# Java Object-Oriented Database
# JOODB

**Joe T. Nartca**

## 1. Introduction

It is not about a try to reinvent the wheel. It is about a lean development that concentrates only on the purpose of storing JAVA *serialized* objects (and *non serialized* data, e.g. String or a simple int). Most of the existing OODB are either too heavy to handle (e.g. own accessing language) or too much leaning on the traditional Relational DB (with SQL). JOODB is designed with the minimal effort to cope with all the necessary without leaving the JAVA standard path. No proprietary APIs, no new or extended accessing language. Simply the most fundamental accessing methods like ADD, DELETE, UPDATE, RENAME, ROLLBACK, etc. The business models can be embedded in the objects or as additional working frames (methods or objects).

## 2. JOODB Architecture and Structure

It is clear that JOODB must be able to run in a distributed networking environment (LAN, WAN and the Web).

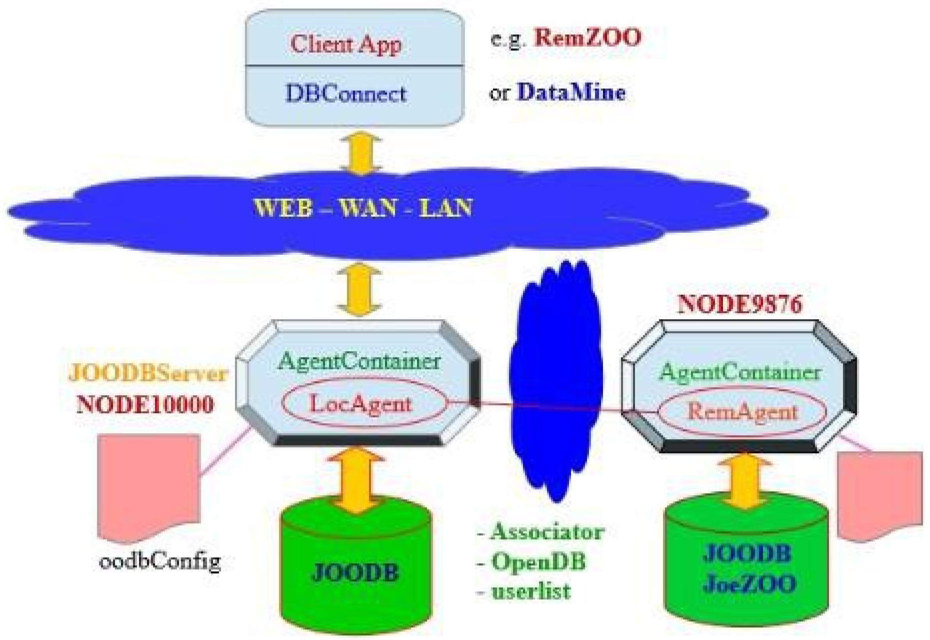


**Fig. 1**

**Fig.1** shows the JOODB accessing structure:

- Every OODB node can possess **n** local databases which are managed exclusively by the local Node Server.
- Each OODB node is initialized by a Configuration file (default: *oodbConfig.txt*) which has the JAVA-Properties format.
- If there are more local Node Servers on the same computer the one that *firstly opened by its Local Agent* (*LocAgent*) is the owner of this OODB and manages it exclusively. Clients of other nodes can **only** access to this OODB as remote clients (*Remote Agent* or *RemAgent*).
- Client can only access a node via an interface DB Connection (*DBConnect*).
- Clients and nodes must be authenticated (*userID* and *userPassword*) before they can start to work with the OODB.
- Clients may have different *access privilege* (read, read/write and read/write/delete).
- UserID, UserPassword and AccessRight are encrypted and kept in the **userlist**.

**Node Architecture**

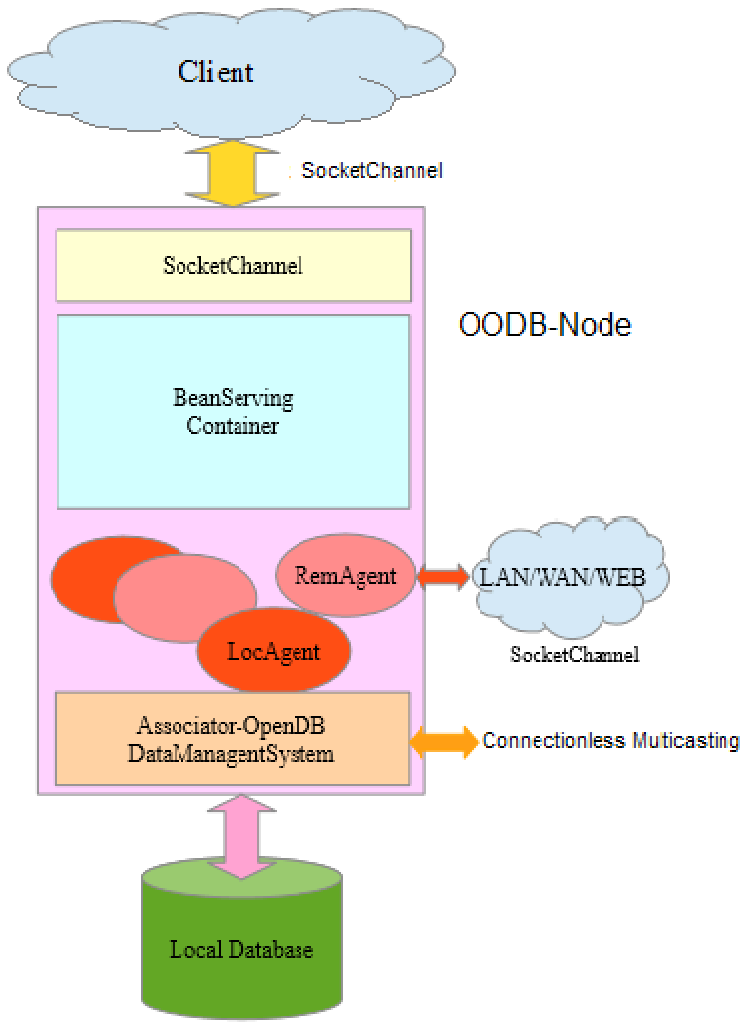


**Fig. 2**

**Fig.2** shows the structure and architecture of an OODB-Node. The networking between Node-Node and Node-Client bases on JAVA *SocketChannel* and *Connectionless Multicasting* (default IP:**224.0.0.3**, default Port: **8888**) for Client-OODB. The involved components (API):

- BeanServing (the server)
- DBBean (superclass of LocAgent and RemAgent)
- DBBroadcaster (each per OODB, invoked by OpenDB)
- LocAgent and RemAgent (started by BeanServing)
- DMS: Associator and OpenDB (accessed by LocAgent or RemAgent)
- All that depends on the **oodbConfig.txt**. Example:

```
################################################
# This is the basic configuraion for the OODB  #
# Project. JAVA properties format              #
# All property keyword must be written in      #
# UPPER CASE. This config file MUST be set in  #
# the PATH or, at least, in the same directory #
# of oodb.jar (the CORE) and the same for the  #
# "userlist" or see UNIX_USERLIST/WIN_USERLIST #
# Joe Nartca (c)                               #
################################################
# NODE = HostName:Port or IP:Port
#
NODE = localhost:9876
#-----------------------------------------------
# sMAX ServerSocketChannel MemoryBuffer
# aMAX Agent MemoryServer
# qMAX Max waiting queues for FixedPool
# pTYPE = 0 FixedPool,
#         1 CachedPool,
#         2 StealingPool
aMAX = 65536
sMAX = 4096
qMAX = 512
pTYPE = 0
#-----------------------------------------------
# Max latent time to wait for Server
# MicroSeconds
#
LATENT = 10
#-----------------------------------------------
# SYNCTIME Snychronizing interval (seconds).
# Default 30 min. or 1800 seconds
SYNCTIME = 1800
#-----------------------------------------------
# Broadcast interval (mSec). Default 500mSec.
# broadcasting Port and IP
# MONITOR 0: disable, >0: enable
MONITOR = 1
BCINTERVAL = 500
MULTICAST_PORT = 8888
MULTICAST_ADDR = 224.0.0.3
#-----------------------------------------------
# Remote classpathes of used objects. The path
# is separated by an ACCLAMATION !
# BACK SLASH is used when the string is too long
# (see bellow). No blank is embedded ! Always
# starting with file:// on LAN or http:// for WAN
#
CLASSPATH=file:///c:/JoeApp/OODB/Node9876/classes/\
!file:///c:/JoeApp/OODB/Node10000/classes/
#-----------------------------------------------
# Path to and for an OODB. Note: the last separator
# MUST be included
# LINUX/UNIX
UNIX_PATH = /media/Data/JoeApp/OODB/joodb/
# WINDOWS
WIN_PATH = c:\\JoeApp\\OODB\\joodb\\
#-----------------------------------------------------
# Path to the userlist
# LINUX/UNIX
UNIX_USERLIST = /media/Data/JoeApp/db/oodb/userlist
# WINDOWS
WIN_USERLIST = c:\\JoeApp\\OODB\\joodb\\userlist
#-----------------------------------------------------
```

**Client Architecture**

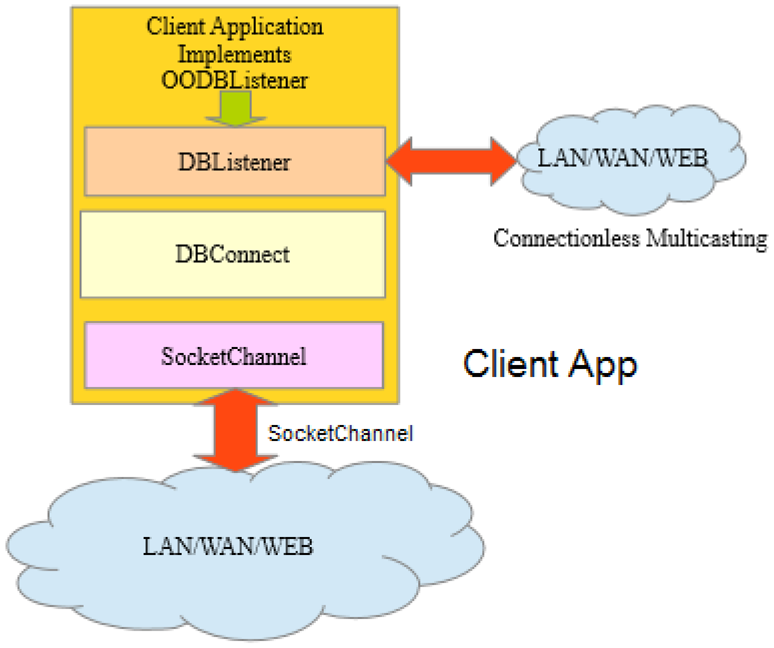


**Fig. 3**

**Fig.3** shows the structure and architecture of a Client Application:

- Client logic (Client application)
- Implementation of OODBListener (optional)
- DBListener (connectionless multicasting, optional)
- DBConnect or DataMine (obligatory, SocketChannel)

