Timely Provisioning of Mobile Services in Critical Pervasive Environments

Filippos Papadopoulos, Apostolos Zarras, Evaggelia Pitoura, and Panos Vassiliadis

Computer Science Department, University of Ioannina, Greece
{filip, zarras, pitoura, pvassil}@cs.uoi.gr

Abstract. Timeliness in conventional real-time systems is addressed by employing well-known scheduling techniques that guarantee the execution of a number of tasks within certain deadlines. However, these classical scheduling techniques do not take into account basic features that characterize today's critical pervasive computing environments.
In this paper, we revisit the issue of timeliness in the context of pervasive computing environments. We propose a middleware service that addresses the timely provisioning of services, while taking into account both the mobility of the entities that constitute pervasive computing environments and the existence of multiple alternative entities, providing semantically compatible services. Specifically, we model the overall behavior of mobile entities in terms of the entities' lifetime. The lifetime of an entity is the duration for which the entity is present and available to other entities. Given a new request coming from a mobile client and a number of semantically compatible mobile entities that can fulfill the request, one of them must be selected. The proposed service realizes three different policies that facilitate the selection. With respect to the first policy, the selection is realized solely on the basis of the client's and the server's lifetimes. The second policy additionally considers the load of each server towards selecting the one that guarantees to serve the new request within the lifetime of both the client and the server. The third policy further deals with periodic service requests.

1 Introduction

Recently, the rapid emergence of WiFi and Bluetooth networks, along with the increasing computing and communication capabilities of mobile devices such as PDAs, Pocket PCs, Smart Phones and wireless-enabled laptops, foster the development of a variety of new applications towards the realization of the overall idea of pervasive computing. Enterprises facilitate their activities for their mobile employees. Airports, railway stations, cafes and shopping centers deploy wireless networks to serve their customers. The evolution of the aforementioned technologies further enables the realization of applications that can be employed to handle certain critical situations like accidents, natural catastrophes, war situations, etc.

Both daily and critical applications are characterized by the following main features:

- They consist of a set of mobile entities, providing a number of services that can be requested by other mobile entities.
- More than one mobile entity may provide semantically equivalent services.

Critical applications are further characterized by the need for *timely provisioning* of services from mobile entities to other mobile entities. Service requests come along with specific *deadlines* that should be met by the mobile entities that serve those requests. Timeliness in conventional real-time systems is addressed by employing well-known scheduling techniques such as the earliest deadline first (EDF) and the rate-monotonic scheduling (RM) [1]. These techniques guarantee that the execution of a number of tasks will take place within the required deadlines. However, the classical scheduling techniques do not take into account basic features of pervasive computing environments.

In this paper, we revisit the issue of timeliness in the context of pervasive computing environments. Specifically, *we propose techniques that address the timely provisioning of services, while taking into account (i) the mobility of the entities that constitute pervasive computing environments and (ii) the existence of multiple alternative entities, providing semantically compatible services.*

The behavior of mobile entities may be rather complicated and depends on several factors [2]. For instance, so far, there has been work towards estimating the physical motion of mobile entities [3]. An entity may become inaccessible by moving into areas that do not belong in the transmission range of a particular wireless network. Another important feature that distinguishes mobile entities from the basic building blocks of conventional distributed systems is their limited resources. For example, the limited battery of a mobile entity may render the entity permanently or temporarily inaccessible. Moreover, the entity may explicitly disable its communication or computation capabilities towards the economization of power, or because of the reception of orders from some external authority.

To facilitate the timely provisioning of services in pervasive computing environments we employ a generic notion for modeling the behavior of mobile entities. This notion shall serve as input to the scheduling techniques that we propose in this paper. Specifically, *we assume that the overall behavior of mobile entities is modeled in terms of the entities' lifetime.* The *lifetime* of an entity is defined as *the time interval during which the entity is available to other entities.* The lifetime is generic enough and can be evaluated using a combination of different means, reflecting various characteristics such as the entity's physical motion and the entity's available resources. The evaluation of the entities' lifetimes is transparent to the proposed scheduling techniques; it is a responsibility of the entities themselves as it depends on their specificities and could not be part of a middleware infrastructure that is going to be used in different kinds of critical situations. It is important to note that the lifetime of the mobile entities is actually the contract between the entities and the scheduling techniques. As long as the lifetime is given as input to the scheduling techniques, it should be respected by the mobile entities (i.e. the entities should not become unavailable earlier). Such a demand may seem quite restrictive for any arbitrary ad-hoc community of mobile entities. However, the proposed approach is aimed at communities that have real-time requirements and it is natural to restrict their arbitrary behavior in order to satisfy them.

The existence of more than one alternative services also plays an important role towards the timely service provisioning in pervasive computing environments. This sort of redundancy must be considered in a systematic way. Before issuing a service request to a mobile entity that shall serve it, the different alternatives must be evaluated so as to select the mobile entity that may possibly guarantee correct service provisioning.

Considering the above, *in this paper we propose three different policies that enable the timely execution of mobile services in the context of critical pervasive computing environments*. The proposed policies are realized in the core of a middleware service that is incorporated within every mobile entity. Specifically, given a new request coming from a mobile client and a number of semantically compatible servers that can fulfill the request, one of them must be selected.

The ultimate goal of the selection process is to guarantee that a response will be sent back to the client within the client's lifetime.

The first of the proposed policies takes into account solely the lifetimes of the client and the server entities; it guarantees that a reply will be sent to the client as long as the server manages to serve the client's request. The second policy provides stronger guarantees by additionally examining the load of available servers towards selecting the one that can serve the new request within the lifetime of both the client and the server. The third policy further deals with periodic service requests. To deal with such cases we *extend classical real-time scheduling techniques to the needs of pervasive computing environments*. Each different policy provides different levels of timeliness in the execution of mobile services and requires different amount of resources. In this particular paper, we concentrate on the *number of messages* required in each policy since communication between mobile devices is amongst the key causes of wasting battery.

