Database Replication with MySQL and PostgreSQL

Fabian Mauchle
Software and Systems
University of Applied Sciences Rapperswil, Switzerland
www.hsr.ch/mse

Abstract

Databases are used very often in business applications. Open source database management systems like MySQL and PostgreSQL are becoming more popular especially in smaller or open source applications. But what if the number of users grows too large to be handled by a single database server? Or if the data should be stored on a second server for secure (public) access? The keyword to this is database replication. And what solutions do MySQL and PostgreSQL provide to enable the growth to larger applications?

1. Introduction and Overview

This article about database replication is divided into two parts. In the first part, the terms and definitions about database replication are described as well as the challenges and some possible solutions to them, focusing only on the utilities provided by the database management system. In the second part, two open source database management systems, MySQL and PostgreSQL are analyzed towards their implementation of replication and a sample setup for each system is being tested.

2. Theory

In this section, we will describe the main terminology and concepts of database replication and discuss the challenges and some solutions to them.

2.1. Problem Domain

When using database replication, some data is stored on two or more database systems. This can be either for improving redundancy, performance or offline availability, or any combination of these. While some of these aims could be easily achieved by just entering the data into multiple databases, this should sound silly to everyone and lead to a further aim: transparency. This means that both the user of the database system and the software engineer should be influenced as less as possible. Even if the data is stored on multiple database systems by any method, the basic database principles of Atomicity, Consistency, Isolation, and Durability (ACID) must be maintained.

2.2. Dimensions of Database Replication

Database replication can have many aspects. From synchronization of mobile databases to enterprise class, high performance database clusters. There are some dimensions in which these scenarios differ:

- **Number of databases** This means just the number of databases storing the same data. When there are exactly two of them, sometimes the term synchronization is used.

- **Network link between the databases** This can be divided into two measures. The available bandwidth, which is normally high nowadays, and the much more important round-trip delay.
- **Online accessibility** The online accessibility means if there exists a permanent connection between the databases or not. This is normally the case when using mobile databases.

2.3. Replication Layout and Type

The replication layout denotes the roles each of the database instances have, describing the operations that can be performed on each. One can distinguish between two major layouts which are described in detail in the following sections. There are further methods especially for high performance applications, where a single database operation is performed by group of servers, each performing the operation on a different part of the data, but these are beyond the scope of this document.

The replication type describes how changes are committed to the other database instances, regardless whether it is a master-slave or a multi-master layout. The main goal is to keep all data consistent. While this also applies to a single database instance when multiple, simultaneous actions should be processed, the solution is the transaction. But when there is more than one instance, things are more complex. There are two highly different approaches to this. Which one to use depends highly on the needs of the application. Both have their pros and cons.

All variants of the replication layout and type can be freely combined leading to four different overall implementations:

Synchronous Master-Slave	Asynchronous Master-Slave
Synchronous Multi-Master	Asynchronous Multi-Master

Figure 1: Variants of replication layout and type

2.3.1.Master-Slave Replication

In a master-slave layout, there are two roles to play. One is the master role. This exists normally only once and behaves much like a standalone database. The other role is the slave with as many instances as you like. The main rule is that any operation that changes the data must only be performed on the master database. All changes are then replicated to all slave databases. If the master fails, one of the slaves is promoted the master and takes its role. In order to re-enable the old master, it has to be degraded to a slave, synchronized with the new master and again be promoted the master role.

This layout can either be used for load-balancing on read operations, where only very few write operations occur, or as a stand-by backup scenario where the slave database keeps up to date, but does not perform any operation directly until it is promoted the master role. All these scenarios are relatively easy to perform because write operations occur only on one instance and can therefore not conflict with other operations.

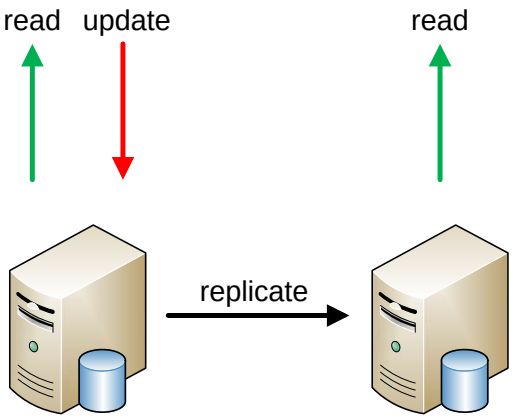

Figure 2: Master-Slave Replication

2.3.2.Multi-Master Replication

Multi-Master layouts are often very complex, but can also perform very good and provide high flexibility. In contrast to the master-slave layout, any operation can be performed on any instance. This means that any change on any instance has to be replicated to any other instance leading to a high demand on synchronization work to keep every instance consistent. Additionally, when changes are performed simultaneously on several instances, conflicts can occur, which to resolve is one of the hardest challenges in database replication. Failing one of the database instances does not influence any of the other databases nor the data, except for the current transaction executed on the failing instance.

