Transparent Scalability with Clustering for Java e-Science Applications*

Pedro Sampaio, Paulo Ferreira, and Luis Veiga
psampaio@gsd.inesc-id.pt, {paulo.ferreira, luis.veiga}@inesc-id.pt
INESC ID/IST, Technical University of Lisbon, Portugal

Abstract. Since object-oriented programming has become dominant in application development, there has been the recurring issue of an impedance mismatch between the way programmers manipulate objects in memory, and the way they are made persistent in secondary storage.

The two-decade long history of events relating object-oriented programming, the development of persistence and transactional support, and the aggregation of multiple nodes in a single-system image cluster, appears to convey the following conclusion: programmers ideally would develop and deploy applications against a single shared global memory space (heap of objects) of mostly unbounded capacity, with implicit support for persistence and concurrency, transparently backed by a possibly large number of clustered physical machines.

In this paper, we propose a new approach to the design of OODB systems for Java applications: $(\mathbf{O}_3)^2$ (pronounced *ozone squared*). It aims at providing to developers a single-system image of virtually unbounded object space/heap with support for object persistence, object querying, transactions and concurrency enforcement, backed by a cluster of multi-core machines with Java VMs that is kept transparent to the user/developer. It is based on an existing persistence framework (ozone-db) and the feasibility and performance of our approach has been validated resorting to the OO7 benchmark.

1 Introduction

Since object-oriented programming has become dominant in application development, there has been the recurring issue of an impedance mismatch between the way programmers manipulate objects in memory, and the way they are made persistent in secondary storage. This mismatch has greater magnitude when using file APIs but it is actually more significant, due to its prevalence, when using relational databases: the object-relational mismatch (addressed in [4], later recounted in [12]).

* This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds, and research projects PTDC/EIA-EIA/102250/2008 and PTDC/EIA-EIA/113613/2009.

Motivation and Background: To address the aforementioned mismatch, a number of object-oriented database (OODB) systems were developed that embodied transparent (or orthogonal) persistence in existing programming languages (e.g., Gemstone [23] introduced transparent persistence in a Smalltalk dialect as early as 1987). However, OODB systems did not achieve the expected predominance, whose causes are still subject to controversy (complexity, unreadiness of programmers, absence of adoption by major players at the time,...) considered either as advantages or disadvantages by different arguing sides. Nevertheless, some of the principles they prescribed were later recovered, albeit in a more limited and simplified fashion, with the widespread usage of object-relational mapping [13] (current examples include OJB [2] and Hibernate [17]).

A few more years after, a kind of *back-to-the-future* trend emerged with the development of new Java-related object persistence standards, such as JDO (Java Data Objects [26]) and related technology. These (re-)introduced many concepts from the original OODB systems to a global audience of Java programmers with wide acceptance, re-evoking the early mismatch argument proposed roughly 20 years before (similar efforts have also been devised in the .NET world with LINQ [7]).

Current Trends and Goals: A similar trend has also been taking place with the rediscovery of the notion of a *single-system image* provided by the transparent clustering of distributed OO storage systems (e.g., from Thor [21] with caching and transactions *ca.* 1992, to present distributed VM systems such as Terracotta).[1] They allow to scale-out systems and overcome the limitations and bottlenecks w.r.t. CPU, memory, bandwidth, availability, scalability, and affordability of employing a single, even if powerful, machine, while attempting to maintain the same abstractions and transparency to the programmers.

The two-decade long history of events relating object-oriented programming, the development of persistence and transactional support, and the aggregation of multiple nodes in a single-system image cluster [24], appears to convey the following conclusion: programmers ideally would develop and deploy applications against a single shared global memory space (heap of objects) of mostly unbounded capacity, with implicit support for persistence and concurrency, transparently backed by a possibly large number of clustered physical machines.

Shortcomings of Current Solutions: Naturally, a number of works in the literature have contributed towards achieving such goals that we address in Section 2. We briefly address existing shortcomings. While existing popular OODB systems (e.g., db4o [22], ozone-db [15]) and persistence frameworks (e.g., Hibernate, JDO compliant) allow programmers to query the object store with declarative languages (e.g., OQL, JDOQL, XQuery/XPath on XML documents containing serialized objects), they do not accommodate the distribution/partition of object graphs across different cluster machines. Replication is sometimes supported only for fault-tolerance purposes, therefore the object heap cannot be increased by

[1] http://www.terracotta.org

aggregating the memory of several machines. Actually, some earlier distributed shared-memory OO systems (such as [21, 25, 27]) partially supported this but, while offering persistence and some transactional support, they forced programmers to state the location of root objects, not offering any support for queries (they only allow the transversal of object graphs through references). A current distributed VM system enjoying moderate success with developers, Terracotta, provides a single-system image to programs but employs local memory only for caching, and secondary storage solely for object swapping purposes at the coordinating node. Furthermore, it offers no support for queries over the objects stored.

