Concurrency Control with Java and Relational Databases

Sérgio Soares and Paulo Borba
Informatics Center
Federal University of Pernambuco
Recife, PE, Brazil
{scbs,phmb}@cin.ufpe.br

Abstract

As web–based information systems usually run in concurrent environment, the complexity for implementing and testing those systems is significantly high. Therefore it is useful to have guidelines to introduce concurrency control, avoiding ad hoc control strategies, which may have a negative impact in efficiency and may not guarantee system safety. This paper defines guidelines for concurrency control in web–based information systems implemented in Java with relational databases. In particular, we show where Java and relational database concurrency control mechanisms should be used in order to implement our concurrency control strategy. Additionally, we analyze the performance of different concurrency controls approaches. The main point of the guidelines is to guarantee system correctness without redundant concurrency control, both increasing performance and guaranteeing safety.

1. Introduction

As web–based information systems usually run in concurrent environment, the complexity for implementing and testing those systems is significantly increased. In fact, subtle implementation errors might appear and are usually difficult to detect and locate. This indicates that we need adequate ways to implement concurrent programs. In particular, it is useful to have guidelines to introduce concurrency control, avoiding ad hoc control strategies, which may have a negative impact in efficiency, making redundant controls. Moreover, ad hoc control strategies may not guarantee system safety, adding new race conditions that lead to invalid executions.

In order to avoid those problems, the guidelines presented here guarantee safety, standardizing the control strategies, favoring system extensibility, and improving maintainability. Contrasting with general well–known [11, 14] design patterns for concurrency control, our guidelines are more effective for developing web–based systems because they are tailored to a specific architecture widely used to develop this kind of system. They also assume the use of relational databases for implementing persistence, and consider the databases concurrency control support. We also analyze the performance of several concurrency controls, identifying the most efficient ones.

Since our architecture demands the manipulation of both Java [10] objects and database tables, we must use programming languages concurrency control mechanisms over the objects. Our guidelines use concurrency control mechanisms of both Java and relational databases, which are widely used for implementing web–based information systems. The guidelines indicate where each mechanism shall be used in order to guarantee system correctness without redundant and expensive concurrency control. Therefore programmers know which concurrency controls must be implemented by the database mechanisms and which ones the programming language features must implement. A pessimist approach can leave all the concurrency control to the database management system. However, this approach does not manipulate objects, avoiding several benefits of the object–orientation.

The negative impact on efficiency caused by programming language concurrency control mechanisms, and the need for guidelines that support the correct and efficient introduction of concurrency control, has been reported by several researchers [3, 1, 6]. They are worried in guaranteeing execution efficiency, avoiding unnecessary synchronization in Java concurrent programs. Our approach differs in the sense that it is not general, but tailored to a specific architecture and considers both language and database concurrency control mechanisms. Although specific, a wide range of application can be developed using this architecture. Some examples are shown in the next section.

We applied our guidelines to two real web–based information systems. In the first one, which was already implemented and running, we used the guidelines as a mechanism for checking whether the system controls concurrency cor-

rectly. This helped us to validate the guidelines, since some concurrency control that our guidelines define was used in those systems, such as using transactions only when it is necessary. In the second system, the guidelines were applied during the implementation phase and were responsible up to 10% for the implementation time, which demonstrate the small impact of them.

This paper is structured in five sections. In Section 2 we present a specific architecture to developing web–based information systems. After that, in Section 3, we define the concurrency control guidelines, which aim to guarantee safety and efficiency. We identify the mechanism (database or programming language) that should be used in each situation of the presented architecture. We discuss the performance impact of different concurrency control approaches in Section 4. Related approaches and conclusions are presented in Section 5.

2. A Specific Architecture for Web–based Systems

This specific architecture, which the guidelines are tailored in, tries to provide some requirements that can be considered general to web–based information systems, including the ability to work in a distributed environment, guarantee data integrity and persistence in production environment, possibility to change the persistence mechanism, or use volatile data, for requirements validation, and be easily maintained. In order to attend those requirements a layer architecture and design patterns were used in a Java implementation. This layer architecture aims to separate data management, business rules, communication (distribution), and presentation (user interface) concerns. Such structure prevents tangled code, for example, business code interlacing with data access code, and design patterns allow us to reach greater reuse and extensibility levels.

