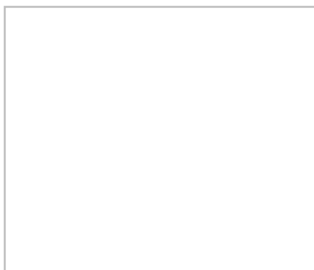
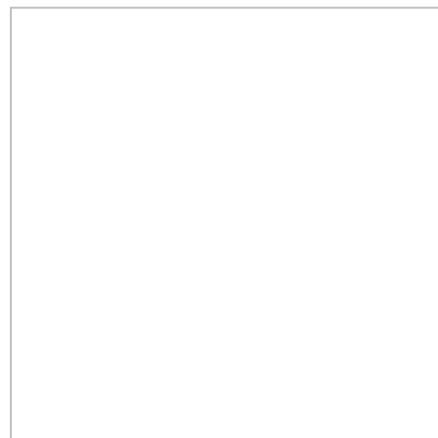
Scalability & Performance Draft
eFortis Relationship Management Suite
Fortis IT Consulting
April / 2007





Altamiro Figueiredo dos Santos
Themis Vassiliadis

eFORTIS Relationship Management Suite

INDEX

1. INTRODUCTION 3
  1.1. The Purpose of Database 3
  1.2. Different Type of Databases 3
  1.3. Relational Databases 4
  1.4. Transactional Databases 5
  1.5. Accessing Databases 5
  1.6. Accessing Databases Without an API 6
  1.7. Performance Issues and Non-Issues 7
  1.8. Performance Solutions 7
2. EFORTIS DATABASE ABSTRACT LAYER 9
  2.1. Looking for Less I/O Overhead 9
  2.2. The Poor Control of PHP Persistent Database Connections 10
  2.3. Working With Non Persistent Connections 11
  2.4. Establishing a connection pooling 12
3. EFORTIS DBSOCK 13
  3.1. Why eFortis DBSock 14
  3.2. eFortis DBSock Parameters 14
  3.3. Flexibility 15
  3.4. Topology – Advantages & Disadvantages 16

1. INTRODUCTION

These days it's rare web-based application that doesn't access some sort of database. This article examines in detail the different tools available for developing web based applications and the database access options a developer has with each.

1.1. The Purpose of Database

First of all, let's take a look at the meaning of database? The term database originated within the computing discipline. A computer database is a structured collection of records or data that is stored in a computer system so that a computer program or person using a query language can consult it to answer queries. The records retrieved in answer to queries are information that can be used to make decisions.

For the purposes of this discussion, a database is an organized store of information that an application can access. That's a very general definition, and there is probably a more specific and perhaps more accurate definition somewhere on the net, but that's what "database" means in this discussion.

1.2. Different Type of Databases

From all types of databases, today two are relevant for developer:

In hierarchical model, data are organized into a tree-like structure. The structure allows repeating information using parent/child relationships: each parent can have many children but each child only has one parent. All attributes of a specific record are listed under an entity type. In a database, an entity type is the equivalent of a table; each individual record is represented as a row and an attribute as a column. Entity types are related to each other using 1: N mapping, also known as one-to-many relationships.

A relational database is a database that conforms to the relational model, and could also be defined as a set of relations or a database built in an RDBMS.

A relational database management system (RDBMS) is a system that manages data using the relational model. Frequently, the term "RDBMS" is inaccurately used as a generic label for the relational database concept. Most current RDBMSs deviate significantly from the relational model and are more accurately called SQL database management products. Many SQL-based DBMS vendors have all but dropped the word relational from their marketing materials and technical documentation. See current usage for an explanation of the requirements for a DBMS to fully support the relational model.

1.3. Relational Databases

There are actually different types of relational databases.

A relational database is a database that conforms to the relational model, and could also be defined as a set of relations or a database built in an RDBMS.

A relational database management system (RDBMS) is a system that manages data using the relational model. Frequently, the term "RDBMS" is inaccurately used as a generic label for the relational database concept. Most current RDBMSs (for example: MySQL, PostgreSQL, Oracle, Microsoft SQL Server, Ingres) deviate significantly from the relational model and are more accurately called SQL database management products.

