all services.
- $S_j$: The set of services running on noje $j$
- $s_k$: A service, such that $s_k \in \mathcal{S}$.
- $\mathcal{A}$: Set of all activities.
- $a_i$: Activity $i$, such that $a_i \in \mathcal{A}$.
- $A_j$: Set of activities of node $j$.
- $na(s_x)$: Number of activities of a service $s_x$.
- $hp(i)$: Set of activities with higher priority than, such that $a_i$.
- $C_i$: Computation time of an activity $i$.
- $T_i$: Activation period of an activity $i$.
- $P_i$: Priority of an activity $i$.
- $U_i$: Utilization of $i$.
- $U$: Utilization of server node.

Let $j$ be a server node in the fog computing environment that can be used for activity offloading. Node $j$ runs a set of activities $A_j$ where $A_j \in \mathcal{A}$ and $\mathcal{A}$ is the set of all activities that can be run in any computation node. A given mobile client may request a set of services $W$ to be run on server $j$.

For the server to provide the requested service to the mobile clients, it must make sure that it has sufficient spare computation power, that is, if the total utilization comprising the newly requested activities is less than a specified system wide threshold value $\gamma$. This is done via a utilization based technique as illustrated in algorithm 1.

The utilization of an activity $a_i$ on a specific node is $U_i$; $U_i$ is calculated as the fraction of the processor that it consumes during its activation period. For simplicity, it is assumed that all activities can follow a periodic activation pattern.

$$U_i = \frac{C_i}{T_i} \tag{1}$$

Given that $W$ is the set of services requested by the mobile client, the utilization of a node $n_j$ is the sum of the partial utilizations of all the activities that it executes, and this is shown in eq. (2).

$$U = \sum_{x=1}^{s_x \in (S_j \cup W)} \sum_{i=1}^{na(s_x)} \frac{C_i}{T_i} \tag{2}$$

**Algorithm 1** Utilization of the tasks offloaded to a server $j$

```
1: procedure UTILIZATION_CALCULATION
2:     $l \leftarrow \gamma$
3:     $U \leftarrow 0$
4:     for $a_i \in (A_j \cup W)$ do
5:         $U_i \leftarrow \frac{C_i}{T_i}$
6:         $U \leftarrow U + U_i$
7:     if $U \leq l$ then
8:         $A_j = A_j \cup A_W$
9:         for $a_i \in A_j$ do
10:            start($a_i$)
11:    return $U$
```

As mentioned before, $W$ is the set of services that a mobile client requests to be run on a server $j$. The set of activities contained in $W$ is $A_W$. Therefore, if the offloading is accepted, the current set of activities of the node is extended with $A_W$ as expressed in equation (3).

$$A_j = A_j \cup A_W \quad \text{such that} \quad A_W \subset \mathcal{A} \tag{3}$$

## *5.2. Architecture*

The proposed architecture (see figure 4) contains a set of modules that enhance the logic of the operating system by adding the capacity to: $(i)$ control the execution of parallelizable activities of the offloaded services; and $(ii)$ to enforce individual activities over specific hardware cores. Data communication and distribution facilities are also available to support the execution

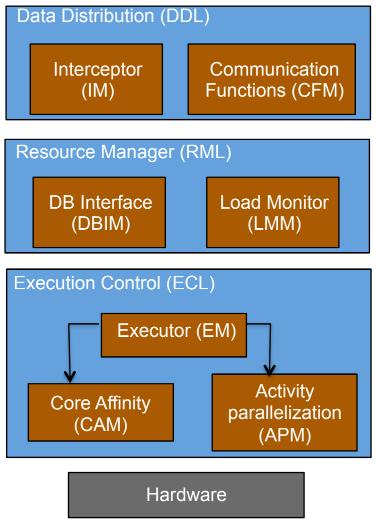

Figure 4: Architecture for hardware accelerated execution of computationally intensive services

of the offloaded activities with the indicated data upon request from external entities such as mobile clients.