Figure 1 presents an UML [4] class diagram that illustrates the software architecture and design patterns [13, 9, 5] considered here, for a simple bank example. Accesses to the system are made through a unique entry point, the system facade [9] (`Bank`). The system facade also implements the Singleton [9] design pattern to guarantee that there is just a single instance of this class, which is the instance to be distributed over the user interfaces. The facade is composed of business collections, target of facade methods delegation. Accounts are registered, updated, and queried in a web client implemented using Java Servlets technology. Section 3 provides more details about the classes of the software architecture.

Although the guidelines presented in this paper have been defined for a specific layer architecture and associated design patterns [13], this architecture can be used to implement several kinds of system, but it is especially useful for the development of web–based information systems.

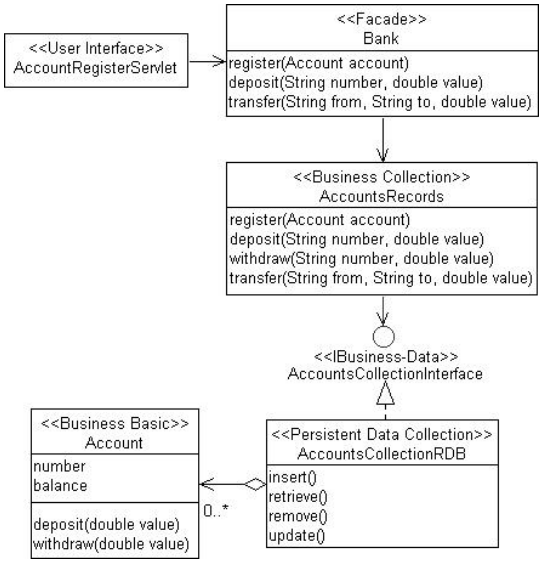

Figure 1. Architecture's class diagram.

3. Concurrency Control Guidelines

In this section we indicate the concurrency control mechanisms (programming language or database) that should be used in each part of the code structured according to the pattern presented in the previous section (see Figure 1). By following those guidelines we can have a safe, non–redundant, and efficient concurrency control. We define specific guidelines for each kind of class of the architecture presented in Section 2. The guidelines prevent naive controls such as synchronizing all the system access methods (facade's methods) or implementing database transactions for all those methods. We also indicate the most commonly applied concurrency control techniques according to our experience acquired through the implementation and analysis of systems that use the same software architecture considered here.

General guidelines, based in basic principles of concurrency control, are needed to avoid concurrent executions of non–atomic methods and methods that read the attributes modified by the non-atomic ones. Examples of this kind of method are the ones that access `long` or `double` attributes, since the Java specification [10] does not guarantee atomic assignments to these types. To avoid such concurrent executions we can use the Java `synchronized` method modifier, which serializes concurrent executions of modified methods in the same object.

3.1. Business Basic Classes

We start to define the specific guidelines with the business basic classes (see Figure 1), which represent system basic objects such as customers and accounts. First of all, we must identify which basic objects might be concurrently accessed.

Identifying Concurrent Access

The data collections classes (see Figure 1) determine if an object may be concurrently accessed because these classes are responsible for storing and retrieving basic objects. Usually, persistent collections that use relational databases create a new instance, with the data retrieved from the database, for each request to search for an object.

Therefore, if two threads, for example, try to retrieve accounts with the same number, the threads will get references to distinct copies of the object stored in the database. This avoids concurrent access on any `Account` instance, since we assume that the concurrency environment allows two or more clients to access the system, but each user access it sequentially.

An alternative to this approach for implementing data collections is to use object caching to guarantee that there will be a single object in memory for each entity stored in the database. This approach is used by some relational and object–oriented database access APIs [16, 17, 7] to prevent inconsistencies that might happen, for example, in the case of a concurrent update of two copies of the same account. In this case, concurrent requests to retrieve an account with the same number will receive a same reference to the object stored in the cache. This allows the concurrent access to basic objects, which is controlled by basic guideline mentioned in the beginning of the section.

Introducing Concurrency Control

After identifying the basic objects that might be concurrently accessed, we must apply the general guidelines to the corresponding classes. In the classes whose objects are not concurrently accessed we must still analyze situations where concurrent updates of copies of the same object should not be allowed. For example, an `Account` class has methods such as `deposit` and `withdraw`, which update the `balance` attribute based on its old value. Concurrent updates of two copies of the same account might take system to an inconsistent state.