Some relational databases like SQL Server, Oracle, MySQL and several others, provide a high level of query language. But there are also relational databases like BerkeleyDB and dBase that didn't provide a high level of query language, providing the access on tables directly on physical files.

Three key terms are used extensively in relational database models: relations, attributes, and domains. A relation is a table with columns and rows. The named columns of the relation are called attributes, and the domain is the set of values the attributes are allowed to take.

The basic data structure of the relational model is the table, where information about a particular entity (say, an employee) is represented in columns and rows (also called tuples). Thus, the "relation" in "relational database" refers to the various tables in the

database; a relation is a set of tuples. The columns enumerate the various attributes of the entity (the employee's name, address or phone number, for example), and a row is an actual instance of the entity (a specific employee) that is represented by the relation. As a result, each tuple of the employee table represents various attributes of a single employee.

All relations (and, thus, tables) in a relational database have to adhere to some basic rules to qualify as relations. First, the ordering of columns is immaterial in a table. Second, there can't be identical tuples or rows in a table. And third, each tuple will contain a single value for each of its attributes i.e. each tuple has an atomic value.

A relational database contains multiple tables, each similar to the one in the "flat" database model. One of the strengths of the relational model is that, in principle, any value occurring in two different records (belonging to the same table or to different tables), implies a relationship among those two records. Yet, in order to enforce explicit integrity constraints, relationships between records in tables can also be defined explicitly, by identifying or non-identifying parent-child relationships characterized by assigning cardinality (1:1, (0) 1:M, M:M). Tables can also have a designated single attribute or a set of attributes that can act as a "key", which can be used to uniquely identify each tuple in the table.

1.4. Transactional Databases

Working with RDBMS, there are two modalities: transactional and non-transactional. A database transaction is a unit of interaction with a database management system or similar system that is treated in a coherent and reliable way independent of other transactions. In general, a database transaction must be atomic, meaning that it must be either entirely completed or aborted. Ideally, a database system will guarantee the properties of Atomicity, Consistency, Isolation and Durability (ACID) for each transaction. In practice, these properties are often relaxed somewhat to provide better performance.

1.5. Accessing Databases

There are several programming languages for writing an application like PHP, Perl, Python, C/C++ and Java. Each one has a way to make their access on databases.

All of then provide filesystem access methods. Some programming languages provide modules to make an interface between database and user (Php, Pear, Python). In others cases, they have several APIs to help then make the access on database.

Usually an API provide access to several databases, they are a bridge, providing commons interfaces between them (ODBC, JDBC).

To make an access on databases we do not need create an application, maybe we just need make some reports, for example. In that cases there are several tools provided by vendors to make that task as easier as possible.

Oracle offers a complete suite for report, management or just simple querys. SQL Server also offers a complete suite to help all of layers on using. There are tools to management, to make simple or complex report and API's for possible integrations with other platforms.

1.6. Accessing Databases Without an API

Application server and web-based application in generally use shared libraries (DLL and SO) or COM (Common Object Model) object methods to establish the communication to the DB. This model is common called connection by API.

The major issue is that API's only exist for a limited set of platforms. Some DBs only have API's for Microsoft OS on Intel-based hardware. There are others DBs that just have it for a few brands of Unixes. Operating system like MacOS, PowerPC, NT for Alpha and others were unattended.

What about the use of an API in a particular language? Crossing language and platform the covered possibilities will be choked.

To bypass it we should work with distributed processing, but this is a generic solution. RPC (Remote Procedure Call), Corba, DCOM (Distributed COM) and Java RMI (Remote Method Invocation) allow a program to invoke methods on other computers and get the results back locally. These are useful if have another computer somewhere to farm out the database work to. C and C++ support RPC and Corba on almost any platform and DCOM on some platforms (including some Unixes, but it's really expensive). Java supports RMI. Perl and Python

support some of these methods outright, and may be extended with modules to support more.

1.7. Performance Issues and Non-Issues

The First version of RDBMS was projected to work with just a few connections procedure. There were effort was made to support procedures like commit/rollback/session/privileges, and the procedure of logon was put aside, with a lower priority.