**JOODB features:**

- JOODB is organized in two parts: ObjectNames in *Associator* and ObjectData in **n** *Clusters*. OODB name is the root. Suffix *_asso* and *_n* (**n** = 0, 1, 2, 3, …) specify the corresponding parts. Both parts are saved in *GZIP* format.
- *Associator* contains **n** *AssoEntries (AE)*. Max. 2147483647 (2GB) AEs. Each AE consists of ObjectNameSize (2 bytes), ObjectDataSize (4 bytes) and the ObjName (max 32KB). Associator is partitioned in 4194304 sets, and each set is corresponding to one certain cluster. The *entire Associator* is loaded after the *first* Open and is kept in memory (performance) during the runtime. Modifications (e.g. new objectName) are saved when the last user quits or system downs.
- *Cluster* is the true database of JAVA serialized objects and grouped in 512 slots of variable size. Each slot can have a size of max. 2GB and is referenced by one *AssoEntry*. Only the referenced Clusters are cached and loaded (on-demand.)
- Quick access using JAVA APIs: *LinkedHashMap and HashMap*.
- *Self-Reorganization*. Deleted objects will be replaced successively by new added objects which occupy the empty (or deleted) slots. No need for any additional reorganization of the data files.
- Modification of object or Renaming of *ObjName* is broadcasting in real-time to all users who work with the *same* database. However, it preconditions that *Client apps* must implement the interface *OODBListener* and run *DBListener* as a thread of their own.
- If *AutoCommit* is set each modification is *committed* directly after *ObjName* is *unlocked*. Or in case of failure on the client site all uncommitted objects will be committed. Otherwise each modification *must be committed separately* in order to make the data permanent.
- Object *cannot be modified twice* before commit. Otherwise Exception No. 19 is thrown.
- *Rollback* (or *undo)* reverses a modification.
- *Multiple independent users*. If a user quits *without* autoCommit *only his modifications* are rolled back and the other users are informed (broadcasting feature).

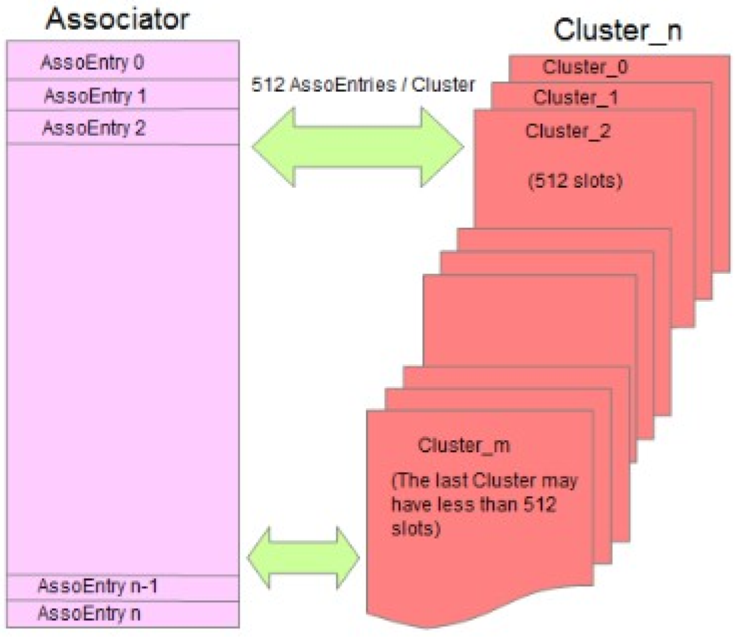


**Fig. 4**

## Object-Oriented Database (OODB) and Relational Database (RDB)

*OODB* fundamentally differs from *Relational Database (RDB)* in many aspects. One of the main aspect is that *OODB is an entity and RDB an atomic*. Or in other words: it is the way how a solution for an existed "problem" are referred, presented, accessed and stored. An entity is the *wholenes*s with own identity. Atomic is a *related singleton* that has no identity if it is isolated.

An object or an entity is in itself a complete thing that can be in itself a complete solution and does not require any further processing. An atomic is an incomplete, meaningless thing if it stands alone without any further attributes. The first is an *Object-Oriented* solution while the other is a *Relational* solution.

Example: if an object *BMW* or *AUDI* is referred it is in itself a *distinct, complete object*: a BMW or an AUDI. If only car is referred it is confusing and meaningless. If the same object car is referred with *all kinds of attributes* (e.g. BMW steer, BMW engine, etc.) then the referred car is now complete: a BMW.

Because this topic is about OODB. RDB is only a cross reference. So, I don't intend to go into details about their differences. However, there are also sameness between OODB and RDB: their *main access method*. The Key in RDB or ObjectName in OODB. For the ease and familiarity for those who are new to OODB the access "*key*" is used and referred here as "***ObjName***". Different name with the same substance and the principle. RDB usually needs *some additional* "keys" in order to be able to assemble *several singletons* in a *recognizable, acceptable* entity. OODB needs only ***one*** *ObjName*.

## OODD-Design

OODB is, if you so will, **the way** how an object is referred, presented, accessed and stored. **The way** is no other thing than a **design** how to present an object in the most optimal form: ***OODB-Design***. It is obvious: the more optimal an object is designed, the faster it can be accessed, presented and stored. Example: an object (or class) in JAVA can have (must not have) several methods. OODB design is the **state of the art** how to separate the needed from the unneeded. The less redundant an object is, the better it is. An OODB that has to carry a load of objects full of redundancies is the nightmare for the users who have to work with it.

Example: Object *Animal* is the *superclass* of *Dog, Parrot, Locust, Whale*, etc. and if it is implemented with 3 methods: *sound(), appearance()* and *description()* then every subclass inherits all 3 methods and that inflates the OODB with *3 redundancies per Animal.* If all 3 methods are separated and arranged in *another object* (or class) which is *neithe*r a part of object Animal, *nor* a part of OODB, but a part of (client) application then all *subclasses* of Animal are reduced to the necessary, without any superfluous redundancy.

JOODB is organized in two different parts (or files): *Associator* that contains the object names, and **n** *Clusters* that contain the serialized data (objects or String, primitives.) The name of the DB is the **root** of *Associator* and *Clusters* and the suffix *_Asso* or *_n* indicates the specified part.

Example: ZOO is the DB name, then *ZOO_Asso* is the *Associator* and *ZOO_n* (where n = 0, 1, 2, etc.) is *Cluster_n*.

## Applied Machine Learning (aML)

Machine Learning is a *big word*. A word that makes people feeling *uneasy*. Machine is not and never becomes a human (or any living species that has a brain.) Also, the term "**learning**" is misleading and wrong. It is simply about how a form of some patterns or a chain of sequences man *teaches* (to be more precise: *programs*) a machine to comply what man sees as it is or should be. And when a machine does exactly what it was taught and what it should do then it is ***aML***. Nothing more, nothing less. Any extrapolation or a deduction from some similar thing into a *new valuable* knowledge that leads to an appropriated action is a myth, not a reality. Or, at least, a very long way to go.

## Self-Organization

If an object is deleted the *ObjNameSize* of *AssoEntry (AE)* is set to 0 and the rest is physically removed (**n** bytes reduced to **2** bytes), the *corresponding slot in Cluster is also deleted*, NO real physical space occupies the storage. If a new object is added, the first AE which has *0-ObjNameSize* will be asserted and a new corresponding slot is then created for the object data. A **commit** (or *unlock* in conjunction with *AutoCommit*) makes the assertion permanent. **Self-Organization.** An *applied Machine Learning*. JOODB is designed with several ML features (e.g. remote access recognition).

## Multiple Users

In a *multiple-user environment* where users might access and modify the same object, it is easy to run into a *deadlock* situation. A proneness to some unexpected problems (e.g. *hidden inconsistency* or *long delay time*.) JOODB uses the *multicasting* possibility to make such a deadlock-situation visible to all users who work with the same object of the same OODB. Precondition is an implementation of the required method *oodbEvent()* of *OODBListener* interface and *DBListener* as an independent *Event-listening* thread.

## Safe Connection and Networking

JAVA-NIO *SocketChannel, ServerSocketChannel* and *ByteBuffer* are the bases for the communication between Client and Server. Only the broadcasting mechanism bases on *multicasting socket* and *UDP protocol*. All that happens in real-time (except some latency.) *Applied Machine Learning* is also the base for the distributed work between *LocAgent* and *RemAgent.*

## Big Data (BD)

Similar to ML Big Data is currently the *most mystic* word in the IT world. And no one can precisely explain or defines what BD exactly is. In reality it might be so: data that we acquire daily by our senses (seeing, hearing, touching and feeling) are manifold and our brain (or brain of any living species) is *able to reduce and to deduct them to the most usable form* so that we can react accordingly. BD is an applied knowledge of our brain: **Info(rmation)Graphics**. To be precise: Information into images. We can rely on our living experience: Data need not to be amassed, but should be let as they are. BD should be let where they are and how they are. The **best** "*Data Miner* or *Data Scientist*" is the one who is able to "*infographicize*" the data and to make them *visible, valuable, veritable, verifiable* and *variable* to the viewers or consumers. Complex mathematical formula or so is just an *obfuscation* of the reality. The application ZOO demonstrates how BD can be managed without having to amass locally the data. Images, Sound tracks and descriptions are *on the web where they are*. They are only accessed when they are needed or referred to. And all that in *real-time*.

