The UniServer Architecture
and its Use for Database Access

D. Schischkin, G. Weber
Institute for Computer Science
Free University Berlin
Germany
email: weber@inf.fu-berlin.de, schisch@inf.fu-berlin.de

Abstract

A key component of a multi-threaded server is the thread manager, which performs the assignment of incoming requests to server threads. We present a new kind of thread manager called UniServer, implemented in Java. UniServer incorporates three innovations: First it solves a great deal of the involved management tasks by intelligent use of the monitor concept. Second, it allows to choose between the main threading strategies by the appropriate adjustment of only three parameters. And finally, it allows the change of the threading strategy at runtime. As an application and demonstration of the usefulness of UniServer we present a server called MetaDriver based on UniServer, which can increase the performance of applications that use the JDBC Java Database Connectivity standard.

Keywords: multi-threaded server, JDBC, Middleware, 2-tier architecture, 3-tier architecture, CORBA, monitors

1 Introduction

With the increasing practical interest in distributed programming, there is also a growing need for standard server implementation techniques. Distributed object-oriented concepts like CORBA provide on the one hand a uniform way of building complex applications by constructing them out of distributed objects communicating with each other [1, 2]. On the other hand there is the need to implement efficient and scalable server processes that offer these objects. One of the choices that has to be made in implementing a server is the choice of the threading strategy, for example the question, if the server shall be single-threaded or multi-threaded. The decision of the threading strategy is often taken during the design phase of the server, and it is hard wired into the code of the server. Surveys of different threading strategies and proposals for their hard wired implementation can be found in the literature [3, 4]. At the first glance one may think that more sophisticated threading strategies are generally advantageous, but there can be several motivations for servers with a simple threading strategies like single-threaded servers. First, in special applications there may be no use for multi-threading, for example a server that wraps a single-threaded legacy application may itself be implemented single-threaded. Second, a more complicated threading strategy may be rejected because its programming and debugging may turn out to be hard. However, choosing a suboptimal threading strategy leads to performance losses and decreased scalability of a system. Valuable data for determining the optimal threading strategy can possibly only be gained during the runtime of the server and by comparing different threading strategies. Moreover the usage intensity of the server may change in time, and the question whether the server can be adapted to changes in the usage intensity or not is exactly the question of the scalability of the server.

Therefore there is a demand for a general purpose thread manager which can be used in server implementations and is able to support different threading strategies. This paper presents such a thread manager called UniServer [5]. The UniServer is implemented in Java, but the basic techniques may be applied in other languages like C++ as well. In our implementation the UniServer consists of a class library that is used by deriving the application-specific classes, thus giving the UniServer architecture.

The UniServer combines three innovations:

- It uses the monitor concept for a great deal of the management tasks involved in the thread managing.
- It allows the choice between the main threading strategies by adjusting only three different parameters.
- The threading strategy can be changed during the runtime of the server.

As an application and demonstration of the usefulness of UniServer, we present a server called MetaDriver based on UniServer. MetaDriver provides a useful functionality for the JDBC Java Database Connectivity standard, namely it combines the generality of a JDBC bridge with the rich functionality of an application middle tier, allowing for example to pool open database connections in order to increase the performance of JDBC applications [5].

In Section 2 we give an introduction into the concept of a multi-threaded server. A short explanation of the concept of monitors and how UniServer makes use of monitors is given in Section 3. In Section 4 we explain, how the threading strategy of the server depends on the choice of two integer and one boolean parameter. Also changes of the threading strategy at runtime are addressed there.

In Section 5 we motivate the development of the MetaDriver and in Section 6 we give an overview over its features. The results of performance measurements with MetaDriver will then be given in Section 7. In Section 8 we finally give some concluding remarks.

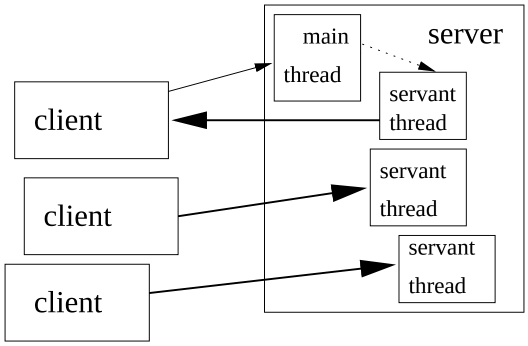

Figure 1: A multi-threaded server (The main thread is assigning a client request to a servant thread).

2 Multi-threaded Servers

In the common model of a multi-threaded server the server has two kinds of threads. There is only one thread of the first kind, called the main thread, and several threads of the second kind, called servant threads. The main thread receives the client requests (usually by listening to a serversocket object) and passes each client request to a servant thread, who takes over the answering of the request (see Figure 1).

