A scheduling framework for enterprise services

Vassileios Tsetsos, Odysseas Sekkas, Ioannis Priggouris, Stathes Hadjiefthymiades *

Department of Informatics and Telecommunications, University of Athens, TYPA Bldg., Panepistimiopolis, Ilissia, GR-15784 Athens, Greece

Received 25 January 2005; received in revised form 28 April 2005; accepted 22 May 2005
Available online 11 July 2005

Abstract

Application-level scheduling is a common process in the enterprise domain, but in order to be productive it should be time-effective, interoperable and reliable. In this paper we present the design and implementation aspects of a modular scheduling solution, which aims to fulfill the above-mentioned requirements. The proposed solution constitutes an integral part of an existing service provisioning platform targeted to the location based services (LBS) domain; however application to other service-oriented architectures is also possible. A careful design process is followed in order to identify the scheduler’s functional subsystems and the relationships between them. The implementation is based on the J2EE framework, thus guaranteeing independence from underlying technologies and applicability in enterprise environments. The high quality and scalability of the scheduler is also validated through an intensive performance analysis.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Time-critical scheduling; Enterprise applications; J2EE; Push paradigm; Middleware platform

1. Introduction

Effective task scheduling is a “hot” topic in the computing community since its early years. Effectiveness, in general, is a vague notion unless we relate it to the different types of scheduling. Roughly speaking, scheduling can be divided in two broad categories: low level scheduling and application (or service) level enterprise scheduling. The term “enterprise” intends to point out the large-scale nature of this category. The first category consists of scheduling techniques and systems in multitasking Operating Systems or network elements such as QoS-aware routers. These systems should be extremely fast and support priority-based scheduling. On the other hand, the second category includes schedulers that should execute tasks either in real-time or some time in the future, periodically or not. Such schedulers may be responsible for the execution of batch jobs or for time-specific service provisioning. Moreover, tasks may be scheduled upon the triggering of events (e.g., in case John Doe logs in the main server, the system informs the administrator by e-mail). They should be reliable and highly scalable in order to be usable in the enterprise domain.

In this paper the architecture of an enterprise scheduling service is presented. This service, referred as Scheduler from now on, was developed as an integrated component of a service provisioning platform used in the mobile networking context. Its role is to support the time/event-triggered service delivery. Although the middleware and service provision platform was focused on mobile-oriented services (e.g., location-based and context-aware services) the design of the Scheduler was based on a much more general-purpose requirements analysis and can support a wide range of real-time applications. A prototype implementation of the Scheduler...

260

The rest of the paper is structured as follows. In Section 2 we provide a brief summary of work related to service scheduling. Section 3 presents the architecture and implementation of the Scheduler, starting from the requirements specification that drove the design phase and concluding with an in-depth analysis of the architecture and key implementation decisions. Section 4 summarizes the results of the various tests carried out in order to assess the performance and stability of the prototype. In Section 5 we describe in brief PoLoS, a middleware platform for location-based services, which hosted the Scheduler service in its architecture. In Section 6, we discuss some scalability and clustering issues. Finally, future work envisaged is presented followed by some conclusions.

2. Requirements analysis and related work

An enterprise scheduler, as part of a broader enterprise solution, has to meet a lot of critical requirements. These requirements can be classified as functional (i.e., company and end-user requirements) and technical. They also vary according to the type of services the scheduler is intended to provide. Existing solutions were evaluated in order to identify the desired characteristics, which apply to all general purpose real-time enterprise schedulers (e.g., large numbers of clients, high quality of offered services, etc.).

The most important requirement is the accuracy of the scheduling engine. A highly accurate engine guarantees that the scheduled tasks will commence execution on time or, at least, with the lowest possible delay. We should also note that, end-users are typically interested in low turnaround times (i.e., from task registration till task completion), since only at the end of task executions they obtain the requested results. From the scheduler perspective, however, we are interested only in starting delays, since turnaround times are affected by other factors beyond scheduler control (e.g., network delays). This delay should, in any case, be very short (e.g., in the order of milliseconds) so as not to decrease the reliability of the scheduling service. Furthermore, the users should be able to schedule both periodic and aperiodic tasks. In the first case, the capability of determining an ending time would be required. At this point, it should be noted that the ways a task can be scheduled vary from very simple to very expressive and flexible. The final decision should be taken according to the requirements of the targeted application.

The performance of the scheduler should be high, even when hundreds of tasks have to be executed concurrently. This requirement stems directly from the large-scale nature of the enterprise applications and services. In general, scalability is one of the most important features that a well-designed scheduler should provide.

Another important feature of an enterprise scheduler is persistence. Scheduled jobs need to be saved in a persistent store (e.g., a relational, object or XML database). This will enable them to automatically be rescheduled after a system crash, thus catering for a smooth failover. Stored information could be further exploited by the enterprise, for management purposes or for issuing statistical reports.

Logging is an efficient and flexible mechanism for monitoring scheduler’s activities. As the scheduler triggers the execution of enterprise jobs, it has a crucial role in the provisioning process which makes it suitable for the maintenance of logs. These logs can be used for administrative, charging and billing purposes. Special provision should be taken, though, so that the logging activity does not degrade the scheduler’s performance.

Apart from the general requirements, there are technical requirements that need to be satisfied. Portability across different operating systems, application servers and databases is the most important of them, as it ensures independence from specific platforms and potential use by a large market share.