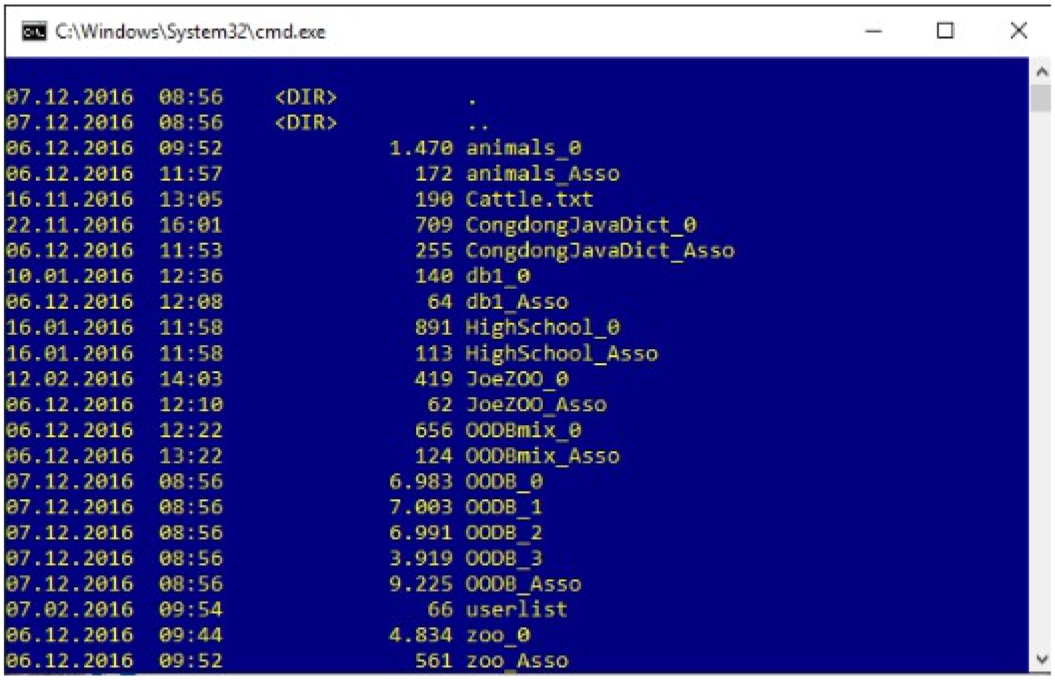
**JOODB highlights**:

- if *autoCommit()* is set (true) modification (*delete, update, rename*), except ADD operation, will be committed after *objectName* is released (i.e. unlocked). *autoCommit* is user-bound and that means only modifications of user who issues *autoCommit* are committed in case of unexpected exit.
- *ADD operation* always requires a *commit().* Otherwise the data is only temporary in cache and could get lost.
- Depend on the Setting of *SYNCTIME* (see *oodbConfig,txt* of Section **ServerNode**) OODB will be synchronized between Cache and Storage.
- OODB *Broadcaster* always works in sync with *DBListener* (see Section **Client Applicatio**n).
- AutoCreate. If an OODB is opened *for the first time* it will be created (and depending on user privilege) when user quits or iinvokes *saveDB()*.
- BeanServing runs its threads in *ThreadPool* which is either a *FixedPool* or a *CachedPool* or a *StealingPool* (concurrent *parallelism*). See **Section 3.2 ServerNode** for details.

The *access time of an object* (read/update) depends on its physical size and on the *geological* accessing path (remote or local). A test on an OODB of 1800 serialized objects of (2x Strings of 10 bytes, 1 int and 1 double) on a local node gives (on **Acer** i5-1.7GHz 8GB RAM.):

- an averaged access time of 0.41 millisecond per access. Or **2500** objects per second.
- selectAll("name", "Joe"): 0.84 millisecond for 402 objects out of 1807.
- selectAll("salary", "GE", 10000.0): 0.158 millisecond for 1717 objects out of 1807.

The following screenshot shows you how JOODB organizes (Associator and Clusters)


C:\Windows\System32\cmd.exe

```
07.12.2016  08:56    <DIR>          .
07.12.2016  08:56    <DIR>          ..
06.12.2016  09:52             1.470 animals_0
06.12.2016  11:57               172 animals_Asso
16.11.2016  13:05               190 Cattle.txt
22.11.2016  16:01               709 CongdongJavaDict_0
06.12.2016  11:53               255 CongdongJavaDict_Asso
10.01.2016  12:36               140 db1_0
06.12.2016  12:08                64 db1_Asso
16.01.2016  11:58               891 HighSchool_0
16.01.2016  11:58               113 HighSchool_Asso
12.02.2016  14:03               419 JoeZOO_0
06.12.2016  12:10                62 JoeZOO_Asso
06.12.2016  12:22               656 OODBmix_0
06.12.2016  13:22               124 OODBmix_Asso
07.12.2016  08:56             6.983 OODB_0
07.12.2016  08:56             7.003 OODB_1
07.12.2016  08:56             6.991 OODB_2
07.12.2016  08:56             3.919 OODB_3
07.12.2016  08:56             9.225 OODB_Asso
07.02.2016  09:54                66 userlist
06.12.2016  09:44             4.834 zoo_0
06.12.2016  09:52               561 zoo_Asso
```

Note: The database *OODB* consists of Associator *OODB_Asso* and four Clusters (*OODB_0*, *OODB_1, OODB_2* and *OODB_3*)

## 3. Programming with JOODB

### 3.1. Client Application

Client application can only access JOODB with the API *DBConnect*. DBConnect is the remote mirror of *OpenDB* (*open, add, update, delete,* etc.). *SocketChannel* is the connection base.

```
        /**
        Contructor, establish a connection to OODB-Server
        @param uID      String, LOGIN ID
        @param pw       String, LOGIN PASSWORD
        @param host     String, OODB Server HostName
        @param port     int, OODB Server port
        @exception      Exception for IO, NIO, etc.
        */
        public DBConnect(String uID, String pw, String host, int port) throws Exception
```

*DataMine* is an Extension of *DBConnect*. Client Application can run either *DBConnect* or *DataMine*. Each connection can handle **n** OODBs. The method *openDB()* must specify the OODB name which is to be opened. Thereafter all subsequent DB actions (e.g. *add, update*, etc.) must always carry this DB name as the first parameter. Example:

```
        // create a connection to Node Localhost:9876 under UserID Joe and password Password
        Dbconnect dbcon = new DBConnect("Joe","Password","localhost",9876);
        // set AutoCommit for all opened DBs
        dbcon.setAutoCommit(true);
        // open the OODB with the name ZOO
        dbcon.openDB("ZOO");
        // check for the existence of Object Bear, if not add it
        if (!dbcon.isExisted("ZOO", "Bear")) dbcon.add("ZOO", "Bear", new Animal(an));
        // get all animal names
        String[] names = dbcon.getObjNames("ZOO");
        …
        try {
            if (dbcon.lock("ZOO", "Bear")) {
                dbcon.update("ZOO", "Bear", new Animal(an));
                dbcon.unlock("ZOO", "Bear");
                boolean updated = dbcon.isUpdated("ZOO", "Bear");
            } else System.out.println("Error. Cannot lock 'Bear'.");
        } catch (Exception ex) {
            ex.printStackTrace();
        }
```

To cope with the integrity in a ***Multiple-Users*** *environment* it is recommended that Client application should implement the interface *OODBListener*, registers and adds it to *DBListener* then runs it as a subthread. Example:

```
public class ZOO extends JFrame implements OODBListener { // interface of OODBListener
    …
    public ZOO( ) { // constructor
        …
        try { // run DBListener
            listener = new DBListener(bIP, bPort);
            listener.register(zas.dm, dbName); // register the Connection and its DB
            listener.addListener(this);        // add it to DBListener
            listener.start();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    …

    // implement the required method of OODBListener
    public boolean oodbEvent(String msg) {
        JOptionPane.showMessageDialog(this, msg, "Info", JOptionPane.PLAIN_MESSAGE);
        redraw();
        return true;
    }
```

OODB messages are all about the *activities* (update, delete, etc.) that were performed on behalf on the clients. The message format is:

```
[NodeHostname:port](actiity) message

Example:
[localhost:9876]Rename ObjName:Cow of OODB:'C:\JoeApp\OODB\joodb\ZOO' to:Cattle
```

The users are informed about the *current activities* around the OODB so that they can avoid collision when an object is being updated by *user A* while *user B* tries to rename it (exception is always costly and it consumes more time and resources than a *broadcast* information).

*DBConnect* offers the users the possibility to execute an object on its local environment *without* having to have the required JAVA classes on their local computer. The method is

```
/**
 Execute a method of an executable serialized Object.
 @param obj    executable serialized Object
 @param method String, invoking method name
 @param pTypes ClassArray, list of Parameter Types (e.g. String.class, etc.)
 @param parms  ObjectArray, list of parameters (e.g. "HelloWorld", etc.)
 @return Object returned value (null for void)
 @exception Exception from HOST or JAVA
*/
public static Object exec(Object obj,String method,Class<?> pTypes[],Object[] parms) throws Exception
```

This possibility works *only in conjunction* with the classpath configuration on the Node Server (see *3.2 ServerNode/oodbConfig*, ObjName: `CLASSPATH`.) Example:

```
public class RemZOO extends JFrame { // SWING with JFrame
    …
    JButton but = new JButton("Animal");
    but.addActionListener(a → {
      try {
         String ani = JOptionPane.showInputDialog(this,"AnimalName");
         if (ani != null) {
            // retrieve the serialized object from OODB "ZOO"
            Object obj = dbcon.getObject(oodb, ani.trim());
            // execute or run the method "print()"
            dbcon.exec(obj, "print", new Class<?>[] { }, new Object[] {});
            // then the method "show()"
            dbcon.exec(obj, "show", new Class<?>[] {JFrame.class}, new Object[] {this});
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
    });
    …
}

// on the remote NodeServer
public class Animal implements java.io.Serializable {
    …
    public void print() {
      System.out.println("ZooAnimal:"+name+" has a weight "+weight);
    }
    …
   public ImageIcon getImage( ) {
     if (pic.indexOf("://") < 0 && (new File(pic)).exists()) return new ImageIcon(pic);
     try {
        return new ImageIcon(new URL(pic));
     } catch (Exception e) { }
     return null;
   }
   …
   public void show(JFrame jf) {
     new Display(jf, getImage( ));
   }
   //
   private class Display extends JDialog {
      public Display(JFrame jf, ImageIcon icon) {
         super(jf, true);
         setImage(jf, icon);
      }
      private void setImage(JFrame jf, ImageIcon img) {
         …
      }
   }
}
```

