# Towards Timely ACID Transactions in DBMS

Marco Vieira[1], António C. Costa[2], Henrique Madeira[1]

[1] CISUC - University of Coimbra, Polo II,
3030 Coimbra, Portugal
{mvieira, henrique}@dei.uc.pt

[2] FC – University of Lisbon,
Lisbon, Portugal
casim@di.fc.ul.pt

**Abstract.** On-time data management is becoming a key difficulty faced by the information infrastructure of most organizations. In fact, database applications for critical areas are increasingly giving more importance to the timely execution of transactions. Database applications with timeliness requirements have to deal with the possible occurrence of timing failures, when the operations specified in the transaction do not complete within the expected deadlines. In spite of the importance of timeliness requirements in database applications, typical commercial DBMS do not assure any temporal properties, not even the detection of the cases when the transaction takes longer than the expected/desired time. This paper discusses the problem of timing failure detection in database applications and proposes a transaction programming approach to help developers in programming database applications with time constraints. The paper illustrates the proposed programming model with a practical example using the Oracle 10g DBMS running a performance benchmark for real-time database applications.

**Keywords:** Databases, transaction processing, performance, timely transactions

## 1 Introduction

Developing database applications with timeliness requirements is a very difficult problem as current database technology does not provide easy programming support that help engineers and programmers in dealing with timing issues. This is true for all the programming layers of a typical database application: the database management system (DBMS), the middle layer software (e.g., web-server, application-server, etc), and the client application. Nevertheless, real database applications very often have to cope with the possible occurrence of timing failures, when the operations specified in a transaction do not complete within the expected deadlines. Without adequate support to help designers and programmers to solve timing requirements, the development of these applications is a very complex task.

The notion of time is completely absent from the classical DBMS transactional model, which is based on the ACID properties (Atomicity, Consistency, Isolation, and Durability) [1]. In this paper we discuss the problem of timing failure detection in database applications and propose a transaction programming approach to help developers in programming database applications with time constraints. According to the timing requirements we classify database applications in different classes, namely: traditional (no temporal requirements), fail-safe, time-elastic, and fail-operational. To implement these classes we propose the following types of transactions, which support different temporal requirements: transactions with no temporal requirements (typical ACID transactions), transactions with strict temporal requirements, and transactions with relaxed temporal requirements.

This paper proposes a new approach for transaction programming, which allows concurrent detection of timing failures during execution, including for distributed transactions. Timing failure detection can be performed at the database clients' interface, in the database server, or in a distributed manner. An application programming interface (API) that implements this new transaction programming approach is provided. It can be used by database programmers to easily implement applications with timeliness requirements. All source code, including an example of utilization, is available at [http://gbd.dei.uc.pt/downloads.php] for public use.

The structure of the paper is as follows. The following section discusses the problem of timing failure detection in database applications and proposes a classification for database transactions and database applications. Section 3 proposes a new transactions programming approach and presents the application programming interface developed. Section 4 presents the experimental and Section 5 concludes the paper.

## 2 Timeliness Requirements in Database Applications

In many situations timeliness is as important as correctness and atomicity. For instance, in a database application designed to manage information about a critical activity, e.g., air traffic control, a transaction that reads and stores the current reading from a positioning radar must be executed in a short time (i.e., the longer it takes to execute the transaction the less useful the reading is). In other words, in many applications, a transaction that does not complete before a specified deadline becomes useless and this situation must be reported to the application/business layer in order to be handled in an adequate manner. But worse than becoming useless, the delayed execution of a transaction can compromise safety properties of a system. In such cases, the detection of a violated deadline would allow the execution of fall-back or recovery actions, isolating and avoiding the propagation of the failure to other components and, consequently, the occurrence of more severe failures.

Real-time databases have emerged some years ago. However, these databases have been designed and implemented for very specific applications [2], [3]. To support real-time applications, real-time databases relax the ACID requirements to allow better support for temporal consistency while maintaining support for data consistency [4], [5]. Semantic information is used to determine to what degree the ACID properties must be enforced.

In real-time DBMS the ACID properties are normally applied only to parts of the transaction. Nevertheless, important features such as timing failure detection or, more generically, timing fault-tolerance, have been completely neglected, which also restricts the application areas for such DBMS. The problem is even worse if we consider the possibility of deploying distributed DBMS over wide-area or open environments. Such environments exhibit poor baseline synchrony and reliability properties, thus making it more difficult to deal with timeliness requirements. Obviously, this uncertainty and lack of timeliness will directly affect the execution of transactions, which, as an immediate effect, will be delayed. However, more severe effects may also be observed on the account of timing failures.