The remainder of this paper is structured as follows. Section 2 presents a motivating example, employed throughout this paper to demonstrate the use of the proposed service. Section 3 presents the overall architecture of the proposed middleware service. Section 4 details the three alternative policies we propose. Section 5 discusses related work and finally Section 6 concludes this paper with a summary of our contribution and future research issues.

2 Motivating Example

In this section we present a scenario where the timely provision of services is essential to confront a critical situation. This scenario serves to exemplify the use of the proposed middleware service. Specifically, we face the case of an accident in a nuclear plant. The plant consists of a number of different laboratories shown in Figure 1. The accident caused a rapid increment in the overall level of radioactivity observed in the plant area. The exact radioactivity measures range from laboratory to laboratory, depending on the physical location of each one of them (i.e., the radioactivity measures are higher in labs that are closer to the area where the accident took place). The accident took place at daytime. Hence, several employees may be trapped within the different labs. Several rescue squads enter the plant area towards dealing with this situation. Each squad is in charge of a different lab and tries to locate trapped employees. Each squad consists of firemen equipped with wireless-enabled radioactivity sensors with limited processing capabilities. Communication between different squads is feasible only through the squads leaders who are additionally equipped with small laptops serving as base stations for the networks formed in each room.

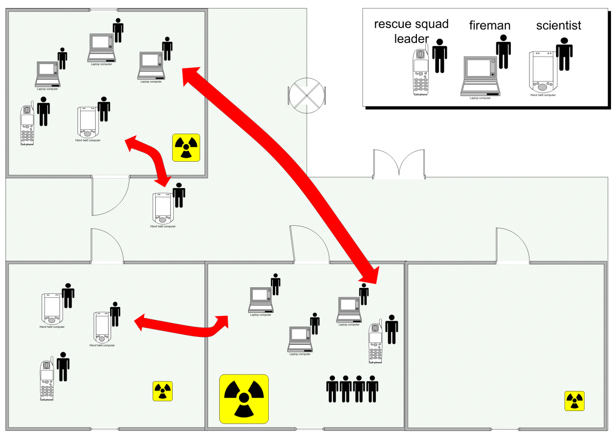


Fig. 1. Overview of the nuclear accident situation.

A group of scientists also enters the accident area. The main goal of the the scientists is to gather radioactivity measures from different labs and use them for the post-mortem analysis of the situation, which shall result in estimating the impact of the accident in the plant territory. Each scientist is assigned to a different lab and carries a PDA, used for contacting the sensor-equipped firemen. Firemen and scientists can not remain in the accident area for long. Each one of them has a strict time-to-leave and within this *deadline* he must accomplish his assigned tasks. Taking for instance the firemen who accept radioactivity measure requests by the scientists, it is important to accept these requests only if they can be served within their time-to-leave and the time-to-leave of the scientists. Failing to serve these requests on-time shall delay the accurate estimation of the impact of the accident, which is critical for the identification of the particular strategy that should be followed to rapidly deal with the accident's consequences. Depending on the availability of replacements, the leaving person may be substituted by new ones. The firemen may also move from lab to lab, depending on the current situation in each one of them. For instance, a fireman may be asked by his leader (who is constantly in contact with other leaders) to move to another lab where there exist injured employees. Locally, the leader may assign several tasks to his firemen.

In our example, the members of the rescue squads and the scientists are mobile entities. In particular, the firemen are service providers used by the scientists and the leaders of the rescue groups, which constitute the mobile clients in our critical situation.

3 Service Architecture

The overall architecture of the proposed middleware service is designed over WSAMI [4]. WSAMI is a lightweight platform developed at INRIA, which aims at facilitating the development of *ad-hoc* communities of mobile entities. WSAMI entities may execute on either stationary or mobile devices. They may provide or use a number of WEB services, conforming to the standard WEB services architecture [5]. Specifically, WSAMI services are specified using a declarative language that extends the features of the standard WEB Service Description Language (WSDL), with additional features that prescribe qualitative properties of the services such as security and transactions. Communication with the services is realized through the exchange of messages, whose format conforms with the Simple Object Access Protocol (SOAP). The WSAMI platform comprises two main subsystems used for the realization of the proposed middleware service: (1) The *CSOAP* broker, which facilitates the exchange of SOAP messages between resource constrained mobile entities; and (2) the *Naming and Discovery* (ND) service, which allows mobile entities to gather information regarding WEB services provided by other available entities.

WSAMI is a highly scalable platform since the realization of the ND service is completely distributed. Every WSAMI entity comprises an instance of the ND service, which periodically checks the environment for other instances of ND services hosted by neighboring WSAMI entities. This task is realized using the standard Service Location Protocol (SLP). The resulted information is kept locally by the service and is used afterwards for the discovery of WEB services provided by the neighboring entities.

Figure 2 gives an overview of the main components that constitute the architecture of the proposed service. A *pervasive computing environment* is a community of WSAMI entities. Each entity may play the role of a client to other mobile entities, playing the role of the server. The entity can be accessed through the use of the WEB services it provides. The mobile entity further includes a *community directory* that contains a local view of the community that corresponds to the entity. Specifically, the directory contains information regarding the WEB services that are provided by other community members, *which can be accessed by the mobile entity.* This information is divided into different *categories* depending on the different types of community members (e.g., scientists, firemen, etc.). The directory is managed by the *community manager* service. Whenever a

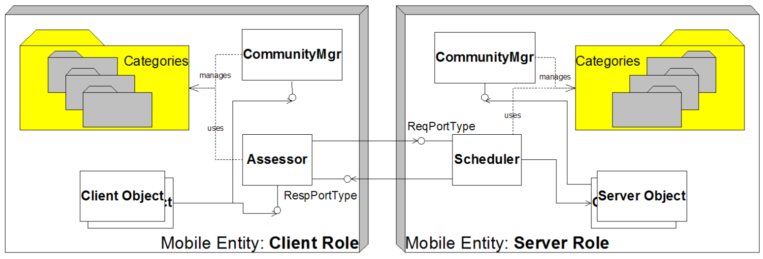


Fig. 2. Overview of the service architecture.

mobile entity joins a pervasive computing environment, the community manager populates its local directory. This takes place as follows: First, the mobile entity queries the community manager for available WEB services belonging to the particular categories that interest the querying entity; the manager forwards the entity's request to ND, which subsequently contacts all other neighboring NDs; the results are collected and stored in the community directory; following, the entity configures the community manager to *periodically refresh* the directory by following the three steps mentioned above; alternatively, the entity may also *explicitly refresh* the directory.