Other requirements, such as stability and support for clustering and load-balancing, are not considered, because most application servers implement such functionalities. In Section 6, we elaborate on the clustering and load-balancing issues. A user-friendly and fully functional graphical user interface for administration purposes is also desirable, but it is more considered as an external utility rather than part of the scheduler’s core architecture. Another potential requirement is the capability to assign priorities to tasks. CPU and other types of schedulers employ priority queues to separate the mission-critical tasks from the less urgent ones. However, this separation is not very important for a real-time scheduler where the only differentiation between its tasks is their time of execution.

The first popular job scheduler was cron, a UNIX utility. Cron executes jobs\(^1\) (mainly shell scripts) scheduled on an hourly, daily, weekly or monthly basis and is still an integral part of any Unix/Linux platform. However, cron is not targeted to the enterprise domain, as its main role is to assist in administering UNIX systems. On the other hand, many companies have produced and provide commercial enterprise schedulers. Below we present briefly the most important of them, highlight their core features and point out the common functionality they have.

\(^1\) Some developers define “job” as a set of “tasks”. In this paper these terms have the same meaning and are used interchangeably.
**Flux** (Sims, 2002) is a commercial job scheduler that can be used either as a standalone program or as a software component. A user can schedule time-driven, event-driven and file-driven tasks (triggered by the creation or update of specific files). **Flux** features a rich and user-friendly graphical user interface (GUI) for creating, editing and monitoring jobs. It also enables the creation of very expressive and flexible schedules, which can be executed even in different time zones. Finally, it includes clustering and failover capabilities. The latter is performed with the assistance of multiple deployed scheduler instances.

**Kronova eScheduler** (Dale, 2003) is a scheduler addressed solely to the enterprise domain. It is used mainly for scheduling batch jobs, such as weekly backups and monthly reports. eScheduler supports many types of tasks: operating system commands, Enterprise JavaBeans (EJBs), local and remote Java objects and e-mail messages. The main component of its scheduling engine is a session EJB and the scheduled jobs are implemented as entity EJBs.

To the best of our knowledge, the only open source J2EE (Java 2 Enterprise Edition) (Cattell et al., 2000) scheduler is Pulsar (Bhattacharya, 2003). The core of Pulsar’s engine is based on the standard Java Timer class (Joy et al., 2000). The scheduler runs as a servlet and the tasks are either stateless sessions EJBs or Java classes, which can be scheduled using an XML (Extensible Markup Language) file containing all the required parameters. Currently, there is support only for one application server and, in general, this scheduler lacks the advanced features of the aforementioned commercial products.

The above-mentioned solutions have much functionality in common; indicative examples are persistence and logging support. Furthermore, most of them come with a graphical management tool, while all of them are based on EJBs in order to take advantage of the facilities provided by application server containers (see Section 3).

Concluding this section we should point out that although task (job) scheduling constitutes a central process in many enterprise systems, there is limited published work on the design and performance evaluation of such software. The present paper tries to describe in detail such a design and evaluation process in order to supplement the related bibliography and act as a guide for future similar implementations.

### 3. Architecture and implementation

#### 3.1. Technology overview

Trying to meet the requirements identified in the previous sections, the Scheduler capitalized on state-of-the-art enterprise technologies and standards, like the J2EE 1.3 application framework (Cattell et al., 2000). Open source tools and software were used for the implementation of the Scheduler’s prototype and extra features (e.g., logging support).

The main advantage of the J2EE application framework is the built-in EJB 2.0 support. EJBs are distributed components running within a container of the application server. They allow the separation of application logic from system-level services thus allowing the developer to concentrate on the business domain issues and not on system programming. In addition, the EJB container takes care for advanced issues such as instance pooling, transactions, security and caching. Three EJB types are currently defined in the J2EE specification: session, entity and message-driven beans (MDBs). Session beans, either stateless or stateful, represent the enterprise processes and implement the application logic. Entity beans perform the object-oriented mapping to an underlying datastore table. There are two entity bean types: those with Container Managed Persistence (CMP) and those with Bean Managed Persistence (BMP). CMP entity beans are more flexible as they are mapped to database entries through an abstraction layer that makes them portable across different databases. Database transactions are also handled transparently by the container. Message-driven beans are special EJB components that can receive JMS (Java Message Service) messages and consume these messages from queues or topics where they are placed by any valid JMS client (Cattell et al., 2000). A message-driven bean is decoupled from the clients that address messages to it. It is the server’s responsibility to provide concurrent message consumption by pooling multiple message-driven bean instances (Roman et al., 2002). These beans are ideal for implementing asynchronous communication and binding with other message-oriented applications.

#### 3.2. Architecture overview

Based on the requirements identified in Section 2, the following core subsystems were identified for the scheduling service:

- **the Scheduling Subsystem**, which includes the client-side scheduler interface and the main registration mechanisms;
- **the Task Execution Subsystem**, which is responsible for the accurate execution of the scheduled tasks;
- **the Persistence Subsystem**, which performs the persistence of the scheduled tasks, thus providing the desired reliability.

The core element in the architecture is the **Scheduler-Bean**, which is responsible for the initialization of the
scheduler and the scheduling/removal of tasks. This component can be used for scheduling two different types of tasks:

1. Time-triggered tasks, which can be either periodic or aperiodic (Franklin and Zdonik, 1998). These tasks have certain time constraints, which are defined upon their registration and should be respected until their expiration. Such tasks are registered on the Timer object, which is responsible for their timely execution.

2. Event-triggered tasks. These are tasks requiring immediate execution, without any time constraints. Although not strictly bound to the scheduling paradigm such tasks are supported by the scheduling service in order to provide a unified service execution interface.