The environments we consider in this paper can be characterized, essentially, as environments of partial synchrony. In fact, their basic synchrony properties are only cluttered from time to time, or by specific parts of the structure. Several partial synchrony models have been proposed, with different solutions to address application timeliness requirements. The idea of using failure detectors that encapsulate the synchrony properties of the system was first proposed in [6]. The work in [7] introduces the notion of Global Stabilization Time (GST), which limits the uncertainty of the environment. The Timed model, proposed in [8], allows the construction of fail-aware services, which always behave timely or else provide awareness of their failure.

Our proposal is to bring timing failure detection to the typical ACID transactions implemented by most commercial DBMS, putting together classic database transactions management and distributed timely computing. The goal is to extend the typical transaction programming approach in order to support transactions with ACID properties together with timing failure detection.

In order to add timing failure detection to the typical ACID transactions, we propose that the basic toolset to be offered to database application programmers should include the following classes of transactions:

– **Transactions with no temporal requirements**: typical ACID transactions implemented by classic database management systems. The database clients do not expect any timeliness guarantees, not even the detection of timing failures.

– **Transactions with strict temporal requirements**: for this class, the database clients can specify a time frame in which the transaction has to be concluded to succeed. In this class, the system must at least provide timing failure detection, even in distributed transactional environments. The transaction is rolled back if it does not commit in the specified time and the client database application is notified in order to cope with the occurrence of the timing failure.

– **Transactions with relaxed temporal requirements**: in this class, the transactions are always executed independently of the specified time frame. However, if the deadline is reached before the transaction commits, the client application is nevertheless informed. This allows the application to execute any task related to the occurrence of the timing failure (e.g., notify the DBA) and continue the execution of the transaction.

Real database applications very often have to deal with different timing requirements, whose execution must be supported by one or more of the classes of transactions proposed before. The following points present our classification for database applications concerning timing constraints, and give some examples of applications from real scenarios:

– **Traditional applications class**: typical applications with no temporal requirements.

– **Fail-safe class**: applications that can switch to a fail-safe state when there is a

timing failure. When a transaction is submitted the application waits for the transaction response or a notification that a timing failure has occurred. The application must be informed about a timing failure occurrence as soon as the deadline specified for the transaction is exceeded. In this case the application executes some conforming actions and enters a fail-safe state. These database applications can be implemented based on transactions with strict temporal requirements. Typical examples include manufacturing industrial processes (electronic industry, automotive manufacturing industry, etc) where it is possible to stop the manufacturing chain in case of delay in the database transaction execution (that fail-safe state is in general necessary because of mechanical issues of manufacturing processes).
– **Time-elastic class**: applications able to adapt timing constraints during execution. In this case, the collection of information about timing failures and the temporal execution of transactions can be used to feed a monitoring component or to tune specific application parameters in order to adapt its behavior to the actual load conditions of the system. The application may decrease the transaction submission rate, increase the transactions deadline if possible, or postpone the execution of transactions to a latter time. In environments with replicated databases, the application can also perform load balancing based on timing failure detection. These applications can be implemented based on transactions with strict temporal requirements or based on transactions with relaxed temporal requirements. Examples of this class include databases that control mobile communication systems, where connection establishment can tolerate some delays (or may be refused) and billing transactions can be postponed, or continuous manufacturing processes such as chemical processes.
– **Fail-operational class**: applications that continue executing transactions without adapting timing constraints during execution, regardless of the timing failures detected. The client application should be notified about the occurrence of the timing failure but the execution of the transaction does not stop. Timing failure detection is used by the application to perform specific actions (e.g., notify the database administrator) when the execution of transactions exceed the deadline. These applications can be implemented based on transactions with strict temporal requirements or transactions with relaxed temporal requirements. Examples of this class include pay-per-view television applications and video streaming.

## 3 New Transactions Programming Approach

Transactions in typical database application are executed by submitting one command at a time or using batches of commands. After submitting a command (or a batch of commands) the client application waits for the corresponding response. The database server can be a single machine or a distributed set of replicated servers. An important aspect is that neither the client layer nor the database server are concerned about the detection of timing failures.