The main functional blocks that support hardware accelerated computation are divided by layers as follows:

- *Data Distribution* Layer (DDL) that contains the functions responsible for the interaction with the outside clients:
    - *Communication Functions* Module (CFM) that provides the basic functions to distribute the data among the participating nodes, basically between the mobile client and the fog server. This module implements a communication interface for asynchronous data communication (efficiency and timeliness is achieved through using UDP/IP). The base technology used for the realization of this module is the data distribution system by means of data-centric topic based transmission.
    - *Interceptor* Module (IM) is an active entity that acts as a first interception point to collect requests, analyze them, and serve them at the fog server side.
- *Resource Manager* Layer (RML) checks the feasibility of the requested services by running the schedulability analysis algorithm of equation

(2). This layer has the view over the current execution load of the server node. This layer detects anomalous situations when the current load does not allow the execution of additional service functions; then, an unavailability response to the mobile client is provided.

  - *Load Monitor* Module (LMM) calculates whether it is feasible to run the requested service by performing a utilization based analysis that considers the current load of the server activities and of the activities of the new service.
  - *Data Base Interface* Module (DBIM) is the entry point to the data base of the server that contains the relevant execution information, i.e., the set of services and activities of the server, and the execution parameters of each of them. The LMM accesses the data base though this interfacing module.

- *Execution Control* Layer (ECL) arbitrates the execution of the activities over the processor cores according to their priority value, security level, and based on if they can be run in parallel or not.

  - *Core Affinity* Module (CAM) controls the execution of specific activities over selected cores of the multicore processor. It reserves specific cores to run given tasks based on priority by means of processor affinities.
  - *Activity Parallelization* Module (APM) performs the parallelization of the application level computations; those activities that can execute in parallel are effectively run in different cores simultaneaously.
  - *Executor* Module (EM) controls the execution of the activities by assigning them to specific cores or by explicit parallelization depending on their nature by using the functions provided by CAM and PAM, respectively.

Figure 5 shows the main system component and their interactions as explained above.

## *5.3. Remote service offloading*

Figure 6 shows the distributed architecture for the communication among mobile client and fog server.

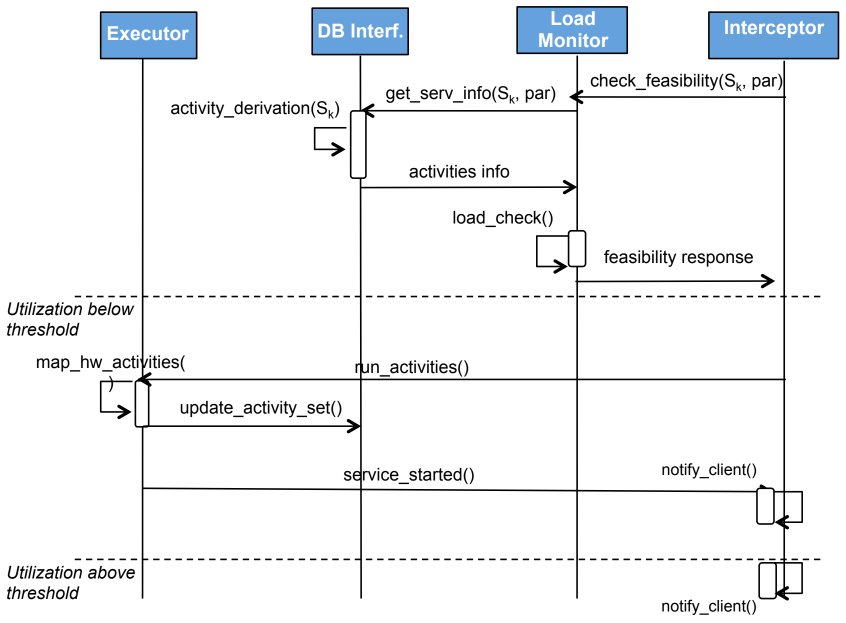

Figure 5: Behavioral diagram: interaction across system components

An asynchronous data-centric scheme is used; the communication is based on publish-subscribe (P/S) interaction on a global data space where mobile client and fog server are part of the same domain and exchange messages based on their type and content through topics. Two topics are created and identified by unique names that store both the communication from mobile client to server (*OffloadRequestType*) and viceversa (*ResponseType*). These topics use keys to handle different instances of each topic so that the receiving entities can differentiate the data source, i.e., the mobile client or server that the data comes from.

The distributed interaction between mobile client and server is organized as a domain; a domain defines an application range where communication among related entities (mobile clients soliciting eHealth recommendation) can be established. A domain becomes alive when a participant is created. A participant is an entity that owns a set of resources such as memory and transport. If an application has different transport needs, then two participants can be created.