The community manager is rather typical and is not further detailed in this paper. On the other hand, the behavior of the rest of the components showed in Figure 2 is actually the one that facilitates the timely execution of services among community members. Briefly, each mobile entity comprises an *assessor* and a *scheduler* service. The assessor service accepts as input requests from client objects encapsulated in the mobile client. For every request, the category of community entities that can serve it is further specified. Given this information, the assessor takes charge of selecting a particular mobile entity from the local community directory. Following, the assessor forwards the client request to the scheduler service of the selected entity. In particular, the scheduler accepts input messages, which are subsequently stored in a message queue maintained by the scheduler. The selection process relies on the lifetime of the community entities, which is actually part of the WSAMI description of the mobile entities. The selection process may follow three alternative policies, providing different timeliness guarantees. The three policies are the focus of the remainder of this paper.

Regarding our motivating example, each member of the rescue squads contains an instance of the server-side architecture shown in the right part of Figure 2. On the other hand, the scientists contain an instance of the client-side architecture given in the left part of the figure.

4 Timeliness Policies for the Provision of Services

Before getting into details regarding the policies of the proposed middleware service, let us formally define our execution environment. As already discussed, a critical pervasive computing environment is a community of mobile entities, $E = \{m : MobileEntity\}$. In the most general case, a mobile entity may play both the client and the server role. Therefore, the mobile entity is a tuple $m = (D, C, S, A, lifetime)$, where D is the community directory, C is the directory manager, S is the scheduler, A is the assessor and $lifetime$ is the lifetime of the entity. Regarding the directory D we have: $D = \{ct : Category\}$ and $\forall ct \in D, ct = \{epa : EndPointAddressInfo\}$. Every element within a category contains information regarding the endpoint address of a scheduler service, provided by a neighboring mobile entity. This information comprises the URI of the service and the lifetime of the neighboring entity, $\forall epa \in ct, epa = (uri, lifetime)$.

The ultimate goal of the three policies is to guarantee that a client will receive a reply to a request made on a server entity, before leaving the community. Hereafter, we assume that the worst case communication delay for sending a SOAP message between two entities can be estimated. This estimation depends on the length of the message that is to be sent and the characteristics of the underlying network protocol. The length of the requests and responses exchanged between two entities are known to the assessor and the scheduler services. Regarding the underlying network protocol, the proposed service relies on IEEE 802.11 for wireless LANs [6]. This particular protocol provides two fundamental mechanisms for accessing the medium. The first one is called Distributed Coordination Function (DCF) and handles the retransmission of collided packets with respect to binary exponential back-off rules. The handling of packet collisions with DCF renders the estimation of the medium access delay difficult (i.e. the time required to obtain access to the medium). Consequently, the estimation of the overall delay for sending a message to a target endpoint is also complicated. The second mechanism that is provided by IEEE 802.11 is called Point Coordination Function (PCF). This mechanism guarantees collision free and time bounded packet transmissions. However, it requires the existence of a Point Coordinator (PC) that resides on an access point [1]. The PC periodically gives the right to transmit messages to each of the mobile entities that constitute a community. During this period, the mobile entities may transmit their messages or part of their messages, depending on the message length. Based on the above, the PCF mechanism is more suitable for the purpose of the proposed service.

4.1 Lifetime Policy

The first of the policies of the proposed service is based solely on the lifetimes of the mobile entities. The schedulers of the mobile servers maintain FIFO queues of requests scheduled according to the classical Round Robin (RR) algorithm.

[1] The time-bounded election or substitution of a PC is an interesting issue that is complementary to our approach and is not further discussed in this paper.

According to the Lifetime policy, every request req is issued by a client object of a mobile entity m_{client} to the assessor service of this entity, $m_{client}.A$. The assessor must select a mobile server out of a category $ct \in m_{client}.D$ that contains information about all the alternative mobile servers that may possibly serve req, and forward req to the selected server. Within ct there may exist servers whose lifetimes are greater than the lifetime of the client and servers whose lifetimes are smaller or equal to the lifetime of the client. Selecting one of the former implies that the request may be served after the end of the client's lifetime. On the other hand, selecting one of the latter implies that the request may be served earlier than the end of the client's lifetime. In both cases, a request may not be served at all, depending on the load of the server, which is not known to the assessor. Considering the above, if there exist one or more servers whose lifetimes are smaller or equal to the client's lifetime, one of them is selected randomly by the assessor. In the opposite case, the assessor selects randomly a server with a longer lifetime. More precisely, let c_{rep} be the worst case communication delay for sending a reply message to req. Let $ct', ct" \subseteq ct | ct' \bigcup ct" = ct$ be two disjoint subsets of ct defined as follows:

$$ct' = \{epa_{m_{server}} \in ct | epa_{m_{server}}.lifetime \leq m_{client}.lifetime - c_{rep}\}$$
$$ct" = ct - ct'$$

Then, for the selected server m_{server} we have:

$$\begin{cases} if\ ct' \neq \emptyset\ then\ m_{server} \in ct' \\ else \qquad\qquad\quad m_{server} \in ct" \end{cases}$$

Hence, if $m_{server} \in ct'$ the Lifetime policy guarantees that *the client will receive a reply on time, as long as the request is served.* However, there is absolutely no guarantee that the request will be served at all. The Lifetime policy is quite simple since it does not introduce any communication overhead apart from the one needed to exchange the request and the reply messages. It requires minimal information regarding the behavior of the different mobile entities of the environment.