In order to provide transactions with temporal requirements we need a new approach for transaction programming. Our proposal is to modify the typical database environment in order to include timing failure detection capabilities. Three possibilities can be considered: detection in the client layer, detection in the database server, and distributed detection. To measure the duration of both local and distributed actions we have adopted the Timely Computing Base (TCB) model [9], which is based on an improved round-trip technique that guarantees not only bounded, but also almost stable measurement errors. Informally speaking, the TCB Distributed Duration Measurement service can be used for monitoring the duration of local or distributed actions. It is the responsibility of the application to indicate the beginning and the end of the action. Local actions are measured in the same site, and in the case of distributed actions the measurement is made in a distributed way, using different TCB instances.

### 3.1 Timing Failure Detection in the Client Database Interface Layer

In a typical database environment the client application communicates with the server through a database interface layer (e.g., Oracle Call Interface). This layer is specific for each DBMS and is responsible for managing all the communication with the database server. The detection of timing failures in the client requires the modification of this layer. As changing the database interface layer itself is difficult (this is normally a piece of proprietary software) our proposal is to add a wrapping layer through which all the communications between the client application and the database interface layer must go by. We call this layer Transactions Timing Failure Detection (TTFD) layer.

Figure 1 shows the architecture we propose for timing failure detection in the client interface layer. In order to implement timing failure detection capabilities, we propose the use of two connections between the client application and the TTFD layer: one to submit commands and receive results and the other to control timing definitions (e.g., the deadline and the type of the transaction) and receive timing failure notifications. These notifications are sent in the form of exceptions that must be handled by the client application. Note that, the TTFD layer can be used to abstract any particular implementation of a timing failure detection service. Therefore, it is possible to generate timing failure notifications at this layer, independently of the specific notification mechanism provided by the specific TTFD service implementation.

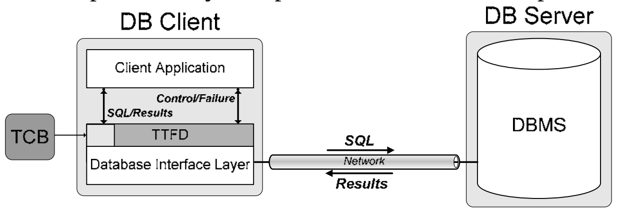

**Fig. 1.** Environment for timing failure detection in the database interface layer.

When the client application begins a time critical transaction (i.e., submits the first SQL command in the transaction) and provides the class of the transaction (strict temporal requirements or relaxed temporal requirements) and the deadline for the execution, the TTFD layer starts counting the elapsed time. The Duration Measurement service of the TCB is used for time measurement. The time critical transaction ends when the application executes a commit or rollback. If the deadline is violated before the end of the execution of the transaction an exception is generated and thrown to the client. If the transaction has strict temporal requirements it is automatically rolled back, otherwise, the transaction execution continues normally.

From the client point-of-view, the measured execution times includes the delays due to client-server network communication. This means that some false positives may occur due to the extra time that it takes for the response to travel from the server to the client. Timing failure detection in the database interface layer should be used in scenarios where it is important to take in consideration the time the client application waits for the response to the last command in the transaction (e.g., when the client application is able to perform load balancing at the network level). However, other solution is needed in scenarios where timeliness requirements apply to timed executions that terminate on the server side.

### 3.2 Timing Failure Detection in the Database Server

The second approach we propose consists on detecting timing failures at database server side. In this case the transactions execution time is measured from the server point-of-view. To provide the detection of timing failures in the database server we need to modify the DBMS implementation or to use a DBMS proxy that intercepts all the messages arriving to and leaving from the database server (see Figure 2). The database proxy also implements timing failure detection and the TCB is used to measure time. In the client side we also need the layer through which all the communications between the client application and the database interface layer must go by (TTFD layer). In this architecture this layer does not deal with timing failure detection. It receives timing definitions from the client application and forwards them to the database proxy. When the proxy detects a timing failure it notifies the TTFD layer, which raises the corresponding exception to the client application.

An important aspect is that all the communication between the TTFD layer and the DBMS proxy is performed using a communication channel different from the one used to send the SQL commands and results. The timing definitions and timing failure notifications are sent using this channel. Figure 2 presents the environment that we propose for timing failure detection in the database server. As mentioned before, the database server can be a single machine or a set of replicated servers.