In many applications, consecutive requests from a client form a session, for example in the case of database transactions consisting of several client requests to a database [6]. With respect to the session concept there are two alternatives how servant threads can be assigned to client requests:

- The *thread-per-request* strategy, i.e., each client request is assigned to its own servant thread, regardless if it relates to some earlier client request or not.
- In the *thread-per-session* strategy all client requests that form a single session are handled by the same servant thread. Therefore the servant thread is reserved for this client during the session.

The creation of servant threads, also called thread spawning, can be performed in two different ways:

- *request-spawning* For each incoming request a new thread is spawned. Therefore the thread creation overhead leads directly to a delay in answering the request.

- *pre-spawning (thread-pool)* There is a pool of ready-to-use threads for future requests. A thread-pool can be used to recycle servant threads: After completing the handling of a client, a servant thread can be reentered in the thread-pool and used to answer further client requests.

The management of the thread-pool is usually left to the main thread. However, we propose a more elegant solution in the following section.

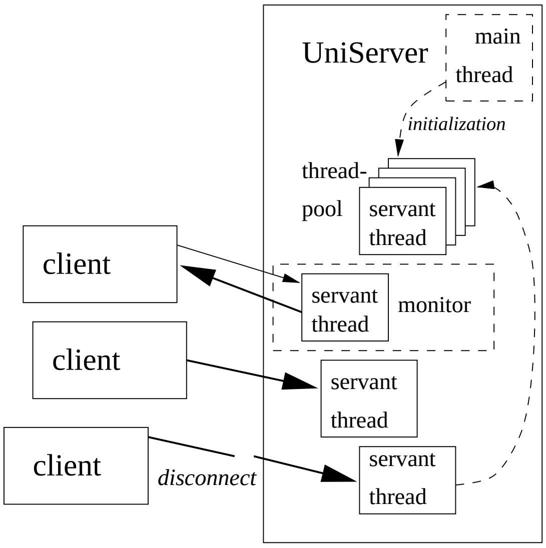

Figure 2: The UniServer.

3 Thread Mangement with Monitors

Monitors [7] are a well known synchronization concept for concurrent programming, that is for example used by the Java language [8, 9]. We will restrict our explanation of the monitor concept only to those very basic features, that are necessary for our application. For our purposes a monitor can be considered as being related to an method of a class, that can be executed only by one thread at the same time. If the method is currently executed by a thread (the thread is said to *possess* the monitor at this time), then all other threads which try to enter the method are blocked and inserted into a waiting queue. If the thread possessing the monitor leaves the method, it releases the monitor and the next thread from the waiting queue can enter the method.

The implementation of UniServer makes use of the surprising result that the concept of monitors can be used to implement a multi-threaded server in which the main thread is no longer involved in the thread management. In common server implementations the main thread receives the client requests by executing a method of an object, let us say the method's name is serversocket.accept(), that blocks the main thread until a new client request comes in. The main thread afterwards passes the client request to a servant thread. Our alternative to this procedure is the following: We make sure that the serversocket.accept() method is related to a monitor. Each servant thread executes as its first action after creation this method. Hence if during the server initialization several servant threads are spawned, these servant threads are inserted into the waiting queue of the monitor for serversocket.accept(). The waiting queue therefore represents the thread-pool (see Figure 2).

The thread management is automatically performed by the monitor implementation, which passes each client request to the next servant thread in the queue. After handling a client each thread returns in the thread-pool by calling the serversocket.accept() method.

4 Threading Strategies

In order to control the number of waiting servant threads in the thread-pool, UniServer possesses two parameters *min* and *max*, which define the minimal and maximal number of waiting threads in the pool. The servant threads themselves take care of this condition. When a thread leaves the pool, he spawns a new thread if less than *min* threads are remaining in the queue. Similarly a servant thread returns to the pool only if less than *max* threads are already waiting, otherwise the thread terminates. However, although in the UniServer the main thread is not involved in the thread management, this thread is still needed for initializing the server by spawning servant threads. In the case $min = 0$ one servant thread must be spawned, in other cases between *min* and *max* servant threads have to be spawned.

The second main achievement of the UniServer is, that he can produce the main threading strategies simply by choosing the appropriate pair of values for *(min, max)*. The following list shows the choices and the resulting threading strategies:

- $0 < min < max$ represents the thread-pool strategy. The thread-pool always contains at least *min* servant threads. However, the number of active servant threads (threads currently handling client requests) is not restricted.

- $0 = min, 1 < max$: The number of servant threads is fixed to the number of threads initially generated by the main thread (for example *max* servant threads). This strategy is therefore a thread-pool strategy, too, but with a fixed number of servant threads.

- $1 \leq min = max$ resembles the request-spawning strategy. Each time a client request comes in, a new servant thread is created. All threads terminate instead of returning into the pool.