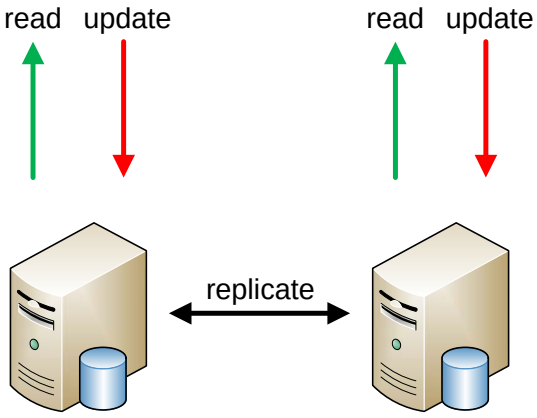

Figure 3: Multi-Master Replication

While previously stating that any operation can be performed on any instance, there is one exception of this: the database schema. Because updating the database schema affects almost the whole database, it is virtually impossible to keep multiple, simultaneous updates synchronized. Therefore updating the schema is often allowed only on one instance, called the schema-master. For schema operations, the layout behaves exactly as a master-slave layout.

The multi-master layout is often used in high performance database clusters or scenarios, where the interconnection between the instances has low bandwidth and high roundtrip time.

2.3.3. Synchronous Replication

Synchronous replication, sometimes also called eager replication, is relatively close to the single database solution using transaction. It fully guarantees all ACID requirements. All operations on one database instance are done simultaneously on all other instances except for read-only operations where only the read-locks are done on the other instances.

Normally, a distributed transaction with a two phase commit is used to achieve this. When a transaction is started on one instance, the same transaction is also started on all other instances and every lock that is achieved on the first instance is also immediately distributed. When there where changes made to data during the transaction, and a commit is called at the end, all modifications are sent to the other database instances. These have to check if the transaction would commit successfully and send this data back to the first instance. If and only if all instances report successful commit behavior, the final commit signal is sent and the changes are applied to all instances. If not, the complete transaction is rolled back.