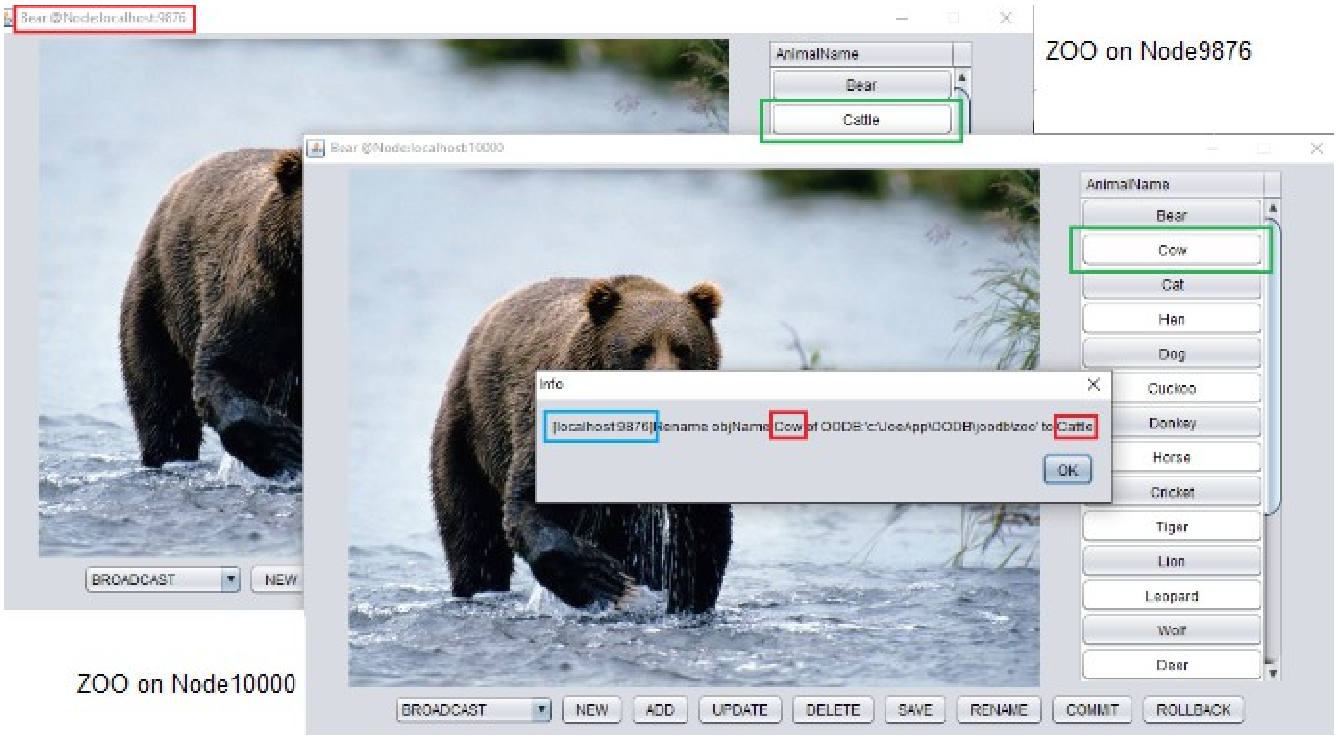
When the button "*Animal*" is pressed a prompt pops up for the name (or *ObjName*) of the animal then two *remote* methods are executed: *print()* and *show()*. Of course, the users *must know* the method names and their required parameters. Of course, it works only when the *remote object is independent on other object(s)*. Example, Access to external file that lies *elsewhere outside the* CLASSPATH will trigger some exception. *DataMine,* an extension of *DBConnect,* offers the users useful methods that are required for their complex application development:

- search (single or multiple)
- select (single or multiple)
- compValue (String or priitives)
- etc.

The *ZOO application* is bundled in the package. ZOO lets you *study* the way how different methods of *DBConnect* under different conditions (*local* or *remote* by starting with the argument *config10000.txt* or *config9876.txt*) are used (*commit, lock, unlock, add, delete*, etc.) Here are two screenshots of ZOO.

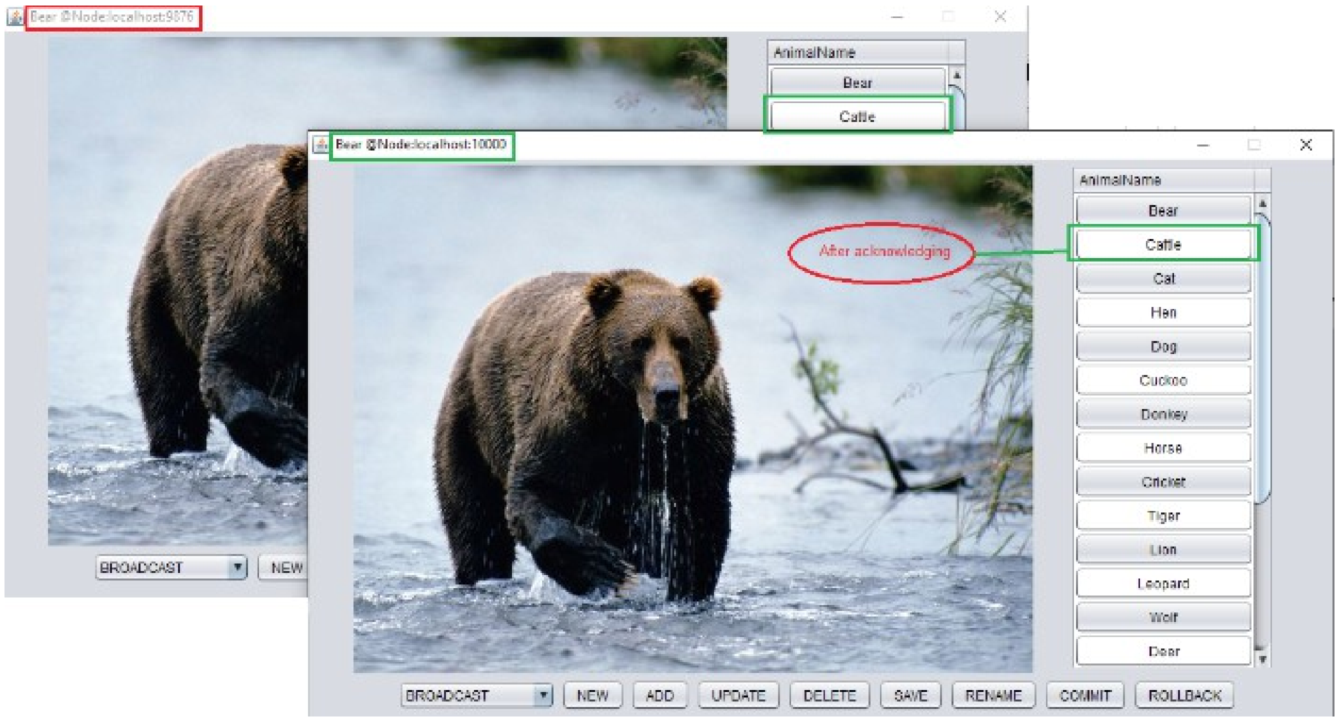
**Note:**

- The broadcasting message reaches ***ALL*** *users who work with the same Database*, ***ECXEPT*** *the one who made the modification.*
- The best practice is to set **autoCommit** to *true* for the case of problem on the client site so that modified objects are committed (except appended/inserted objects which require an explicit commit)
- In case of starting exception "java.lang.Exception: [1][*null*]c:\JoeApp\OODB\joodb\OODBmix is locked by another NODE. @OODB:OODBmix" the locked file (suffix ***_Locked***) must be deleted on the node site (here: *OODBmix_Locked*). In this case you have 3 choices:
  - use the built-in button **FREE DB** of *JOODBServer* (see JOODBServer-MainScreen)
  - use the method ***freeDB(String dbName)*** of API *BeanServing* to free it (your own Server)
  - ask the admin on the Server-site to delete the file with the suffix *_Locked.*



```
|localhost:9876|Rename objName Cow of ODDB 'c:\UeeApp\ODDB\joodb\zoo' to Cattle
```

DBBroadcasting after RENAME: **Cow** to **Cattle** on Node9876



**Cattle** shows up after acknowledgement

The 2 config files for the aforementioned argument (see next page).

```
#---------------------------------------
# Joe Nartca (c)
# config9876.txt
# RUN: java ZOO config9876.txt or java ZOO
# (default: config9876.txt)
#
NODE=localhost
NODE_PORT=9876
BROADCAST_IP=224.0.0.3
BC_PORT=8888
#---------------------------------------
```

```
#---------------------------------------
# Joe Nartca (c)
# config10000.txt
# RUN: java ZOO config10000.txt
#
NODE=localhost
NODE_PORT=10000
BROADCAST_IP=224.0.0.3
BC_PORT=8888
#---------------------------------------
```

With the method "*broadcast(dbName, msg)*" of DBConnect every user can directly inform the others about his or her intention for his/her next move.

```
        /**
        Broadcasting a message to all users of the specified OODB
        @param dbName   String, name of OODB
        @param msg      String, message to be broadcast
        @return boolean true (done)
        @exception Exception from HOST
        */
        public boolean broadcast(String dbName, String msg) throws Exception
```

(more: see the APIs documents)

As aforementioned, the following **Fig. 5** shows you how the same ZOO application runs with different config file: *config9867.txt* as **local app** and *config10000.txt* as **remote app**.