The SchedulerBean exports a well-defined interface towards client applications that need to perform operations on the Scheduler. The SchedulerBean uses the TaskBean component in order to persist tasks in a relational database. The primary key of each stored task is a unique identifier generated by the SequenceBean. Event-triggered tasks are not stored in the database, as their execution is not likely to be repeated in the future; instead they are directly forwarded for execution to the TaskHandler.

When a new task needs to be executed the SchedulerBean forwards it to the Task Execution Subsystem. Communication between the two subsystems is implemented asynchronously using the message queuing model. The SchedulerBean places the new task in the Queue from where it is instantly consumed by the TaskHandler. Of course the time consumption can potentially vary depending on the load experienced by the Queue. Subsequently the TaskHandler acts as a simple client for the service/task to be invoked by commencing its execution.

The scheduler architecture along with the relationships between the different subsystems and their components is depicted in Fig. 1. Further elaboration on each subsystem and implementation details are provided below.

### 3.3. Detailed design and implementation

The components comprising the different subsystems of the Scheduler are implemented as EJBs, thus allowing their operation in distributed environments. All subsystems were designed and implemented so that they can be deployed inside any J2EE-compliant application server, either on the same machine or on remotely connected hosts. The former approach is more efficient in terms of delay and time-response (due to the local interfaces defined in EJB 2.0 specification), while the latter provides all the advantages of a distributed and, thus, highly scalable architecture. The implementation details for the Scheduler are presented in the following sections. Implementation took place on the Jboss (Stark et al., 2002) application server, which was also used for the testing phase. The core scheduling engine was implemented using the standard Java Timer class (Joy et al., 2000) which is highly scalable. As it will be shown in the section of the performance analysis later on, this implementation proved to be very efficient. Finally, the lightweight Log4J (Gülcü, 2003) framework, developed by the Jakarta Project (Goodwill, 2001) team, was used for the logging mechanism.

#### 3.3.1. The scheduling subsystem

The core component of the subsystem is the SchedulerBean, which is modeled as a stateless session bean. The Subsystem, which decouples the core scheduling engine from the execution of the tasks, has a significant effect on the overall performance of the scheduling mechanism.

Finally, the Scheduler features a flexible and reconfigurable logging mechanism, which can log every internal activity with little additional performance overhead. The scheduler architecture along with the relationships between the different subsystems and their components is depicted in Fig. 1. Further elaboration on each subsystem and implementation details are provided below.
SchedulerBean is responsible for scheduling tasks in the Timer or sending tasks directly for execution in the TaskHandler. Furthermore, it collaborates with the Persistence Subsystem, which stores the information of each task in a database in case of system failure. The SchedulerBean has three basic methods: “initialize”, “schedule” and “remove” (see Table 1).

Method “initialize”. This method initializes the Scheduler. Initialization includes among others the rescheduling of all tasks that are stored in the database. Every task retrieved from the database is checked for the expiration date and if it has not expired yet is rescheduled, or else is removed from the database. The relevant UML Sequence Diagram can be viewed in Fig. 2.

Method “schedule”. This method is invoked when a new task needs to be scheduled. After retrieving a unique sequence number (primary key) from the SequenceBean, the task is scheduled in the Timer. This number is used as the primary key of the new record in the database, where all the information about the task is stored. Persistent data include start and stop time and the execution period. Fig. 3 shows the steps performed by this method. The procedure just described stands for time-triggered tasks. In the case of an event-triggered task,

<table>
<thead>
<tr>
<th>Method</th>
<th>Arguments</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Schedule</td>
<td>userID: the identity of the user that scheduled the task; invocationParameters: information regarding the start time, stop time and execution period of the scheduled task; taskData: parameters related to the task execution logic; scheduleID: a unique ID identifying the specific task. It is also used as a primary key for the persistence mechanism</td>
<td>scheduleID</td>
</tr>
<tr>
<td>Remove</td>
<td>scheduleID: the ID of the task to be removed</td>
<td>–</td>
</tr>
</tbody>
</table>

Fig. 2. The sequence diagram of method “initialize”.

Fig. 3. The sequence diagram of method “schedule” (time-triggered).
execution takes place immediately without the need for Timer registrations. Execution data is sent directly to the TaskHandler.

Method "remove". The method is called for removing a scheduled task from the Timer. After its removal the task is removed from the database and will not be executed again.

Each task is implemented as object of the class Task and has a start time, a stop time and a period. We refer to all of them as "invocation parameters". Start is the time that the scheduled task will be invoked for the first time. If not specified, immediate execution is performed. Stop is the time that the task will be invoked for the last time and, then, will be removed from the Scheduler. If not specified, infinite execution is assumed. Period defines the period in the case of periodic events. If not specified (or has value equal to 0), indicates that the task will be executed at most twice (once at the start and another one at the stop time, unless those two are the same). If not specified (or has value equal to 0) and stop is not specified also the event will take place once at the start time. There are eight possible combinations of the invocation parameters, which are presented in Table 2. E stands for "Exists" and N stands for "Not specified". For example, if invocation parameters of a task are ENE this means that the task has a start time, the stop time is set to "Not specified" and has a period different from 0. All these combinations after being processed by the SchedulerBean, are reduced to two (highlighted in Table 2) for easier handling and better performance. So, if start time is N then as start time is set the current time and thus it is transformed to E. The EEN and ENE cases are also transformed to EEE. Apart from invocation parameters and ID, the object Task has another attribute which defines the data of the service to be executed. A service can be represented by any Java object or servlet.

The Timer is the most important component of the architecture. It extends the java.util.Timer class. This class provides a scheduling engine that is responsible for the execution of tasks. Each Timer object uses a single background thread for executing all of the timer’s tasks. Internally, it uses a binary heap to represent its task queue, so the cost to schedule a task is $O(\log n)$, where $n$ is the number of concurrently scheduled tasks.