Some very old RDBMS leave the responsibility to access the physical files of the databases to the programmer. Those kinds of RDBMS moor the programmer to use a lower level of query commands, consequently forcing the programmer to use several others querys to compensate the limitations.

There is databases also designed thinking about performance access. For those databases, the important thing is speed, fast communication between the clients and the server. Procedures like update were put in a lower priority.

Nowadays, RDBMS are made for boot proposes. The vendors leave a whole configuration and tools to help programmers or managers project their databases for a particularly use. There are indices, summary views to turn this task as easier as possible.

1.8. Performance Solutions

For critical missions, RDBMS have several performance issues associated with access between RDBMS's and web-based applications, to fix it, several solutions were developed.

The first one is to tune the database to make logins process faster. This solution should appear a nice idea, but certainly other process of the server will be affected.

A second solution should be to attach the web-server or application server directly to the RDBMS. The most databases don’t allow it and others have limitations anyway. In the other hand this solution should be very expensive.

The third and most popular solution is to maintain pools of RDBMS connections and loan them out to needy executables. Web application servers do this by logging in when the application server starts up and allowing application threads to access the database through the open connection. When run as an Apache module, PHP has the same capability. Persistent CGI's written using the FastCGI library can be written to do the same thing. EFortis DBSock provides a solution along these lines for accessing many different databases.

Other examples exist, but those come to mind immediately. This solution suffers with regard to dynamic scalability. As the demand on an application increases, more immediately available database connections are needed. Otherwise, the application stalls while new connections start up or old ones finish their current job. Application load must be predictable ahead of time for this solution to be flawlessly effective.

2. EFORTIS DATABASE ABSTRACT LAYER

Originally developed over PHP PEAR::DB framework, the architecture has changed to a new model.

PEAR::DB is simply a container class with some static methods for creating DB objects as well as some utility functions common to all parts of DB.

The common functions are a mix of internal functions and specific databases API encapsulated in a default syntax, independently of the client in use.

2.1. Looking for Less I/O Overhead

PEAR::DB class is covered by 16 files witch represent a great overhead to load for each request. Thinking about that was developed a new class embedding functions existing in PEAR::DB. Some piece of the original code was copied in a single file and new functions were rewrite without depreciated and unusual futures.

The main class is responsible to manager connections, transactions, fetch cursors and report errors message.

Additional futures like a complete log and tracking for connections errors and query problems was build inside.

As PEAR::DB, several control functions and facilities was developed:

- autoCommit: Enables or disables automatic commits;
- autoExecute: Automatically generates an insert or update query and call prepare() and execute() with it
- autoPrepare: Automatically generates an insert or update query and pass it to prepare();
- nextID: Returns the next free id in a sequence;
- numRows: Determines the number of rows in a query result;
- affectedRows: Determines the number of rows affected by a data maniuplation query;
- autoFree: Automatically close an opened cursor and free allocated memory;

2.2. The Poor Control of PHP Persistent Database Connections

Persistent connections are links that do not close when the execution of script ends. When a persistent connection is requested, PHP checks if there's already an identical persistent connection (that remained open from earlier) - and if it exists, it uses it. If it does not exist, it creates the link. An 'identical' connection is a connection that was opened to the same host, with the same username and the same password (where applicable).

People who aren't thoroughly familiar with the way web servers work and distribute the load may mistake persistent connects for what they're not. In particular, they do not give an ability to open 'user sessions' on the same link, they do not give an ability to build up a transaction efficiently, and they don't do a whole lot of other things. In fact, to be extremely clear about the subject, persistent connections don't give any functionality that wasn't possible with their non-persistent brothers.

This has to do with the way web servers work. There are three ways in which the web server can utilize PHP to generate web pages.

The first method is to use PHP as a CGI "wrapper". When run this way, an instance of the PHP interpreter is created and destroyed for every page request (for a PHP page) to the web server. Because it is destroyed after every request, any resources that it acquires (such as a link to an SQL database server) are closed when it is destroyed. In this case, no gain is giving using persistent connections -- they simply don't persist.