After the selection of m_{server}, the request is forwarded towards the selected entity. Eventually, req is received by the scheduler service of m_{server} and it is placed in the request queue maintained by this service.

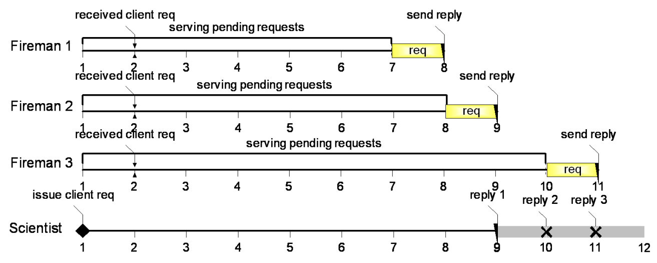


Fig. 3. Applying the Lifetime policy.

Getting to our case study example, assume that a scientist enters a lab in the plant area and wants to request from a fireman the current value of radioactivity. In the same lab there exist 3 possible firemen and the scientist is supposed to select one of them and issue his request. Suppose that the lifetimes of the scientist and the firemen are respectively 9, 8, 9 and 11 time units (Figure 3)[2]. The communication overhead is at most 1 time unit. According to the Lifetime policy, the scientist selects Fireman 1. If the fireman manages to serve the request, in the worst case this will happen at time = 8. Consequently, the scientist will receive a reply at time = 9. On the other hand, if the scientist selects the second fireman, in the worst case his request will be completed at time = 9 and the reply will arrive at time = 10, which is too late for the scientist. Similarly, if the scientist selects the third fireman, in the worst case his request will finish at time = 9 and the reply will arrive at time = 11, which is also too late.

4.2 LifetimeLoad Policy

The second policy of the proposed service provides stronger timeliness guarantees. Still, the scheduler services of the mobile servers manage FIFO queues of requests, which are scheduled according to the classical Round Robin (RR) algorithm. However, with this policy we examine both the lifetimes and the current load of the mobile entities that may serve a client request.

As with the Lifetime policy, a request *req* is issued by a client object of a mobile entity m_{client} to the assessor service $m_{client}.A$. The assessor forwards directly *req* to the scheduler service of an accessible mobile entity, m_{server}, which is randomly selected. The scheduler service eventually receives *req* and examines the feasibility of its execution based on the load and the lifetime of m_{server}.

Specifically, let $N_{m_{server}}$ be the total number of pending requests for m_{server}, including *req*. C_{req_i} denotes the time units required for serving each pending request req_i, issued by m_{client_i}. Moreover, c_{rep_i} denotes the worst-case communication delay required to send a response back to m_{client_i}. Given that the scheduler queue is FIFO, *req* is the $N_{m_{server}}$-th pending request. Every new request added to the scheduler queue introduces an additional overhead in the execution of all the other pending requests. This is due to the fact that requests are scheduled in a RR fashion. Hence, *before adding a new request* in the scheduler queue we have to verify that this additional overhead shall not delay the rest of the pending requests too long, making it impossible to send the corresponding replies back to the clients that issued the requests. Moreover, we have to verify that the new request will be served within the server's lifetime and a reply will be send back to the client within the client's lifetime. To achieve the previous, the scheduler performs the following:

1. For every request $req_i, i = 1, \ldots, N_{m_{server}}$ (including the new request, $req_{N_{m_{server}}}$), the scheduler calculates the overall time D_{req_i} required for serving it.

[2] Note that each one of the Figures 3, 4, 5 examines three different scenarios that correspond to the selection of Fireman 1, 2 and 3 respectively

2. Then, to accept the new request the scheduler must verify that the following constrain holds:
$\forall req_i, i = 1, \ldots, N_{m_{server}} | D_{req_i} \leq m_{client_i}.lifetime - c_{rep_i}$

Upon the verification of the above constraint the scheduler service reports back to the client assessor. If the constraint holds the report is positive and the scheduler continues serving pending requests including req. On the opposite case, the report is negative and the scheduler forgets req. Eventually, the assessor of m_{client} receives the report from m_{server}. In case the report is negative, another mobile entity is selected and the same procedure is followed. If the reports of all the available mobile entities are negative, req is dropped by m_{client}.

Calculating D_{req_i} is realized as follows. Given that req_i is in the i-th position of the FIFO queue and that serving it relies on RR, it takes i time units for req_i to be placed at the end of the queue. Let $Q = \{A_{req_k} | k = 1, \ldots, N'_{m_{server}}\}$ be the remaining time units required for the execution of each one of the pending requests at the time when req_i will be placed at the end of the queue. Note that the cardinality of Q may be less than $N_{m_{server}}$ given that there may be requests from the original queue that will be completed at the time when req_i will be placed at the end of the queue. For every A_{req_k}, we have that $A_{req_k} \leq C_{req_k}$. Let $Q' = [B_i | i = 0, \ldots M]$ be a sequence whose $B_0 = 0$. The remaining elements of Q' result from Q by removing the duplicate values existing in Q and sorting the remaining values in increasing order. Specifically, for Q' the following hold:

(1) $B_0 = 0$
(2) $\forall B_i, B_j \in Q' | i < j \Rightarrow B_i < B_j$
(3) $\forall B_i > B_0$ there exists at least one $A_{req_k} \in Q | A_{req_k} = B_i$
(4) $\forall A_{req_k} \in Q | (\exists B_i \in Q' | B_i = A_{req_k})$

For all $B_i \in Q'$ we define their multiplicity $mult_i$ as follows:

$$\begin{cases} mult_i = |Q_{B_i}|, & i > 0 \\ mult_0 = 0, & i = 0 \end{cases}$$

where $Q_{B_i} \subseteq Q \wedge (\forall A_{req_k} \in Q_{B_i} | A_{req_k} = B_i)$.