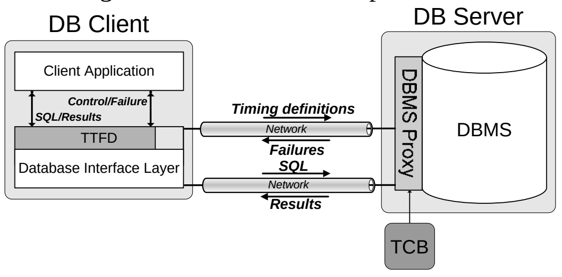

**Fig. 2.** Environment for timing failure detection in the database server.

When the client application begins a time critical transaction (i.e., submits the first SQL command) and provides the class of the transaction and the deadline for the execution, the DBMS proxy starts counting the elapsed time using the TCB. If the deadline is exceeded before the end of the execution of the transaction an exception is sent to the client application through the TTFD layer. Transactions with strict temporal requirements are automatically rolled back. On the other hand, for transactions with relaxed temporal requirements the execution continues normally. Note that, rolling back the transaction is not influenced by any timing constraints as the state of the database only changes if the transaction commits.

Timing failure detection from the server point-of-view does not consider the amount of time that goes from the moment when the client submits the first command and the moment the server receives that command. This means that the transaction execution time is counted only after the first command is received by the DBMS proxy. Thus, if the communication between the client and the server is slow, there are some cases where a timing failure is not detected because the client/server communication time is not considered. Timing failure detection in the database server is useful in scenarios where the network delays should not be taken into account or are always so small that, from the client application point-of-view, have no impact in the transaction execution. It may also be useful if one decides to enrich the database server with real-time modules that are autonomously and immediately executed by the TCB upon failure detection.

## 3.3 Distributed Detection of Timing Failures

A transaction starts when the client application submits the first command and ends immediately after the server finishes the execution of the last command (and not when the client application receives the response). Thus, some database applications may require the execution time to be counted from the moment when the client submits the first command and the moment when the execution of last command ends at the server side. In this case the two solutions proposed before cannot be applied. It is necessary to use a form of timing failure detection based on distributed duration measurements. We will simply call it distributed timing failure detection.

An obvious problem raised by this approach is the distributed measurement of time. As it is well known, it is quite difficult to have synchronized clocks in different machines. To solve this problem we have decided to use the Duration Measurement service of the TCB model [9]. This service obviously requires local clocks of TCB modules to be read, and timestamps to be used and disseminated among relevant nodes of the system. Unlike the measurement of local actions, measuring distributed durations is quite more difficult because simply reading the clocks to get two timestamps is not sufficient. The distributed duration measurement service of the TCB is based on an improved round-trip technique [9] that guarantees not only bounded, but also almost stable measurement errors.

As shown in Figure 3, to provide distributed detection of timing failures we need to modify the DBMS implementation or to use a DBMS proxy and to include an additional layer in the database client that handles all the communications between the client application and the database interface layer (TTFD layer). This layer does not detect timing failures. It receives timing definitions from the client application and instructs the TCB to start measuring the time. Timing definitions are sent to the

DBMS proxy that detects timing failures using the distributed duration measurement capabilities of the TCB. When a timing failure is detected the DBMS proxy notifies the TTFD layer, which raises the corresponding exception to the client application.

As in the solutions proposed before, two connections are needed between the client application and the TTFD layer and the communication between the TTFD layer and the DBMS proxy also uses a communication channel different from the one used to send the SQL commands and receive the responses.

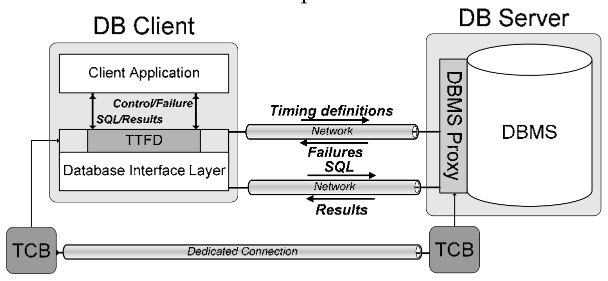

**Fig. 3.** Environment for distributed timing failure detection.

## 3.4 Transactions Programming Interface

In order to allow database programmers to easily implement applications with timeliness requirements we have developed an application programming interface (API) that implements the new transaction programming approach proposed in this paper. In this work we have implemented this interface for JAVA and we are now starting to implement it for other languages (C++ and Delphi). An important aspect is that this API is as close as possible to the ones normally used by the programmers when implementing typical database applications. In fact, we tried to implement classes and methods similar to the ones typically used by programmers in terms of the names, parameters, and form of use (see example in section 4). Table 1 presents the most important methods provided.

