$(O_3)^2$: From "poor-man's persistence" to Transparent Clustering for Java 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

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed Applications; D.3.4 [**Processors**]: Run-time environments; D.3.2 [**Language Classifications**]: Object-oriented languages

Keywords

Virtual machines, single-system image, transparent cluster

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. 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 in 1987), with current albeit simplified successors such as OJB and Hibernate.

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) and related technology, with similar efforts in .NET LINQ. 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 [4, 6] with caching and transactions *ca.* 1992, to [1] and present distributed VM systems such as Terracotta).

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. Example applications include web, business, science and engineering (e.g., architecture, engineering, electronic system design, network analysis, molecular modeling), and even games, virtual simulation environments.

While existing popular OODB systems and persistence frameworks (e.g., Hibernate, JDO compliant) allow programmers to query the object store with declarative languages (e.g., OQL, JDOQL), 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 aggregating the memory of several machines. Some earlier distributed shared-memory OO systems (such as [4, 6]) 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. 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.

We propose a new approach to the design of OODB systems for Java applications: $\mathbf{(O_3)^2}$ [5] (*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. Transparency regarding developers and their interface with the OODB system is untouched. Our approach has been validated by employing a benchmark (OO7) relevant in the literature.

2. $(O_3)^2$ARCHITECTURE

The architecture of $\mathbf{(O_3)^2}$ is an extension of an existing middleware, ozone-db [3], 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 [2], 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).

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

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

Middleware'10 Posters and Demos Track December 1, 2010 Bangalore, India
Copyright 2010 ACM 978-1-4503-0601-0/10/12 ...$10.00.

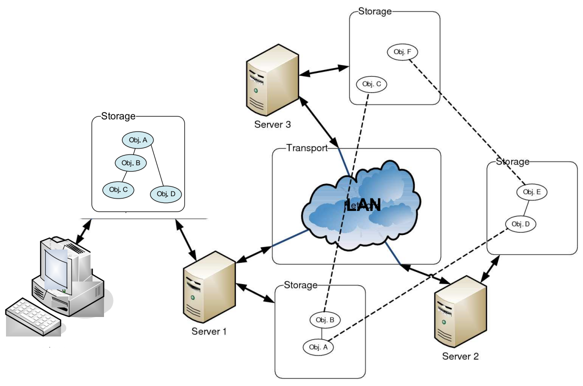

Figure 1: application scenario in $(\mathbf{O}_3)^2$ architecture.

(namely Java VM), and application development models.

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.

Figure 1 describes a typical scenario of application execution in $(\mathbf{O}_3)^2$, with relevant differences from ozone-db: 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 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: 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. 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. 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. First, 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.

Alternatively, 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.

In the implementation of $(\mathbf{O}_3)^2$, the application interface of ozone-db is unchanged, therefore applications need not be modified, nor even recompiled.

3. 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] Falko Braeutigam and Gerd Mueller and Per Nyfelt and Leo Mekenkamp. The ozone-db Object Database System, www.ozone-db.org, 2002.

[4] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in thor. In *International Workshop on Distributed Object Management*, pages 79–91, 1992.

[5] Pedro Sampaio, Paulo Ferreira, and Luis Veiga. Ozone-squared : From "poor-man´s persistence" to transparent clustering for java applications. Technical Report 38, INESC-ID, August 2010.

[6] 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.