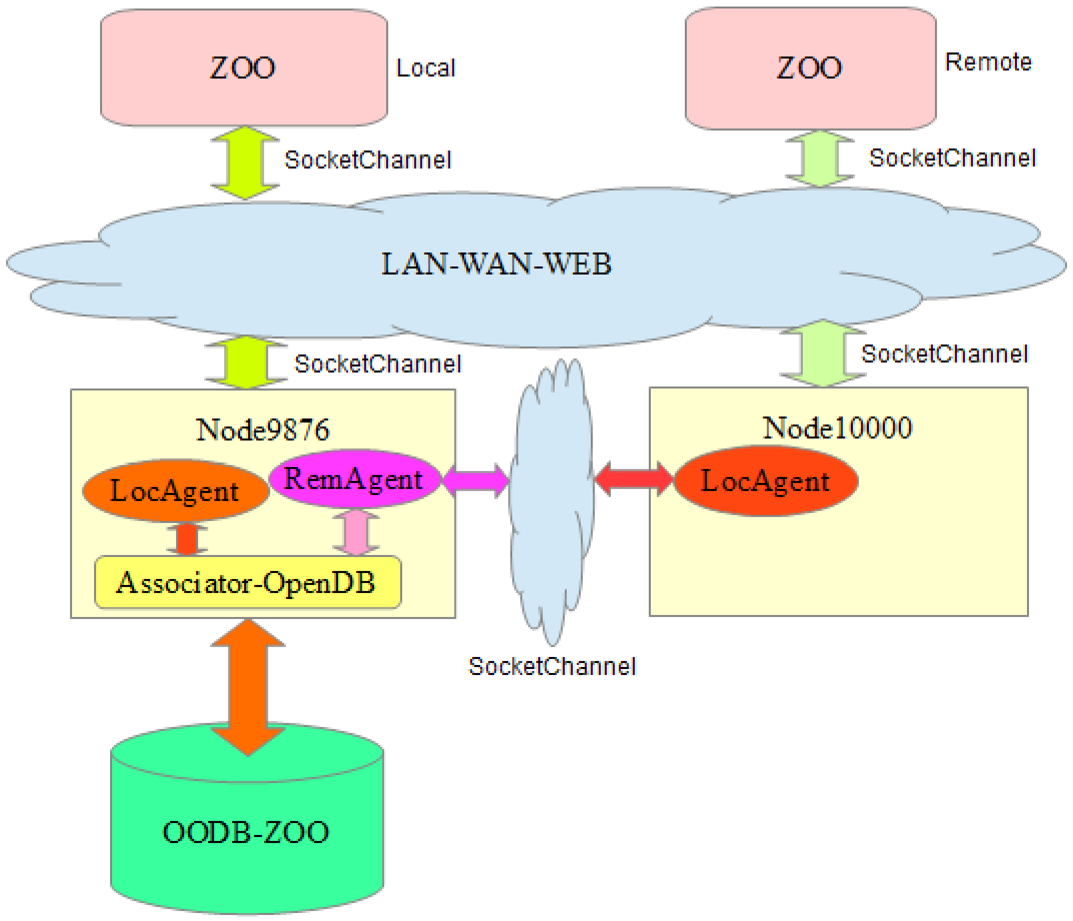


**Fig.5**

*LocAgent* on the *Node1000*0 initiates a communication with *Node9876*. *Node9876* receives the request and its internal Agent launches a *RemAgent* which works *exclusively* with *LocAgent* of *Node10000* and is responsible for the communication, and for the DB accesses. *RemAgent9876* is for *LocAgent10000* "the *requested* OODB ZOO". OODB-ZOO is shared by *LocAgent* and *RemAgent*.

## 3.2 ServerNode

Only the JOODB-API *BeanServing* and a *oodbConfig* file are needed for a functioning OODBServer. Example:

```
import oodb.*;
public class Server  {
   public Server() {
     try {
       System.out.println("CRTL-C to quit");
       // Config file, userID and userPassword
       BeanServing bs = new oodb.BeanServing("oodbConfig.txt", "joe", "congdongjava");
       bs.setLogger(System.out); // use the console as the error logfile
       System.out.println(bs.getMessage()+"\n");
     } catch (Exception e) {
       e.printStackTrace();
     }
   }
   //
   public static void main(String... a) throws Exception {
     new Server( );
   }
}
```

As you see, a *NodeServer development* is a simple and straight-forward business. However, you can beautify your own server in SWING or JFX.

However, OODB server always needs an *userlist* that bases on the provided API *UserList* and with it you can create a NodeServer and set up your own node environment.

| Constructor and Methods | comment |
|---|---|
| `public UserList(String fName) throws Exception` | Constructor |
| `public int isUser(String uid, String password)` | Check for an user |
| `public boolean addUser(String uid, String password, int right)` | Create a new user |
| `public int updateUser(String uidOld, String passwordOld, String uidNew, String passwordNew)` | UpdateUser |
| `public int updateUser(String uidOld, String passwordOld, String uidNew, String passwordNew,int priv)` | UpdateUser with new access rights |
| `public boolean deleteUser(String uid, String pw)` | Delete a user |
| `public static String encrypt(String s)` | Internal encrypting |
| `public static String decrypt(String s)` | Internal decrypting |

(more: see the API-Documents)

If *BeanServing* is successfully launched it acts a *OODBServer*. Incoming Client is firstly verified for its authenticity (ID and Password) before it hands out the client to an *internal agent* that decides whom who works with the client: *Local Agent* (*LocAgent)* or *Remote Agent* (*RemAgent.)* Depending on the *dbName LocAgent* (***no*** *prefix*) or *RemAgent* (***with*** *prefix* **[host:port]**) is responsible for the OODB accesses. *RemAgent* is initiated by *LocAgent* and works exclusively for this *LocAgent*, If the *requested OODB* is **new** (i.e. not in Cache) *LocAgent/RemAgent opens* it as *owner* for the client and caches it, or shares the cached OODB. *OpenDB*, opened by *LocAgent/RemAgent*, processes the request and hands out the *result* to its owner who wraps it in *POJO-Result* and sends packet to its client. *RemAgent* is run by another *BeanServing* on the node of **host:port.**

If you design your own OODB Server you need only to care about the authentication (API *UserList*) and the server (API *BeanServing* -see JOODBServer). Some *useful tools* are bundled in the package for the ease of your development (e.g. Authenticate, Login in SWING). The API-core is:

JOODB core for a Node:

1. BeanServing: Server API
2. DBBean: Superclass of LocAgent and RemAgent
3. Broadcaster: Node broadcaster
4. DBParm: POJO between BeanServing and DBBean
5. LocAgent: Local Agent that interfaces directly with the underneath OODB-DMS (OpenDB)
6. RemAgent: Counterpart of LocAgent on remote site
7. Result: POJO for the result delivered by OODB-DMS

JOODB DMS core

1. Associator: The Keys container
2. OpenDB: The data part (serialized objects)
3. DBBroadcaster: Modification Broadcaster

JOODB Authentication

1. Authenticate (SWING), interface JDialog.
2. UserList (base)

BeanServing can be instantiated by 2 following constructors and :

```
        /**
        Constructor, API using "oodbConfig.txt" default
        @param uID      String, UserID
        @param password String, UserPassword
        @exception      Exception thrown by JAVA
        */
        public BeanServing(String uID, String password) throws Exception {
                beanServing("oodbConfig.txt", uID, password);
        }
        /**
        Constructor, API using the given configuration file
        @param config   String, Config-filename for this Seever Node
        @param uID      String, UserID
        @param password String, UserPassword
        @exception      Exception thrown by JAVA
        */
        public BeanServing(String config, String uID, String password) throws Exception {
                beanServing(config, uID, password);
        }
```

The first constructor always looks for the ***oodbConfig.txt*** in the current node paths. The methods allow the users to access or to view the *running state* of *BeanServing* (e.g. number of active OODB, shared or opened, number of *LocAgents*, etc.).

**Caution:** For your own *BeanServing* Node-Server you must develop an app that runs the API *UserList.* It lets you set up your own **userlist** required by *BeanServing*. UserList data are encrypted and *2 static methods* are available for encrypting and decrypting.

The meanings of **oodbConfig.txt**

| Keyword | Explanation and example |
|---|---|
| NODE | OODB node. A combination of HostName (or IP) and Port.<br>Example: NODE = localhost:9876 |
| sMAX | Max buffer in Bytes for the ServerSocketChannel, default: 4096 |
| aMAX | Max buffer in Bytes for LocAgent and RemAgent, default: 65536 |
| qMAX | Max waiting clients in a queue. Default: 512 (FixedPool. See pTYPE) |
| pTYPE | Internal ThreadPool model (0: FixedPool, 1: CachePool, 2: StealingPool).<br>Default: 0 |
| LATENT | The latency time for Server (microSeconds), default 10 |
| SYNCTIME | Synchronizing interval of OODB in seconds, default: 1800 or 30 minutes |
| MONITOR | Broadcaster flag (0: disable, 1: enable), default 1 |
| BCINTERVAL | Broadcasting interval (delay) in milliseconds, default: 500 |
| MULTICAST_PORT | Broadcaster port, default: 8888 |
| MULTICAST_ADDR | Broadcaster IP, default: 224.0.0.3 |
| CLASSPATH | (remote or local) Client classpaths, separator is an acclamation ! |
| UNIX_PATH/WIN_PATH | The OODB path according to OS convention |
| UNIX_USERLIST/WIN_USERLIST | The path to the userlist (UserID and Password) |

**<u>Note:</u>** Any *missing* keyword will trigger BeanServing to throw a *NullPointerException.*

*Exception* thrown by *OpenDB* is always preceded by a *number* enclosed by *Square-brackets* **[*n*]**. Here is the list of all Exceptions.

```
[1](dbName) is locked by another NODE.
[2](objName) existed already!
[3](objName) isn't owned by (owner)
[4]Unknown objName:(objName)
[5]Unable to rollback. Unknown objName:(objName)
[6](priv):Insufficient UserPrivilege!
[7]Unknown/invalid command:(command_String)
[8]Invalid or Unprivileged:(userID) or (password)
[9]Negative Cluster!!!
[10]Unknown/Invalid ClusterNo.(clusterNo)
[11]Unable to access Remote Node:(node)
[12]Unable to access OODB from (node)
[13]Exception @(node):
   (ErrorMessage)
[14]Unable to start DBListener
[15]DBListenerException: (ErrorMessage)
[16]Unknown or Unopened DB:(dbName)
[17]SyntaxError: Missing DB-Name !
[18](Java thrown ExceptionMessage)
[19]Modified an uncommitted Object @(objName)
[20]DBBrocaster for (dbName) is diabled !
[21]Invalid objName:(objName)
[22]Invalid ClassName
[23]Invalid Pattern
[24]Invalid VariableName
[25]OODB must be opened before setting AutoCommit
```

Term enclosed by brackets is the name of the "problem". Example: ObjName is *Cow*, the message could be "[2] *Cow* existed already". Note: ErrorMessage is here the Exception message thrown by JAVA-JVM (e.g. *NullPointerException*, etc.)