Contribution and Proposal: In this paper, we propose a new approach to the design of OODB systems for Java applications: $\mathbf{(O_3)}^2$ (pronounced *ozone squared*). It aims at providing to developers a single-system image of virtually unbounded object space/heap with support for object persistence, object querying, transactions and concurrency enforcement, backed by a cluster of multi-core machines with Java VMs that is kept transparent to the user/developer. While embodying some of the principal goals of the original OODB systems (orthogonal persistence, transparency to developers, transactional support), it *reprises* them in the context of contemporary computing infrastructures (such as cluster, grid and cloud computing), execution environments (namely Java VM), and application development models, described next. It is based on an existing persistence framework (ozone-db [15]).

In fact, today more and more applications are developed resorting to OO languages and execution environments, encompassing common desktop and web applications, commercial business applications on application servers, applications for science and engineering (e.g., architecture, engineering, electronic system design, network analysis, molecular modeling), and even games, virtual simulation environments. This is due to the universality of the programming model and performance offered by present JIT[2] technology. Such applications essentially maintain, navigate and update object graphs with increasingly larger (main) memory requirements, more than a single machine has available or can manage efficiently. For storage, reliability and sharing purposes, these objects graphs also need be made persistent to a repository. Thus, being OO applications now prevalent in most domains, improvements to their common underlying execution environments (VMs and related middleware) can therefore have a wider impact by offering scalability and performance gains transparently.

In short, we propose a platform that supports object persistence with no impedance mismatch by providing single-system image view to the programmers. The rest of the paper is organized as follows. In the next Section, we address the relevant related work in some areas intersecting with our work goals. In Section 3, we describe the architecture of $\mathbf{(O_3)}^2$. Section 4 describes the main implementation details and Section 5 the performance results obtained with a

[2] Just-in time compilation.

benchmark from the literature. Section 6 closes the paper with some conclusions and future work.

2 Related Work

There is abundant work in the literature in a number of intersecting and overlapping themes related to the work proposed in this paper: i) OODB systems, ii) object-relation mappers and persistence frameworks, iii) distributed shared-memory systems, iv) distributed object-oriented systems and virtual machines, v) cluster and parallel computing. We address each theme individually with previous work, within paper length limitations, regarding how the following relevant properties are provided by the system: i) true *single-system semantics* (i.e., distributed aggregation of resources - namely memory, and distribution/partition of object graphs across cluster machines), ii) *transparency*/orthogonality to developers w.r.t. object-orientation in the programming model and object persistence, and iii) support for *object querying* (e.g., OQL), and object transversal with declarative declarative languages (e.g., XQuery/XPath).

OODB Systems: OODB systems traditionally designate those systems that are simultaneously database systems and object-based systems. They provide support for orthogonal (transparent) persistence of object graphs, querying to the object store (usually a single server machine), and frequently object caching. This is achieved without requiring an extra mapping step to a relational database. They also enable navigation through object graphs, type inheritance, polymorphism, etc.). Earlier examples include Exodus [10], O2 [19], Gemstone [8]. Examples of recent work include ozone-db [15] and db4o [22]. They provide transparency and object querying. The main limitation of past and current OODB systems is that they do not offer true single-system image semantics. A repository must fit in its entirety on a single machine; other machines may only be used as backup replicas for fault-tolerance purposes, but the object heap cannot be increased by aggregating the memory of several machines.

O/R Mappers and Persistence Frameworks: O/R mappers [13] and OO persistence frameworks comprise more recent approaches to achieve persistence in object-oriented systems, albeit with less flexibility, by leveraging existing relational databases, employing object-relational mapping. Examples include OJB [2], Hibernate [17], and implementations of the JDO (Java Data Objects) [26] specifications. LINQ.Net [7] is a related proposal for object persistence in Windows platforms. Most allow object querying using various languages (e.g., OQL, LINQ integrated with C#). Transparency to developers is mostly provided, albeit the dependence on external RDBMS, usually requiring the identification of specific tables or views. As with classical OODB systems, single-system image semantics is not supported, as RDBMS are usually confined to a single high-performance machine (with possible backups).

Distributed Shared-Memory Systems: One of the earliest ways of providing single-system image semantics was through the use of distributed shared-memory (DSM). This was achieved by leveraging virtual memory support in the architecture, translating global virtual addresses into physical addresses of local pages or triggering access to remote ones. Influential examples of DSM systems include Munin [6] and TreadMarks [18]. Nonetheless, they lacked transparency since developers needed to be aware of consistency models and algorithms (e.g., release consistency) different from those they were used to [20]. This was also a source of some overhead and performance penalties. No support for object persistence nor object querying was available.