Consider a possible execution where the balances of copies of the same `Account` are concurrently modified, where two threads concurrently execute three operations: to request an account to the data collection, to deposit a value in the retrieved object, and to update the object in the data collection. After executing, the account's `balance` information may be inconsistent because each thread works with a different copy of the same object, and the second copy to be updated overwrites the first one.

In order to avoid this problem with concurrent updates of object copies, our guideline suggests the implementation of an update control for each persistent object. This can be done using a timestamp–based technique that adds version information to those objects. Note that this technique is not an implementation of a database algorithm such as timestamp ordering [12]. The idea is to allow object updating just if there is not a newer version of it stored in the database. Otherwise, the operation must be restarted.

3.2. Data Collection Classes

The data collections classes (see Figure 1) are responsible to store and retrieve basic objects. They implement the business–data interfaces, which are abstractions of the system's data management. These interfaces allow us to easily extend the system changing the data management mechanism, by providing implementations of the business–data interfaces to the respectively data management mechanism.

In the persistent data collections we can also find concurrency problems when an object is inserted, updated, or removed from the database. In these cases we must guarantee that database features are properly used; we must include, update, or remove an object inside a database transaction. In this way, we can assume that great part of the concurrency control, regarding to data update, is implemented by the database. Considering this, we must guarantee that the methods of the facade are atomic regarding database access. We can do this by implementing database accesses with a single SQL command, or by implementing transactions in the facade's methods, using the persistence mechanism interface services. To guarantee the atomicity of the data collection methods we must follow the following steps:

- Identify data collection methods that directly or indirectly execute two or more SQL commands;
- Identify business collection methods that call the data collection methods identified in the last step;
- Identify facade methods that call business collection methods identified in the last step;
- The facade methods identified in the last step must use the persistence mechanism interface methods `beginTransaction`, `commitTransaction` and `rollbackTransaction` to implement a transaction on its body.

For example, in the following `update` method of the `Bank` class, the underlined pieces of code are responsible for the transaction mechanism implementation.

```
public class Bank {
  private AccountsRecords accounts;
  private PersistenceMechanismInterface pm;
  public void update(Account account) {
    try {
      pm.beginTransaction();
      accounts.update(account);
      pm.commitTransaction();
    }
    catch (DBTransactionException e) {
      pm.rollbackTransaction();
    }
  } ...
}
```

Assuming that in order to update an account we had to invoke two SQL commands.

3.3. Facade and Business Collection Classes

The business collection classes are responsible for implementing verifications and validations according to the application business logic. Such classes use the business–data interface services to store and retrieving basic objects. The facade class [9] provides an unified interface with all system services, grouping all instances of the business collection classes.

Our guidelines for business collections and facade are mainly concerned with identifying business logic that might lead to race conditions. An example of this kind of rule is verifying, before insert an object in a collection, if there is an object with the same code, or any sort of information that is used in a primary key sense, of the object to be inserted. A concurrent execution that tries to register two objects with the same code may lead the system to an inconsistent state. Automatically allocating a code for each object, for example, using a relational database sequence, or implementing this sequence in the business collection, which eliminates the need for the code verification, can avoid this problem.

For other business verifications that generate situations like the one described in the last paragraph we must prevent the concurrent execution synchronizing the methods responsible for the verification. Therefore we should use the `synchronized` method modifier or the *Concurrency Manager* pattern [15], which, provides an alternative to method synchronization aiming performance increasing. *Concurrency Manager* uses knowledge about the semantics of the methods in order to block only conflicting execution flows, allowing the non–conflicting ones to execute concurrently.

Another concurrency problem occurs when, for example, the facade class implements some operation with multiple calls to methods of business collections. This will indirectly call methods of the data collections, which implies in executing more than one SQL command to the database. As we mentioned in Section 3.2, when an operation cannot be executed with a single SQL command, we must implement transactions in the facade's methods, using the persistence mechanism interface services. The same control is valid when a business collection implement an operation with multiplies calls to data collections methods. One example of this kind of method is the business collection method `transfer`, which might be implemented by a call to `withdraw` followed by a call to `deposit`.