Tasks may be scheduled for one-time execution (ENN case), or for repeated execution at regular intervals (EEE case) in two ways: fixed-delay execution and fixed-rate execution. In fixed-delay execution, each execution is scheduled relative to the actual execution time of the previous execution. If an execution is delayed for any reason, such as garbage collection or other background activity, subsequent executions will be delayed as well. In fixed-rate execution, each execution is scheduled relative to the scheduled execution time of the initial execution. If an execution is delayed, two or more executions will occur in rapid succession to “catch up” (Joy et al., 2000). In Scheduler we chose fixed-delay execution because this method is appropriate for activities that require “smoothness” keeping the frequency accurate in the short run than in the long run. If a task takes excessive time to complete, it “hogs” the timer’s task execution thread. Therefore, in Scheduler, tasks were designed to complete quickly as it will be shown in subsequent paragraph.

Scheduled tasks can be canceled at any time, using the reference ID assigned to them upon their scheduling. Canceling a task results to its removal from both the Timer and the database.

### 3.3.2. The persistence subsystem

The Persistence Subsystem, comprised of the entity beans TaskBean and SequenceBean (see Fig. 1), is responsible for storing data pertaining to time-triggered tasks in order to maximize the reliability of the scheduling service. It consists of two main components. The TaskBean is the interface of the relational database used for storing the task details. It is implemented as a CMP 2.0 entity bean. CMPs can be developed much more rapidly because the EJB container performs storage operations, handles all data access logic and generates the JDBC code. There is no need to implement any persistence logic (such as JDBC or SQL/J). This simplifies development and enhances scalability offering an abstraction layer over JDBC (Eckel, 2000). The SequenceBean, which is responsible for generating unique identifiers (IDs) for the scheduled tasks, is also considered as part of this subsystem. The generated IDs are used as primary keys for the records stored in the database. Stored data includes the start and stop time as well as the execution period for each task and is used during the initialization of the Scheduler.

### 3.3.3. The task execution subsystem

The basic component of this subsystem is the message-driven bean TaskHandler, which is bound to a message queue (see Fig. 1). When a Task “runs”, it acts like a JMS client that sends a message to the Queue. This message contains information pertaining to the service.

<table>
<thead>
<tr>
<th>Start</th>
<th>Stop</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>N</td>
</tr>
<tr>
<td>E</td>
<td>N</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>N</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>E</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>
execution. The SchedulerBean has also the same role (i.e., JMS client) in the case of event-triggered tasks. This asynchronous implementation has significant impact on performance because tasks complete quickly without delay. Specifically, the actual operation performed by each scheduled task is degraded to the simple insertion of a message in the JMS queue. This is considered as a very lightweight operation, compared to what a service might have involved. Execution of the service takes place immediately after the message is consumed from the queue and is completely decoupled from the Scheduling Subsystem. Thus it does not affect the scheduler’s performance.

4. Performance evaluation

The performance evaluation of the scheduler component was performed in a PC powered by an Athlon 1.3 MHz CPU. The operating system was Windows 2000 Professional, configured to execute a minimum set of background services. JBoss 3.0.4 was the J2EE-compliant application server which was used for both the development and the performance evaluation of the software component. Its embedded Hypersonic SQL database was used for persisting tasks. All subsystems were deployed on a single application server with no special performance tuning applied.

For the presented analysis, different test cases were developed and used. In most of these tests the time intervals between the scheduling of two successive tasks follow the exponential distribution (thus, simulating a random arrival model). The exact testing configurations, as well as the metrics used, are described in the following sections.

4.1. Metrics

The performance metrics of a scheduler can be divided in system metrics and user metrics (Bapat, 2001; Jain, 1991; Feitelson and Rudolph, 1998). The first category includes throughput, utilization (the fraction of time that the system remains busy), makespan and efficacy. The two latter are used mainly in distributed computing environments hence they were not considered in our case. Utilization was also excluded as it tries to capture how efficiently the system resources are being utilized and is best suited for low-level schedulers (e.g., CPU schedulers). However throughput was measured as it gives a feeling of the Scheduler’s capabilities. The second category, which is of prime importance for an application scheduler, includes turnaround time and average delay. These metrics focus on the measurement of the performance from the end user’s perspective. The inclusion of the above metrics in a test suite, can give a quite clear view of the scheduler’s capabilities and limitations.

The most important metric in a real-time scheduler is the actual time that a task execution has delayed. The delay metric is defined as the elapsed time between the trigger of a task and the start of its execution. In our case, delay includes both the delay of the timer and the time a task has been waiting in a queue. One good thing about delay is that it does not have to deal with the nature of the jobs (unless they are classified into priority classes). Hence, a simple arithmetic mean is enough to capture the implications of average delay (provided that outliers have been excluded from the set of samples). This time, as already stated, should be less than a second for the vast majority of the tasks. Hence, the appropriate time measurement unit is the millisecond. In order to measure the maximum, minimum and average task delays and indicate their correlation with the scalability of the scheduler, several tests were performed.

The throughput, which is defined as the number of jobs that are completed per unit time, has also been measured. This metric, is not very important for end users, but is strongly related with the scalability of the scheduler.