The topic *OffloadRequestType* (see code 2) is used to communicate the data sent from mobile clients to server. The information provided refers to

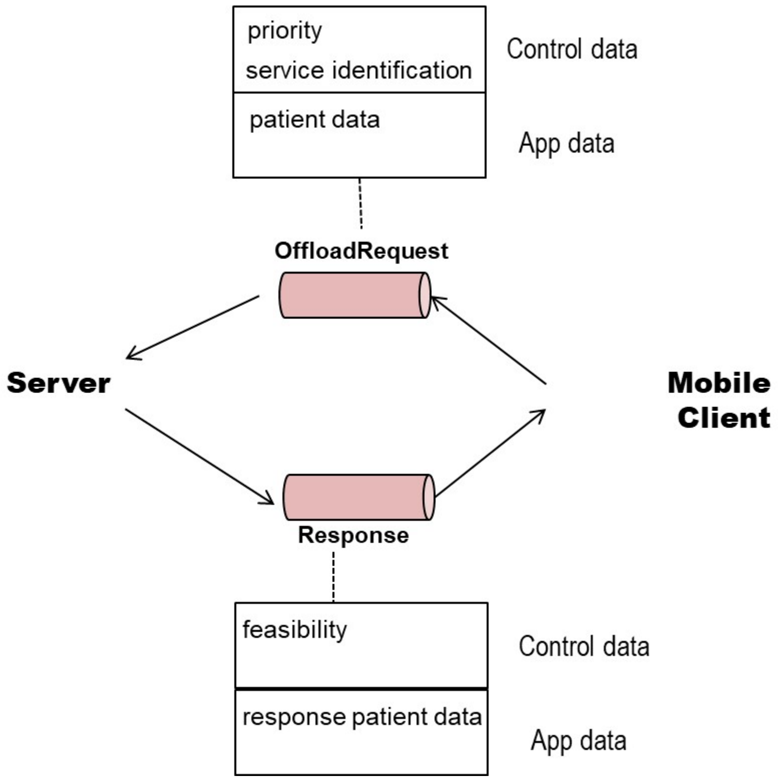

Figure 6: Distributed configuration for the remote interaction between mobile clients and offloading servers

the identifer of the source, the requested priority, the unique string with the requested service name (e.g., *eHealth recommender* or *VoiceInterpreter*, etc.) and a data stream that provides the mobile client data for the server (e.g. the voice stream or the health parameters of the day).

Code 2: Topic structure template

```
struct OffloadRequestType{
 short nodeid;
 short mcid;
 short priority;
 string serv;
 sequence<octed, TOPIC_MAX_INFOREQ_SIZE> reqdata;
};
#pragma keylist OffloadRequestType nodeid
```

The topic *ResponseType* (see code 3) stores the information sent from server to the mobile client in response to the request to offload a service. The contained data refers to the feasibility of such offloading and the processed data produced by the service that has been executed. In the event that the server is malfunctioning or its load is too high to serve the client request, the feasability string returns an indication of *service temporarily unavailable*.

Code 3: Topic structure template

```
struct ResponseType{
 short mcid;
 short nodeid;
 string feasibility;
 sequence<octed, TOPIC_MAX_INFOFLOW_SIZE> data;
};
#pragma keylist OffloadRequestType mcid
```

A set of quality of service policies are put in place to ensure that the communication channel is reliable:

- *RELIABILITY* is set to RELIABLE for guaranteed message delivery, and it is set both for data reader and data writer that are the server or mobile client depending on the communication direction.

- *HISTORY* is set to KEEP_ALL that guarantees that samples will be retained until the subscriber retrieves them. It is set for data reader that is also the server or mobile client in this case.

## *5.4. Execution control*

Hardware reservation techniques are used to allow mobile clients to express whether or not they require priority service. As a result, the system can reserve some specific processing core for the execution of selected activities. Reserving cores improves the processing time, i.e., the response time to clients. Also, hardware reservation provides enhanced security as it minimizes the interference with the code and data from other activities running simultaneously on the server.

