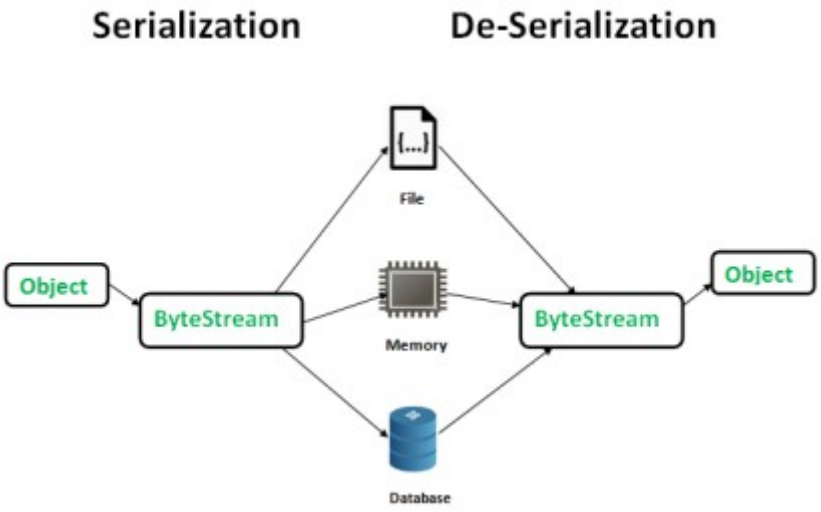
Java Serialized Object Database JODB

1. Java Serialized Object

If a Java object is to be stored in a file for later use, it must be formatted so that its data can be retrieved correctly later. This process is generally called serialization, and the stored object is called a serialized object. Java serialized objects can be implemented using the Serializable interface. Example:

```
public class Employee implements Serializable {
  private static final long serialVersionUID = 1234L;
  public People(String name, int age, String address,
                String qualification, double income, String pic) {
    this.age = age;
    this.pic = url;
    this.name = name;
    this.income = income;
    this.address = address;
    this.qualification = qualification;
  }
  private int age;
  private double income;
  private String name, pic, address, profession;
  public String toString() {
    return "Name: "+name+", Age: "+age+", Addr.: "+address+", Income: "+income+
           ", Profession: "+profession;
  }
}
```

I assume that you are familiar with the implementation of Java interfaces and in particular with the Serializable interface, so let's skip explaining some details and focus on the fact that such a serialized object is usually a so-called POJO (Plain Old Java Object) which generally consists of data and some methods. The data are similar to DB data: strings and numbers. And the methods are generally uncomplicated.



(picture: Geeksforgeeks)

The operation (object to storage medium or vice versa) is simple in JAVA. However, this operation requires the physical presence of a bytecode class of the object. And this complicates the serialization process because the class of the object must be known on each side in its CLASSPATH as required by ObjectOutputStream and ObjectInputStream. In

a large LAN/WAN network environment, such redundancy of bytecode classes is not only difficult to maintain but also error-prone (e.g. actual version). In short:

- Serialization: ObjectOutputStream with writeObject() method.
- Deserialization: ObjectInputStream with readObject() method.

The problem in a network environment is not only the presence of a bytecode class on each node, but also the network traffic load when looking for a specific object among many same objects with different content and the bytecode class of these objects is not available. The only solution is to send all objects to the requester, but this may cause a crash or a standstill in network traffic. This is probably the main reason why there is no JAVA database for serialized objects, although Serialized Object DB has many advantages. For example, each serialized object is an independent entity in itself, so there is no need to implement atomicity (ACID) or Common Schema Definition Language (CSDL) for the data description because serialized objects' data and field names (variables) are independent of the other objects' and do not need to be normalized or to define the data relationship as is usually the case with RDB data.