- $min = 0, max = 1$ results in the single-threaded strategy. The single thread never spawns a new thread, an therefore no new client request will be handled before the thread returns to the pool.

All these different threading strategies can be used in the thread-per-session or in the thread-per-request mode, depending on whether the implementation of the client request handling in the servant thread suports these modes. Once again it is possible to implement the request handling in such a flexible way, that the choice between these both alternatives can be made by setting a boolean variable *sessionmode*.

In this way the main threading strategies can be selcted by the appropiate choice of three parameters *min, max* and *sessionmode*. Figure 3 gives a hierarchical classification of the different threading strategies.

The fact that the threading strategy of uniserver can be changed by changing the values of these three parameters facilitates the third main feature of UniServer, namely that the threading strategy can be changed at run-time. However in using this feature there has to be taken care, that the changes between the threading strategies are compatible with the paricular application.

In a concrete server implementation project in Java the UniServer architecture is applied by inheriting the servant thread class provided by the Java UniServer and implementing the client request handling for the specific application. The UniServer architecture is in fact used in a pure-Java server called MetaDriver, that demonstrates to the full extend the flexibility

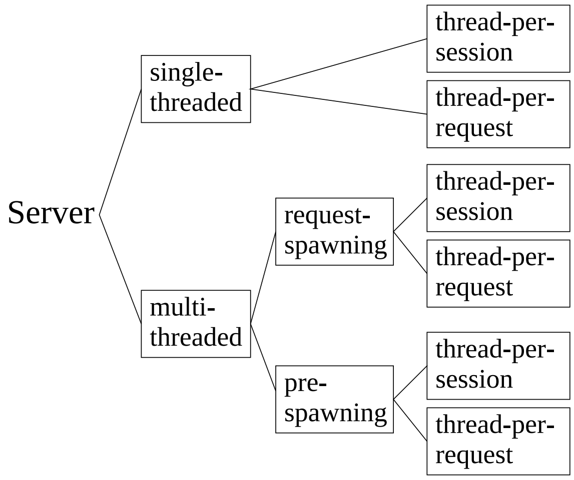

Figure 3: The different threading strategies that can be applied in a server.

of UniServer, but also introduces a number of interesting features specific to its application domain, i.e., database access. The MetaDriver will be introduced in the following chapters.

5 The MetaDriver for JDBC

In order to explain the functionality of the MetaDriver it is convenient to start with some remarks about the Java Database Connectivity standard JDBC. JDBC is Java's vendor independent call-level interface for access to relational databases [10], that is supported by many database vendors. JDBC allows a Java client, for example an applet, to connect directly to a database server on a host for building 2-tier client/server applications. If the clients are applets, this architecture has the disadvantage that due to applet security restrictions the applet must be loaded from the host on which the database server runs. To overcome this problem many database vendors offer products called JDBC bridges or JDBC gateways, that reside on the host from which the applet is loaded and have the only purpose to pass the JDBC communication from the applet to a database server on a different host. Therefore the use of these JDBC bridges turns the 2-tier architecture into a 3-tier architecture with a middle tier of very restricted functionality, and this process is almost transparent to the database application itself. Independent from this problem there are good reasons to implement database applications as 3-tier systems: It is convenient to restrict the use of the relational database backend to standard relational database concepts that have been proven to be portable in practice between the products of different and to deposit all other application into an application middle tier written in Java, instead of coding it in stored procedures proprietary to a certain database product. The Java middle tier can communicate with the Java client with a proprietary protocol. The whole 3-tier application can remain platform-independent in this way. But the use of a proprietary protocol for the client-to-middle-tier communication is a disadvantage with respect to later redesign of the application.

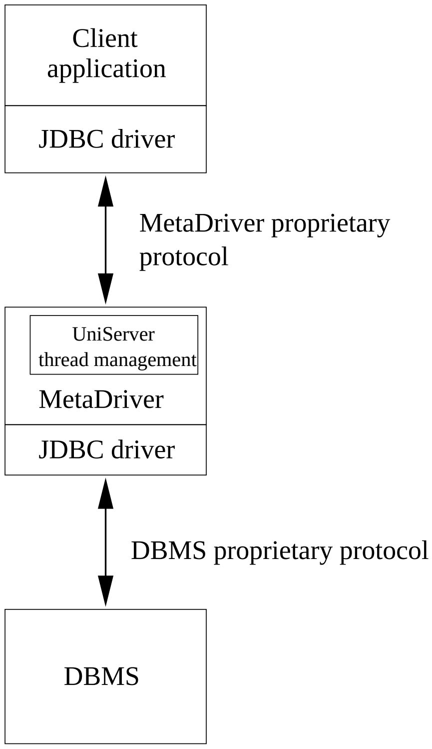

Figure 4: The MetaDriver for database access.