To control the hardware cores and to enforce the execution of activities in specific cores the following functions are used:

- `setCPUaff(int cpu)`; a given entity will run exclusively in the designated processing core (`cpu`). This enables the control of the execution of higher priority activities over lower priority ones as the entity (thread) servicing low priority activities is not assigned to reserved cores. A subsequent effect is that the context switches during the execution of higher priority requests are reduced.

- `setCPUsched(int priority, int policy)` is used to set the priority of specific threads and the scheduler policy for thread dispatching. For the high priority threads, the real time scheduling policies (SCHED_FIFO and SCHED_RR) are used which allows to set some selected threads as real-time and these will always be scheduled before the non real-time ones. As a consequence, the execution is further controlled.

Code 4 shows the runtime allocation of an activity run by a thread to a specific thread set.

Code 4: Activity allocation

```
cpu_set_t cpuset;
CPU_ZERO(&cpuset);
CPU_SET(i, &cpuset);
ActivityThreadPtr t = new ActivityThread;
pthread_t t_id = t->start().id();
pthread_setaffinity_np(t_id, sizeof(cpuset), &cpuset);
```

The eHealth service running in the fog server has a set of activities of two types: dependant and independent. The first ones have data dependencies and have to be run sequentially, whereas the later ones are decoupled and can be run simultaneously in different cores.

There are three main strategies for expressing binding of activities to cores:

- filling up one processor socket before allocating activities to other sockets.

- allocating activities evenly across all sockets and cores.

- explicit allocation only to those cores and sockets that are indicated.

For the first alternative, activities are allocated one per core, to a single socket prior to binding to other sockets. Linux follows the same pattern to start at socket 0 regardless of whether other activities have already been bound to that socket. Setting an offset to force the binding to another thread is possible if the hardware configuration is known. For example, to explicitly allocate to cores 0 and 2, the following envirornment variable can be used: `export KMP_AFFINITY='proclist=[0,2],explicit'`.

Parallelization infrastructures (e.g. OpenMP) also allow to define activity affinity at runtime. For example to run on cores 0 and 2, and environment variable can be used such as `export GOMP_CPU_AFFINITY='0,2'`.

## 6. Experimental validation

The proposed framework has been implemented in C++ on a POSIX compliant middleware based on DDS standard [27] that provides publish-subscribe communication. The prototype setting is partly simulated; a real distributed deployment with homogeneous nodes of medium processing capacity is used, which replicates the capacity of the hardware used in a legacy fog server. The server runs on an Intel E3400 double core with 1MB cache, 2.60 GHz, 800 MHz FSB, and 2GB RAM, running Ubuntu 10.04 Linux of 32 bits and kernel 2.6.32. Experiments have been carried out for a fog server running synthetic eHealth offloaded services to which only a limited $n$ number of remote patients (that are mobile clients) can perform simultaneous requests.

The architecture supports prioritization of certain patient requests; therefore, the system supports high and low priority mobile clients. Priorization is provided by the system as an interface call. In the experiments, half of the tested remote patients are high priority and half are normal priority. This is performed to test the effect of the designed system in a proper synthetic example with multiple remote patients operating simultaneously that belong to different priority groups and, therefore, compete for service.

To test the achieved response time values, the system is stressed in a way that each mobile patient performs performs periodic requests. The chosen period is $100ms$. Upon each request, the fog server launches the requested eHealth service that is implemented by a set of concurrent activities run by operating system threads. High priority requests are serviced by high priority threads for which processing cores are reserved; these high priority threads are real-time ones and they use the SCHED_RR scheduling policy at priority

99. Low priority requests are serviced with the default CFS (*completely fair scheduler*) scheduling policy without core reservation.

A number of experiments have been carried out to test the response time (i.e., the time taken to perform the processing of the requested service and associated data) and the capacity to tolerate noise at the fog server. On the one hand, improvement of the respose time is important for increasing the interaction capacity between patient nodes and the fog servers; it will support faster and more timely results delivery to patients. On the other hand, handling noise is key to offering a robust execution environment where the normal operation on behalf of patient nodes is not altered by the rest of unrelated activities that may be already running on the fog servers.

In the first tests, the service time for multiple clients is measured. To show the support of priorization, these mobile patients are placed in two different priority groups. The requested offloaded services run by the fog server are computationally intensive operations simulating the computations required by the eHealth services. Results are displayed in groups of 10 executions, that are expressed as *client position*.