3.4. Commonly Applied Controls

After implementing and analyzing some running web–based systems that use the layer architecture and the design patterns presented here, we can identify which concurrency controls were frequently applied. Some of the systems we implemented and analyzed are a system to manage a telecommunication company's clients, a system for performing on–line exams, a the system for registering health complaints to the health authorities.

The only concurrency control in the facade class, usually, is implementing transactions. In the business collections there are some calls to the concurrency manager methods in order to avoid interference by business verifications. Method synchronization is made only in the `update` method of the data collections classes that implement the timestamp mechanism, and in some business collections methods that do not use the concurrency manager. This happens when the system has few simultaneous users accessing the system and these methods are lightweight, as discussed in the following section. In the basic classes, the concurrency control commonly applied is to implement the timestamp mechanism in the classes whose copies of the same object cannot be concurrently updated. This is our alternative to intuitive controls that tend to synchronize and to implement transactions in all facade methods, which is not efficient nor safety.

4. Performance Evaluation

In this section we present and analyze performance tests with different techniques and concurrency control approaches, including the ones suggested by the guidelines definition. The tests show that some of these approaches are not recommended, validating the advantages of the approach suggested by our guidelines. Moreover, the tests support the decision of which alternative to use for concurrency control, since some of our guidelines offer more than one solution for some problems.

4.1. Performance Tests

We implemented a small customers registering system, which allows customers registering, retrieving, and updat-

ing. The tests execute these three operations in different versions of the system, each one implementing different concurrency controls. We first compare the following versions:

- No control: system without concurrency control;
- Synchronized facade: all facade class methods synchronized;
- Facade with transactions: all facade class methods implementing transactions;
- Suggested Control: applying the defined guidelines, with the concurrency manager in the business collection and the timestamp for the basic class.

Only the synchronized facade approach and the one that applies our guidelines guarantee system correctness, if applied separately. In fact, the former just guarantees system correctness if two copies of a basic object can be concurrently updated, otherwise the approach is not safety. The no control approach is used as a reference to measure the controls impact.

The second test measures the impact of the timestamp mechanism, so we compare the following system versions:

- No control: system without concurrency control;
- Timestamp: timestamp mechanism implemented for the `Customer` basic class;

We also analyze our alternative for method synchronization, comparing the following system versions:

- Synchronization in the Business Collection: business collection `insert` method synchronized;
- Concurrency Manager in the Business Collection: business collection `insert` method using the *Concurrency Manager* pattern;

For each of the different versions we also analyze variations in the controlled methods. Thus, we can measure the impact of different concurrency controls in different types of systems represented by those variations. For example, variations on method execution time were implemented by including a loop that increases the execution time in approximately 100%, which we called heavyweight methods. Another variation in the tests is an increasing in the system workload. We tested the system with workloads between 3 and 600 threads concurrently accessing the system. Therefore we could simulate a situation of extreme concurrency, hardly found in access rates of real systems. For instance, we collected numbers for a considerably used web system that has a reasonable access rate of more than 400 users (not hits) accessing the system in an hour, whereas our experiments execute in few seconds. In our tests there were 12 available connections with the persistence mechanism, which were shared between the threads, without concurrent access to them.

4.2. Performance Analysis

The following paragraphs summarize the tests with the different variations such as system workload and method weight.

General Approaches for Concurrency Control

Figure 2 presents a bar chart that compares the no concurrency control system with the approach that synchronizes all facade methods, the one that implements transactions in all facade methods, and the approach that applies our guidelines.

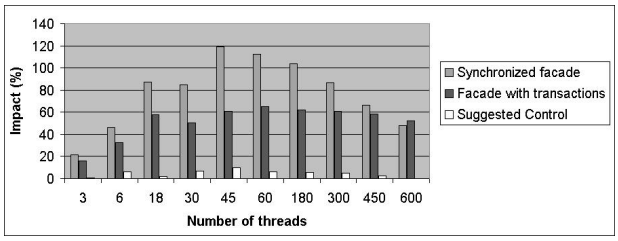

Figure 2. Impact of concurrency controls.