2. Java Serialized Object Database – JODB

The thought of having to transfer a huge pile of serialized objects (or files) to the requester's site is very intimidating when the requester is only interested in one specific serialized object among them. As mentioned earlier, the server site has either all bytecode classes of all different serialized objects or a tool that can be used to inspect the contents of serialized objects. The de-serialized process using ObjectInputStream-readObject() recreates an instance of a serialized object with full functionality (with all executable methods). And this, as mentioned above, requires the presence of a bytecode class. However, if we closely examine the JAVA serialization protocol and analyze the serialized objects in the file, we can see that the data (e.g. strings, numbers, etc.) are in their frozen state and thus can be read and used. There is no need to recreate a full instance for a serialized object because the methods are useless on the server side. An example of the simple Java Employee object about a description of an employee (which is usually used in RDB):

- Name (String)
- Age (int)
- Income (double)
- Address (String)
- Profession (String)
- Image (String of an URL to the image or image file)

The getdata, print and display methods are useful on the client side, but useless on the server side. Thre following table has three parts:

1. Java Object, an implementation of Serializable
2. serialized object file (in hex format)
3. Analyse of serialized object of 2) with explanation

```
public class Employee implements java.io.Serializable {
  private static final long serialVersionUID = 1234L;
  public Employee(String name, int age, String address, String profession, double income, String url) {
    this.age = age;
    this.url = url;
    this.name = name;
    this.income = income;
    this.address = address;
    this.profession = profession;
  }
  private int age;
  private double income;
  private String name, url, address, profession;
  ...
  public String toString() {
    ...
  }
  ...
  public void print() {
    ...
  }
  public void picture(JFrame jf) {
    ...
  }
}
```

```
0        4        8        C        10
ACED0005 73720008 456D706C 6F796565 00000000 | .... sr.. Empl oyee .... | 0
000004D2 02000649 00036167 65440006 696E636F | .... ...I ..ag eD.. inco | 1
6D654C00 07616464 72657373 7400124C 6A617661 | meL. .add ress t..L java | 2
2F6C616E 672F5374 72696E67 3B4C0004 6E616D65 | /lan g/St ring ;L.. name | 3
71007E00 014C000A 70726F66 65737369 6F6E7100 | q.~. .L.. prof essi onq. | 4
7E00014C 00037572 6C71007E 00017870 00000049 | ~..L ..ur lq.~ ..xp ...I | 5
40ED4C00 00000000 74000E46 72616E6B 66757274 | @.L. .... t..F rank furt | 6
20612E4D 2E74000E 4A6F6520 4F6C6420 4765657A |  a.M .t.. Joe  Old  Geez | 7
65727400 184D5363 2E20696E 20436F6D 70757465 | ert. .MSc . in  Com pute | 8
72205363 69656E63 65740007 4A6F652E 6A7067   | r Sc ienc et.. Joe. jpg  | 9
```

```
0        4        8        C        10
ACED0005                                  STREAM_MAGIC (ACED), STREAM_VERSION (0005)
         7372                             TC_OBJECT TC_CLASSDESC
             0008 456D706C 6F796565       len:0008 string:Employee                    <---Object name
                                  00000000 serialized ID
000004D2
         02                               SC_SERIALIZABLE
           0006                           # fields: 0006                             <---6 fields
               49 00036167 65             I(nteger) len:0003 string:age               <---int
                             440006 696E636F D(ouble) len:0006 string:income           <---double
6D65
    4C00 07616464 72657373                L(for Object) len:0007 string:address       <---string
                           7400124C 6A617661 TC_STRING len:0012 name:Ljava/lang/String;
2F6C616E 672F5374 72696E67 3B
                             4C0004 6E616D65 L(for Object) len:0004 string:name        <---string
71007E00 01                               TC_REFERENCE null baseWireHandle
           4C000A 70726F66 65737369 6F6E     L(for Object) len:000A string:profession  <---string
                                     7100 TC_REFERENCE null baseWireHandle
7E0001
      4C 00037572 6C                      L(for Object) len:0003 string:url           <---string
                    71007E 0001           TC_REFERENCE null baseWireHandle
                               7870       TC_ENDBLOCKDATA TC_NULL
                                    00000049 73                                  <----age value
40ED4C00 00000000                         60000                                  <----income value
                  74                      TC_STRING
                    000E46 72616E6B 66757274 len:000E string:Frankfurt a.M.      <----address content
20612E4D 2E
           74                             TC_STRING
             000E 4A6F6520 542E2053 63687761 len:000E string:Joe Old Geezer      <----name content
727A
    74                                    TC_STRING
      00 184D5363 2E20696E 20436F6D 70757465 len:0018 string:MSc. in Computer Science <---Profession content
72205363 69656E63 65
                    74                    TC_STRING
                      0007 4A6F652E 6A7067 len:0007 string:Joe.jpg                <---url content
```