To show the advantage of the framework, first the baseline experiment is performed as follows. Figure 7 shows the response times on a fully distributed setting in a situation where all mobile clients have equal priority. The figure shows the response times experienced by dispatched clients over an unmodified communication library that is not aware of the hardware architecture nor the core reservation. This is the baseline scenario representing the best effort case that is put in place by current server side eHealth servicing techniques.

Figure 8 displays the response times experienced by mobile clients that are dispatched by the proposed architecture that is aware of the multicore structure of the hardware. Requesting clients are divided in two groups: a high priority one and a low priority one.

It is observed that the core reservation logic for given activities of $\mathcal{A}$ set has a negligible effect on the service time of the clients. The average cost of the reservation is in the order of 1 to 10 $\mu s$. Higher priority clients are allocated to cores that are reserved for them; consequently, they experience shorter service times with a smaller variation between their minimum and maximum values. Also, it can be seen that low priority clients have shorter service times.

Without service reservation, all clients have similar service times regardless of their priority; this is not an ideal situation in the presence of mobile patients that require faster response time and priority service. Figure 8 shows

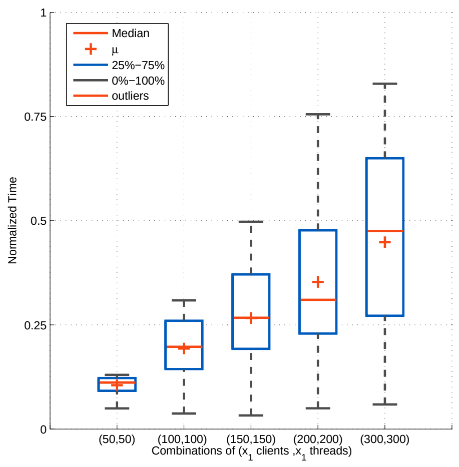

Figure 7: Remote processing time (in milliseconds) for different load conditions

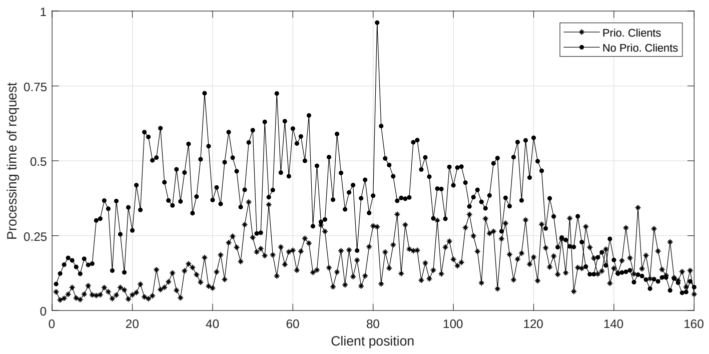

Figure 8: Processing time for priority and non-priority activities with core reservation

the service times for clients over the implemented architecture that has processing priorization through core reservation. It is seen that high priority clients (shown on the upper curve) have shorter service times as compared to low priority clients (shown on the botton curve of the graph); in fact, high priority clients service times are improved by an average of 3x as comparted to the situation where the proposed architecture is not run. The dispersion of the service times is also improved and it is smaller than that of low priority one; and the worst case service time priority is in the order of 2.5x smaller than the service time for low priority clients. In summary, low priority clients experience larger service times and also the variation in their service times is larger than that of higher priority clients.

Also the evaluation and analysis of the performance for distributed eHealth service offloading is performed. The proposed architecture execution layer that is aware of the multicore uses parallelization functions for running specific computation activites of the eHealth services within specific cores.

To provide a more realistic setting, the execution hardware is now more powerful by using Intel i7-5600U processor at 2.60 GHz with 2 cores and hyperthreading, meaning that 4 threads can run in parallel. A Linux 64 bit and 3.16 kernel is used.

The data exchange is done by using octets of 1024 elements in a stream like transmission simulating the communication of big data. Communication is unidirectional through topics and one topic is used for each direction (among fog server and mobile patients). By using a publish-subscribe data centric communication, several mobile patients can transmit data to the server side that will handle the requests through the executor component and service the request by checking the available spare processing capacity of the server. Simultaneous requests are handled concurrently at the server and the requested service involving the received data is processed efficiently to guarantee a maximum bound on the delay of the service processing.