The MetaDriver is a server intended to be used as a middle tier in such an application, but he combines the advantages of a JDBC bridge with those of an application middle tier. In order to achieve this the MetaDriver uses JDBC for the communication with a relational database backend, but also for the communication between the client and MetaDriver. So MetaDriver comes along with its own JDBC Driver, that can be dynamically loaded by the client, like every other JDBC driver (Figure 4).

In the next section we introduce the different features that are provided by the MetaDriver by the use of the UniServer architecture.

6 Features of the MetaDriver

The opening of a connection to a relational database is a time consuming process that can take up to one second. In an application where the database connection must be established after the user has aleady sent his request, the user will recognize a delay in the turnaround time of the application. MetaDriver can keep a number of open database connections, thus increasing the turnaround time for user requests. If MetaDriver is used in the thread-per-session model, one of the preopened database connections is assigned to a client for the duration of the session of this client. This may be necessary for example if clients perform transactions consisting of several user requests. However, for other applications it can be supefluous to reserve database connections for single clients. In this case MetaDriver can be used in the thread-per-request model. Then it is possible for MetaDriver to handle many clients with few open database connections. The JDBC driver, that is loaded into clients connecting to the MetaDriver is 100-percent JDBC compatible, but it also inlcudes advanced features like repeated access on the same result set.

7 Experimental Results

The MetaDriver and the UniServer architecture within the MetaDriver were evaluated in a test environment that comprises of a Sun Sparcstation 10/52 with 96 Mbyte RAM for both the relational database server (Sybase) a well as for the MetaDriver and Sun X Terminals for the Clients.

The first focus of interest was, whether the introduction of a pure-Java middle tier into a former 2-tier relational database application with a Java client can affect the performance in a negative way. Therfore we compared a 2-tier architecture and a 3-tier architecture with MetaDriver as middle tier. The middle tier was executed with the standard JDK Java interpreter without further optimizations. For test purposes a simple query returning a result set with 10 rows was used.

The results of the measurements give rise to a differenciated view. First the complete average turnaround time was 0.04 seconds in the case of the 2-tier architecture and 0.2 seconds for the 3-tier architecture. This shows a decrease of performance due to the introduced middle tier.

On the other hand, the output of the result set needed 0.03 seconds in the case of the 2-tier architecture, but only 0.02 seconds in the case of the 3-tier architecture. This shows, that the JDBC Driver, that is loaded into the client for the MetaDriver provides a more efficient access to the data than the driver used in the 2-tier system.

The process of opening a JDBC connection from the client side takes 0.2 seconds if the client connects to the MetaDriver, but 1 second if the client connects directly to the database. It is here that the MetaDriver shows its clearest advantage.

Summarizing the results of the measurements, it can be said that the additional middle tier introduces a moderate performance loss for operations, that can be already executed almost optimally in a 2-tier architecture, but on the other hand the middle tier can increase the performance decisively in situations, where a middle tier can solve time consuming tasks before the client request actually arrives (as it is the case for the preparation of database connections).

8 Conclusion

Pure-Java middletier server will become commonly used in the future. For the sake of stability and performance, implementations with terse code and little indirection even in each single tier are desirable. With the UniServer architecture an exemplary implementation of multithreaded servers is given, that unveils the very nature of the management tasks in such servers. With MetaDriver a useful product for building database applicatiions in Java was implemented, that shows, how new features can be implemented without further increasing the number of standards that are already in use in the area. Both products exemplify, that software improvements should be modular, i.e.

transparent to other parts of the software, and runtime configurable, and that both goals can be reached without inacceptable performance losses.

References

[1] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. Wiley, NY, NY, 1996.

[2] J. Siegel. *CORBA Fundamentals and Programming*. Wiley, NY, NY, 1996.

[3] C. D. Schmidt and S. Vinoski. Object Interconnections (Comparing Alternative Programming Techniques for Multi-Treaded CORBA Servers, Column 5, 6, 7). *SIGS C++ Report magazine*, 1996.

[4] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. Wiley, NY, NY, 1997.

[5] D. Schischkin and G. Weber. `http://www.inf.fu-berlin.de/~weber/uniserver/` Welcome Page for UniServer and MetaDriver.

[6] J. Gray and A. Reuter. *Transaction Processing, concepts and techniques*. Morgan Kaufmann, San Mateo, Calif, (1993).

[7] C. A. R. Hoare. Monitors: an Operating System Structuring Concept. *CACM*, 17(10):549–557, 1974.

[8] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.

[9] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, Reading, Massachusetts, 1997.

[10] G. Hamilton, R. Cattell, and M. Fisher. *JDBC Database Access with Java*. Addison-Wesley, Reading, Massachusetts, 1996.