Based on the JAVA serialization protocol, the hex analysis clearly shows us that the data of the object fields (variables) are available if we only have a tool to read it:

- age: 73
- income: 60000
- name: Joe Old Geezer
- address: Frankfurt a.M.
- profession: M.Sc. in Computer Science
- url: Joe.jpg

Using such a tool, we can design and implement a Java Object Database (JODB). And the tool is ODBObjectView, which allows you to access the (initialized) fields by field names.

Let's talk about this tool. As we know, most of the JAVA APIs are serializable and they are in very large numbers. Besides the primitives (int, double, long, String, etc.), custom serializable objects are usually based on Java APIs like ArrayList, BigInteger, BigDecimal, etc. To view and access the data of the fields, the tool must be able to initialize or instantiate these fields correctly. In our Employee example, it is relatively easy with int, double, and String. It gets more complicated with Array and APIs. Therefore, a tool should limit itself to the manageable size with the most commonly used APIs, primitives (including string) and primitive arrays with a maximum of three dimensions. The supported APIs are ArrayList, BigInteger, BigDecimal, Long, Double, Float, Integer and Short. And I believe that a JODB can be easily designed and implemented with this limited design. In a database, an object is usually identified and accessed by an unique key. The key can be a BigInteger, a long or a String (of any length).

With this idea I have designed and implemented a distributed database for serialized objects and the tool is an important element of the access design. The database is in a separate server (or distributed as cluster across multiple servers on network) and the clients (with their local bytecode classes of serialized objects) can request an object with certain characteristics without the server overloading the network with objects. As with RDB, Java objects are accessed via an unique key (and only one).

The keys are associated with objects and are a part of objects. For example, the object Employee for a person named John Doe has a key, which can be: a personal ID (a long or a BigInteger) or as a String of the name as "John Doe" and the data are accessed by this "John Doe" key and stored in JODB as a byte array (see ODMS)

3. JODB Architecture and Structure

The following images show you how JODB position itself in a Network environment and how JODB communicates between client apps and JODB servers.