Let $B_l \in Q'$ be the specific value that corresponds to the time units required for completing req_i, i.e., $B_l = A_{req_i}$, then the first B_1 units of B_l will actually execute in: $|Q| * B_1$ time units. After $|Q| * B_1$ time units, all the requests that required B_1 units to complete will be removed from the queue. Hence, the length of the scheduler's queue will become $|Q| - mult_1$. Moreover, all the requests that required B_2 time units to complete will now require $B_2 - B_1$ units. Consequently, the next $B_2 - B_1$ units of B_l will actually execute in $(|Q| - mult_1) * (B_2 - B_1)$. By induction, we can conclude that the overall service time D_{req_i} can be calculated using the following formula:

$$D_{req_i} = i + \textstyle\sum_{k=1,\ldots,l}((|Q| - \sum_{m=0,\ldots,l-1} mult_m) * (B_k - B_{k-1}))$$

In Figure 4, we revisit our case study scenario. Assume the situation discussed in Section 4.1 where the scientist wants to select out of three firemen the one

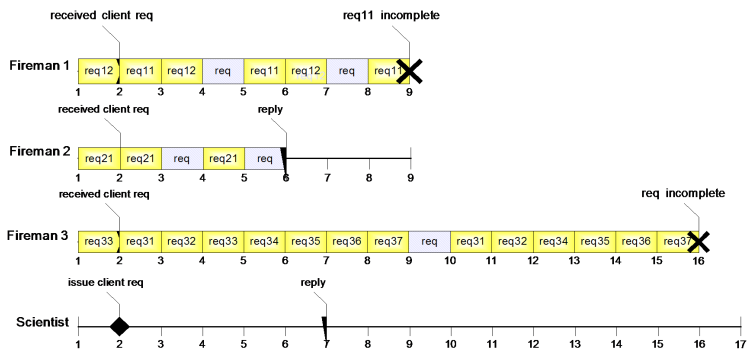


Fig. 4. Applying the LifetimeLoad policy.

that can fulfill his request, *req*, within the scientist's lifetime. This time the lifetimes of the three firemen are 9, 9 and 16 time units, respectively. The worst case execution time for *req* is 2 and the worst-case communication delay is 1. Suppose that the client assessor chooses Fireman 1 first. The scheduler of the fireman has two pending requests in his queue. At the time when *req* arrives, req_{11} requires 4 more time units to complete, while req_{12} requires 2. With the addition of *req* in the queue, the overall delay for completing it shall be 6. Given that *req* arrives at time $= 2$, its execution will complete at time $= 8$. Consequently, the client will receive a reply at 9, which is legal. However, with the addition of *req*, the overall delay for completing req_{11} shall be 8 time units. Hence, req_{11} will finish at time 10, which is greater than the lifetime of the first fireman. Accepting, thus, *req* causes a missing deadline for the first fireman. Let us assume instead that the client assessor selects Fireman 2 first. The queue of his scheduler contains only one pending request that requires 2 more time units to complete at the arrival of *req*. The overall delays for completing *req* and req_{21} are 4 and 3, respectively. These values are legal with respect to the lifetimes of the client and Fireman 2. Consequently, *req* can be accepted.

4.3 EDFTB Policy

As we already discussed the LifetimeLoad policy provides quite strong guarantees. However, it can only provide them in cases of requests that are served once during the lifetime of a mobile server. In practice it is often the case that a mobile client requests a mobile server to perform a particular task more than once, usually with a certain period. In our example, for instance, the scientists may request the firemen to measure the level of radioactivity periodically and produce a certain amount of measures, which should be returned back to them towards performing more accurate analysis and estimations. Such kind of requests, leading to the execution of periodic activities, can not be guarantied by the LifetimeLoad policy.

The EDFTB policy detailed here is suitable for both requests leading to periodic activities and typical requests that are served once. Hereafter, we call the former *periodic requests* and the latter *aperiodic requests*. As implied by the name of the policy, it relies on the classical EDF (Earliest Deadline First) [1] and the TB [7](Total Bandwidth) algorithms, which are customized here for the specific purpose of critical pervasive computing environments.

Briefly, Liu and Layland [1] examine a typical real-time system that executes only periodic tasks, which arrive dynamically in a queue. The execution of the tasks is preemptive. Every task t_i is characterized by a period T_i. Every instance of t_i must complete within T_i. Hence, T_i is the deadline for the completion of t_i. Moreover, t_i is characterized by a worst case execution time C_{t_i}. According to EDF, a task is scheduled if it is the one with the earliest deadline amongst all the periodic tasks in the queue. Liu and Layland proved that a particular set of tasks $\{t_1, t_2, \ldots t_N\}$ is scheduleable if and only if the system's utilization is at most 1. Formally:

$$U_P = \sum_{i=1,\ldots N}(C_i/T_i) \leq 1$$

In the TB algorithm, Spuri *et al.* [7] further consider a real-time system with both periodic and aperiodic tasks. To deal with this combination they propose dividing the system utilization into U_P, used for executing periodic tasks, and U_S, used for the execution of aperiodic tasks. Aperiodic tasks t_{a_i} are given a deadline d_{ta_i}, on the basis of U_S. The given deadline is the shortest possible and does not jeopardize the execution of periodic tasks. Specifically:

$$d_{ta_i} = max(r_k, d_{ta_{i-1}}) + C_{ta_i}/U_S$$

In the above formula, r_k denotes the time instant that ta_i arrived and $d_{ta_{i-1}}$ denotes the deadline given to the aperiodic task whose arrival immediately preceded the arrival of ta_i. Based on its given deadline, ta_i is scheduled by EDF as any periodic task. Given the previous, an overall set of periodic and aperiodic tasks is scheduleable if and only if $U_P + U_S \leq 1$.