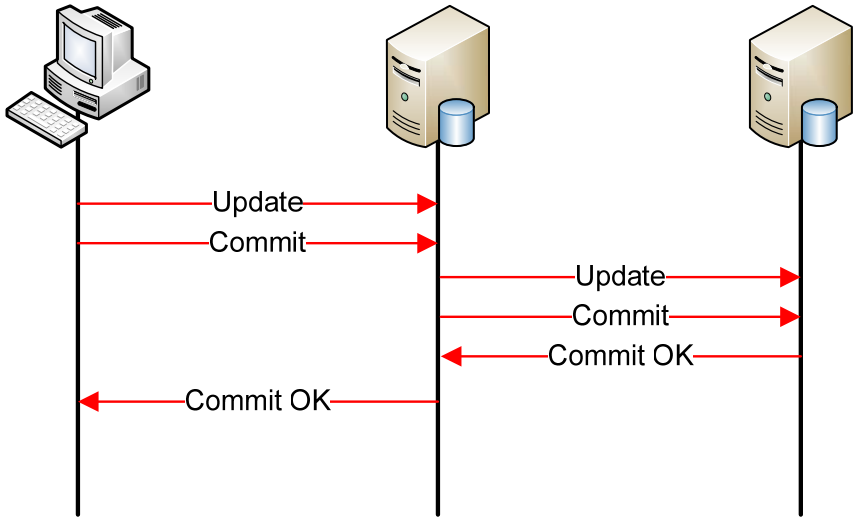

Figure 4: Synchronous Replication

At first, this sounds very nice because all ACID requirements are met and therefore, conflicts cannot occur. But there are disadvantages over this. One thing is that because for every transaction, the different instances have to communicate with each other, and there are several question and answer rounds, this can slow everything down. This effect becomes the more relevant the higher the roundtrip time is, but in the most literature it denoted to be generally slower than asynchronous replication. Another problematic case is the event that one instance fails. When this occurs, no transactions will commit anymore, because the failed instance will never report a successful commit behavior. To recover from this state, the failing instance must be detected and excluded from the replication. As this requires some timeout, it may affect the system performance very hardly. Re-enabling a failed instance or adding a new one is also relatively complex. First, the existing data has to be replicated to the new instance, and then it has to be synchronized to the transactions. Mostly, this requires the whole system to stop accepting new transactions, synchronize the remaining data, and return to the operative state. This can cause some application outage, so that adding new instances have to be done at nonworking times.

2.3.4. Asynchronous Replication

The asynchronous replication, also called lazy replication, is a completely different approach. At first, every database instance is acting for its own, without affecting any other instance. After committing a transaction on one instance the relevant changes, if any, are distributed to the other instances, which then replicate these changes. The distribution can be either immediately or delayed. One can see easily, that this leads to temporary inconsistencies between the databases, and therefore violates the ACID requirements. One can define it as being stressed, when it is guaranteed that a single database client always uses the same server during a session. With that, subsequent writing and reading queries will always fulfill the ACID requirements.

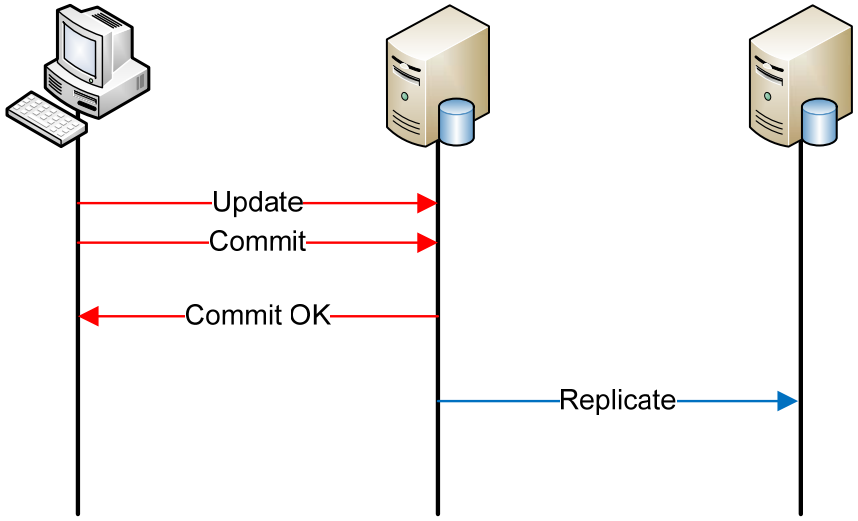

Figure 5: Asynchronous Replication

The way to find the data that has to be replicated and recreating it on the other instances is very open. One possibility is to use the transaction log and replay them, or to replay exactly the issued SQL statements. Another way would be to find the delta in the data blocks and distribute them. Whichever method is used, there is a basic problem if asynchronous replication is used in a multi master layout: conflicts. Whenever the same data is changed in any way at the same time on different servers, before the replication has taken place, the database system does not know, which change it should use (see next section).

Beneath the disadvantage of possible conflicts there are many advantages with this replication type. One mayor advantage is that it uses relatively less network bandwidth, and, it can work well in networks with high latency. Another advantage is that temporary network outages do not necessary lead to a complete unavailability of the database system. In master slave layouts, network outages of several days are allowed, if the application does not absolutely relay on the very newest data. Finally, failing, re-enabling and adding new instances do not affect the rest of the database system in terms of consistency or availability. It may certainly affect performance.

2.4. Conflicts

As already mentioned before, there may be conflicts in some scenarios. The term conflict is used to describe the case, when the same data or dataset is changed on two or more database instances, not knowing each other changes, and thus, not avoiding simultaneous changes. When the data is synchronized later, the system does not know which change is the ‘truth’. Resolving

such conflicts is mostly a very hard work and often requires knowledge of the semantics of the data. Although some conflicts cannot be solved automatically and require an administrator to make a decision.