## ThreadPool

BeanServing runs its threads (Agent, LocAgent and RemAgent) in a ThreadPool which is specified in the *oodbConfig.txt.*

```
# sMAX ServerSocketChannel MemoryBuffer
# aMAX Agent MemoryServer
# qMAX Max waiting queues for FixedPool
# pTYPE = 0 FixedPool,
#         1 CachedPool,
#         2 StealingPool
#
aMAX = 65536
sMAX = 4096
qMAX = 512
pTYPE = 0
```

## FixedPool (pType 0)

If client applications are "*long-living*" their LocAgents or RemAgents live with them. *FixedPool* is the best choice. Depending on the "number" of clients the **qMax** should be varied to match with the need. Too big is sometime *disadvantageous*. Default is 512 . That means if the number of clients exceeds 512 the incoming clients must wait until one of the running quits. To find out what number is the best you have to do some *statistics* and measure the *averaged lifespan* of client applications.

## CachedPool  (pType 1)

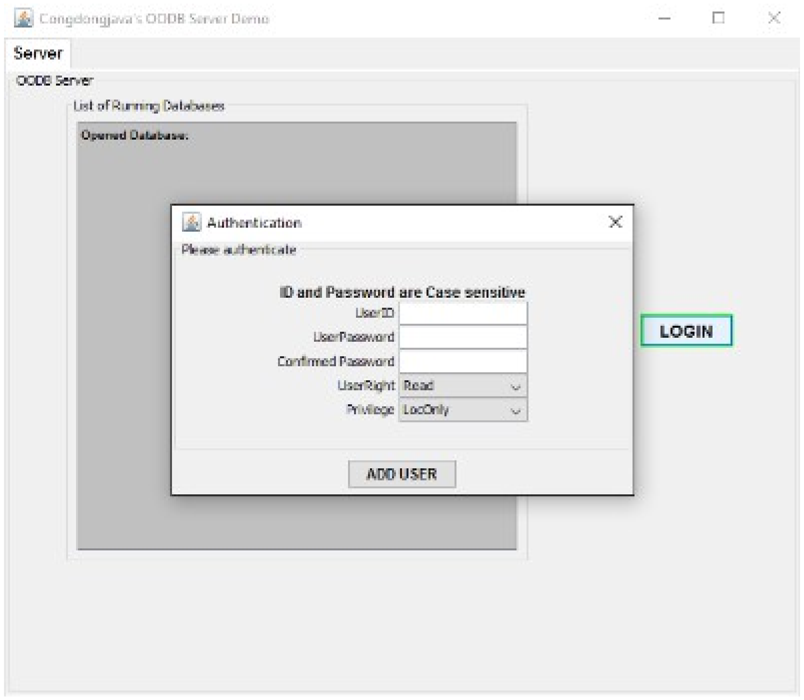
*CachedPool* is a *FixedPool* with an **unlimited number** of clients. CachedPool is only fine if the clients are "relatively *short-living*". However "short-living" is a relative perception, hence you have to tinker or to experience with the last so that you could achieve the most optimal decision.

## StealingPool  (pType 2)

*StealingPool i*s a special form of *ForkJoinPool*. JVM is just a Virtual Machine that runs on a Guest-OS and that is the *main drawback of parallelism with JAVA:* JVM must relies on its Guest-OS and because JVM "shields" the users from direct accessing to the underlying OS JVM parallelism is not optimally implemented. Direct Control (e.g. exception within the pool) is almost impossible. Thing such as CPU (or processor) affinity is not available for JAVA. Hence, StealingPool is only the best choice if client applications are *really short-lived*.

## JOODBServer

JOODB bundles the prototype *SWING-JOODBServer* (*JOODBServer, Login, Maint, SysMon, Tune*). If you use *JOODBServer* and you want your ***own userlist*** you need only to delete the *userlist file* (in the *joodb* folder) and start *JOODBServer* and click *Login*. *JOODBServer* prompts the "creating" menu (see screenshot) so that you can establish your *own new userlist* (see screenshots.) The login ID is **joe** and the password is **joenartca** (both in *lowercase.*)


Congdongjava's OODB Server Demo

Server

OODB Server

List of Running Databases

Opened Database:

Authentication

Please authenticate

ID and Password are Case sensitive

UserID:

UserPassword:

Confirmed Password:

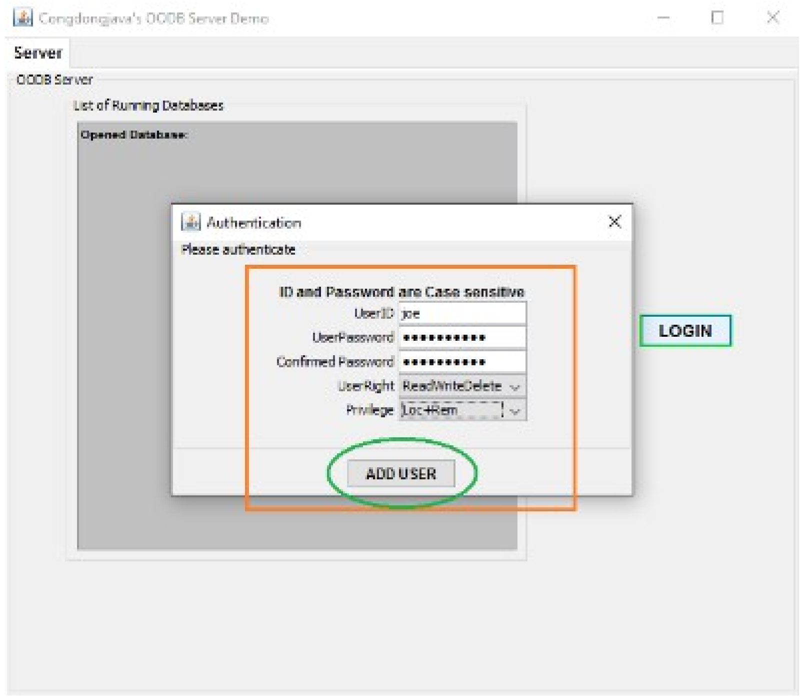
UserRight: Read

Privilege: LocOnly

ADD USER

LOGIN

The prompt for creation of *new userlist*


Congdongjava's OODB Server Demo

Server

OODB Server

List of Running Databases

Opened Database:

Authentication

Please authenticate

ID and Password are Case sensitive

UserID: joe

UserPassword: ••••••••••

Confirmed Password: ••••••••••

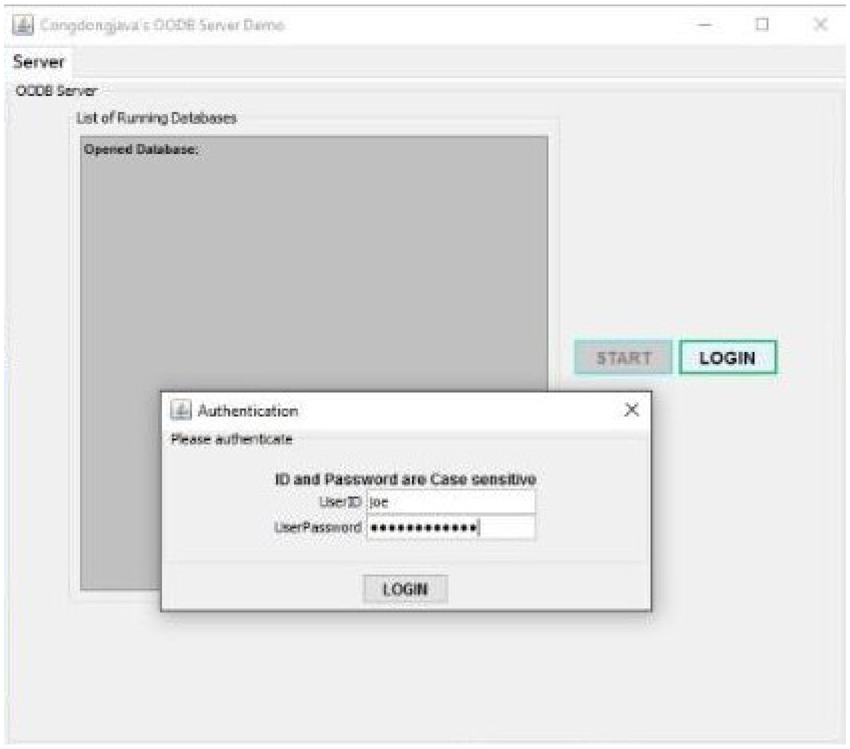
UserRight: ReadWriteDelete

Privilege: Loc+Rem

ADD USER

LOGIN

With *ADD USER* a new *userlist* is created.