Turnaround time is the time from the moment the user submits the job to the system till the system finishes its execution. From the user’s perspective, turnaround time is a crucial metric, which must be minimized. Turnaround time has two components, waiting time and execution time (or run time). The waiting time is the time that the job spends in a queue waiting to be assigned to a server while the run time is the time the host takes to complete executing the task. In the context of the present test suite, we measure the turnaround times for event-triggered tasks that are submitted initially to a heavily loaded scheduler and subsequently to an idle one. Finally we introduce a new metric, named Normalized Delay. This metric is defined as follows:

\[
\text{Normalized Delay} = \frac{\text{delay per task group}}{\text{group period}}
\]  

where task group stands for a set of tasks with the same execution period (called group period). The delay in the numerator can be either an average or maximum value. Normalized Delay gives a qualitative view of the delays imposed by the Scheduler.

4.2. Test descriptions and results

Before performing the actual tests, test cases were examined in order to determine the behavior of the Scheduler under heavy load conditions. The results of these stress tests allowed the determination of the limits for the performance of our Scheduler. During the initial stress test 300 tasks were scheduled for simultaneous execution (at the same millisecond). The delays observed were between 0 and 30 ms and the average delay was
approximately 19 ms. This time value is an approximation of the minimum interval required between the executions of two successive tasks so as to keep delay in low levels. It also gives a rough approximation of the maximum throughput the Scheduler can handle before delay starts to increase exponentially, which was found to be in the order of 50 tasks/s (1000 ms divided by a 19 ms task interval). Indeed, when 50 tasks were scheduled, each one of them having an 20 ms interval from the previous one, the average delay was 17 ms. Subsequent similar experiments showed that the delay is a linear function of the number of tasks scheduled.

4.2.1. Test 1: delay and normalized delay (ND)

This test demonstrates the delays imposed on the tasks by the scheduler. Apparently, hourly or daily periodic tasks, even in a large number, would not induce an overload problem on the scheduler. We are interested in more frequent executing activity. In this respect, the task periods were randomly selected from the set [2, 3, 5, 10, 30] seconds. It should be noted that, such periods are not representative for real-world tasks, but are considered appropriate for the desired load tests.

As tasks are assumed to commence execution at random times (thus simulating random arrivals) the interval time \( X \) between the scheduling of two successive tasks was modeled as exponentially distributed variable with mean arrival rate \( \lambda = 5 \) tasks/s. The formula used is:

\[
X = -\frac{1}{\lambda} \ln(u), \quad \lambda: \text{arrival rate}, \quad u: \text{a uniformly distributed random number in the } [0,1] \text{ interval} \quad (2)
\]

Provision has also been taken so as not to exceed the limit of 50 task executions/s. Thus, the number of tasks in each period group was adjusted accordingly. Fig. 4 depicts the average and maximum delay observed. The minimum delay was kept stable at 10 ms, which reflects the delay for inserting and consuming a message from the queue. Table 3 shows the normalized delays. ND\( \times \) denotes the normalized delay of the tasks with period \( \times \) seconds. These values were calculated from samples taken during the last minute of test execution, when all the tasks were active. The average throughput during this minute was 2500 tasks/min (or 41.67 tasks/s). A first observation from this test is that smaller values of the group period lead to more rapid congestion, which, of course, is normal. However for period values in the order of 30 s, which again can be considered excessive for application-level scheduling, the average ND remains steadily very low (<0.02), while its maximum value does not exceed 0.1. Taking into consideration that typical periods for scheduling application-level services are normally much bigger, we can safely conclude that the Scheduler does not impose bottleneck to a service provisioning system.

4.2.2. Test 2: delay with variable offered load

In this test, we exceeded the 50 tasks/s limit and observe how performance varies. The offered load (tasks/s) is increased on steps of 5 tasks/s and the periods (the same as in Test 1) are evenly distributed among them.

As shown in Fig. 5, maximum delay increases substantially as throughput increases. Moreover, Fig. 6 shows that the delay is a rough approximation of a lin-

![Figure 4](image.png)

**Fig. 4.** Average and maximum delay.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>ND2</th>
<th>ND3</th>
<th>ND5</th>
<th>ND10</th>
<th>ND30</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.02/0.57</td>
<td>0.01/0.25</td>
<td>0.01/0.17</td>
<td>0.00/0.13</td>
<td>0.00/0.04</td>
</tr>
<tr>
<td>200</td>
<td>0.06/0.69</td>
<td>0.03/0.43</td>
<td>0.03/0.31</td>
<td>0.02/0.15</td>
<td>0.01/0.06</td>
</tr>
<tr>
<td>300</td>
<td>0.04/0.39</td>
<td>0.04/0.30</td>
<td>0.02/0.14</td>
<td>0.01/0.10</td>
<td>0.01/0.05</td>
</tr>
<tr>
<td>400</td>
<td>0.05/0.76</td>
<td>0.04/0.27</td>
<td>0.02/0.28</td>
<td>0.02/0.22</td>
<td>0.01/0.08</td>
</tr>
<tr>
<td>500</td>
<td>0.09/0.54</td>
<td>0.05/0.34</td>
<td>0.03/0.40</td>
<td>0.02/0.25</td>
<td>0.01/0.06</td>
</tr>
<tr>
<td>600</td>
<td>0.17/1.14</td>
<td>0.07/0.34</td>
<td>0.03/0.26</td>
<td>0.02/0.19</td>
<td>0.01/0.05</td>
</tr>
<tr>
<td>700</td>
<td>0.13/0.98</td>
<td>0.06/0.33</td>
<td>0.06/0.58</td>
<td>0.03/0.29</td>
<td>0.02/0.08</td>
</tr>
<tr>
<td>800</td>
<td>–</td>
<td>–</td>
<td>0.16/0.71</td>
<td>0.04/0.34</td>
<td>0.02/0.06</td>
</tr>
<tr>
<td>900</td>
<td>–</td>
<td>–</td>
<td>0.09/0.43</td>
<td>0.03/0.16</td>
<td>0.02/0.07</td>
</tr>
<tr>
<td>1000</td>
<td>–</td>
<td>–</td>
<td>0.03/0.12</td>
<td>0.02/0.22</td>
<td>0.01/0.09</td>
</tr>
</tbody>
</table>