2.4.1. Conflict Prevention

Basically, it is better to avoid a conflict then to have to resolve one. A synchronous replication avoids conflicts by design. This is also true to master-slave replication because changes occur only on one instance. The last scenario to look at is therefore the asynchronous multi-master replication. While the system itself does not prevent conflicts to happen, the application design can. This could be achieved by dividing the data into partitions and allow only one database instance at a time to change a particular partition. Different partitions can be managed by different servers at different times. Although this is not a perfect solution and is not transparent to the software engineer in the way that he has to exactly know the database infrastructure.

2.4.2. Conflict Resolution

Whenever a conflict cannot be avoided, it has to be resolved. In other words, some mechanism has to decide, which change is the 'truth', or combining all changes to a new 'truth'. In (Keating, 2001) there are three different conflicts mentioned:

At first the classic 'update' conflict, where the same field of a tuple is changed to two different values. A good way to resolve this is to keep the exact timestamp of each change, and use the last one. This method requires precise time synchronization between the database instances. Another is to define priorities to each database instance and use the change made on instance with the highest priority. There are other methods which however require semantic knowledge of the data. If the value is for example a counter, and each action changes the value n+1, the final result would be n+2. Another example is an average value; in this case, the resolved result is the average between the two new values.

The second conflict is the 'uniqueness' conflict. That is, when a column has a uniqueness constraint and two rows with the same value are inserted. Resolving such a conflict is very hard. The only way is to change the constrained value and append a value defined by the database instance. If this is not feasible by the application, the administrator's decision is required.

The last type is the 'delete' conflict. This occurs when a row is changed on one instance, while it is deleted on another instance. As stated by (Keating, 2001) there is no possibility to automatically resolve such conflicts. It is best to design the application such that these conflicts cannot occur by marking rows as deleted instead of really deleting them.

3. Enterprise Usage Scenarios

The replication layouts presented in the previous are mostly theoretical, but for most of them, there exists a realistic usage scenario. The only layout, which normally does not make much sense, is the synchronous master-slave layout. As, by definition, there is only one point where changes can occur, it is not necessary to perform the changes synchronously on other database instances. In the following sections some scenarios are presented where one or more layouts can be used.

3.1. High Performance Cluster

One of the basic reasons to use more than one database server is when a single one would not be able to handle the workload. To resolve this, a multi-master layout is needed. The choice of a synchronous or asynchronous layout depends highly on the application and the structure of the database. If there are a lot of concurrent updates, which may conflict with a high probability, a synchronous layout should be used. This is the usual case for high performance clusters. For such a system, the distance between the servers must be very low. Normally on the same IP subnet, with a high speed connection (>= 1000 Mbit/s preferred). Also, ne number of servers has some influence. The larger this number is, the lower must be the distance (or the faster the network). If this is not the case, see section 3.2 Distributed Databases.

To be able to use an asynchronous layout, which is normally faster than the synchronous, the software engineer has to fulfill to some requirements. He has either to design automatic conflict resolution algorithms (or adapt his design to the existing ones), or make sure that conflicts cannot occur, for example by manual locking or by allowing only certain updates on each server.

3.2. Distributed Databases

The term distributed is here understood as geographic distribution, with high network delays between the sites, so the distance between the servers can be denoted as large. This scenario has to be divided into two sub-forms where updates can be issued only on a single master server, or on multiple sites.

In the first case, an asynchronous master-slave layout is the usually used. This allows read access to the database with normal speed, without influence from any network parameter. The second form is one of the most difficult scenarios to implement. As with high performance usage, the developer has to take care to keep the number of conflicts low or avoid them. Since manual locking slows down the complete system, only the division of the updatable data is a valuable solution. Note also, that due to the use of distributed transactions in combination with high network delay, a synchronous multi-master layout is not an option.

3.3. Backup and Data Warehouse

The last scenarios to consider are a backup database for a failover system or a dedicated data warehouse server. For one of these scenarios an asynchronous master-slave layout should be used.

For backup databases, the advantage is that they are always up to date, without affecting the performance too much. If the primary database fails, it can be immediately replaced. In a data warehouse scenario, complex and often very demanding queries can be issued without affecting the database for its primary use.

4. Implementations

In this section, a closer look at the replication features of MySQL and PostgreSQL and a sample setup. Due to time constraints, only MySQL could be tested in detail. The tests done here aim only to test what features are provided, and how they behave. The tests will not look at replication speed or maximum number of simultaneous operations at all.