## 4 Practical Example of Implementation

The example presented in this section has resulted from the study meant to demonstrate and evaluate the usefulness of the transactions programming approach proposed. As we are particularly interested in the aspects related to transaction execution time, we have decided to use a performance benchmark for real-time database systems for telecommunications [10]. This benchmark represents a telecommunications operator that includes several service providers, each one having its own database in a distributed environment.

The benchmark has been implemented using both the standard Java Database Connectivity (JDBC) API and the API proposed in this paper (considering both strict temporal requirements and relaxed temporal requirements). The first goal is to evaluate the effort needed to migrate from the traditional approach to the approach

**Table 1.** API provided for JAVA programmers. All source code, including an example of utilization, is available at [http://gbd.dei.uc.pt/downloads.php] for public use.

| TACID Method | JDBC Method | Short Description |
|---|---|---|
| TACIDConnection getConnection(String URL, int approach) | Connection getConnection(String url) | Establishes a new TACID connection to the database. *url* represents the name of the database. *approach* is the timing failure detection level (client, server, or distributed) |
| void close() throws SQLException | void close() throws SQLException | Closes the database connection |
| void commit() throws SQLException, TACIDTimeoutException | void commit() throws SQLException | Commits the transaction. Throws an timeout exception if a timing failure has occurred meanwhile |
| void rollback() throws SQLException, TACIDTimeoutException | void rollback() throws SQLException | Rollbacks the transaction. Throws an timeout exception if a timing failure has occurred meanwhile |
| ResultSet startTimeQuery(int type, int timeout, String sql) throws SQLException, TACIDTimeoutException | ResultSet startTimeQuery(int type, int timeout, String sql) throws SQLException | Starts a new time critical transaction. SQL represents the first command (select command) in the transaction. *type* is the type of the transaction (strict temporal requirements or relaxed temporal requirements). *timeout* represents the deadline for the transaction Returns the result of the command in a result set |
| int startTimeUpdate(int type, int timeout, String sql) throws SQLException, TACIDTimeoutException | int executeUpdate(String sql) throws SQLException | Starts a new time critical transaction. SQL represents the first command (insert, update or delete command) in the transaction |
| ResultSet executeQuery(String sql) throws SQLException, TACIDTimeoutException | ResultSet startTimeQuery(int type, int timeout, String sql) throws SQLException | Executes a query. The time critical transaction has already been started by another command |
| int executeUpdate(String sql) throws SQLException, TACIDTimeoutException | int executeUpdate(String sql) throws SQLException | Executes an insert, update, or delete command. The time critical transaction has already been started by another command |
| ResultSet getResultSet() | – | Returns the result of the last query executed |
| int getResultUpdate() | – | Get the result of the last insert, update, or delete executed |
| long getExecTime() | – | Returns the execution time for the last transaction |

proposed in this paper. Table 2 presents a simple example of the use of timing failure detection in the benchmark.

As we can see, the TACID implementation is very similar (in both structure and commands) to the typical implementation, which facilitates the database programmers work. During the benchmark implementation we have observed a similar implementation time for both approaches. In fact, an experienced programmer tokes around two days for each implementation. Obviously this implementation time depends strongly on the programmer's experience.

In order to evaluate the efficiency of the time failure detection approaches, we have performed several experiments. The basic experimental platform consists of three machines. Two machines are used as database servers and one as database client. The machines are connected using two dedicated fast-Ethernet networks. One is used for the SQL/results communication and the other is used by the TCB. Six service providers are considered, by implementing three databases in each database server.

The Oracle™ DBMS is one of the leading databases in the market and as one of the most complete and complex database it represents very well all the sophisticated relational DBMS available today. For that reason we have chosen Oracle 10g [11], which has been tuned based on the results from a previous work on the evaluation of the Oracle performance and recoverability [12].