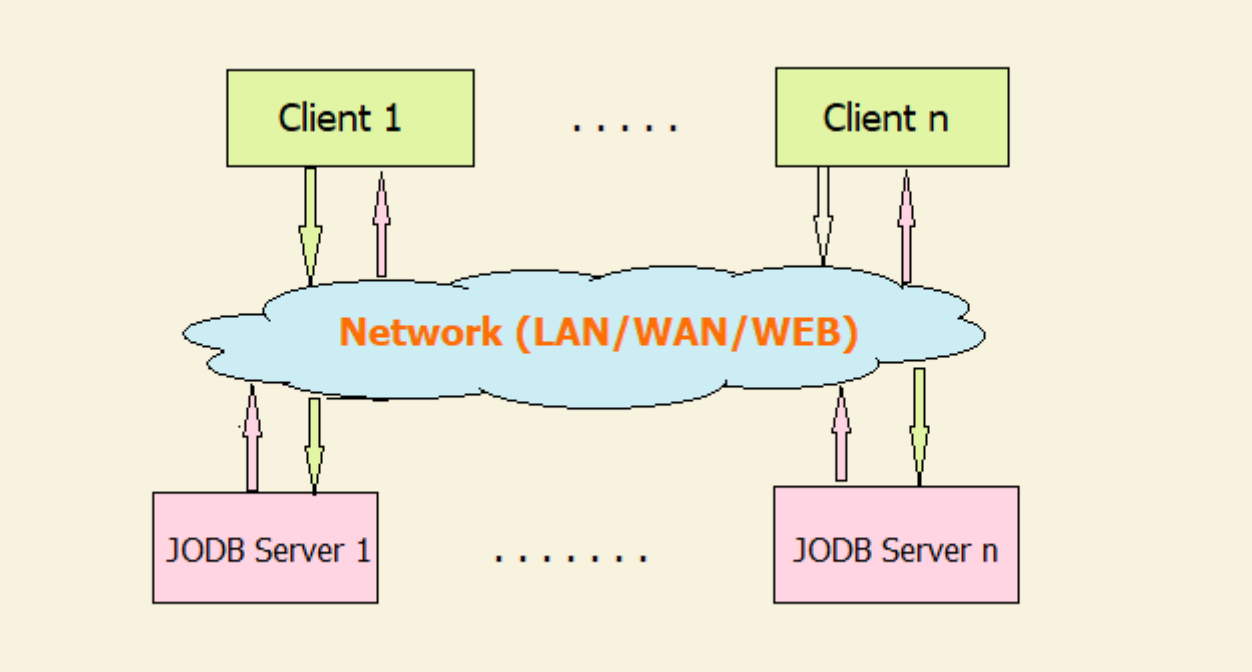


Figure 1

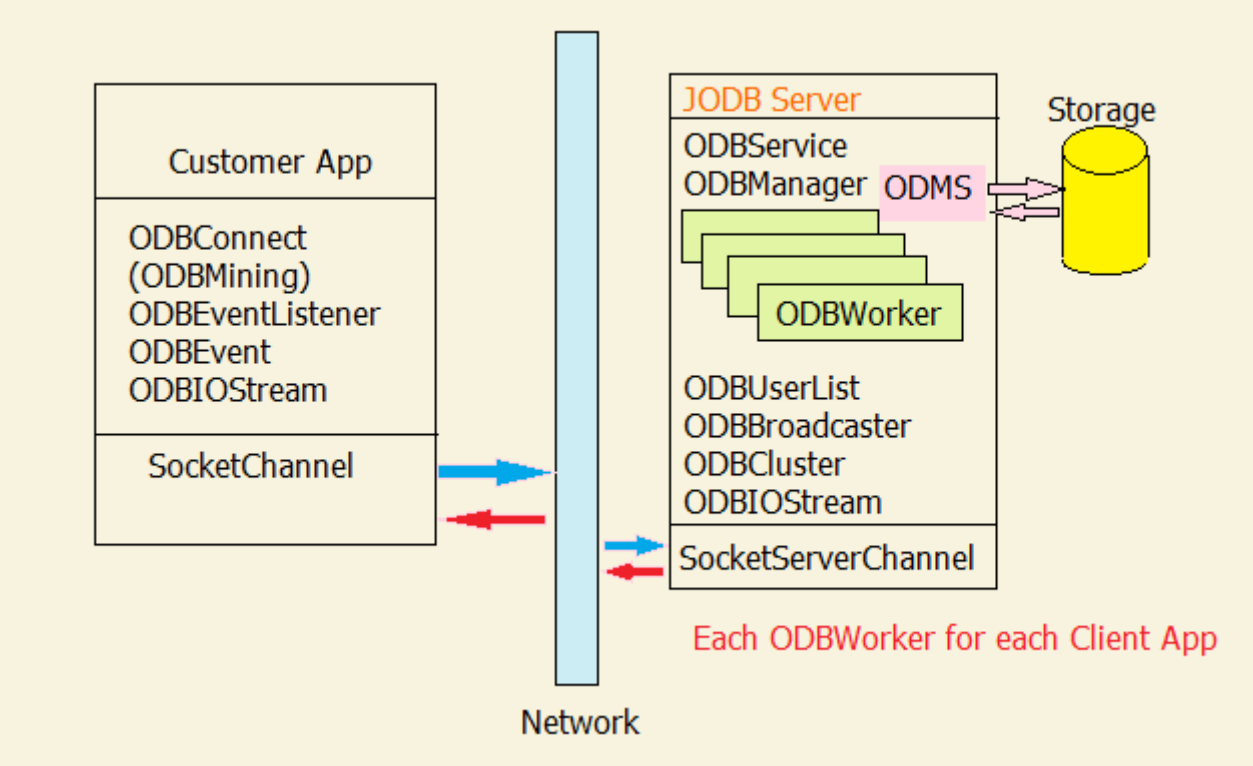


Figure 2

Figure 1 shows you how JODB clients access JODB servers. JODB clients access only one JODB server known to them directly via hostname and port. If the JODB is distributed across different JODB servers and these servers are online (active), the entire JODB is

automatically available to the clients. Otherwise, only the JODBs are part of the online servers. This process is possible (Figure 2, left side) because each JODB client is served by an ODBWorker, which automatically searches the network for JODB servers that are online and then establishes a connection via an ODB Agent, which represents it as its proxy on this JODB server. The connection is done via SocketChannel (java.nio.channels) to ensure fast communication. The procedure is as follows:

- ODBConnect with hostname, port, user ID and password (ID and password are encrypted by a private encryption/decryption algorithm). If the connection is successful, then it can issues a connect() to an ODB via its name (dbName),
- client apps communicate with the JODB server via the provided ODBConnect methods like add, delete, read, update etc. Or for the more complex SQL-like formulations with ODBMining. ODBMining is an extension of ODBConnect.
- IO communication is done via ODBIOSstream to SocketChannel.

If the client app wants to know more about JODB process, e.g. which JODB servers are currently online and if a JODB server is currently down, it can implement the ODBListening interface which is connected to ODBEvent. The ODBEvent informs client apps about the status of JODB they are working with (e.g. who is updating what or which JODB server is currently online or down, etc.). On the right side of Figure 2, the JODB server structure is shown. The server is an user implemented app that wraps ODBService, which is based on a server configuration in an XML-like script. The server configuration provides all relevant data for JODB server, such as server name and port, path to JODB, etc. A configuration script example:

```
<!-----------------------------------------------------------------------------------------
  @author Joe T. Schwarz (C)
  An example (Case Sensitive)

   - LOGGING     0: disabled, 1: enabled
   - MAX_THREADS max. threads ExecutorsPool (min. 1024, max. unlimited)
   - USERLIST    the encrypted file of all odb users (abs. path)
   - PORT        server's port
   - WEB_HOST/IP is the WEB name or IP of this server.
     localhost is the web name if it's local only
   - NULTICASTING the MultiCast IP for MulticastSocket
   - DELAY in milliseconds before shudown node (min. 1 millisecond, max. 1 second)
   - ODB_PATH/LOG_PATH  the abs. path for all ODB and for Logging

  Messaging facility for the nodes using the Multicast IP
  The multicast datagram socket class is useful for sending and receiving IP multi-
  cast packets. A MulticastSocket is a (UDP) DatagramSocket, with additional capa-
  bilities for joining "groups" of other multicast hosts on the internet.
  More about MulticastSocket: see MulticastSocket API
  About Global Internet Multicast IP: http://www.tcpipguide.com/free/t_IPMulticastAddressing.htm
------------------------------------------------------------------------------------------!>
<WEB_HOST/IP> localhost />
<PORT> 9999 />
<MAX_THREADS> 4096 />
<DELAY> 2 />
<MULTICASTING> 224.0.1.3:7777 />
<!-----------------------------------------------------------------------------------------
   working environment
------------------------------------------------------------------------------------------!>
<LOGGING> 0 />
<ODB_PATH>C:/JoeApp/ODB/Nodes/oodb_Node1 />
<LOG_PATH>c:/JoeApp/ODB/Nodes/log_Node1/ />
<USERLIST>c:/JoeApp/ODB/Nodes/oodb_Node1/userlist />
<!-----------------------------------------------------------------------------------------!>
```

When JODB server starts, it initializes various instances defined in the configuration file:

- setup a server with Hostname/Port
- ODBBroadcaster, which is responsible for broadcasting ODBEvents to other JODB servers and clients and ODBListener for listening on other JODB servers
- ODBManger, which instantiates ODMS (interface to storage), manages and synchronizes JOBB activities like lock, unlock, write, read, etc.

When a client request comes in, ODBService starts an ODBWorker, which first verifies client authenticity (encrypted user ID and password), then associates with the client until it quits. When the client accesses an object that does not exist in the local ODMS, ODBManager instantiates and starts ODBCluster for all nodes to set up ODB Agents that perform the access work on behalf of ODBWorker on those nodes. Communication between clients and workers is done through ODBIOStream in conjunction with SocketChannel (see Figure 3).

