Transparent Scalability with Clustering for Java e-Science Applications*

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

Abstract. 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

A trend 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 [6] with caching and transactions *ca.* 1992, to present distributed VM systems such as Terracotta). 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 [8], 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.

* This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

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[1] 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.

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 [5]).

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 the performance results obtained with a benchmark from the literature. Section 5 closes the paper with some conclusions and future work.

2 Related Work

OODB systems traditionally designate those systems simultaneously databases 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 Gemstone [3]. Examples of recent work include ozone-db [5] and db4o [7]. 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.

[1] Just-in time compilation.

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 [6], OBIWAN [9], 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 provided 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 VM for an object oriented language. A distributed VM 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 [4], with virtually unbounded memory available to applications. Examples include Jessica [10] 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.

3 Architecture

In this section, we describe the architecture of $(\mathbf{O}_3)^2$. It is an extension of an existing middleware, ozone-db [5], 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.

$(\mathbf{O}_3)^2$ 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.

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 e-Science applications ported to make use of persistent objects (e.g., [2]). 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.

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.

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. 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 bytecode format only.

Figure 1 describes a typical scenario of application execution in $(\mathbf{O}_3)^2$. 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 local caches at clients, for improved performance and bandwidth savings (and, additionally some support for disconnection). In Figure 1, 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.

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

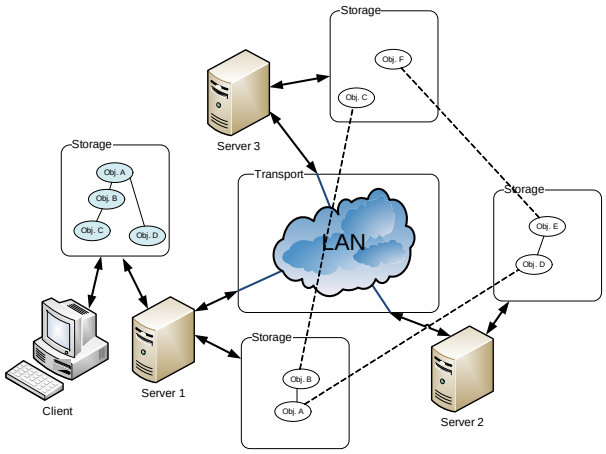

Fig. 1. Typical application in the $(\mathbf{O}_3)^2$ architecture with a larger graph of objects at the servers, and a subset of objects cached locally

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. The coordinator may trigger migration of subsets of objects among servers, may decide to keep 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.

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 leverage 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.

4 Implementation Issues and Results

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.

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. 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).

Evaluation: The evaluation of $(\mathbf{O}_3)^2$ was performed by executing a known benchmark for OODBs (OO7) with dimension of objects, number of references and connections per objects 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) three-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. 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 2 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 a third 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

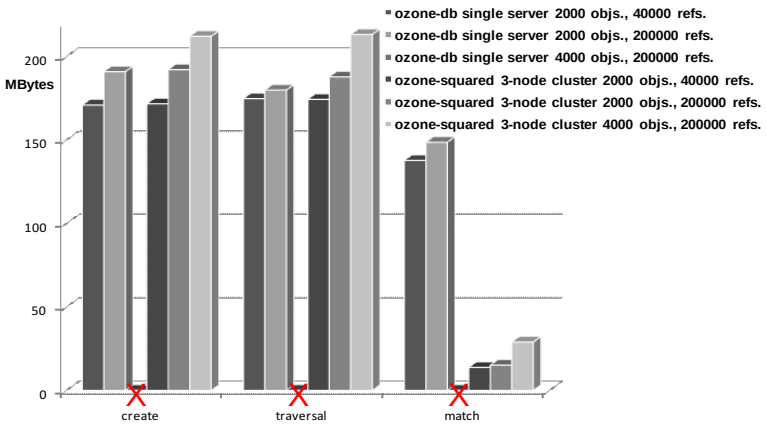

Fig. 2. Memory usage tests (total memory used by single node and whole cluster)

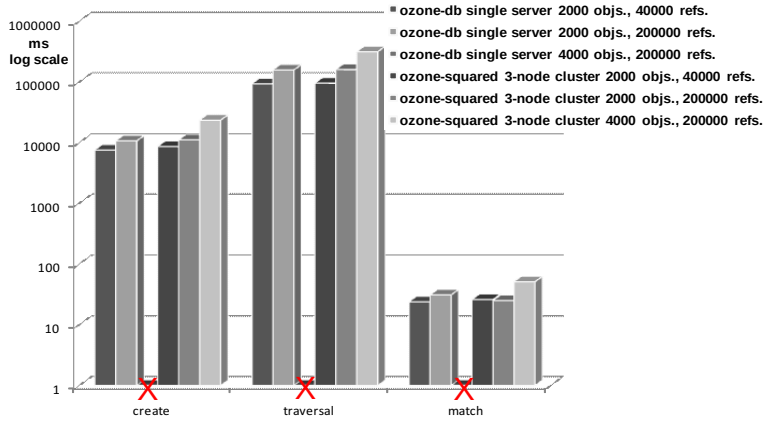

Fig. 3. Execution time tests

shows that for the most memory intensive tests with $(\mathbf{O}_3)^2$, the global memory available for 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. $(\mathbf{O}_3)^2$ scales making it possible to execute this test with increased memory load without applications crash.

The results in Figure 3 show that total execution times for the benchmark tests remain similar across configurations. This demonstrates that $(\mathbf{O}_3)^2$ management 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, so no speed-up was to be expected. 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.

5 Conclusion

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. 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.
3. P. Butterwoth, A. Otis, and J. Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.
4. Rajkumar Buyya, Toni Cortes, and Hai Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2), 2001.
5. Falko Braeutigam and Gerd Mueller and Per Nyfelt and Leo Mekenkamp. The ozone-db Object Database System, www.ozone-db.org, 2002.
6. Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in thor. In *International Workshop on Distributed Object Management*, pages 79–91, 1992.
7. J. Paterson, S. Edlich, H. Hörning, and R. Hörning. The Definitive Guide to db4o. 2006.
8. 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.
9. 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.
10. 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.