In the rest of this section, we describe in detail our specific extension to the EDF and the TB algorithms to handle the case of critical pervasive computing environments.

With the EDFTB policy, the scheduler of mobile entities uses the EDF algorithm. However, the scheduleability of periodic and aperiodic requests further involves additional constraints, which are verified by the scheduler service upon the arrival of a new request req coming from the assessor of a mobile client. In particular, if req is periodic it will result in the execution of a periodic activity on the side of the mobile server. Since the lifetimes of both the client and the server are limited, the client is obliged to associate req with a period T_{req} and a required number of instances n_{req} of the periodic activity that is going to be executed. For example, a scientist should request a fireman to measure the level of radioactivity 3 times with a period of 2 time units. The request req is eventually received by the scheduler of m_{server}, which further assumes that the overall server utilization is divided into U_P and U_S for the execution of periodic and aperiodic requests, respectively. Given the previous, the goal of the scheduler is to verify whether the server can preserve the following constraints:

- For periodic requests:
  1. *req* will be served n_{req} times within the lifetime of the server.
  2. The replies to *req* will be send back to the client within the client's lifetime.
- For aperiodic requests:
  1. *req* will be served within the lifetime of the server.
  2. The reply to *req* will be send back to the client within the client's lifetime.

Based on the outcome of the verification procedure the scheduler sends a positive or a negative report to the client assessor. Depending on the report, the assessor behaves as in the case of the LifetimeLoad policy.

Periodic requests: In case *req* is periodic, the scheduler of m_{server} performs the following steps:

First it checks whether the utilization of m_{server} remains lower than 1 if the new request is accepted for service. Formally, if there exist $N_{m_{server}}$ pending periodic requests on the server, the scheduler verifies the following:

$$U_P = \sum_{i=1,\ldots N_{m_{server}}} (C_{req_i}/T_{req_i}) + (C_{req}/T_{req}) \leq 1 - U_S$$

In the above formula, C_{req_i} and T_{req_i} denote the worst case execution time and the period of each pending request. If this formula holds it means that the execution of *req* shall not jeopardize the execution of the rest of periodic requests that already exist in the scheduler's queue. However, the scheduler must further verify that the server's lifetime is sufficient to allow executing *req* n_{req} times. This is accomplished in the second step by evaluating the following:

$$m_{server}.lifetime/T_{req} \geq n_{req}$$

Hence, in the first two steps the scheduler of m_{server} verifies whether the server can guarantee the first of the two constraints stated for periodic requests. The third step performed by the scheduler amounts in checking the second constraint. Given that the execution of the activities that serve periodic requests is preemptive, the only way to assure that the client will get the replies to *req* within $m_{client}.lifetime$ is by checking whether m_{client} lives longer than m_{server}. More precisely, if c_{rep} denotes the worst case communication delay for sending a reply to *req*, then the following must hold:

$$m_{server}.lifetime \leq m_{client}.lifetime - c_{rep}$$

If all the above hold, the server reports back to the client with a positive answer. In the opposite case, the answer is negative and the client selects another candidate mobile entity. If the answers of all the alternative entities are negative the request is dropped by the client.

In our case study scenario assume the following situation, depicted in Figure 5. At time 2 a scientist wants to issue a periodic request *req* with $C_{req} = 1$ and $T_{req} = 4$ to one of the three firemen shown in the figure. The scientist further requires that *req* is executed 3 times. Suppose that Fireman 1 is selected first by the scientist. Fireman 1 periodically executes two requests req_{11} and req_{12} with $C_{req_{11}} = 2$, $T_{req_{11}} = 4$ and $C_{req_{12}} = 2$, $T_{req_{12}} = 4$, respectively. With the inclusion of *req* in the queue of Fireman 1, the U_P utilization shall become 1.5. Hence, Fireman 1 can not execute *req* without jeopardizing the execution

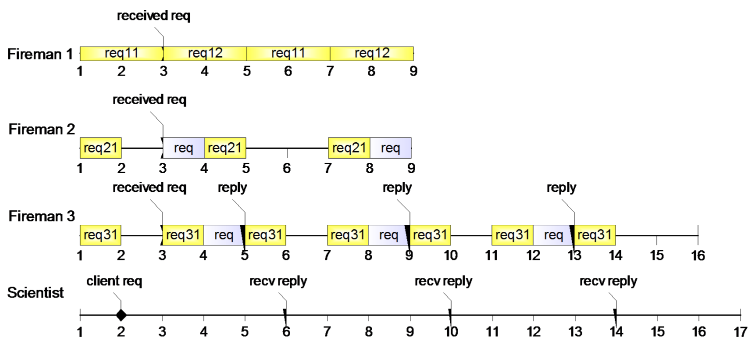


Fig. 5. Applying the EDFTB policy for periodic requests.

of req_{11} and req_{12}. Assume instead that Fireman 2 is selected first by the scientist. Fireman 2 is already responsible for the execution of one periodic request, req_{21}, with $C_{req_{21}} = 1$ and $T_{req_{21}} = 3$. Hence, with the inclusion of req the U_P utilization for Fireman 2 shall become 0.59. However, the remaining lifetime of the fireman at the time when he receives req is 6. Consequently, he can execute req at most 2 times. Fireman 3 in our example periodically executes one request with $C_{req_{31}} = 1$ and $T_{req_{31}} = 2$. With req, its utilization U_P, shall become 0.75. Moreover, the remaining lifetime of the fireman when he receives req is 13. Thus he can execute req 3 times. Finally, the lifetime of Fireman 3 is less than the lifetime of the scientist. If the communication overhead c_{rep} is at most 1, the fireman will deliver the replies to the scientist, within the scientist's lifetime. Summarizing, if the third fireman is the first entity contacted by the assessor component of the scientist, the report will be positive and req will be successfully executed.