Opened Database:

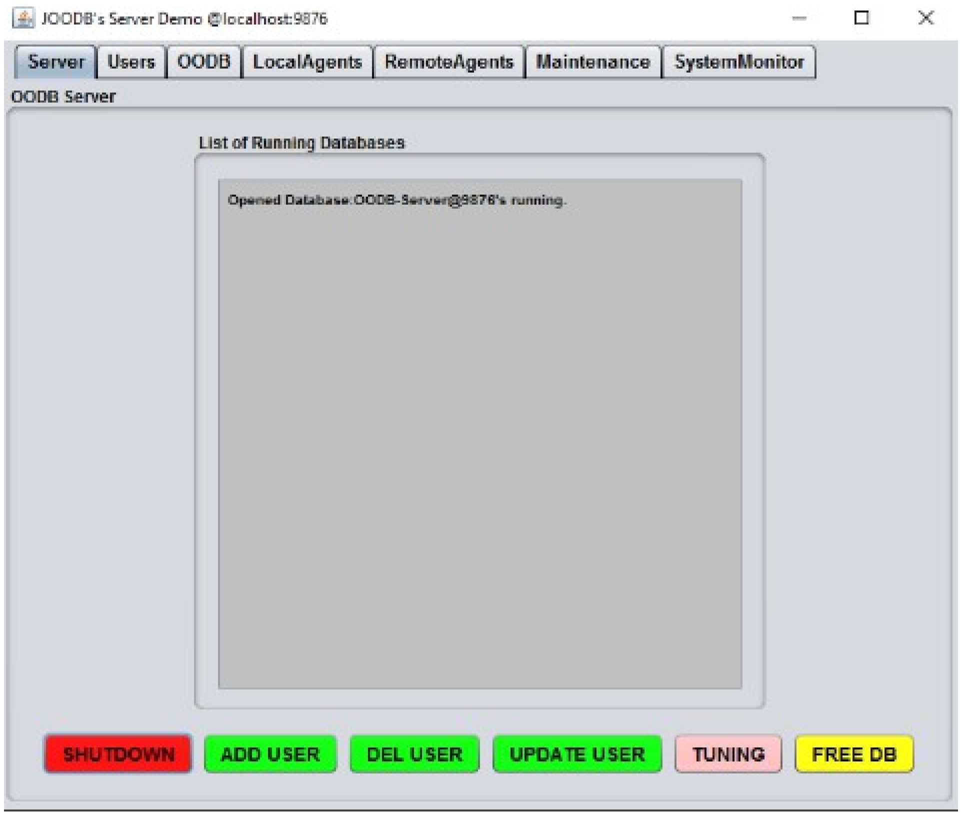
Please authenticate

ID and Password are Case sensitive

UserID joe

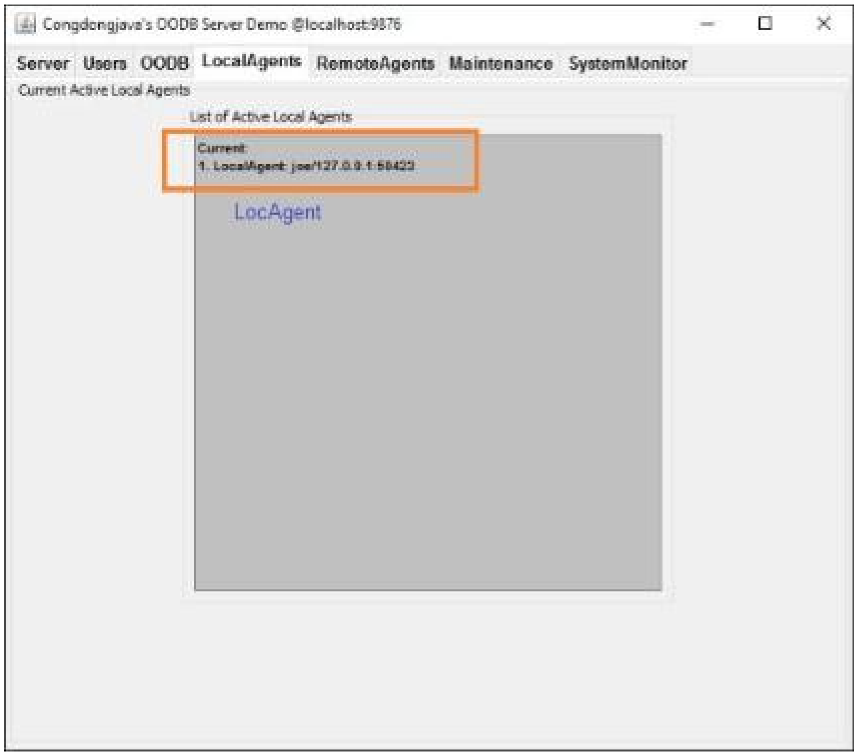
UserPassword •••••••••••••

Then the “usual” LOGIN screen



Opened Database:OODB-Server@9876's running.

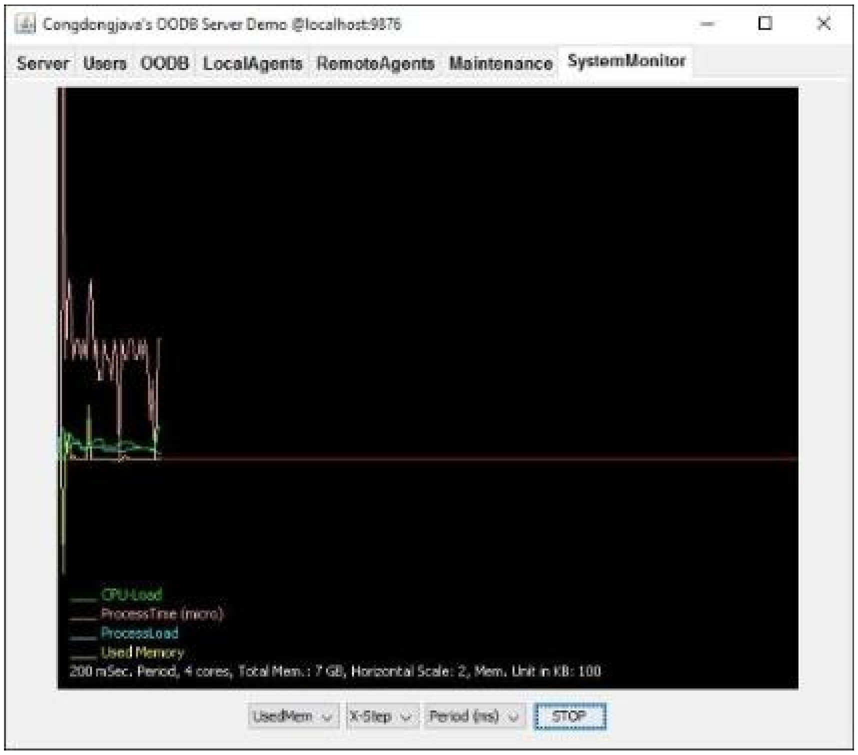
The main screen



Current Active Local Agents

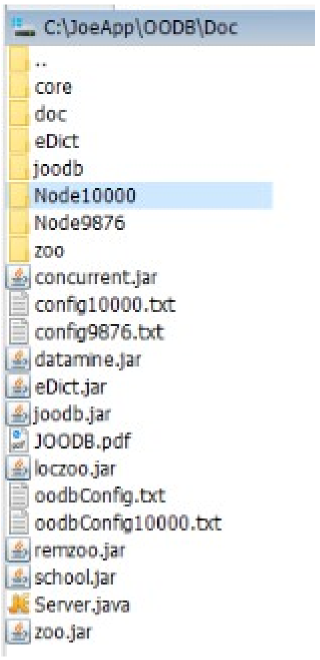
List of Active Local Agents

Current:
1. LocalAgent: joe/127.0.0.1:50423



200 mSec. Period, 4 cores, Total Mem.: 7 GB, Horizontal Scale: 2, Mem. Unit in KB: 100

The package includes:



```
C:\JoeApp\OODB\Doc
..
core
doc
eDict
joodb
Node10000
Node9876
zoo
concurrent.jar
config10000.txt
config9876.txt
datamine.jar
eDict.jar
joodb.jar
JOODB.pdf
loczoo.jar
oodbConfig.txt
oodbConfig10000.txt
remzoo.jar
school.jar
Server.java
zoo.jar
```

- **core:** JOODB sources
- **doc:** JAVA-styled documents
- **joodb:** some OODB for the testing purposes
- **Node9876**: DataMining and Concurrency test sources
- **Node10000:** some more test programs (remote object, etc.)
- **zoo:** the ZOO basic test program (basic test remote and local)
- **joodb.jar**: JOODB APIs and executable JOODBServer (***java -jar joodb.jar***)
- some documents and *config files* and the simple Server source.
- QA executable jar files (*concrrent.jar, datamine.jar, zoo.jar, school.jar, loczoo.jar, remzoo.jar*)

How to run the jar files ?

First sou have set the CLASSPATH that includes **joodb.jar.** Example on Windows

```
set CLASSPATH = c:\myOODB\joodb.jar;%CLASSPATH%
```

Then open a CMD widows and start JOODB server (or ***double-click*** at *joodb.jar*) as following

```
javaw -jar joodb.jar [your oodbConfig file]
```

javaw runs "off-line" like LINUX/UNIX as "java -jar joodb.jar **&**". The ampersand sends the execution into background (off-line).

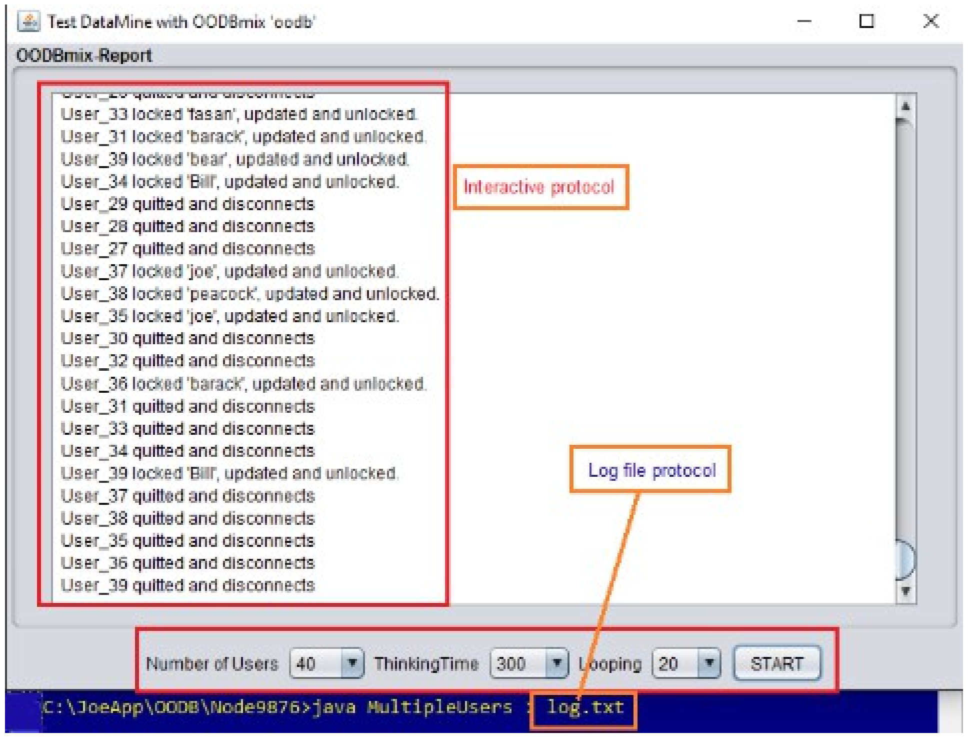
- The *multiple-users* ***concurrent.jar*** simulates **n users** who are either REMOTE (node 10000) or LOCAL (node 10000). Therefore *it requires the second OODB server* that runs at **port 10000**.
- For some useful works (e.g. comparison of field name, looking for pattern) ***datamine.jar*** is dedicated for that and it shows you how the *API DataMine* is coded.
- ***loczoo.jar*** and ***remzoo.jar*** run against the database *OODBmix* and shows you how different objects (*Animal* and *Member*) and a simple String can be in the same database.
- ***school.jar*** and ***eDict.jar*** are two additional examples showing you how JOODB works with *Tables* and *JFX*

All the sources are included. They are NOT organized as "projects" that you may usually see in your IDE (Netbeans or Eclipse).