The chart shows that synchronizing all facade methods is a very expensive approach, increasing the execution time up to 120%. We can notice a significant overhead, more than 50%, when implementing transaction for all facade methods. This is a motivation to apply our guidelines for concurrency control, because it indicates exactly which facade methods must implement transactions. We can conclude that our guidelines impact is small, less than 10%, if compared with the others approaches. However, this is a necessary impact to guarantee the system safety.

This chart shows data observed by executing lightweight method. In systems with heavyweight methods the execution time increases 50%, but the difference of execution time between the approaches is lower.

Timestamp

Similar to the results of the previous test, we can see in Figure 3 the performance impact of the timestamp mechanism. The impact with lightweight methods is smaller then with heavyweight methods. This occurs because of the number of connections, which restricts the number of concurrent requests to the persistence mechanism. If the methods of the data collection are heavier they take more time to execute

and keep the connections for a bigger time, delaying the execution of others threads. In these tests we had 12 available connections with the persistence mechanism. Although a considerable impact in the case of heavyweight methods, this is a necessary control to guarantee system correctness. However, increasing the number of available connections adding more connections might decrease this impact.

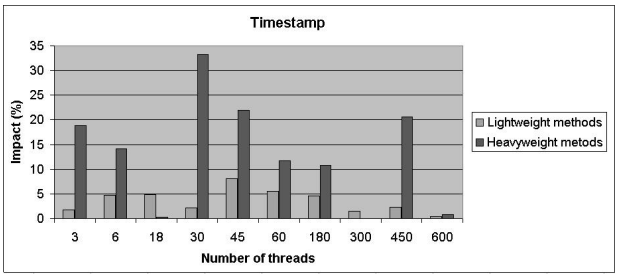

Figure 3. Timestamp mechanism impact.

Concurrency Manager

The last test compares the concurrency manager design pattern and the `synchronized` modifier performance to control the concurrency in the business collection methods. We considered another variation besides the system workload and methods weight: some of the tests lead to race condition, on the same basic object, allowing the evaluation of our solution according to this aspect. This variation is implemented creating threads that try to insert, to update, and to retrieve customers with the same code. We considered this variation because it suggests the use of different alternatives in specific situations, since the concurrency manager uses this information (methods semantics) to synchronize the threads only when it is necessary.

Figure 4 shows the negative performance impact caused by the `synchronized` modifier versus compared with the concurrency manager. When few users are accessing the system concurrently the `synchronized` modified is slightly worst (up to 3%) than concurrency manager. For systems that expect this access rate we suggest the adoption of the `synchronized` modifier solution, since it is simpler to implement and to maintain. For systems that expect higher access rates, we suggest the use of the concurrency manager, which offer a performance going of up to 20%. This gain is bigger (up to 30%) when there is no race condition over the same object, which is typically the case for systems like the one we analyzed. In the tests with heavyweight methods the control impact of the `synchronized` modifier is less than with lightweight methods, but still relevant (10% to 20%).

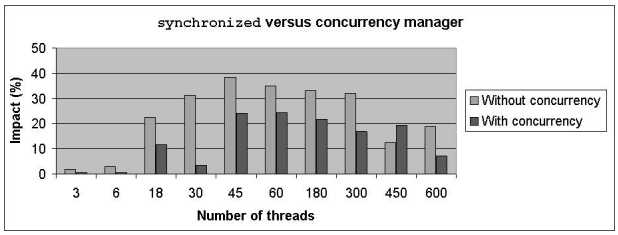

Figure 4. Concurrency Manager versus `synchronized`.

5. Conclusions

Contrasting with general well–know [11, 14] approaches for concurrency control, we defined guidelines for developing web–based systems tailored to a specific architecture widely used to develop this kind of system. We also assume the use of relational databases, which are also widely used for implementing web–based information systems. Our guidelines use the concurrency control mechanism of both Java [10], the programming language, and relational databases. However, those approaches [11, 14] cover a bigger portion of the systems concurrency problems, since it is not tailored to a specific architecture for web–based systems using relational databases, as our guidelines.

The main point of the guidelines is to guarantee system correctness without redundant concurrency control, both increasing performance and guaranteeing safety. The database management systems (DBMS) deal with a big portion of the system concurrency control reducing the need for programming language concurrency control features. This is the case for basic classes, where language features are only necessary if their objects are concurrently accessed. According to our experience this is not the case for many web–based systems developed with JDBC. However, the DBMS do not solve all problems related to concurrency control. Therefore we need to know where to use programming language features in order to avoid redundant controls and their negative performance impact. We have shown that those problems can be solved through our guidelines for concurrency control, preventing losses from 10% to 110% in the execution time, when compared with naive solutions such as synchronizing facade methods.