Aperiodic requests: If req is aperiodic, the scheduler follows the TB approach [7]. Based on U_S, the scheduler assigns a deadline to req and checks whether this deadline is consistent with respect to the lifetimes of the client and the server. Formally, the deadline is given according to the following formula:

$$d_{req} = max(r, d_{req'}) + C_{req}/U_S$$

In the above, r is the moment when req arrived and $d_{req'}$ is the deadline given to the aperiodic request req' whose arrival immediately preceded the arrival of req. Moreover, the following must hold to guarantee that with the given deadline a reply to req will be delivered back to the client, within the client's lifetime:

$$d_{req} \leq min(m_{server}.lifetime, (m_{client}.lifetime - c_{req}))$$

In Figure 6 we revisit the situation discussed in Figure 5. In particular, the periodic request previously issued by the scientist is scheduled in Fireman 3 (req_{32} in the figure). Suppose now that the scientist further issues an aperiodic request req to the same fireman. For Fireman 3 we have $U_P = 0.75$ and $U_S = 0.25$. The request arrives at time 5. Since it is the first aperiodic request,

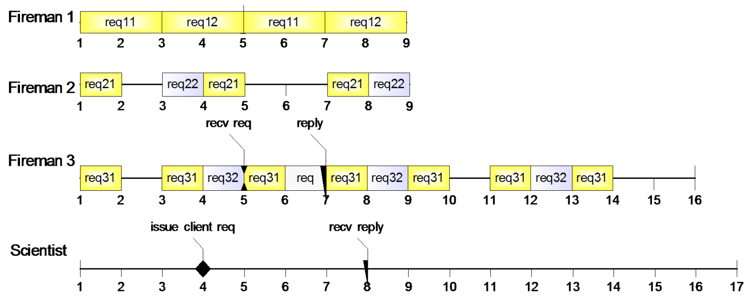


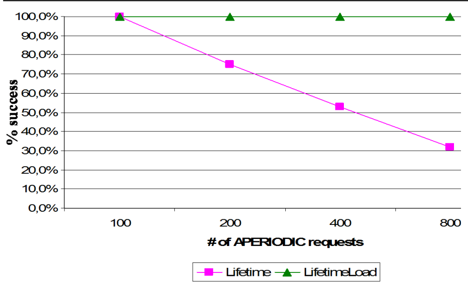
Fig. 6. Applying the EDFTB policy for aperiodic requests.

it is given a deadline that is equal to 9 (i.e., $5 + 1/0.25$). If the worst case communication overhead is 1, the scientist will receive a reply at time 10, at the latest. Consequently, *req* can be scheduled on Fireman 3 and the report sent to the scientist is positive. Actually, we can observe in the figure that *req* can be served earlier than 9 and the scientist may get the reply at time = 8.

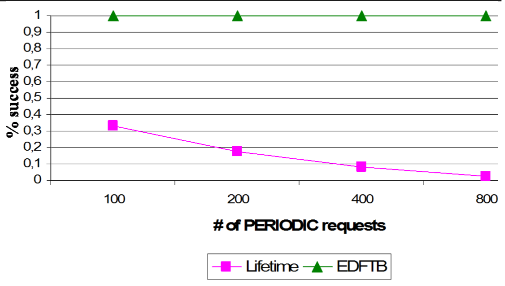
4.4 Assessment

In Figure 7 we summarize our first experimental results produced by the application of the proposed policies in a simulated environment realized using the PARASOL simulator [8]. The environment consists of 3 mobile server entities, providing compatible services. We performed 2 different classes of experiments. In the first one (Figure 7(a)), we compare the performance of the Lifetime and the LifetimeLoad policies with 4 different workloads of *aperiodic* requests (100, 200, 400 and 800 requests, respectively). Similarly, in the second class of experiments (Figure 7(b)) we compare the performance of the Lifetime and the EDFTB policies with 4 different workloads of *periodic* requests (100, 200, 400 and 800 requests, respectively). The lifetimes of the mobile entities, the worst-case execution times for the requests and the periods of the periodic requests are randomly generated by following a uniform distribution. Our performance metric is *the percentage of accepted requests (i.e., the requests that are actually stored in the queue of a scheduler) that complete on time*, with respect to the lifetimes of the clients and the mobile servers.

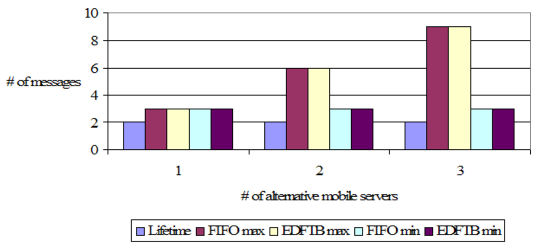
Specifically, in Figure 7(a) we observe that the percentage of aperiodic requests that complete on time in the case of the Lifetime policy, linearly decreases as we increase the number of requests that constitute the overall workload issued to the mobile servers. As opposed to that, the LifetimeLoad policy guarantees that all the accepted requests are executed on time (i.e., we do not have any missed deadlines). In Figure 7(b) we observe that the behavior of the Lifetime policy gets even worst, given that the requests are periodic and require more time to complete. The EDFTB policy behaves perfectly in this case as we observe that all the accepted requests finish on time. The price to pay for avoiding missed deadlines is given in Figure 7(c). As mentioned in Section 1, we can estimate the battery overhead introduced by the three policies by examining the number of messages exchanged between a mobile client and a mobile server. In the figure we can observe that the overall number of messages exchanged between a mobile client and the alternative mobile servers is constant in the Lifetime policy, which is the cheapest. For the other two policies the minimum number of messages exchanged is greater but is still constant. On the other hand, the maximum number of messages for the LifetimeLoad and the EDFTB policies linearly increases with the number of alternative mobile servers.