The processing of eHealth parameters runs as a service (eHealth service) in the fog server. The eHealth service runs as a nested loop; the outher loop updates a control variable that runs the part of the health analysis that is not parallelizable, i.e., that performs calculations on previous results. The inner loop performs column-wise operations on the patient data matrix. This inner loop is parallelized with an OpenMP pragma directive like the one showed in code 5.

Code 5: Robustness validation through controled noise generation and corresponding core assignment

```
noiseLevel = loadLevel;
for (int i = 0; i < sysconf(_SC_NPROCESSORS_ONLN); ++i)
    {
  cpu_set_t cpuset;
  CPU_ZERO(&cpuset);
  CPU_SET(i, &cpuset);
  NoiseThreadPtr t = new NoiseThread;
  pthread_t t_id = t->start().id();
  pthread_setaffinity_np(t_id, sizeof(cpuset), &cpuset)
      ;
}
```

In this code, the logic to enforce code assignment is partly shown as part of the evaluation of the robustness of the eHealth service execution in the presence of different load conditions. Threads are assigned core affinities to measure the actual exeution times and interference for the eHealth activities. To replicate realistic execution conditions at the fog server, noise control is tested; for this, interfering execution is created explicitly.

A realistic set of load conditions at the fog server are synthesized by designing an internal hook that can dynamically run a thread that creates the noise (DummyService); this enables the validation of a range of execution conditions that affect the processor load. One noisy thread is created per core and started in the system and the core affinity binding is set for all the threads to different processors. These perform operations that merely consume processor cycles with a microsecond idle time in between two sonsecutiove repetitions. The complexity of the generated load is controlled with a parameter to set the noise level that can be modified as desired. When the maximulm noise level is set, the noisy threads skip their waiting and start consuming all their processor slice before they are preempted. similarly, when the noise level is set to zero the noisy threads exit the noise generation loop and terminate their execution.

The parallelization can be configured either dividing the computations statically or dynamically, and with different number of threads. The most straightforward solution is to split the computations in equal parts with one thread per core in order to maximize the utilization of the system and load balancing. It was evidenced experimentally that there is no significant difference between static or dynamic splitting and the best performance is achieved with one thread per core. Although this is true for the best case, it was found experimentally that dynamic splitting leaving one core free is a more robust configuration that leads to a shorter worst case processing time for all noise levels with a minimum difference in the best case performance.

The maximum number of simultaneous streams that the fog server can process is analyzed, i.e., the capacity of the system. As the intention is to obtain the timeliness information of the server, a real-time aproximation is used: each stream is processed by a task (i.e. an activity that is run by a real-time thread) and its packets (i.e., the data) are modeled as jobs. In this

Table 2: Processing times of the patient transmitted data in $ms$

| Noise level | 0 | | 50 | | 100 | |
|---|---|---|---|---|---|---|
| data | Serial | Parallel | Serial | Parallel | Serial | Parallel |
| mean | 1.83 | 1.33 | 1.85 | 1.28 | 3.30 | 0.98 |
| std | 0.39 | 0.29 | 0.04 | 1.10 | 0.17 | 0.79 |
| min | 1.3 | 0.83 | 1.72 | 0.96 | 2.13 | 0.86 |
| 25-perc. | 1.43 | 1.01 | 1.8 | 0.99 | 2.86 | 0.99 |
| median | 1.54 | 1.1 | 1.88 | 1.02 | 3.20 | 1.04 |
| 75-perc. | 1.59 | 1.19 | 1.92 | 1.16 | 3.39 | 1.12 |
| max | 6.02 | 4.21 | 2.42 | 17.66 | 3.74 | 16.55 |
| Number of measures | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |

way, it is possible to use a utilization based analysis as indicated in the logic for the load monitor component; also, it is possible to use a response time analysis to check whether all tasks are schedulable: a task $i$ is schedulable if it finishes before its deadline (being the deadline equal to its next activitation $T_i$ ).

The cost of stream processing depends on the actual data sent through the stream and, therefore, of the quality and precision of the transmitted data. Therefore, the capacity is given by the equation that follows:

$$nt(N) \leq \frac{1024}{d_c \cdot t_p} \tag{4}$$

where $nt(N)$ is the number of streams of the system, $d_c$ is the number of data items sent by the mobile client and $t_p$ is the time taken by the fog server to process a packet of data items sent by a mobile client. A stream generates a data item every $t_d$ seconds and these are sent to the server in groups of 1024 data items per packet.