Table 3
Normalized delays (average/max) for each period group
ear function of time (or task executions). The solid line shows the first execution of the last scheduled task, whereas the dashed line shows the time this task should normally have been executed. If the offered load is higher than 50 tasks/s, the delay systematically increases. Finally, Fig. 7 shows the delay values and behavior for a "normal" throughput of 50 tasks/s. In this figure, the first execution of the last scheduled task and the time it should normally have been executed are almost identical (dashed line), as there is only a 23 ms delay between them.

4.2.3. Test 3: turnaround time of event-triggered tasks

This test is quite similar to Test 1. The only difference is that during its last minute around 360 event-triggered tasks were scheduled at random intervals. The rationale behind this test case was to measure the turnaround times of these event-triggered tasks. The measured average and maximum turnaround times are depicted in Table 4, as well as the percentage of tasks with execution times greater than 20 ms. Finally these measurements are compared with those measured on an idle scheduler.

4.2.4. Test 4: heavy load conditions

This test case aims to demonstrate the scheduler's behavior under more realistic conditions regarding the service provisioning context, which is after all the targeted environment for the Scheduler. The periods shown in Table 5 were used.

The number of scheduled tasks was also increased significantly (10,000, 15,000 and 20,000 tasks) and the scheduling rate was raised to 10 tasks/s. To compare the results of the experiments both the delay and the number of scheduled tasks were monitored, 40 min after the initial scheduling.

For 10,000 tasks, the results were quite satisfactory: the average delay was below 200 ms, and all the tasks have been scheduled 15 min after the start. For 15,000

Table 4

<table>
<thead>
<tr>
<th>Turnaround times</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average turnaround time (ms)</td>
<td>Maximum turnaround time (ms)</td>
<td>Tasks with execution times &gt;20 ms (%)</td>
</tr>
<tr>
<td>Idle Scheduler</td>
<td>1.6</td>
<td>1.6</td>
<td>&lt;1</td>
</tr>
<tr>
<td>Loaded Scheduler</td>
<td>3–50</td>
<td>800–2640</td>
<td>~10</td>
</tr>
</tbody>
</table>

Table 5

<table>
<thead>
<tr>
<th>Periods of tasks for Test 4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage of scheduled tasks</td>
<td>Period (min)</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>70</td>
<td>10</td>
</tr>
</tbody>
</table>
tasks, the delay was very short (below 300 ms) until the number of scheduled tasks reached 13,800. After that point the delay increased rapidly, and 40 min after the test start, delay was about 8 min with only 14,600 out of the 15,000 tasks having been scheduled. Finally, for 20,000 tasks the delay started to increase uncontrollably when the number of tasks reached 12,000 and after 40 min of test execution it reached the 10 min, while only 12,350 tasks had been scheduled.

The above results are quite promising, as the majority of the tasks in real-world conditions should have even larger periods.

4.2.5. Final comments

The final experiment was a variation of Test 3. Its only difference was that instead of submitting event-triggered tasks to the scheduler, we scheduled the same number of tasks. Our intention was to estimate the impact that the scheduling procedure, with the database insertion overhead, would have on the delay times. The results showed that the imposed overhead did not lead to significant performance degradation.

In general, what became obvious from this study, is that the main factor that affects the performance is not the number of scheduled tasks but the load that was offered by the scheduler's clients. Another observation is that the task delays, in tests with low offered load, form a saw-like pattern (see Fig. 8).

Fig. 9 proves that the Timer is responsible for this behavior, as it shows that the delay of a single Timer, that executes dummy tasks, follows the same pattern. From this and other similar experiments, we can safely conclude that the Timer imposes a maximum 10 ms delay, regardless of the number of scheduled tasks.

Moreover, in Fig. 8 only a small percentage of the tasks have delay higher than the average. This percentage is manifested in the plot through the shown peaks and is, mainly, due to operating system services, which run on the background. The times shown in this figure have been collected after the execution of 1000 non-overlapping tasks (the tasks were triggered sequentially).

The aforementioned results prove that only a minimal performance overhead is imposed by the Scheduler. In Section 6 some possible future improvements for optimizing the operation of the Scheduler are discussed.

5. Scheduler application in mobile service provision

The presented Scheduler architecture has been developed as an integrated component of PoLoS (Ioannidis et al., 2003), a generic location based services (LBS) platform. The main objectives of PoLoS are the development, deployment and provision of LBS. It can be deployed over any J2EE compliant server, thus being an inherently open solution. Aiming to provide an integrated solution for delivering LBSs, PoLoS features multiple interfaces with positioning systems, GIS (Geographic Information System) servers or other network elements. It supports all major transport protocols from the wireless domain (e.g., HTTP, WAP or SMS), thus allowing access from a wide range of end terminals, such as mobile phones, smart phones, personal digital assistants (PDAs) and notebooks. PoLoS aims to be independent from the underlying technologies and as diverse as possible and towards this direction the positioning interface covers a wide spectrum of existing positioning techniques, from 2G/3G and WLAN environments, while GPS location is also supported.