Distributed OO Systems and VMs: Akin to DSM systems, distributed object systems were able to aggregate memory (heaps) of several machines across the network in order to offer applications a shared object space with uniform referencing across process boundaries, together with some runtime services (e.g., micro transactions, long-running transactions, possibly while disconnected using cached and replicated objects, distributed garbage collection). Examples include work in Thor [21], and OBIWAN [27], and Sinfonia [1]. These systems provide object persistence and transparency to developers w.r.t. programming model. However, support for single-system image semantics is not fully supported since distribution is made known to application developers, who must know where special (root) objects are located in the network. No object querying is supported, only root object look-up.

The same approach can be applied to the notion of a virtual machine for an object oriented language. A distributed virtual machine aggregates the resources of machines in a cluster, able to provide, e.g., a Java VM with a larger heap encompassing part (or all) of the individual machines' object heaps. This provides a single-system image with shared global object space [9], with virtually unbounded memory available to applications. Examples include cJVM [3], Jessica [28] and Terracotta (which, albeit its success, holds the entire object graph in a coordinator machine and employs others solely for caching). Persistence is not offered at all or is limited to support for object swapping. Furthermore, no support for object querying is provided.

Cluster and Parallel Computing: Current scientific computing and intensive data processing is also performed relying on the execution of parallel computations, e.g., in the context of Grid and cluster computing, by employing approaches that do not manipulate object graphs. Examples include MPI protocols [16], and programming/application models such as Map-Reduce [14]. These approaches allow queries over data and may offer persistence but only in files or tabular data (e.g., Google FS, BigTable), not object-oriented storage. Therefore, as object-oriented application developers are concerned, they do not offer transparency as they must adhere to such specific protocols, programming models, and data structures (no support for transversal of object graphs). Support for single-system image semantics is limited with MPI (a single, coordinator node holds the complete data in memory, e.g., a matrix), while BigTable supports it by partitioning the entire store over all cluster nodes but not with object-oriented data.

3 Architecture

In this section, we describe the architecture of $(\mathbf{O}_3)^2$. It is an extension of an existing middleware, ozone-db [15], simply because it is open-source and we can leverage some of its properties: persistence in object storage, transparency to developers who just have to code Java applications, support for transversal on object graphs using both a programmatic, as well as a declarative and query-based approach (using XML, W3C-DOM, and allowing XPapth/XQuery usage).

However, ozone-db lacks support for single-system image semantics, i.e., currently an object store must reside fully in a single server machine, and objects cannot be cached outside this central server (in some small installations, applications and object server are collocated in the same physical machine).

$(\mathbf{O}_3)^2$ provides single-system image semantics by employing a cluster of machines executing middleware that: i) aggregates the memory of all machines into a global uniformly addressed object heap, ii) modifies how object references are handled in order to maintain transparency to developers, regardless of where objects are located across the cluster, iii) manages object allocation and placement in the cluster globally, with support for inclusion of more specific policies (e.g., caching objects in client machines for disconnection support). We first describe the fundamental aspects regarding original ozone-db architecture and then describe the architecture of $(\mathbf{O}_3)^2$, and the referred mechanisms.

3.1 Background: ozone-db Base Architecture

The ozone-db is an open source object oriented database project, totally written in Java and aimed to allow the execution of Java applications that manipulate graphs of persistent objects in a transactional environment (including optimistic long-running transactions). Ozone-db has a sizable user base of application developers, and numerous applications ported to make use of persistent objects (e.g., [5]). The main goal of ozone-db is to provide object persistence and transactional support in a transparent way to programmers and without requiring external technology such as RDBMS. The middleware is completely implemented in Java, portable, and executes on virtually all implementations of the Java VM. A ozone-db database (or object repository) is in essence a server machine that manages and maintains the object repository. Programmers develop object oriented applications in a traditional, straightforward way. Ozone-db also has support for object graphs stored in XML repositories, compatible with W3C-DOM, suitable for XPath declarative transversal and XQuery object querying.

The ozone-db overall architecture is depicted in Figure 1, displaying the four main entities involved in executing an application. Broadly, all objects manipulated by applications (except those newly created) reside in the server memory (and ulteriorly made persistent to disk). Object methods are also executed at the server. Client applications mostly execute user interface code, hold variables and calculate expressions outside of methods of persistent objects. Client and server are independent applications but in many scenarios they actually execute on the same physical machine.

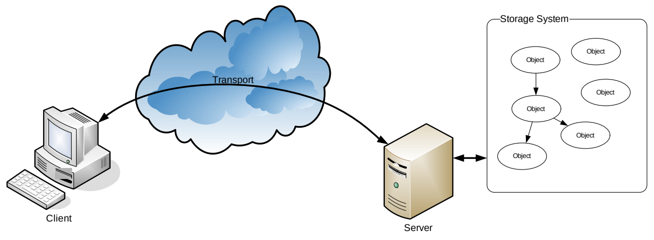

Fig. 1. ozone-db overall architecture

Client: represents the application launched by the user that manipulates the object repository. The client connects to the ozone-db server to address the persistent objects that are stored in the object database. The objects are loaded in the server when first accessed by client applications and are invoked by the client remotely. A client side ozone-db Java library is loaded with the applications.