The second, and most popular, method is to run PHP as a module in a multi-process web server, which currently only includes Apache. A multi-process server typically has one process (the parent) which coordinates a set of processes (its children) actually do the work of serving up web pages. When a request comes in from a client, it is handed off to one of the children that are not already serving another client. This means that when the same client makes a second request to the server, it may be served by a different child process than the first time. When opening a persistent connection, every following page requesting SQL services can reuse the same established connection to the SQL server.

The last method is to use PHP as a plug-in for a multithreaded web server. Currently PHP 4 has support for ISAPI, WSAPI, and NSAPI (on Windows), which all allow PHP to be used as a plug-in on

multithreaded servers like Netscape FastTrack (iPlanet), Microsoft's Internet Information Server (IIS), IBM Websphere and O'Reilly's WebSite Pro. The behavior is essentially the same as for the multiprocess model described before. Note that SAPI support is not available in PHP 3.

However, that this can have some drawbacks if using a database with connection limits that are exceeded by persistent child connections. For example, if the database has a limit of 16 simultaneous connections, and in the course of a busy server session, 17 child threads attempt to connect, one will not be able to.

Another very serious matter is that PHP can not control the upper limit of connections, nether the total amount connections independent of the user in use. In other words, there are no limits for the number of connections, so fatally the application will down when a big number of process will be running.

2.3. Working With Non Persistent Connections

Work with non-persistent connections has advantages and disadvantage.

The advantages are:

- Low memory allocation;
- High security;
- Easy transaction control;
- All opened cursors are forced to be closed, so no process remain live on the server;
- No mid layer through client and database;

The obvious disadvantage is the latency during the process of establishment connection. Many data are changed, user must be validated, grants are conceded and finally memory is allocated in the both sides (client and server). This result in a high process overhead and consequently in a process, bandwidth and memory excessive consume.

For many cases, non-persistent connections should be used specially for low volume transactions.

2.4. Establishing a connection pooling

A simple but efficient solution to cover non-persistent connections is to establish one or more non-interrupt connection between the client and server and share these connections for all demand. This is common called Connection Pooling.

The advantages are:

- Low memory and process overhead;
- Control centralized;
- Fast access to database;
- Cache storage;
- Load balance and failover possibilities;

You probably won't use a Connection Pooling if your application doesn't need a constantly access to the database or in other words only for low demand application.

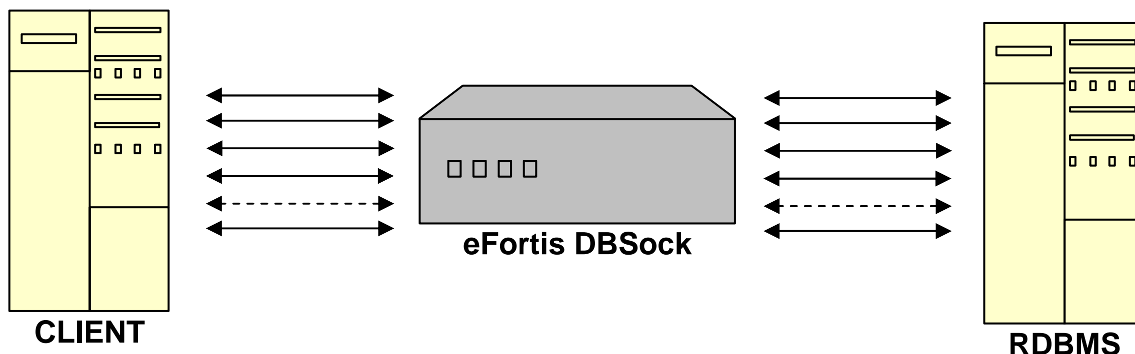
3. EFORTIS DBSOCK

EFortis DBSock is a persistent database connection pooling, failover and load balancing system for Windows, Unix and Linux plataform.

EFortis DBSock is ideal for:

- speeding up database-driven web-based applications;
- enhancing the scalability of database-driven web-based applications;
- distributing access to replicated databases;
- throttling database access;
- accessing databases from unsupported platforms;
- migrating applications from one database to another;
- use and security log;

EFortis DBSock provides another solution for accessing many different databases from software on unsupported platforms. The eFortis DBSock API Is compiled in Java which can run in almost all type of OS and hardware. The server can be run on a different machine where it communicates over the network with the client on one hand and the database on the other. EFortis DBSock provides a service similar to the distributed processing solutions above, only through a highly specialized interface.



