



Building Better Performance: Distributed Applications Development With Persistence

by Christopher Keene, Founder & CEO

Persistence Software, Inc.

www.persistence.com

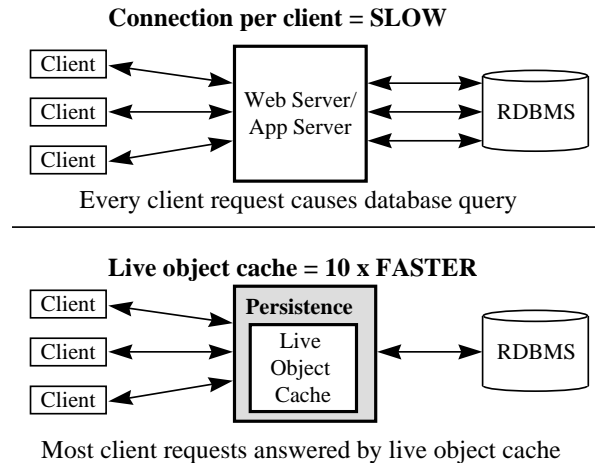
This white paper describes how to use the Persistence Live Object Cache to build high performance distributed systems. It discusses the issues involved in caching object data in an application server, provides benchmark data on live object caching, and includes an application example using a Persistence Live Object Cache with an Object Request Broker (ORB).

Persistence Software provides object-to-relational middleware for building mission critical applications based on the concept of a “live object cache.” A live object cache maps information from relational tables into business objects. Applications built to access business objects in the live object cache can deliver a 10 times performance improvement over traditional client/server solutions.

The findings in this document reflect five years of experience helping customers deploy a variety of high performance distributed systems. AT&T built a system with Persistence for end-to-end network provisioning which achieves 150 transactions per second against a 600 GB Oracle database by storing network configuration information in a live object cache. Federal Express uses Persistence for their Memphis command and control system, storing flight schedule information in a live object cache. The traffic systems built by TRW for the Olympics depend on Persistence. Even the systems to manage the video feeds for the Olympics were built using Persistence.

Need For Live Object Caching

When building a distributed system, performance is often the highest risk area. Live object caching provides better scalability by enabling multiple clients to share critical business data instead of constantly querying the database for the same information. Applications for which each client request triggers a database query are not scaleable past a few clients. In contrast, applications which use a live object cache to pre-fetch and share commonly used information among many clients can provide 10 times better performance.



Without live object caching, a web server or application server can easily become a performance bottleneck. For example, in an Internet catalog application, product information might include text, pictures, diagrams and pricing from several different relational tables. With direct database access, every client wanting product information must connect to the database and perform complex query for each product they are interested in seeing.

Live object caching is a critical requirement for achieving performance in distributed systems. Using a live object cache, “high activity” product information is pre-fetched, from the database, mapped into business objects, and shared among many clients. This has been demonstrated to provide over ten times performance improvement over direct database access for the applications listed in the following table:

Industry	Sample Application	Live object cache data
Telecom	Network management, billing	Network configuration, billing structure
Financial	Trading system, risk management	Security portfolios, risk positions
Transportation	Flight scheduling, package routing	Flight legs, truck routes

Alternatives To Live Object Caching

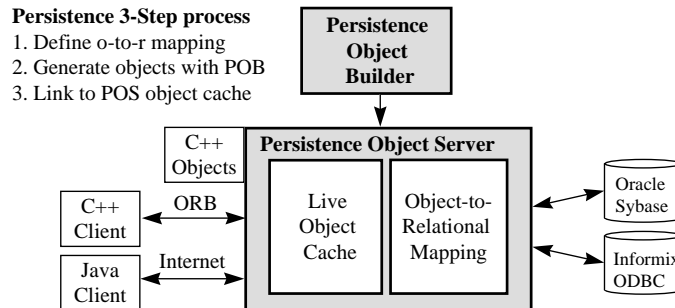
The page level caching provided by relational databases is complementary to live object caching, which can combine information from several different databases. While the relational page cache resides on the database server, the live object cache is maintained on an application server and can combine information from several databases. Accessing data in the live object cache is done through a simple object query or navigation. Accessing data cached by the relational database still requires significant database processing to return information.

In addition, many object databases provide caching capabilities for distributed systems. This means that their performance may in some cases be comparable to a live object cache. Object databases, however, require that transactions and data storage are managed by their own proprietary engine, a requirement which is typically not feasible for a distributed business application.

Elements Of The Persistence Solution

Building a distributed application with Persistence is a three step process. First, specify the mapping from relational data to business objects. This mapping can be imported from existing relational tables, read from

a CASE tool or entered directly into Persistence. Second, generate object-to-relational mapping objects using the Persistence Object Builder (POB). The Persistence Object Builder can also generate language independent Interface Definition Language (IDL) class definitions for each data mapping class, along with a default IDL stubs implementation. Third, link these objects to the Persistence Object Server (POS) and the appropriate ORB or transaction monitor libraries to create a high performance distributed object server.