The concern about avoiding unnecessary concurrent control is topic of many works [3, 1, 6]. One of them [3] uses global data flow analyses to identify what objects with synchronized methods cannot be concurrently accessed in a specific program. An advantage of this approach is that it is completely automatized. In fact, a big portion of our guidelines can also be automatized. Our approach differs from this one because we guide the system implementation

to avoid unnecessary synchronization, also giving guidelines to control race conditions added by business polices. However, contrasting with our approach, this related work doesn't guarantee system safety. It guarantees just the safety of the optimizations made by the analyses, therefore, the system implementation must guarantee safety before applying the analyses (optimizations). We might say that our guidelines and this approach are complementary, since the guidelines can be applied to guarantee the system safety before execute the data flow analyses and the optimization.

In this paper we analyze some alternatives to solve concurrency problems showing, in general, that the concurrency manager is more efficient than the `synchronized` modifier. Moreover, we show the negative impact of the widespread use of the `synchronized` method qualifier, as well as of the unnecessary implementation of transactions in facade methods. The experiments also allow us to state that the impact of the guidelines application in a system is relatively small; mainly when comparing with the other approaches (see Section 4). Another advantage of our proposal being based on specific software architecture, is allowing the exact definition and application of the guidelines, giving better support to programmers. Although specific, the software architecture has been and can be used to implement a wide range of web–based information system. Development productivity is increased because the guidelines precisely indicate the points where concurrency control code must be applied, identifying classes and situations passive of control, and which mechanism should be used to control such problem.

We applied our guidelines in two real web–based information systems in order to validate them. The first system was already implemented and running, and we used the guidelines as a mechanism for checking whether the system controls concurrency correctly. This helped us to validate the guidelines, since some concurrency control that our guidelines define was used in those systems, such as using transactions only when it is necessary. We also found some naive controls, which could be avoided if our guidelines were used. In the second system, which is another implementation of the application implemented by the first system, the guidelines were applied during the implementation phase. We made this to analyze the impact of using the guidelines in the implementation time. In this case the guidelines were responsible to 10% of the implementation time, which shows a small impact of them.

References

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 207–222, November 1999.

[2] I. M. Author. Some related article I wrote. *Some Fine Journal*, 99(7):1–100, January 1999.

[3] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–46, November 1999.

[4] G. Booch, I. Jacobson, and J. Rumbaugh. *Unified Modeling Language – User's Guide*. Addison–Wesley, 1999.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Pattern–Oriented Software Architecture*. John Wiley & Sons, 1996.

[6] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–19. ACM, November 1999.

[7] O. Design. Pse pro for java api user guide, 2001. Avaiable at http://support.odi.com/i/documentation/doc/-psepro/pse-java/doc/pdf/pseug.pdf.

[8] A. N. Expert. *A Book He Wrote*. His Publisher, Erewhon, NC, 1999.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison–Wesley, second edition, 2000.

[11] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, second edition, 1999.

[12] V. Li. Performance models of timestamp-ordering concurrency control algorithms in distributed databases. *IEEE Transactions on Computers*, 36(9):1041–1051, 1987.

[13] T. Massoni, V. Alves, S. Soares, and P. Borba. PDC: Persistent Data Collections pattern. In *First Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, 3th–5th October 2001. To appear in UERJ Magazine: Special Issue on Software Patterns.

[14] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Vol. 2: Patterns for Concurrent and Networked Objects*. Wiley & Sons, 2000.

[15] S. Soares and P. Borba. Concurrency Manager. In *First Latin American Conference on Pattern Languages Programming — SugarLoafPLoP*, Rio de Janeiro, Brazil, 3th–5th October 2001. To appear in UERJ Magazine: Special Issue on Software Patterns.

[16] A. Software. O2 technology user manual: Java relational binding. Version 2.0, July 1997.

[17] Sun Microsystems. The Enterprise Java Beans Specification, October 2000. Avaiable at http://java.sun.com/products/-ejb/docs.html.