---

The work presented in this paper has been performed in the framework of the project IST-2001-35283 “PoLoS”, which is partly funded by the European Commission and the Swiss BBW (Bundesamt für Bildung und Wissenschaft).
(Prigouris et al., 2003). Moreover interfaces with GIS systems, SMS gateways and other external entities have been developed using the Web Services model and standardized frameworks such as 3GPP’s OSA (Open Service Access) (OSA, 2002). For assisting the service creation process PoLoS offers an Integrated Development Environment (IDE), which supports the process of writing new services and deploying them inside the provisioning platform. Services are created through an XML-based Service Creation Language (SCL), which was defined specifically for this reason (Ioannidis et al., 2004).

The heart of the PoLoS architecture is its Kernel (Spanoudakis et al., 2003), which is responsible for coordinating the collaboration between the peripheral entities (e.g., GIS server, positioning systems, billing systems, interfaces components) and its internal components. Moreover the kernel handles issues like service deployment, service management and service removal by maintaining interfaces towards the appropriate functional entities. A bird’s eye view of the PoLoS framework is provided in Fig. 10. The Scheduler is a core module of the Kernel, which supports the Push paradigm (Chevest et al., 2002). The scheduler acts as a dummy client, on behalf of the user, which, triggers the execution of services, collects the results and dispatches them to the designated end user via the available WAP or SMS interface. Registrations to the Scheduler are performed through a convenient management interface, which allows users to register services for future execution according to the different invocation patterns supported by the scheduling mechanism.

6. Scalability issues

Every enterprise system should be scalable enough in order to support large numbers of concurrent users. The proposed scheduler is no exception to this rule, since its main target application domain is service provisioning. From the discussed experiments we observed that for very high loads (see Test 4 in Section 4.2), the application server resources (i.e., computing power, memory) were saturated. According to the Java documentation the Timer class, has high capacity which does not degrade the performance of the system (“...this class scales to large numbers of concurrently scheduled tasks; thousands should present no problem...”). The EJB components, on the other hand, although “intelligently” managed by the EJB container, certainly, impose scalability limitations.

The most common way to address scalability is through load-balancing techniques. Load-balancing refers to the distribution of workload to multiple application instances executing in different physical systems. Clustering is probably the most well-known load-balancing technique used by modern application servers. Unfortunately, scalability design issues are not, yet, covered by the J2EE specification. Thus, in order to make a clustered version of the Scheduler we have to consider some additional design aspects. The main challenge regarding clustered (time-triggered) applications is the existence of the Timer component itself.

Two different scenarios can be considered for handling potential scalability problems that may arise: The first scenario, which is depicted in Fig. 11, involves the deployment of the Timer outside the cluster of the EJB containers. The Timer is a singleton object, common for all the members of the cluster. Each container within the cluster (hereinafter referred to as cluster node) hosts the SchedulerBean, the entity beans (i.e., TaskBeans) and the TaskHandler along with the corresponding Queue (i.e., one per cluster node). Placement of the Timer outside the cluster is justified on the basis of its increased scalability and the fact that task registrations are not demanding and resource consuming operations. In this scenario, the SchedulerBean, has to be enhanced with additional methods, since apart from task registration, it handles the queue dispatching process, inside its local cluster node.
In the second scenario, which is illustrated in Fig. 12, all components of the Scheduler’s architecture, including the Timer, are clustered. This means that each cluster node has a local Timer, which is used for triggering tasks. An important issue, with this approach is that assignment of tasks to cluster nodes is handled by the load-balancing engine of the application server, without taking into account any information about the scheduled tasks. Since, such engines usually adopt simplistic mechanisms for distributing the experienced load (e.g., a round-robin scheme), it is possible that certain cluster nodes become overloaded, while others experience lighter load. For example, imagine the extreme case where we have a cluster of two nodes and the scheduling requests arrive in such a manner, that for each two arriving requests the first one is resource-demanding (i.e., requesting frequent execution for an extensive time period), while the second is significantly lighter (i.e., requesting infrequent
execution or execution for a much shorter period). After a while, the first cluster node will reach saturation, while the second node will have very low utilization. In order to avoid, such unbalanced operation, each Timer inside the proposed cluster, does not perform direct task execution, through its local queue, but rather directs each triggered task back to the external load-balancing engine. Hence, the task execution process is decoupled from task scheduling and is redistributed between available containers, thus, increasing the fairness and efficiency of the system. As in the previous scenario, the Scheduler-Bean should be enhanced with functionality responsible for the queue dispatching process.

The discussion in this section highlights the important issue of clustering a scheduling service. The J2EE community has recognized that this topic is of high interest for enterprise applications and has included a container-managed Timer Service in its EJB 2.1 specification. Such service is persistent and has been adopted by several vendors. However, it is more coarse-grained than the default Java Timer and not suited for real-time events.

7. Further work

Results of performance analysis showed that the message queue, to which the TaskHandler is bound, constitutes a bottleneck. When a large amount of tasks arrives in the queue a lot of time elapses before these tasks are consumed by TaskHandler. Multiple message queues can be used to eliminate this undesirable delay. Tasks for execution will be distributed among these queues and as a result the wait time will be effectively reduced. But this is useless when all queues are maintained by the same server (i.e., in a single machine with only one CPU). A clustered environment where queues, Task-Handlers (and possibly Timers), reside in different servers (and machines) would be ideal for such an implementation. Installing and testing the Scheduler in such a multi-CPU distributed environment is in our immediate future plans.

Improvements can be done also in the section of persistence. We can use the Java Data Objects (JDO) API that is a standard interface-based Java model abstraction of persistence. JDO (Tyagi et al., 2003) is used to directly store Java domain model instances in the database. CMP beans require active transactions for all business methods. Non-transactional access is not standard or portable. JDO allows choosing whether transactions are needed. JDO requires insertions, removals and updates to be performed within transactions, but read-only applications, such as caching, can also be implemented without transactions. The most important benefits when using JDO however include ease of use, high performance, and integration with EJBs.