The performance benchmark used has been implemented using the traditional approach (no timing failure detection) and considering timing failure at the three layers (clients' interface, database server, and distributed). Both transactions with strict temporal requirements and transactions with relaxed temporal requirements have been implemented. The performance benchmark was executed five times for each configuration (a total of 35 runs) with a duration of 10 minutes for each run.

**Table 2.** TACID implementation *vs* typical implementation: excerpt from the update subscriber transaction. The TACID implementation uses transactions with strict temporal requirements.

| TACID Implementation | Typical Implementation |
|---|---|
| ... | ... |
| // Establish the connection | // Establish the connection |
| Class.forName(driverName); | Class.forName(driverName); |
| con = **TACIDDriverManager.getConnection** | con = **DriverManager.getConnection** (db); |
| (db, **distributedDetection**); | Statement con = con.createStatement(); |
| try { | try { |
| // First command (timeout 100ms) | // First command |
| sql="select addr from profile where | sql="select addr from profile where |
| sid="+sid; | sid="+sid; |
| rs=con.**startTimeQuery**(**strictTempReq**,100,sql); | rs=con.**executeQuery**(sql); |
| ... | ... |
| sql="update profile set addr='Coimbra' | sql="update profile set addr='Coimbra' |
| where sid="+sid; | where sid="+sid; |
| recCount = con.**executeUpdate**(sql); | recCount = con.**executeUpdate**(sql); |
| con.**commit**(); | con.**commit**(); |
| } | } |
| catch(**TACIDTimeoutException** e) { ... } | |
| ... | ... |

Results have shown that timing failure detection does not introduce any overhead in transactions execution. For example, for the baseline configuration we have observed an average of 3769.9 transactions per minute (with a standard deviation of 23.17 transactions), while for timing failure detection at the client interface layer we have observed an average of 3784.5 transactions per minute (with a standard deviation of 38.96 transactions). The small deviations in the measures in successive runs are normal and just reflect the asynchronous nature of transactions. For the other layers (server and distributed) similar results have been observed.

Concerning execution time, the average using the baseline configuration was of 46.62 milliseconds (with a standard deviation of 0.25 milliseconds), while the average with timing failure detection at the client interface layer is around 41.58 milliseconds (with a standard deviation of 0.35 ms). This shows that the execution time when using time failure detection at the client interface layer is lower than the one observed for traditional transactions. This is due to the fact that when a transaction exceeds the deadline it is immediately rolled back and the client application continues the execution to the next transaction. Similar results were observed for the other layers.

Figure 4 presents an example of the execution profile for one of the transactions (*roaming user*) during one run of the benchmark using timing failure detection at the client interface layer and strict temporal requirements (similar profiles were observed for the other layers of timing failures detection). As we can see some transactions exceed the deadline, however in all the cases the timing failure was detected and the client application notified. These transactions are automatically rolled back. As show in the figure, the transactions that exceed the deadline are detected a little bit after the deadline. This is due to the small latency of the detection mechanism (less than 20 milliseconds). Note that, database applications are characterized by long execution times (in some cases several seconds), thus a latency of 20 ms is quite acceptable.

Another important aspect is that, by the analyses of the results shown in Figure 4 we can see that even in sophisticated DBMS like Oracle 10g it is quite difficult to predict the execution time of the transactions. In fact, although most of the transactions are executed in before the deadline some of them exceeded that deadline. This demonstrates the importance of timing failure detection in database applications.

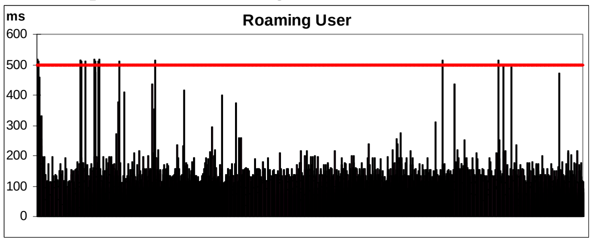

**Fig. 4.** Execution profile using timing failure detection. The horizontal line represents the deadline (500 ms) and each vertical bar represents the execution time of a single transaction. The vertical bars that cross the horizontal line represent transactions that exceeded the deadline.