When looking at the different replication layouts, one can see that any master-slave or synchronous layout should not provide any difficulties by design. But there is one layout, asynchronous multi-master replication, which is interesting to look at. Therefore in the following tests we will only look at this layout in detail.

4.1. MySQL

The MySQL distribution includes basic replication support. Synchronous multi-master replication, which is called clustering in MySQL is supported using distributed Transactions. For asynchronous replication, MySQL basically only supports a master-slave schema.

	Synchronous	**Asynchronous**
Master-Slave	Use Muli-Master	Supported
Multi-Master	Supported	Not supported *

Table 1: Replication support in MySQL

* In (Maxia, 2006) a way to create a multi master replication schema is proposed, but without conflict resolution. This is basically done by circularly defining master-slave relationships, and avoiding to replicate data, which originated from the same server (data, that traveled the full circle).

The way how asynchronous replication is basically implemented, is by regularly exchanging the binary transaction logs. These contain the same information, including the issued SQL-statements, as normal transaction logs, but in a more efficient way. When the replication occurs on the slave databases, the SQL-statements are basically issued on the local database as normal SQL-statements would be done. Therefore, any stored-procedure or similar actions are executed locally on the slave database.

4.2. PostgreSQL

PostgreSQL itself does not provide any replication support at all. But there are numerous third parity tools which extend PostgreSQL to this feature. In Table 2, some implementations are listed for each replication layout. This list is not intended to be complete. A commercial package including PostgreSQL and Slony is provided by PostgresPlus (EnterpriseDB).

	Synchronous	**Asynchronous**
Master-Slave	Use Multi-Master	Slony (Slony-I) Bucardo (Bucardo)
Multi-Master	Pgpool (pgpool) Pgcluster (PGCluster)	PgReplicator (pgReplicator) Bucardo (Bucardo)

Table 2: Replication support in PostgreSQL

Most asynchronous replication extensions are implemented in a way, that every table which should be replicated is equipped with a per row trigger, which records every change on a row to a special replication table. After that, a daemon running on every replication instance is notified,

and will send the newly changed rows to the other servers. Both PgReplicator and Bucardo provide simple conflict resolution algorithms, manly based on the exact timestamp when changes were made and per server assigned priority if the timestamps are identical.

4.3. Test Setup and Scenarios

For these tests, a system is built out of three servers. These servers run as virtual machines within a single computer using VirtualBox OSE 2.0.4 (VirtualBox). The servers are installed with Ubuntu Server 8.10 (Ubuntu), MySQL 5.2 (MySQL). All virtual servers communicate over a single network segment.

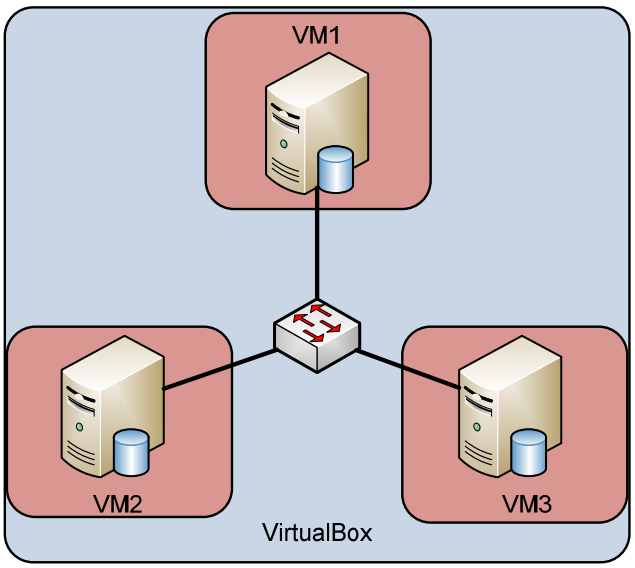

Figure 6: virtual servers

For the tests a simple database schema consisting of a single table is used. This table provides an auto incrementing primary key, a unique text value and an integer used as counter. The SQL DDL statement for MySQL follows:

```
CREATE TABLE test(
      INT id PRIMARY KEY AUTO_INCREMENT,
      CHAR(10) text NOT NULL UNIQUE,
      INT counter NOT NULL);
```

The behavior of the database management systems is tested against the following scenarios. All actions are taken simultaneously on at least two database instances. In order to get reproducible results, replication is suspended while updating the data, and resumed afterwards. After the replication has fully updated all servers (as far as eventual errors allow this) the state of each server can be interpreted.