As mentioned at begin, JOODB core is neat and simple. The core consists of:

- BeanServing (server),
- DBParm (POJO, server),
- DBBean (server),
- LocAgent (server),
- RemAgent (server)
- UserList (server and client),
- Result (POJO, server and client),
- OOClassLoader (server and client),
- DBConnect (client),
- DataMine (client),
- Associator (DataManagementSystem or DMS),
- OpenDB(DataManagementSystem or DMS),
- Broadcaster (thread, server),
- DBListener (thread, client),
- OODBListener (interface, client),
- the rest is just additional API for your development work. They are independent from the core.

Some more screenshots



```
User_33 locked 'fasan', updated and unlocked.
User_31 locked 'barack', updated and unlocked.
User_39 locked 'bear', updated and unlocked.
User_34 locked 'Bill', updated and unlocked.
User_29 quitted and disconnects
User_28 quitted and disconnects
User_27 quitted and disconnects
User_37 locked 'joe', updated and unlocked.
User_38 locked 'peacock', updated and unlocked.
User_35 locked 'joe', updated and unlocked.
User_30 quitted and disconnects
User_32 quitted and disconnects
User_36 locked 'barack', updated and unlocked.
User_31 quitted and disconnects
User_33 quitted and disconnects
User_34 quitted and disconnects
User_39 locked 'Bill', updated and unlocked.
User_37 quitted and disconnects
User_38 quitted and disconnects
User_35 quitted and disconnects
User_36 quitted and disconnects
User_39 quitted and disconnects
```

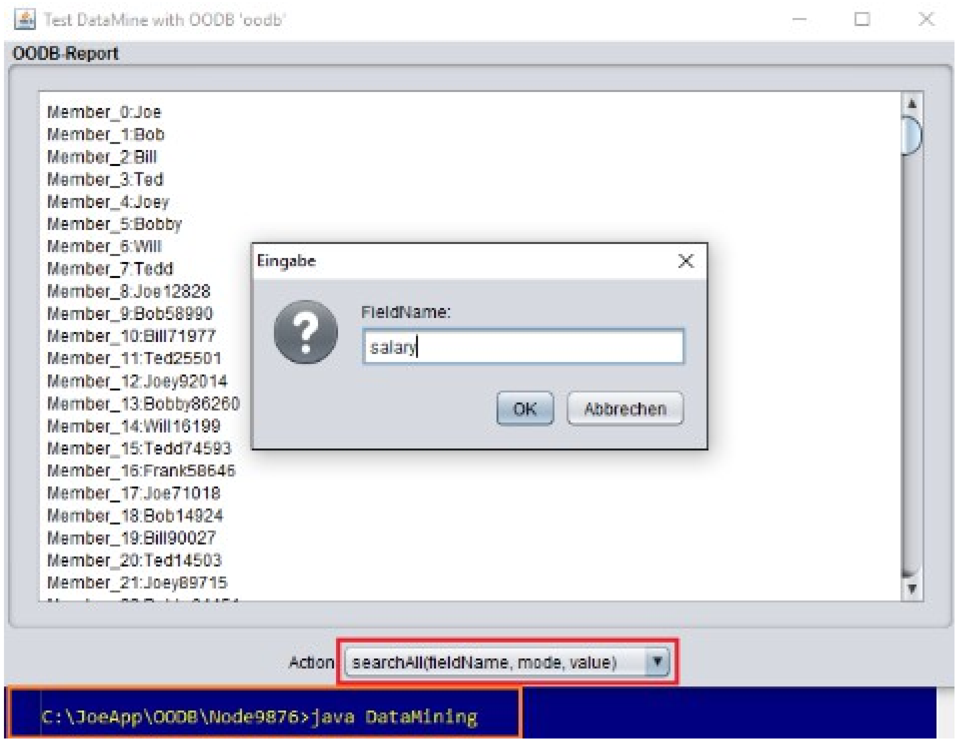
Number of Users 40 ThinkingTime 300 Looping 20

```
C:\JoeApp\OODB\Node9876>java MultipleUsers > log.txt
```

concurrent.jar (MultipleUsers)

and the **log.txt** looks like this

```
...
User_5 locked 'vladimir', updated and unlocked.
User_6 locked 'vladimir', updated and unlocked.
User_0 locked 'Joe', updated and unlocked.
User_2 locked 'macaw', updated and unlocked.
User_7 locked 'Joe', updated and unlocked.
User_8 locked 'macaw', updated and unlocked.
User_3 locked 'fasan', updated and unlocked.
>>>Thread_4 locked 'vladimir' but failed
User_9 locked 'vladimir', updated and unlocked.
User_1 locked 'fasan', updated and unlocked.
User_5 locked 'joe', updated and unlocked.
User_0 locked 'vladimir', updated and unlocked.
User_6 locked 'Joe', updated and unlocked.
User_2 locked 'Ted', updated and unlocked.
User_7 locked 'macaw', updated and unlocked.
User_8 locked 'Bill', updated and unlocked.
User_3 locked 'joe', updated and unlocked.
>>>Thread_1 locked 'barack' but failed
User_9 locked 'barack', updated and unlocked.
User_4 locked 'Bill', updated and unlocked.
User_5 locked 'fasan', updated and unlocked.
User_6 locked 'vladimir', updated and unlocked.
...
```



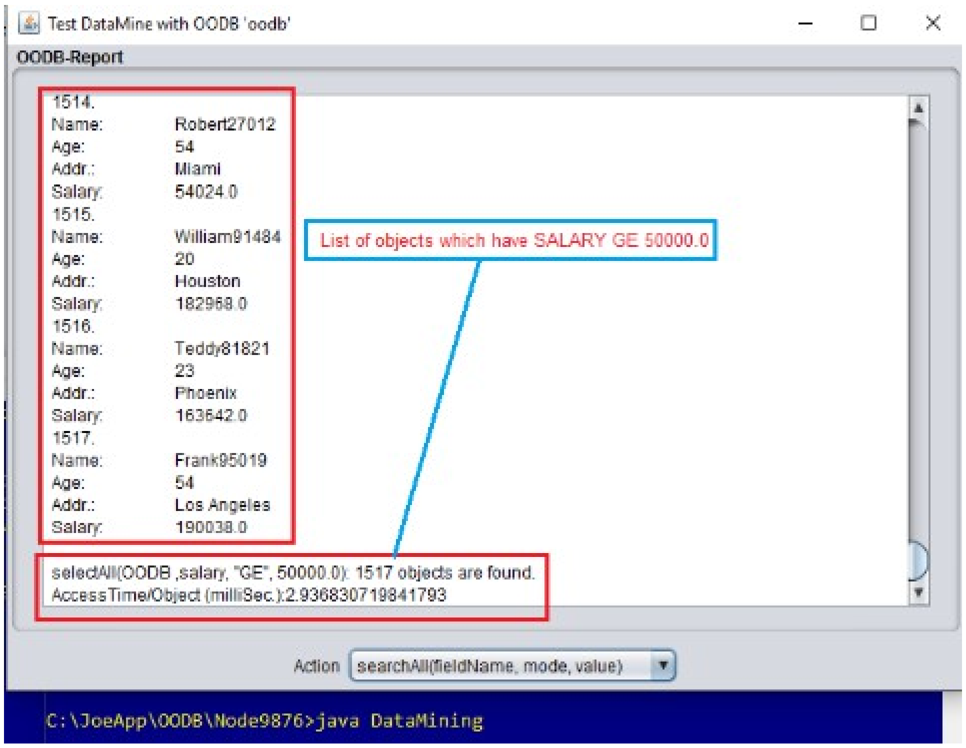
```
Member_0:Joe
Member_1:Bob
Member_2:Bill
Member_3:Ted
Member_4:Joey
Member_5:Bobby
Member_6:Will
Member_7:Tedd
Member_8:Joe12828
Member_9:Bob58990
Member_10:Bill71977
Member_11:Ted25501
Member_12:Joey92014
Member_13:Bobby86260
Member_14:Will16199
Member_15:Tedd74593
Member_16:Frank58646
Member_17:Joe71018
Member_18:Bob14924
Member_19:Bill90027
Member_20:Ted14503
Member_21:Joey89715
```

FieldName: salary

Action: searchAll(fieldName, mode, value)

```
C:\JoeApp\OODB\Node9876>java DataMining
```

datamine.jar or (DataMining)



Test DataMine with OODB 'oodb'

OODB-Report

```
1514.
Name:       Robert27012
Age:        54
Addr.:      Miami
Salary:     54024.0
1515.
Name:       William91484
Age:        20
Addr.:      Houston
Salary:     182968.0
1516.
Name:       Teddy81821
Age:        23
Addr.:      Phoenix
Salary:     163642.0
1517.
Name:       Frank95019
Age:        54
Addr.:      Los Angeles
Salary:     190038.0
selectAll(OODB ,salary, "GE", 50000.0): 1517 objects are found.
AccessTime/Object (milliSec.):2.936830719841793
```

List of objects which have SALARY GE 50000.0

Action: searchAll(fieldName, mode, value)

```
C:\JoeApp\OODB\Node9876>java DataMining
```

the query: selectAll(OODB, slary, “GE”, 50000.0)

Joe