Transport: represents information transfer between client and server, mainly for method invocations, parameters and results. It implements a protocol similar to, yet simpler than, the Java Remote Method Invocation (RMI). It depends only on the Java serialization mechanism, hence it is portable to virtually any Java VM implementation.

Server: represents the instance of the OODB system that performs overall management of object repository, holds objects in memory, and executes their methods, while they are being manipulated by client applications (that, as said, invoke them remotely). It manages client connections, security, and concurrency control with transactions initiated by client applications. The server rules the access of several clients to the persistent objects, while guaranteeing their consistency in a transactional environment.

Storage: represents how ozone-db ensures the physical persistence of the objects in the database. Its implementation relies solely on the server's file system, ensuring its portability to most platforms. To optimize access, a repository may be scattered across a number of individual files, each one containing a set of objects.[3] It is an aggregation of objects in a way to simplify storage organization and optimize read and write access, as well as I/O bandwidth of the file system.

With ozone-db architecture, it is possible to instantiate the server and client applications in the same machine or in different ones, depending on the computing resources available to the user, the size of the object repository, number of applications and application instances. The access to objects stored in the server is mediated by proxy objects, a common approach in most related systems. They make the remote access to server objects transparent to applications, which need not be made aware of the different physical location of the objects being manipulated. Figure 2 illustrates a common example situation where an application is

[3] A bunch or *cluster* in ozone-db terminology

connected to a server, and manipulates objects being instantiated there, usually a fraction of a larger graph of objects kept persistent in the database storage.

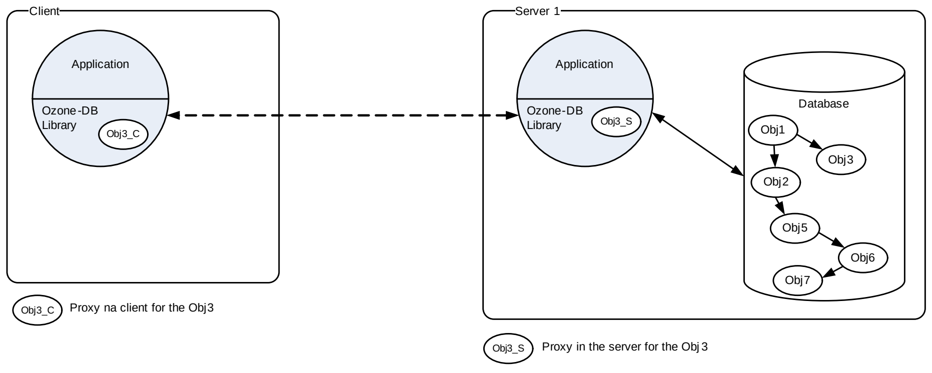

Fig. 2. typical ozone-db application manipulating a graph of objects at the server

3.2 $(\mathbf{O_3})^2$ Architecture

The current architecture of ozone-db offers a number of interesting properties but still suffers from important limitations. Mainly, its deployment is limited to a single server machine which may become a bottleneck in terms of memory, CPU, and I/O bandwidth. A medium range server machine may have 4 or 8 GB of main memory (with some operating system configurations and architectures, only half of that is available to applications and for that matter, to the Java VM object heap), one or two quad-core CPUs (with technology such as *hyper threading*, the number of hardware concurrent threads can double the number of cores), and several large capacity hard disks. While for small and medium size applications, such resources may be enough, they quickly become scarce when applications manipulate larger object graphs and/or several applications are executing concurrently.

Therefore, it would be advantageous to be able to aggregate the available memory of several server machines for increased scalability, and their extended CPU capability for increased performance. Furthermore, for a limited number of highly accessed objects, they should be cached at the client machine (eventually also with persistence), also for increased performance. In essence, allowing this while ensuring transparency to the application developers and compatibility with ozone-db, will provide the intended single-system image semantics. This requires that all interventions be made within the scope of $(\mathbf{O}_3)^2$ middleware, without imposing customized Java VMs nor modifications to Java application code. This last option might even be unfeasible, as applications may be distributed in byte-code format only.