Evaluating the Live Object Cache

Maintaining objects in a live object cache can greatly enhance performance for distributed applications, yet it introduces a number of technical considerations as well. These include enforcing object constraints, managing underlying database locks, mapping object changes into relational transactions and optimizing object flushing.

Because objects may be composed of information from multiple tables, it is critical that certain integrity constraints be managed at the object level. Persistence ensures that any relational data retrieved or stored follows the object constraints expressed in the object model. Another critical constraint for all objects is that each object be unique. It is easy to build applications which through multiple queries return several copies of the same object. Persistence avoids this problem by registering each object returned from the database in the live object cache.

For high-transaction systems, Persistence supports either pessimistic or optimistic locking models. In the pessimistic model, data is automatically locked in the database as it is accessed or updated by the application. In the optimistic model, Persistence manages a version stamp stored in the database to ensure no other user has changed an object since it was last accessed.

The developer may also specify on a class or object level which information should stay in the cache after a commit. Typically this is information which will only change under controlled circumstances, such as pricing information or a network configuration. Persistence also employs “smart flushing” to ensure that relationships between instances in the live object cache are preserved or flushed appropriately when the transaction commits.

Performance Of The Live Object Cache

A live object cache restructures data from a relational database into in memory business objects. These objects may include information from several different tables, which can impose significant performance overhead to retrieve from a relational database. Querying an in-memory object structure is approximately 500 times faster than querying a relational database. For example, retrieving a Stock and all of its Trades requires a database join. Retrieving the Stock object and its related Trades from the live object cache is an in-memory operation.

The following benchmark illustrates the performance offered by live object caching. The benchmark consists of two programs: the Persistence-generated interface and the hand-coded interface. The

Persistence-generated interface implements the read and update tests using Persistence generated classes and the Live Object Cache. The hand-coded interface program implements these tests by programming directly to the native database interface (using dblib for Sybase).

The benchmark tests were performed using a simple object class, containing five attributes. For the benchmark, each operation was performed 1,000 times within a single database transaction. The following table presents a summary of the results.

Performance Benchmark

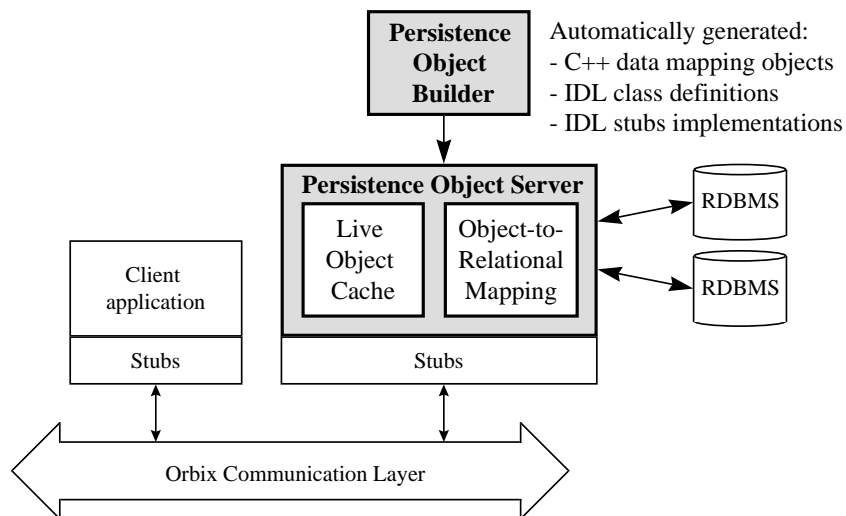
Operation	Persistence	Hand-coded	Percent Difference (%)
Read from database	22.81	21.32	(7.0)
Read from live object cache	0.04	21.32	533 x faster
Update to database	27.75	25.30	(9.7)
Update to live object cache	0.06	25.30	422 x faster

Persistence consultants have built applications which demonstrate the performance benefits of the live object cache. One such application, the flight leg server, shows the value of live object caching for transportation logistics. The application manages a network of airports, flights, flight legs and aircraft in a live object cache. For update-oriented transactions which make moderate use of the cache - such as making flight reservations - the live object cache provides a performance benefit of 5 to 10 times over straight relational access. For decision support operations which can make extensive use of the cache - such as load balancing an aircraft - the live object cache can provide over 100 times performance improvement.

Using Persistence In Distributed Application

The performance benefits of live object caching are magnified in a distributed environment. For these applications, many clients can share a common set of critical information in a live object cache, greatly reducing database traffic and eliminating the need to have a database connection for each client.