EFortis DBSock supports all databases supported by Java JDBC driver.

3.1. Why eFortis DBSock

The main difference between eFortis DBSock and a traditional connection pooling is that eFortis DBSock can run in one or more machines and essentially in different machine from the application. So the use resource should be shared living the application machine working in a soft way, just for the original purpose: to serve the application.

Additional motivation for eFortis DBSock stemmed from the following challenges:

- Oracle API it is really hard to understand and use;
- Oracle databases can take a relatively long time to log into, reducing the snappiness of transient programs such as CGI's;
- Transient programs need to be small and statically linked to be efficient. The Oracle libraries are large and one of them (libclntsh.so) is only available as a shared object library on Linux;
- It would be nice to be able to run a pool of web-servers using heterogeneous hardware and operating systems against on databases, and not be limited to specific platforms and language;
- Failover, load balance, complete tracking and log are futures hard to find in just one solution;

3.2. eFortis DBSock Parameters

Several parameters should be configured for best performance, security and availability.

In the list below these parameters are explained:

- DefaultReadOnly: Sets defaultReadonly property;
- DefaultTransactionIsolation: Sets the default transaction isolation state for returned connections;
- InitialSize: Sets the initial size of the connection pool;
- LoginTimeout: Set the login timeout (in seconds) for connecting to the database;
- LogWriter: Sets the log writer being used by this data source;
- MaxActive: Sets the maximum number of active connections that can be allocated at the same time;

- MaxIdle: Sets the maximum number of connections that can remail idle in the pool;
- MaxWait: Sets the maxWait property;
- MinIdle: Sets the minimum number of idle connections in the pool;
- Username: The username to connect;
- Password: The password to connect;
- ValidationQuery: Query used to check the connection state;
- PrimaryServer: Define the address of the primary server;
- SecondaryServer: Define the address of the secondary server if the primary fail (for failover mode) or the second server (for load balance mode);
- TertiaryServer: Define the address of the tertiary server if the primary and secondary fails (for failover mode) or the third server (for load balance mode);
- PoolMode: "0" none, "1" for failover mode, "2" for load balance mode;

3.3. Flexibility

DBSock is written in Java, compatibility with JRE 5 or earlier.

The API functionality is like a common SO services. There are 2 scripts (startup and shutdown), enabling an easy management of the service.

Different of usually service, DBSock doesn't have an "ini" file or a ".conf" file, the manager can make the whole configuration through a XML file. That XML must be set on the startup service.

The Manager can also make a detailed accompaniment of the service through a large log engine. Through the XML configuration file, the manager can choose between the most detailed log and a small log, storing just exceptions.

The DBSock service can use any port that the manager wants, also, the manager can startup more than one instance of DBSock in a different port (With different XML configuration files).

Is not the propose of this document to explain how to design a java application running parameters that turn the execution as faster as possible, we shall focus on how the DBSock works and how it can

offer a fast and reliable service. There are two ways to configure the service to work.

First, DBSock can be programmed to start with a few connections in memory, and increase it according to demand of application that consume the service. This configuration it's recommended when the application consume is variable.

Second, DBSock can be programmed to start with several connections already in memory, this configuration it's recommended when the application consume is constant.

In boot ways the user can configure how the polling will work, It can hold all connection already created in memory ready to use or those connection after few minutes of inactivate can be closed (time configured on the XML configuration file), that way deallocating the connection on the database.