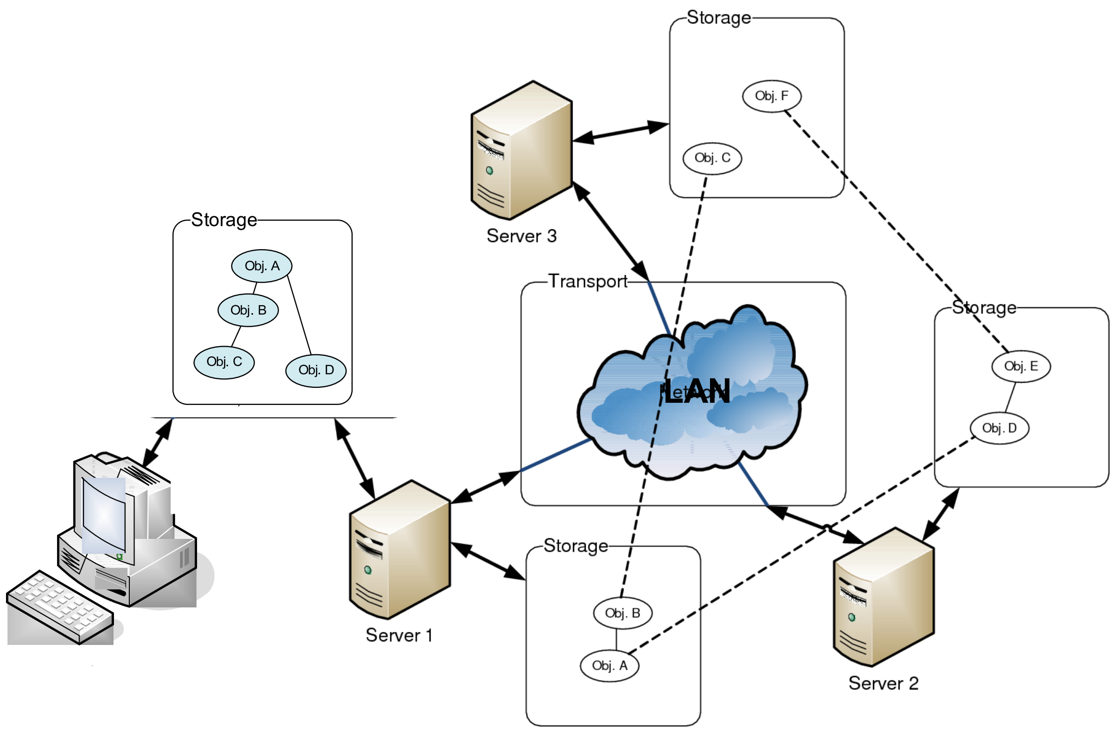

Fig. 3. typical application in the $(\mathbf{O}_3)^2$ architecture with a larger graph of objects at the servers, and a subset of objects cached locally

Figure 3 describes a typical scenario of application execution in $(\mathbf{O}_3)^2$. Regarding the example portrayed in Figure 2, we highlight the following differences: i) the object graph is distributed in main memory and in storage, partitioned among a group of servers (for simplicity, only three are shown), this being completely transparent to applications that need not know the server group membership, and ii) a set of heavily accessed objects can reside in a local caches at clients, for improved performance and bandwidth savings (and, additionally some support for disconnection). In Figure 3, the application while connected to Server 1 has accessed objects A, B, C and D of the graph with relevant frequency. Therefore, these objects are cached at the client in order to improve performance.

The extensions to ozone-db required by the $(\mathbf{O}_3)^2$ architecture are performed at the following levels described in the following paragraphs: i) transport, ii) server, and iii) storage, leaving the application interface unchanged for transparency w.r.t. developers.

Regarding transport, its architecture must be extended in order to be able to fulfill the following additional requirements. Method invocations on objects (originally simply relayed always to servers via proxies) must be registered to determine frequently accessed objects that could (and should) be cached locally. Subsequent invocations are performed against the cache and do not result in immediate communication with the servers, reducing server load and increasing execution speed. Several replacement policies may be used (not the topic of this work); currently a threshold of invocations is used to trigger caching of an object and the cache is preemptively flushed periodically.

The $\mathbf{(O_3)^2}$middleware running at servers is designed in the following manner. Each server now holds in its main memory only a fraction of the objects currently in use. The graph of objects is thus scattered across all servers to improve scalability w.r.t. available memory capacity and performance by employing extra CPUs to perform object invocation. The servers are launched in sequence and join a group before the cluster becomes available for client access. Regardless of object placement strategy, once a client gets a reference to an object, its proxy targets directly the server where the object is loaded. Two strategies may be adopted for object management and placement:

Coordinated: One of the servers acts as a coordinator holds a primary copy of metadata in memory, registering object location (indexed by objectID) and locking information (clients can be connected to any server, though, e.g., with some server side redirecting scheme). This information is lazily replicated to the other servers in the cluster. Modifications to this information (namely for locking) are only performed by the primary. This enables greater flexibility at the expense of some overhead. The coordinator may trigger migration of subsets of objects among servers, may decide to keep the the memory occupation of all servers leveled or, in alternative, only start to allocate objects in a server when the heap of the servers currently in use reaches certain thresholds. This will make the information in some proxies invalid but all servers check with the coordinator for new object location. On method return, the proxy at the client is reset appropriately.