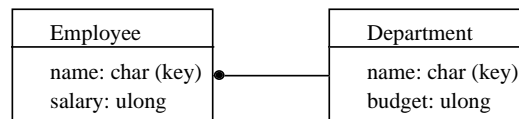
The Persistence Live Object Cache may be integrated with leading ORBs and Transaction Monitors, including Iona's Orbix, SunSoft's Neo, Expertsoft's Powerbroker, BEA's Tuxedo and Transarc's Encina. For example, Persistence for Orbix is an example of a tight coupling between Persistence and Orbix from Iona.



ORB Application Example

One of the most valuable features of ORBs is that client applications need not concern themselves with the physical location of servers. A server object may reside on the same network as the client, or it may reside on a remote host on the Internet or Intranet. Another ORB feature is language and platform independence: the client and server programs need not be developed in the same programming language, so the client can be written in Java while the server is written in C++.

To give an example of how a developer would build an application using Persistence with an ORB, consider the following application, CorpApp. The CorpApp application contains two classes, Employee and Department, with two attributes each; there is a one-to-many relationship between the classes, with the foreign key attribute residing in the Employee class.



The ORB client and server programs manipulate instances of these two classes. The client program is given operations for creating, reading, updating, and deleting instances of these classes. There are operations for creating / destroying database connections and beginning / committing / aborting transactions. Finally, relationship traversal operations are also available. The server program supplies the implementations for all of these operations, using objects stored in the live object cache.

When working with an ORB, the Persistence Object Builder generates everything needed to create a live object cache server which can be accessed by any client on the network or over the Internet. This includes generating the IDL required by the ORB and all the skeleton implementations needed to link Persistence object mapping classes to their corresponding IDL classes. Finally, Persistence generates a server main program and makefile, effectively automating the development of ORB-ready object-to-relational servers.

Building Distributed Web Applications

The performance benefits of live object caching are particularly important for Internet and Intranet applications. A web server which constantly queries a relational database to support browser requests will have very low performance and little scalability. For these applications, it is critical to pre-fetch reference information into the live object cache, then make this information available to browsers on a real-time basis through the web server.

Using the Persistence for Iona ORB Server product, developers can link Java clients to a live object cache across the Internet. Today, a number of Persistence customers, including Healtheon for benefits administration and NET for network management, are implementing web applications which incorporate Persistence and ORBs.

Summary

Persistence enables the development of distributed applications which achieve high performance from live object caching. Both Sun and Sybase have endorsed the Persistence live object cache technology, and deliver Persistence as part of their distributed product solutions.

For development teams building distributed or multi-tier applications, architecting an application server that

combines performance with flexibility is a major design consideration. If implemented correctly, the middle tier application server can provide performance, integrity, consistent interfaces and the flexibility of open standards, making the promise of distributed computing a reality.

For more information on the Persistence, contact Persistence at (415) 372 3600, info@persistence.com, or www.persistence.com.

Appendix: Persistence For Orbix Source Code Listing

Here is an example of the client code the developer would write to connect to a database through Orbix and work with the Persistence data mapping objects:

```
//bind to ORB interfaces
Persistence::ObjectServer_var gObjectServer;
gObjectServer = Persistence::ObjectServer::_bind("orb", hostName);
gDepartmentIF = CorpAppServer::DepartmentFactory::_bind("orb", hostName);

//login to database
Persistence::LoginParams login(dbname, dbpassword);
Persistence::Connection_var gConnection;
gConnection = gObjectServer->connect(login);
gConnection->beginTransaction();

// Create Department
CorpAppServer::Department_var department;
department = gDepartmentIF->create(CORBA::string_cupl(dept));
department->budget(100);

//Show all Employees in the Engineering Department
CorpAppServer::Department_var dept;
dept = gDepartmentIF->queryKey("Engineering");
CorpAppServer::EmployeeCollection* emps;
emps = dept->employs();
CORBA::Long index;
CORBA::Ulong len = emps->length();
for (index=0; index<len; index++)
{
    CorpAppServer::Employee_var emp = (*emps)[index];
    cout << "Employee name = " << emp->name() << endl;
}

// Logoff from database
gConnection->commitTransaction();
gConnection->disconnect();
```

Notice that this listing introduces a few interfaces. The Persistence module contains interfaces for object server, connection and error handling classes. The CorpAppServer module contains interfaces for each class. The EmployeeFactory and DepartmentFactory interfaces declare the instance-independent (static) methods of Employee and Department, respectively. For example, the DepartmentFactory::create() method constructs a new Department instance, and the DepartmentFactory::queryKey() method returns an object from the database which matches the specified primary key. EmployeeCollection and DepartmentCollection are just type aliases for sequences of Employees and Departments.