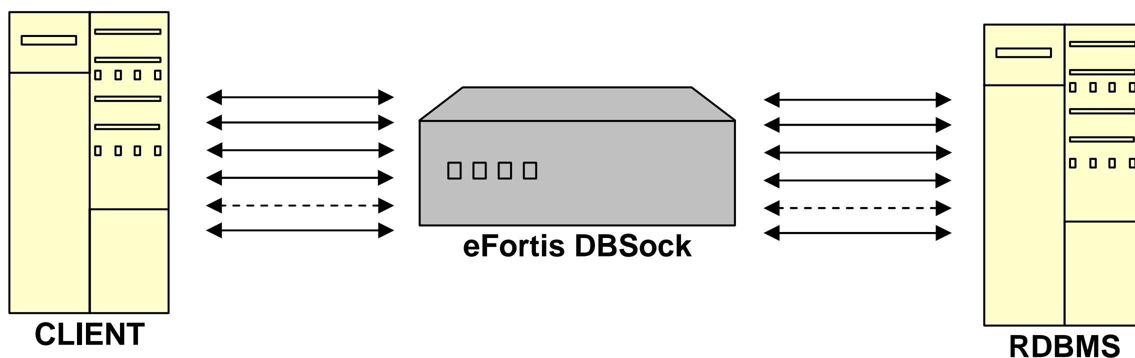
As almost all applications developed in Java, the compatibility, scalability, reliability and platform freedom are the best goals of this solution.

3.4. Topology – Advantages & Disadvantages

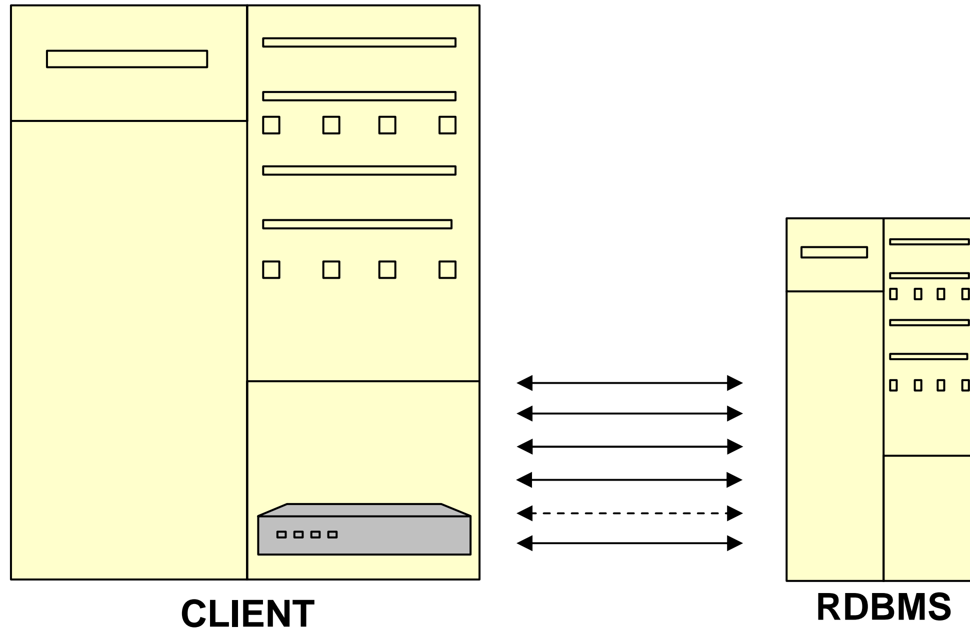
Several topology of DBSock should be used between the application and database.

On the next pages we will show some of then and will discuss the advantages and when should be used.

The classical topology is to use the DBSock as a proxy between the client server and database server as follow:

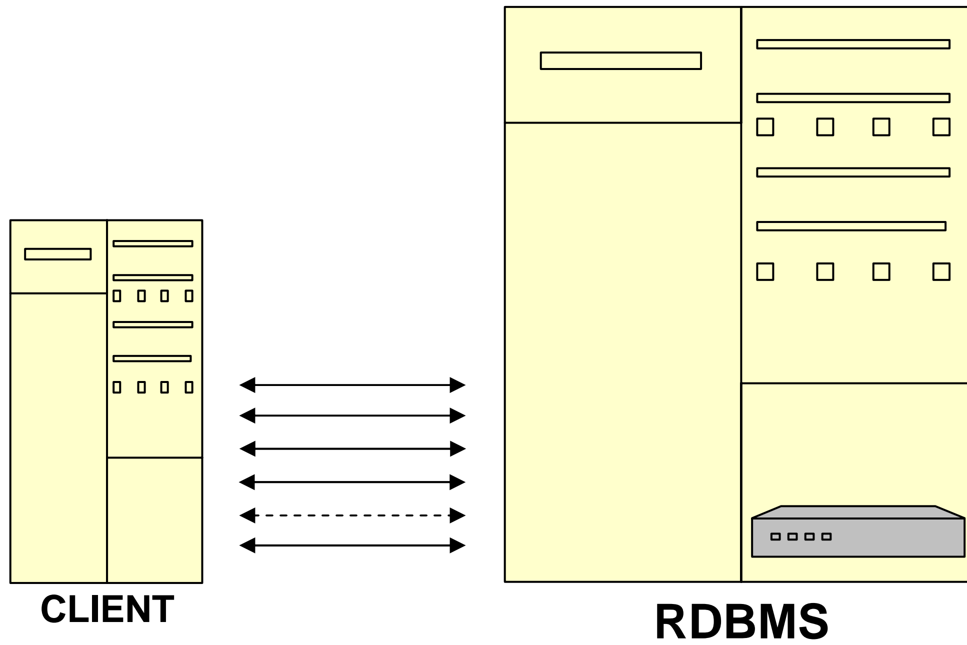


In almost situations this format will attend with performance and availability. But for cost reasons DBSock should be used running at same server of one side (client or database).



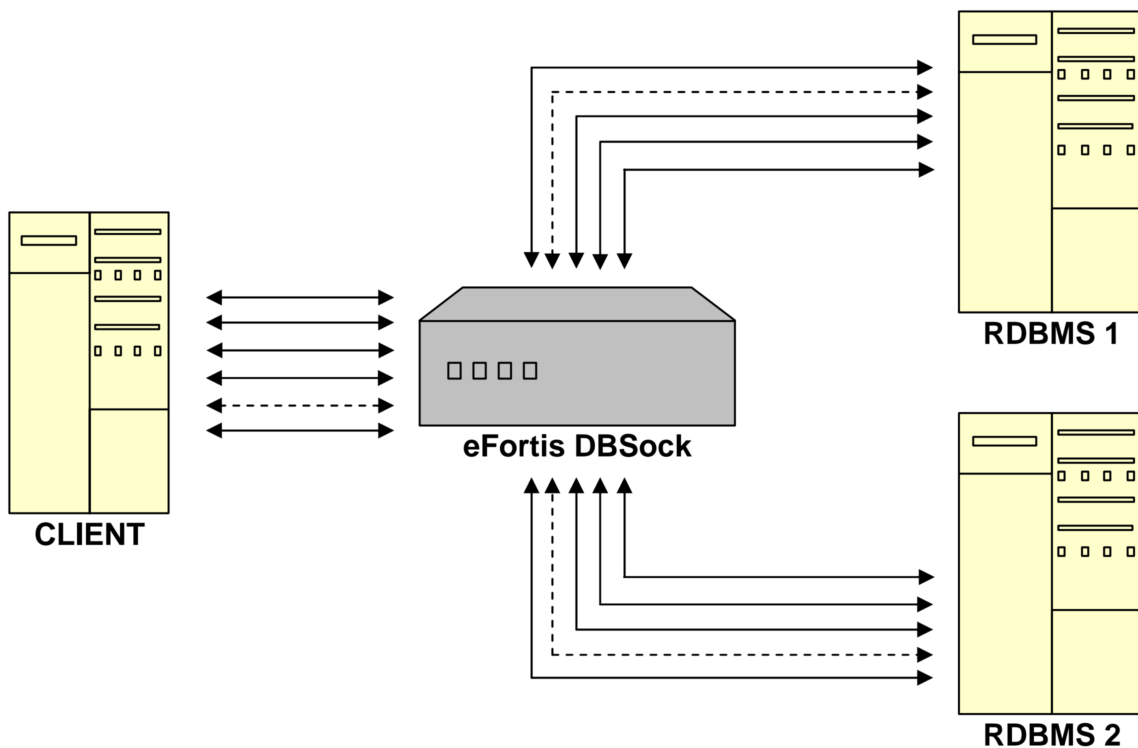
Running together the client server win speedy to connect to DBSock as the service run at the same server. The network bandwidth is less than the other topology. DBSock has a low demand of process and this topology should be extremely useful, specially if you are looking for cost/benefit.