Decentralized: No server needs to act as coordinator for the metadata. When an object is about to be loaded from persistent store, its objectID is fed to a hash function that determines the server where it must be placed, and where its metadata will reside. This is a deterministic operation that all servers in the cluster can perform independently. A simple round-robin approach would be correct but utterly inefficient as it would not any locality of reference. Instead, a tunable parameter in the hashing function decides broadly how many objects created in sequence (i.e., a subset of objects with very high probability of having references among them) are placed at a server before allocation is performed at another server. When objects are invoked later, this locality will be preserved. The overhead in this approach is lower at the expense of reduced flexibility as objects may not be migrated among servers.

Regarding storage, the persistent storage of objects is also balanced among the servers in the cluster using subsets of objects as the *quanta* of deployment. Currently, servers and storage must adhere to the same strategy of the two just described. With coordination, metadata contains information about location of the object itself (its contents). In the decentralized approach, when an object is about to be loaded from the store (when they are being accessed by the application for the first time), the hashing function is used to decide the server responsible for loading the enclosing subset of objects from disk and maintain those objects in its memory heap.

4 Implementation

This section describes the most relevant implementation details of $(\mathbf{O}_3)^2$. The application interface of ozone-db is unchanged, therefore applications need not be modified, nor even recompiled. The major aspects addressed are: i) server group management, and ii) object referencing.

Server Group Management: The $(\mathbf{O}_3)^2$ middleware running at each server in the cluster includes new classes `OzoneServer`, and `OzoneCluster` that allow each server to reference and communicate with other servers, and maintain information about the identity and number of servers cooperating in the $(\mathbf{O}_3)^2$ cluster. Presently, cluster management and fault-tolerance operate with the following approach. A designated cluster manager (just for these purposes but that may double as coordinator as described in Section 3) keeps `OzoneCluster` data updated and forwards notifications to the other servers. Any server, upon communication error may denounce another server to the cluster manager. After re-verification (to avoid wasted work) by the primary, the server is deemed as failed and a new instance of $(\mathbf{O}_3)^2$ is may be launched to take its place. To avoid the penalty of reloading all objects from persistent store, each server may have its own backup server to whom it propagates object invocations and modifications to metadata.

Object Referencing: Object referencing allows servers to redirect accesses to objects loaded in other servers. To avoid performing this repeatedly, after the appropriate server for an object is determined (via coordinated or decentralized strategies), an object proxy is set up in order to reference that server directly, without further indirection. The proxy just created is then returned to the client. Object tables and object metadata classes are extended to include attributes referencing `OzoneServer` objects, to register object location.

This process is described next. In the original ozone-db architecture, when a reference of an object proxy does not exist in memory, it is necessary to load it. This is done by instructing the server to load de the object as the following pseudo-code illustrates:

```
...
if (RefProxyObj == NULL)
    RefProxyObj = LoadProxyObjInServer();
...
return RefProxyObj;
```

In $(\mathbf{O}_3)^2$ implementation an extra step is inserted that triggers the determination of the server where the object proxy is, according to the specified strategy (others may be developed by extending this behavior). Only after that, a message is sent to the determined server to load the object in memory. The follow pseudo-code illustrates this:

```
...
if (RefProxyObj == NULL) {
   server = FindServer();
   RefProxyObj = LoadProxyObjInServer(server);
}
...
return RefProxyObj;
```

This extra-step is necessary because the object graph is distributed/partitioned among all servers in the $(\mathbf{O}_3)^2$cluster.

The way users connect to the database, delete objects from the database and delete whole stores is not affected.

5 Evaluation

The evaluation of $(\mathbf{O}_3)^2$ was performed by executing a known benchmark for OODBs (OO7 [11]) with dimension of objects, number of references and increased in order to make execution times longer (topping at 200 roughly seconds). Both the original ozone-db and $(\mathbf{O}_3)^2$ architecture were used to execute the benchmark tests in two scenarios: i) single server, and ii) two-node cluster (when testing ozone-db, only one of the machines is actually used as server, the other as a client). The machines used are Intel Core2 Quad with 8 GB RAM and 1 TB HD each, running Linux ubuntu server edition for extended address space for applications. The tests purpose is to show that $(\mathbf{O}_3)^2$ clustered architecture, while improving scalability and memory capacity, does not introduce significant overhead in application execution, and that it reduces memory usage in the servers.

The tests evaluate memory usage at each server and execution time for three OO7 benchmark tests: i) consecutive object creation, ii) complete transversal of an object graph, and iii) transversal of the object graph searching for an object (matching). The test database of OO7 consists of several linked objects in a tree structure as depicted in Figure 4. The tree structure has three levels, 2000 or 4000 child objects for the two first levels, and either 40000 or 200000 references among those objects to simulate different object graph densities.