1. Insertion of different values using the auto increment for the primary key
2. Incrementing the counter value on the same tuple (using "`set counter = counter + 1`")
3. Changing the text value on the same tuple to different values.
4. Deleting a tuple based on its text value and changing the text value at the same time.

The first action tests for the consistence of the auto increment features. The actions 2 and 3 test for update- and unique-conflicts. The last action tests for the delete conflict. For details about these conflicts see section 2.4.2.

4.4. Results

For MySQL, the asynchronous multi-master schema provided by (Maxia, 2006) has been set up. For details how to install and configure such a setup see the link provided in the reference. The following results could be observed when executing the actions described above:

1. The auto increment values are handled correctly. When using the `auto_increment_increment` and `auto_increment_offset` values described in (Maxia, 2006) correctly, there should not be any problem.
2. As long as the increments (or decrements) are done by the provided statement above, the changes are replicated correctly and the data remains consistent on all instances. However, when simultaneously setting the column directly to different values, the databases left inconsistent. Due to the way replication is done in MySQL, such conflicts will not be detected, and the databases will remain inconsistent.
3. When a unique conflict occurs during replication, it causes an error at the time when the slave databases try to replay the original SQL statements. Any error that occurs during this phase will block the replication completely. From this time on, no changes will be replicated anymore, until the conflict is solved and the replication is restarted manually. No changes made meanwhile will be lost and can be replicated correctly afterwards.
4. Since only the original SQL statements are replayed on the slave servers, when executing a delete statement, the originally affected rows are already changed, and the statement will doo nothing. Due to the lack any check for the correct result, no conflict will be detected, and the databases will remain inconsistent.

5. Conclusion

Using open source database management systems with replication for large scale applications is definitely possible but requires careful analysis what is needed by the application and planning of the complete system. Eventually, the support for replicated databases should already be considered when designing the application. It should also be noted, that with an asynchronous multi-master schema, the ACID requirements are violated for some period of time while the replication takes place.

When using the layout described by (Maxia, 2006), great care has to be taken when writing update statements. Such systems have to be observed very closely to detect inconsistencies and replication locks (see section 4.4).

Bibliography

Bucardo. (n.d.). *Bucardo*. Retrieved 2008, from www.bucardo.org

EnterpriseDB. (n.d.). *Postgres Plus*. Retrieved 2008, from www.enterprisedb.com

Keating, B. (2001). *Challenges Involved in Multimaster Replication*. Retrieved 2008, from www.dbspecialists.com/files/presentations/mm_replication.html

Maxia, G. (2006, 20 4). *Advanced MySQL Replication*. Retrieved 2008, from www.onlamp.com/pub/a/onlamp/2006/04/20/advanced-mysql-replication.html

Momjian, B. (2008, 12). *Replication Solutions.* Retrieved 2008, from http://momjian.us/main/writings/pgsql/replication.pdf

MySQL. (n.d.). *Mysql Documentation*. Retrieved 11 17, 2008, from http://dev.mysql.com/doc/

PGCluster. (n.d.). *PGCluster*. Retrieved 2008, from http://pgcluster.projects.postgresql.org

pgpool. (n.d.). *pgpool-II*. Retrieved 2008, from http://pgpool.projects.postgresql.org

pgReplicator. (n.d.). *PostgreSQL Replicator*. Retrieved 2008, from http://pgreplicator.sourceforge.net

Postgres-R. (n.d.). *Terms and Definitons for Database Replication*. Retrieved 2008, from www.postgres-r.org/documentation/terms

Slony-I. (n.d.). *Slony-I*. Retrieved 2008, from www.slony.info

Ubuntu. (n.d.). *Ubuntu*. Retrieved 2008, from www.ubuntu.com

VirtualBox. (n.d.). *VirtualBox*. Retrieved 2008, from www.virtualbox.de

Wikipedia. (2008, 6 27). *Optimistic replication*. Retrieved 11 17, 2008, from http://en.wikipedia.org/wiki/Lazy_replication

Wikipedia. (2008, 11 7). *Replication (computer science)*. Retrieved 11 17, 2008, from http://en.wikipedia.org/wiki/Replication_(computer_science)