Like the client side, the DBSock service should run at the database server side.



These two topologies apparently are very similar except that the client will must connect several times to the DBSock. In fact it will run like a traditional client server connection life, with DBSock running like a database listener.

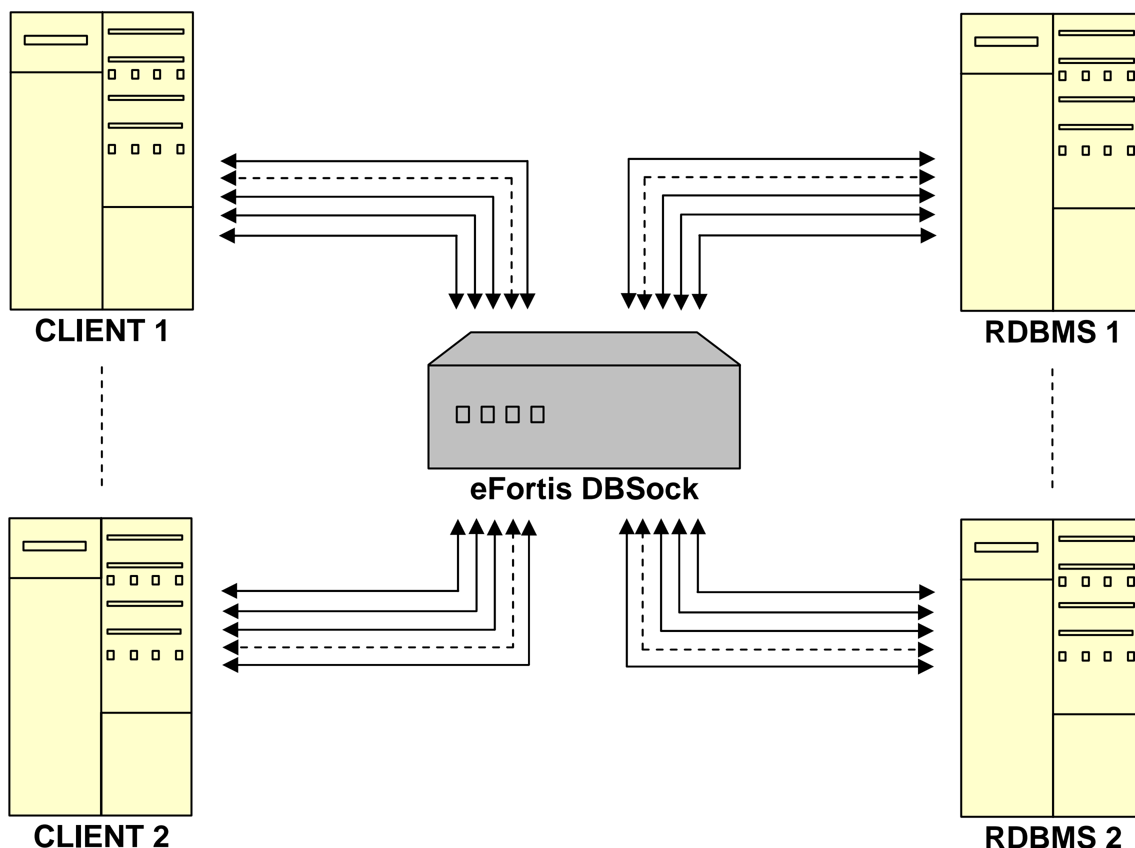
This topology is recommended just when the client side doesn't support the before topology.



For load balance and failover futures it's necessary to use the topology with two databases or two clients at least connected by DBSock.

In load balance systems, each request to client or database is shared by other client or database. DBSock accept an infinity number of servers in threads. The limit is just the OS and hardware performance.

Failover work like load balance mode except that just the primary server is used until a problem be detected. In this case the second server accepts all connections and after that all new connections will send to this server. Like load balance DBSock accept an infinity number of servers.



Finally the last topology is a variation with one or more clients linked to one or more databases thru DBSock.

DBSock accepts a large possibility of use adjusting to all situations and requirements.