The results in Figure 5 show that total memory usage is similar across the configurations for create and transversal tests. These tests occupy the most memory and $(\mathbf{O}_3)^2$ does not introduce relevant overhead. Note that memory occupation is reduced as servers are added because with 3-node $(\mathbf{O}_3)^2$ cluster, the memory effectively used by each server is roughly half of the total shown. With ozone-db, all objects are loaded at one of the machines, the other only used to offload client application (hence slightly reduced memory usage). This shows that for the most memory intensive tests with $(\mathbf{O}_3)^2$, the global memory available for

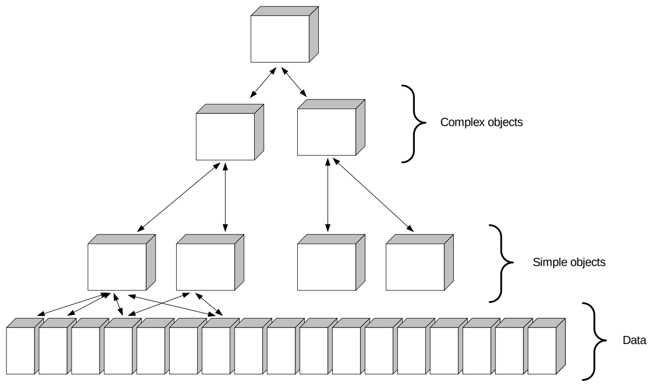

Fig. 4. Tree structure of the testing database

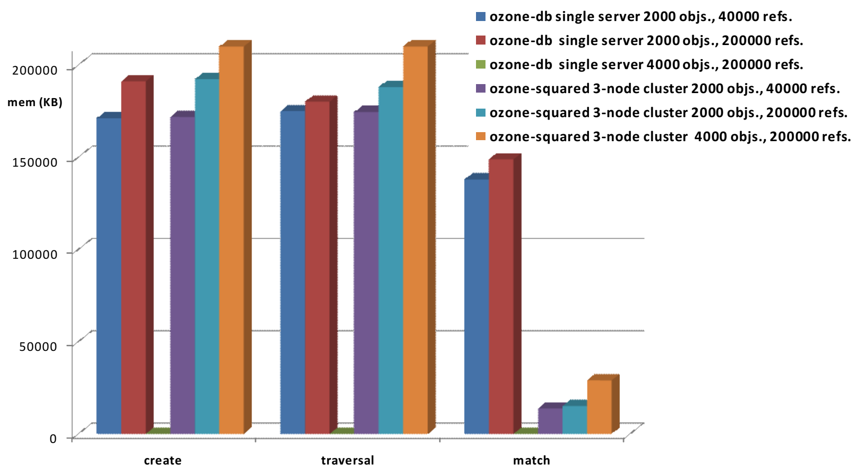

Fig. 5. Memory usage tests

applications can indeed by multiplied without any significant overhead at each server instance.

In the case of original ozone-db, when the objects are 4000 and the references are 200000 it is not possible to execute the application, because the server has not enough memory. The tests show that with $(\mathbf{O}_3)^2$it is possible to execute this test without applications crash.

The improvements in the matching test are due to additional factors. Original ozone-db, when searching objects preloads all object and metadata. In $(\mathbf{O}_3)^2$, this is performed on demand, and failed matches are garbage collected and memory occupation is kept significantly lower. This is relevant for the performance of object querying which is also improved with $(\mathbf{O}_3)^2$.

The results in Figure 6 show that total execution times for the benchmark tests remain similar across configurations. This demonstrates that $(\mathbf{O}_3)^2$ man-

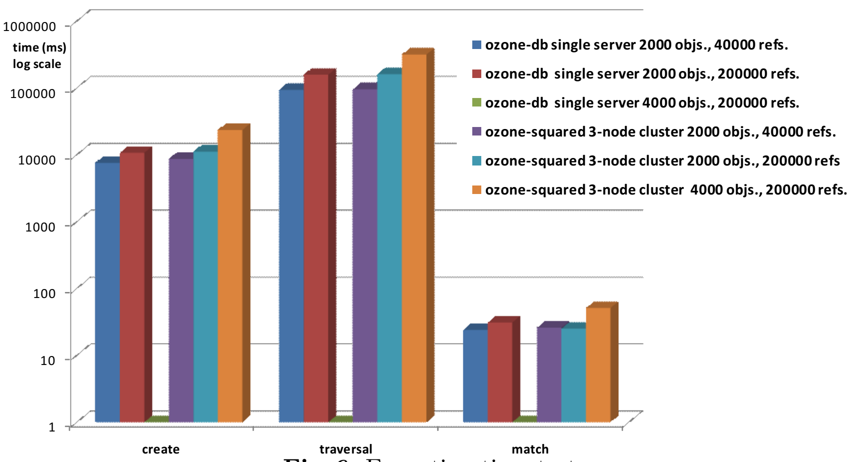

Fig. 6. Execution time tests

agement of several servers and distribution/partitioning of object graphs does not introduce any noticeable overhead to application execution times. However, we must bear in mind that OO7 benchmark is a single threaded application. If there are multiple threads in execution and/or multiple applications accessing the database, the extra CPU capability leveraged by $\mathbf{(O_3)}^2$ will keep processors' load low and increase system throughput, if not reduce individual application execution times.