(a) Lifetime vs. LifetimeLoad



(b)Lifetime vs. EDFTB



(c) Number of messages exchanged with respect to the timeliness properties.

Fig. 7. Experimental results.

5 Related Work

Up to know, there have been several approaches dealing with various dependability attributes in the context of pervasive computing environments [9, 10]. These approaches most frequently concentrate on reliability, performance, availability, security, reputation, etc. However, to our knowledge the approach proposed in this paper is the first attempt that focuses on timeliness.

In the past, the middleware community proposed standards for real-time middleware platforms, which were aimed at conventional distributing computing environments where both the client and the server entities are deployed on top of stationary workstations. Among these standard approaches we have the one proposed by the OMG for Real-Time CORBA and related implementations like TAO and ZEN [11, 12], which proved to be useful in conventional systems. However, they can not be directly employed in the environments examined in this paper.

In the remainder of this section we concentrate on work that is complementary to the approach we propose. In particular, the issue of calculating the lifetime of mobile entities is central to our approach. Currently, we have useful techniques that aim at estimating the physical motion of mobile entities [3] and motion independent techniques that estimate the unavailability of mobile entities [13]. Several classical algorithms for scheduling real-time tasks have been proposed in the past. Usually their are divided into static and dynamic. Obviously, static algorithms can not be used in critical pervasive computing environments. Dynamic scheduling algorithms, schedule tasks on-the-fly. The EDF algorithm that we employed in this paper is among the most widely known ones. However, several others like MLF (minimum-laxity-first) and MUF (maximum-urgency-first) [14, 15] may prove useful in the context of critical pervasive computing environments. As in the case of EDF, these algorithms should also be appropriately enhanced before they are introduced in such environments. In our particular case, we consider using MUF instead of EDF to deal with cases where all of the alternative mobile servers are incapable of serving a new client request. MUF associates tasks with an importance factor, which may serve as a criterion for rejecting pending requests towards serving new ones of greater importance.

6 Conclusion

In this paper, we proposed a middleware service that facilitates the timely execution of mobile services in critical pervasive computing environments. The overall service architecture relies on the WSAMI platform and supports three different timeliness policies for the execution of requests issued by mobile clients to mobile servers. The first policy takes into account the lifetimes of the client and the server entities and guarantees that a reply will be sent to the client as long as the server manages to serve the client's request. The second policy guarantees both that the client request will be served and that a reply will be sent back within the client's lifetime. This is achieved by examining the servers' load along with the client and the servers lifetimes. The third policy extends the classical EDF and TB algorithms for the purpose of pervasive computing environments, to deal with periodic requests in such environments. So far, the proposed service deals with the timely execution of independent client requests. An interesting extension would be to further incorporate support for the timely execution of workflows [16].

Acknowledgments. This work is partially funded by the MobWS GSRT grant for Cooperation in S&T areas with European countries.

References

1. C. L. Liu and J. W. Layland: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM **20** (1973) 46–61
2. E. Pitoura and G. Samaras: Data Management for Mobile Computing. Kluwer Academic Publishers (1998)
3. H. M. O. Mokhtar and J. Su: Universal Trajectory Queries for Moving Object Databases. In: Proceedings of the IEEE International Conference on Mobile Data Management (MDM'04). (2004) 133–146
4. V. Issarny, D. Sacchetti, F. Tartanoglou, F. Sailhan, R. Chibout, N. Levy and A. Talamona: Developing Ambient Intelligence Systems:A Solution Based on Web Services. Journal of Automated Software Engineering **12** (2005) 101–137
5. W3C: Web Services Architecture. Technical report, (W3C) http://www.w3.org/TR/ws-arch/.
6. IEEE: IEEE Standard for Wireless LAN Medium Access Control (MAC). Technical report, IEEE (1997)
7. M. Spuri, G. Buttazzo and F. Sensini: Robust Aperiodic Scheduling under Dynamic Priority Systems. In: Proceedings of the 16th IEEE Real Time Systems Symposium (RTSS'95). (1995) 210–221
8. J. Neilson: PARASOL Users' Manual (v 3.1.). Technical report, (School of Computer Science - Carleton University - Ottawa) K1S5B6.
9. J. Liu and V. Issarny: QoS-Aware Service Location in Mobile Ad-Hoc Networks. In: Proceedings of the 5th IEEE International Conference on Mobile Data Management (MDM'04). (2003)
10. L. Zeng, B. Benatallah and M. Dumas: Quality Driven Web Services Composition. In: Proceedings of the 12th ACM International Conference on the World Wide Web (WWW'03). (2003) 411–421
11. R. E. Schantz, J. P. Loyall, D. C. Schmidt, C. Rodrigues, Y. Krishnamurthy, and I. Pyarali: Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware. In: Proceedings of the 4th IFIP/ACM/USENIX International Conference on Distributed Systems Platforms (Middleware'03). (2003)
12. A. Krishna, D. C. Schmidt, and R. Klefstad: Enhancing Real-Time CORBA via Real-Time Java. In: Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04). (2004)
13. Y. Xiong, X. Lin and J. Rowson: Estimating Device Availability in Pervasive Peer-to-Peer Environment. In: Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'04). (2004)
14. M. L. Dertouzos and A. K. L. Mok: Multiprocessor On-Line Scheduling of Hard Real-Time Tasks. IEEE Transactions on Software Engineering **15** (1989) 1497–1506
15. D. B. Stewart and P. K. Khosla: Real-Time Scheduling of Sensor-Based Control Systems. In: Proceedings of the 8th IEEE International Workshop on Real-Time Operating Systems and Software (RTOSS'91). (1991)
16. A. Zarras, P. Vassiliadis and V. Issarny: Model-Driven Dependability Analysis of Web Services. In: Proceedings of the 6th International Conference on Distributed Objects and Applications (DOA'04). (2004) 1608–1625