To further understand the behavior of the timing failure detection mechanism we have executed the benchmark in the presence of an additional load that stresses the network and the server machines. This way, we have executed the real-time performance benchmark and, in random moments, we have executed the additional workload during a random amount of time. The additional workload has been adopted from the TPC-C performance benchmark [13] (this workload has been chosen due to practical reasons and any other workload could have been selected). The average number of transactions executed per minute decreased about 40%) and the average transactions execution time increased around 30%. The latency remained the same.

Figure 5 presents the execution profile for the roaming user transaction during one execution of the benchmark in the presence of the additional load. Note that, there are now many more transactions whose execution times exceed the deadline or gets closer to it. Nevertheless, all the timing failures were detected.

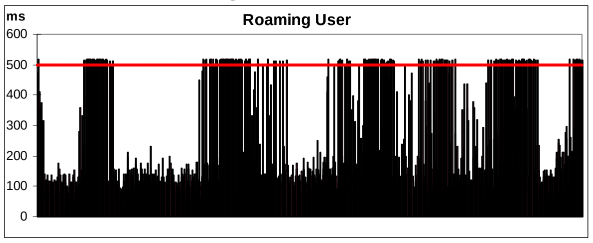

**Fig. 5.** Execution profile using timing failure in the presence of an additional database load.

## 5 Conclusion

This paper discussed the problem of timing failure detection in database applications and proposes a transaction programming approach to help developers in programming database applications with time constraints. Three classes of transactions are considered concerning temporal requirements: transactions with no temporal requirements (typical ACID transactions), transactions with strict temporal requirements, and transactions with relaxed temporal requirements. The approach proposed implements these classes of transactions by allowing concurrent detection of timing failures during transaction execution. Timing failure detection can be performed at the database clients' interface, in the database server, or in a distributed manner. The paper illustrates the proposed programming models in a practical example using the Oracle 10g DBMS. A performance benchmarks for real-time database applications is used to validate the approach and to show the advantage of timing failure detection.

From the results presented in this paper it is clear that it is useful to consider a new transaction programming approach aimed at supporting timing specifications for the execution of transactions. On the other hand, the work done so far was instrumental to uncover some of the problems that must be addressed to solve the temporal issues related to timing failure detection is DBMS settings. We intend to pursue this work and redesign or complement the mechanisms provided by the TCB for timing failure detection, so they become better suited to support the several classes of timed transactions that we identified as the fundamental ones.

## References

1. J. Gray and A. Reuter, "Transaction Processing: Concepts and Techniques", The Morgan Kauf-mann Series in Data Management Systems, Jim Gray, 1993.
2. K. Ramamritham, "Real-Time Databases", Intl Journal of Distributed and Parallel Databases, 1996.
3. G. Ijzsoyoilu, R. T. Snodgrass, "Temporal and Real-Time Databases: A Survey", IEEE Transactions On Knowledge and Data Engineering, 1995.
4. L. DiPippo, V. Wolfe, "Real-Time Databases", Database Systems Handbook, Multiscience Press, 1997.
5. SIGMOD Record, Special Section on Advances in Real-Time Database Systems, Vol 25, number 1, pp.3-40, 1996.
6. T. Chandra, S. Toueg, Unreliable Failure Detectors for Reliable Distributed Systems, Journal of the ACM, 43(2), 225–267, 1996.
7. L. Dwork, L. Stockmeyer, "Consensus in the Presence of Partial Synchrony", Journal of the ACM, 1988.
8. F. Cristian, C. Fetzer, "The Timed Asynchronous Distributed System Model", IEEE Transactions on Parallel and Distributed Systems, 1999.
9. P. Veríssimo, A. Casimiro, "The Timely Computing Base Model and Architecture", Trans. on Computers - Special Issue on Asynch. Real-Time Systems, 2002.
10. J. Lindström and T. Niklander, Benchmark for Real-time Database Systems for Telecom., VLDB 2001 Intl Workshop on DB in Telecom. II, Rome, Italy, 2001.
11. Oracle Corporation, "Oracle® Database Concepts 10g Release 1 (10.1)", 2003.
12. M. Vieira and H. Madeira, "Recovery and Performance Balance of a COTS DBMS in the Presence of Operator Faults", Intl Performance and Dependability Symposium (jointly organized with DSN-2002), IPDS2002, Bethesda, Maryland, USA, June 2002.
13. Transaction Processing Performance Council, "TPC Benchmark C, Standard Specification, Version 5.4", 2005, available at: http://www.tpc.org/tpcc/.