6 Conclusion

The impedance mismatch between the way programmers manipulate objects in memory, and the way they are made persistent in secondary storage has been a two-decade long history of events relating object-oriented programming, the development of persistence and transactional support, and the aggregation of multiple nodes in a single-system image cluster.

In this paper, we propose a new approach to the design of OODB systems for Java applications: $\mathbf{(O_3)}^2$ (pronounced *ozone squared*) that addresses the limitations of previous work in the literature. It provides developers with a single-system image of virtually unbounded object space/heap with support for object persistence, object querying, transactions and concurrency enforcement, backed by a cluster of multi-core machines with Java VMs. Transparency regarding developers and their interface with the OODB system is untouched. Applications need not be modified nor recompiled. Our approach has been validating by employing a benchmark (OO7) relevant in the literature.

Future work includes more refined strategies for object placement (namely based on traces of previous runs of the same application) and address the incompleteness and unsoundness of the memory management of persistence stores in ozone-db (based on explicit delete operations).

References

1. M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. 21 st ACM SOSP, 2007.
2. Apache Foundation. Apache object relational bridge - OJB, 2002.
3. Yariv Aridor, Michael Factor, Avi Teperman, Tamar Eilam, and Assaf Schuster. Transparently obtaining scalability for java applications on a cluster. *Journal of Parallel and Distributed Computing*, 60(10):1159 – 1193, 2000.
4. Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, and Stanley B. Zdonik. The object-oriented database system manifesto. In Hector Garcia-Molina and H. V. Jagadish, editors, *SIGMOD Conference*, page 395. ACM Press, 1990.
5. Richard T. Baldwin. Views, objects, and persistence for accessing a high volume global data set. In *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 77, Washington, DC, USA, 2003. IEEE Computer Society.
6. J. Bennet, J. Carter, and W. Zwaenepoel. Munin: Dist. shared memory based on type-specific memory coherence. In *ACM Symposium on Principles and Practice of Parallel Programming*, volume 30 of *ACM SIGPLAN Notices*, pages 168–176. ACM Press, March 1990.
7. Don Box and Anders Hejlsberg. The linq project: .net language integrated query. Technical report, March 2006.
8. P. Butterwoth, A. Otis, and J. Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.
9. Rajkumar Buyya, Toni Cortes, and Hai Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2), 2001.
10. Michael J. Carey and David DeWitt. The architecture of the EXODUS extensible DBMS. In *Proc. Int. Workshop on Object-Oriented Database Systems*, pages 52–65, Pacific Grove, CA (USA), September 1986. IEEE.
11. Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The oo7 benchmark. In *SIGMOD Conference*, pages 12–21.
12. M.J. Carey and D.J. DeWitt. Of objects and databases: A decade of turmoil. In *PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES*, pages 3–15. Citeseer, 1996.
13. M.J. Carey, D.J. DeWitt, J.F. Naughton, M. Asgarian, P. Brown, J.E. Gehrke, and D.N. Shah. The BUCKY object-relational benchmark. *ACM SIGMOD Record*, 26(2):146, 1997.
14. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
15. Falko Braeutigam and Gerd Mueller and Per Nyfelt and Leo Mekenkamp. The ozone-db Object Database System, www.ozone-db.org, 2002.
16. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.1*. University of Tennessee, Knoxville, 2008.
17. W. Iverson. Hibernate: A J2EE (TM) Developer's Guide. 2004.
18. P. Keleher, A. Cox, and W. Zwaenepoel. TreadMarks: Dist. shared memory on standard workstations and operating systems. *Proc. of the 1994 Winter USENIX Conf.*, January 1994.
19. C. Lecluse, P. Richard, and F. Velez. O2, an object-oriented data model. In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 424–433, New York, NY, USA, 1988. ACM Press.
20. Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.
21. Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in thor. In *International Workshop on Distributed Object Management*, pages 79–91, 1992.
22. J. Paterson, S. Edlich, H. Hörning, and R. Hörning. The Definitive Guide to db4o. 2006.
23. D.J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. *ACM SIGPLAN Notices*, 22(12):117, 1987.
24. GF Pfister, I.B.M.A. Workstations, S. Div, and TX Austin. The varieties of single system image. In *Advances in Parallel and Distributed Systems, 1993., Proceedings of the IEEE Workshop on*, pages 59–63, 1993.
25. M. Shapiro, P. Ferreira, and N. Richer. Experience with the PerDiS large-scale data-sharing middleware. *Lecture notes in computer science*, pages 55–69, 2001.
26. Sameer Tyagi, Michael Vorburger, Keiron McCammon, and Heiko Bobzin. Core java data objects. Prentice Hall PTR / Sun Microsystems Press, 2004.
27. L. Veiga and P. Ferreira. Incremental replication for mobility support in OBIWAN. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 249–256, 2002.
28. Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *IEEE Fourth International Conference on Cluster Computing*, Chicago, USA, September 2002.