8. Conclusions

We have presented a working scheduling model, which supports the scheduled execution of services or other tasks and is targeted to the enterprise domain. Automated and scheduled task execution is an essential ingredient of many modern enterprise systems, as its presence can significantly augment their application range. In the mobile market, where new types of services are continuously emerging everyday, such systems can provide a real boost to their absorption and use by a wide portion of end users. Within the LBS context, the ability to schedule the delivery of service content, while the user is moving between locations, either on a time basis or when certain events occur (e.g., entering an archeological site or a commercial center) can lead to the development of next-generation value added services. A fine-grained architecture of a Scheduler was presented along with the requirements that guided us throughout its design. Implementation aspects were covered in detail while, an extensive performance evaluation was performed in order to demonstrate the scalability, reliability and efficiency of the architecture, thus, assuring its applicability for enterprise-scale systems. The fact that its design and implementation was based on sound techniques and state-of-the-art open technologies makes it an ideal solution for integration with existing service provisioning platforms running over heterogeneous infrastructures.

References

Odysseas Sekkas received his B.Sc. in Informatics from the Department of Informatics and Telecommunications at the University of Athens, Greece, in 2003. He was accepted in the M.Sc. program in Communication Systems and Data Networks from the same Department. Currently, he is in the first year of his M.Sc. studies. He is involved as programmer in the development of the PoLoS (Integrated Platform for Location Based Services) project implemented in the context of IST. His research interests include wireless and mobile computing, QoS and mobility support for IP networks, and web engineering.

Stathes Hadjieftymiades received his B.Sc. in Informatics from the Department of Informatics at the University of Athens, Greece, in 1993 and his M.Sc. in Advanced Information Systems from the same department in 1996. In 1999 he received his Ph.D. from the University of Athens (Department of Informatics and Telecommunications). In 2002 he received a joint engineering-economics M.Sc. degree from the National Technical University of Athens. In 1992 he joined the Greek consulting firm Advanced Services Group, Ltd., where he was extensively involved in the analysis and specification of information systems and the design-implementation of telematic applications. In 1995 he became a member of the Communication Networks Laboratory of the University of Athens. During the period September 2001–July 2002, he served as a visiting assistant professor at the University of Aegean, Department of Information and Communication Systems Engineering.


Tyagi, S. et al., 2003. Core Java Data Objects. Prentice-Hall, USA.

Vassileios Tsetsos received his B.Sc. in Informatics from the Department of Informatics and Telecommunications at the University of Athens, Greece, in 2003. He is currently a postgraduate student at the same Department (Communication Systems and Data Networks division). He is involved as software developer in the PoLoS (Integrated Platform for Location Based Services) project implemented in the context of IST. His research interests include wireless/mobile computing, distributed computing, and web applications.


Tyagi, S. et al., 2003. Core Java Data Objects. Prentice-Hall, USA.

Vassileios Tsetsos received his B.Sc. in Informatics from the Department of Informatics and Telecommunications at the University of Athens, Greece, in 2003. He is currently a postgraduate student at the same Department (Communication Systems and Data Networks division). He is involved as software developer in the PoLoS (Integrated Platform for Location Based Services) project implemented in the context of IST. His research interests include wireless and mobile computing, QoS and mobility support for IP networks, and web engineering.

Ioannis Prigouris received his B.Sc. in Informatics from the Department of Informatics and Telecommunications at the University of Athens, Athens—Greece in 1997 and his M.Sc. in Communication Systems and Data Networks from the same Department in 2000. Over the last two years he has been a Ph.D. candidate in the department. Since 1999, he has been a member of the Communication Networks Laboratory (CNL) of the University of Athens. He has participated in the RAINBOW (Radio Access INdependent BrOadband on Wireless) and the EUROCTTI (EUROpean CITIes platform for on-line transaction services) projects implemented in the context of ACTS and IST correspondingly, as well as in several other national and European projects. Currently, he is extensively involved in the development of the PoLoS (Integrated Platform for Location Based Services) project implemented in the context of IST. His research interests are in the areas of mobile computing, QoS and mobility support for IP networks and location-sensitive resource management. He is the author of many publications in the above areas.

Stathes Hadjieftymiades received his B.Sc. in Informatics from the Department of Informatics at the University of Athens, Greece, in 1993 and his M.Sc. in Advanced Information Systems from the same department in 1996. In 1999 he received his Ph.D. from the University of Athens (Department of Informatics and Telecommunications). In 2002 he received a joint engineering-economics M.Sc. degree from the National Technical University of Athens. In 1992 he joined the Greek consulting firm Advanced Services Group, Ltd., where he was extensively involved in the analysis and specification of information systems and the design-implementation of telematic applications. In 1995 he became a member of the Communication Networks Laboratory of the University of Athens. During the period September 2001–July 2002, he served as a visiting assistant professor at the University of Aegean, Department of Information and Communication Systems Engineering. In July 2002 he joined the faculty of the Hellenic Open University (Department of Informatics), Patras, Greece, as an assistant professor in telecommunications and computer networks. Since December 2003, he belongs to the faculty of the University of Athens, Department of Informatics and Telecommunications. He has participated in numerous projects realized in the context of EU programs (ACTS, ORA, TAP, and IST), EURESCOM projects, as well as national initiatives. His research interests are in the areas of wireless/mobile computing, web engineering, and networked multimedia applications. He is the author of over 70 publications in these areas.