As it is not possible to have a deterministic processing time, it is possible to use the $(100 - \delta)$ percentile, where $\delta$ is the tolerance of the system to data loss. For a tolerance to losses of 25%, a capacity of 6 streams is estimated for the serial data processing and up to 15 when parallelizing the data processing, depending of the parallelization scheme. The higher value of the 75-percentile is used among the different noise levels.

The waiting times caused by the preemptions can be dramatically reduced with a flexible parallelization configuration as the one provided by our proposed system. The increased robustness is explained by the fact that the engineered system is based on a parallelizing software infrastructure which reduces the waiting time of activities; the proposed architecture then reduces the time that tasks are waiting to be dispatched into a core. Even the parallel threads altogether suffer a total number of preemptions equal to those of a single process performing the same task.

## 7. Conclusion and future work

This paper has presented the design, implementation and evaluation of a framework for accelerating the response to mobile patients requiring the execution of smart eHealth services. The model supports distributed offloading to fog servers, using the capacity of multicore processors to accelerate its execution. Services are realized by a number of activities and those activities that are parallelizable can be allocated to reserved cores. The architecture integrates facilities for distributed offloading requests, for core allocation, and for activity parallelization. The framework is validated over a prototype implementation on a fully distributed scenario with synthetic services. Results show that response times of the mobile patient nodes are decreased when core reservation is activated, and that priority clients experience decreased service times. The presence of noise due to other unrelated activities running on the fog servers is managed in a robust manner.

Goal-oriented models can be used to raise the awareness of our proposed framework with respect to application logic [28]. As proposed in [29], application-oriented plans are ellaborated by deliberation and runtime engines, and they are later passed to the operating system scheduler. We have identified as future work the integration of this framework into a higher level structure comprising a runtime engine and a deliberation engine. Moreover, this can be combined with an additional logic to calculate the slack and gain times (e.g. [9]) of the multicore processor to be used for further accelerating the execution.

## Acknowledgement

This work has been primarily funded by the M2C2 (TIN2014-56158-C4-3-P) and PRECON-I4 (TIN2017-86520-C3-2-R), both funded by the Spanish Ministry of Economy and Competitiveness.

## References

## References

[1] Message Passing Interface Forum. http://www.mpi-forum.org/. (Accessed June 2017).

[2] The OpenMP® API specification for parallel programming. http://www.openmp.org/. (Accessed June 2017).

[3] M. Aazam and E. Huh. Fog computing and smart gateway based communication for cloud of things. In *2014 IEEE International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 464–470. IEEE, 2016.

[4] A. Abdelaziz, M. Elhoseny, A. S. Salama, and A. Riad. A machine learning model for improving healthcare services on cloud computing environment. *Measurement*, 119:117 – 128, 2018.

[5] S. H. Ahmed and S. Rani. A hybrid approach, smart street use case and future aspects for internet of things in smart cities. *Future Generation Computer Systems*, 79(Part 3):941 – 951, 2018.

[6] A. Bahtovski and M. Gusev. Cloudlet challenges. *Procedia Engineering*, 69:704–711, 2014.

[7] S. Biggs and S. Vidalis. Cloud computing: The impact on digital forensic investigations. In *2009 International Conference for Internet Technology and Secured Transactions, (ICITST)*, pages 1–6, Nov 2009.

[8] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu. Fog computing: A platform for internet of things and analytics. volume 546 of *Studies in Computational Intelligence*, March 2014.

[9] L. Búrdalo, A. Terrasa, A. Espinosa, and A. García-Fornes. Analyzing the effect of gain time on soft-task scheduling policies in real-time systems. *IEEE Transactions on Software Engineering*, 38(6):1305–1318, Nov 2012.

[10] B. Chapman, G. Jost, and R. Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[11] M. Elhoseny, A. Abdelaziz, A. S. Salama, A. Riad, K. Muhammad, and A. K. Sangaiah. A hybrid model of internet of things and cloud computing to manage big data in health services applications. *Future Generation Computer Systems*, 2018.

[12] M. Elhoseny, G. Ramírez-González, O. M. Abu-Elnasr, S. A. Shawkat, A. N, and A. Farouk. Secure medical data transmission model for iot-based healthcare systems. *IEEE Access*, 6:20596–20608, 2018.

[13] B. Eze, C. Kuziemsky, and L. Peyton. Cloud-based performance management of community care services. *Journal of Software: Evolution and Process*, pages e1897–n/a.

[14] M. Firdhous, O. Ghazali, and S. Hassan. Fog computing: Will it be the future of cloud computing? In $3^{rd}$ *International Conference on Informatics and Applications*, pages 8–15, 2014.

[15] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[16] M. García-Valls, T. Cucinotta, and C. Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726 – 740, 2014.

[17] M. García-Valls, A. Dubey, and V. Botti. Introducing the new paradigm of Social Dispersed Computing: Applications, technologies, and challenges. *Journal of Systems Architecture*, 2018.

[18] M. García-Valls, I. R. Lopez, and L. Fernández-Villar. iland: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. *IEEE Trans. Industrial Informatics*, 9(1):228–236, 2013.

[19] M. García-Valls, D. Perez-Palacin, and R. Mirandola. Pragmatic cyber physical systems design based on parametric models. *Journal of Software and Systems*, –(–):–, 2018.

[20] N. Golov and L. Rönnbäck. Big data normalization for massively parallel processing databases. *Computer Standards & Interfaces*, 54(Part 2):86 – 93, 2017. SI: New modeling in Big Data.

[21] R. Hamza, K. Muhammad, A. Nachiappan, and G. R. González. Hash based encryption for keyframes of diagnostic hysteroscopy. *IEEE Access*, pages 1–1, 2017.

[22] M. Intel. Benchmarks: Users guide and methodology description. *Intel GmbH, Germany.*

[23] A. Jadhav, D. Andrews, A. Fiksdal, A. Kumbamu, J. McCormick, A. Misitano, L. Nelsen, E. Ryu, A. Sheth, S. Wu, and J. Pathak. Comparative analysis of online health queries originating from personal computers and smart devices on a consumer health information portal. *Journal of Medical Internet Research*, 16(7), 2014.

[24] B. Kuhn, P. Petersen, and E. O'Toole. Openmp versus threading in c/c+. *Concurrency: Pract. Exper*, 12:1165–1176, 2000.

[25] K. Muhammad, R. Hamza, J. Ahmad, J. Lloret, H. H. G. Wang, and S. W. Baik. Secure surveillance framework for iot systems using probabilistic image encryption. *IEEE Transactions on Industrial Informatics*, pages 1–1, 2018.

[26] M. Mukherjee, R. Matam, L. Shu, L. Maglaras, M. A. Ferrag, N. Choudhury, and V. Kumar. Security and privacy in fog computing: Challenges. *IEEE Access*, 5:19293–19304, 2017.

[27] Object Management Group. A data distribution service for real-time systems version 1.4, Jan 2007.

[28] J. Palanca, M. Navarro, A. García-Fornes, and V. Julian. Deadline prediction scheduling based on benefits. *Future Generation Computer Systems*, 29(1):61 – 73, 2013.

[29] J. Palanca, M. Navarro, V. Julian, and A. García-Fornes. Distributed goal-oriented computing. *Journal of Systems and Software*, 85(7):1540 – 1557, 2012.

[30] J. Qi, P. Yang, G. Min, O. Amft, F. Dong, and L. Xu. Advanced internet of things for personalised healthcare systems: A survey. *Pervasive and Mobile Computing*, 41(Supplement C):132 – 149, 2017.

[31] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[32] M. Sajjad, M. Nasir, K. Muhammad, S. Khan, Z. Jan, A. K. Sangaiah, M. Elhoseny, and S. W. Baik. Raspberry pi assisted face recognition framework for enhanced law-enforcement services in smart cities. *Future Generation Computer Systems*, 2017.

[33] A. Shehab, M. Elhoseny, K. Muhammad, A. K. Sangaiah, P. Yang, H. Huang, and G. Hou. Secure and robust fragile watermarking scheme for medical images. *IEEE Access*, 6:10269–10278, 2018.

[34] Y. Wang, T. Uehara, and R. Sasaki. Fog computing: Issues and challenges in security and forensics. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 3, pages 53–59, July 2015.