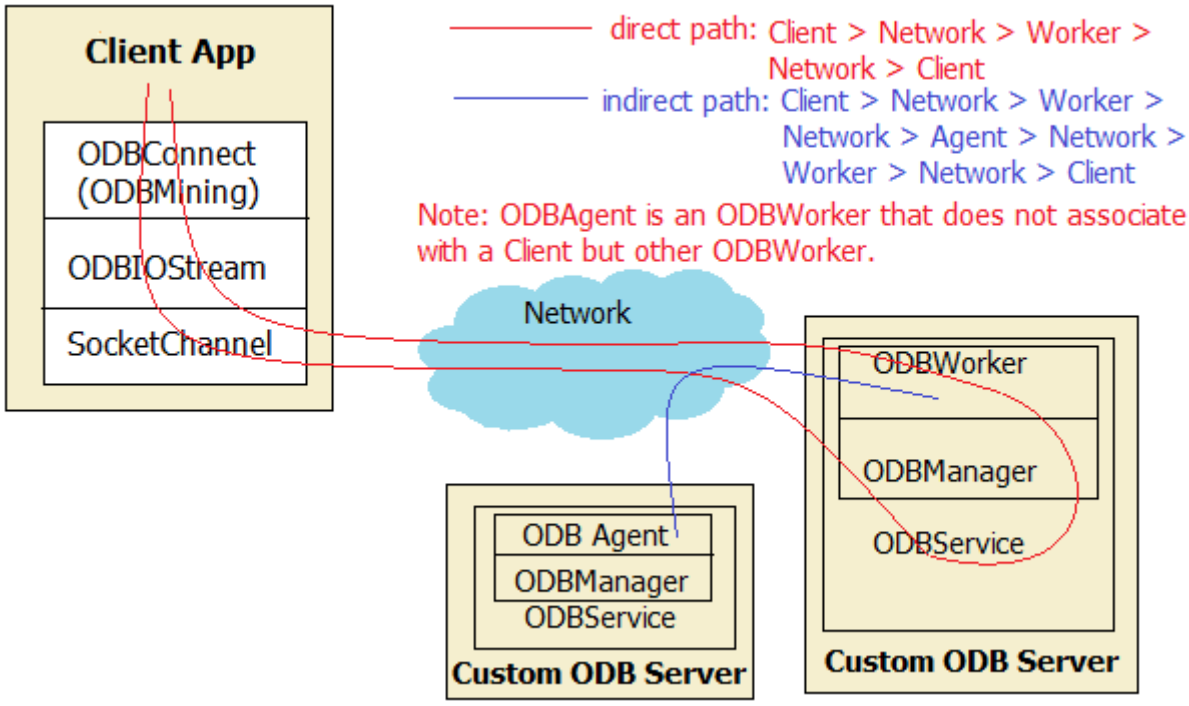


Figure 3

As you can see in the configuration file, the line <MULTICASTING> 224.0.1.3:7777 /> is the IP + port number for the ODBListener and ODBBroadcaster, which broadcasts its status as events to the other ODBServers within the cluster. The events are:

- ODBServer down or up
- new ODBServer joins cluster
- ODBServer is removed from cluster
- and some administrative tasks (like removing an orphaned agent, unlocking a key of a no longer active app, etc.)

With this broadcasting feature, the clients do not need to know how many ODB servers are online and which servers have a part of ODB.

4. Working with JODB package

Let's start with an example:

Client:

```
    String dbName = "Members";
    ODBMining mining = new ODBMining("localhost", 9999, "erika", "joe"); // uID: joe, pw:erika
    mining.connect(dbName); // connect to serialized ODB "Members"
    ArrayList<Object> members = mining.getKeys(dbName);
    for (int i = 0, sz = members.size(); i < sz; ++i)
    System.out.println("Member_"+i+":"+(String)members.get(i)+"\n");
```

The output:

```
Member_0:Bob33732
Member_1:Joe20963
Member_2:Frank96113
Member_3:Frank72144
Member_4:Frank96116
Member_5:Bobby90252
Member_6:Joey49836
Member_7:Joey78397
Member_8:Willy99933
```

For the clients only two JODB APIs are relevant:

- ODBConnect: covers the basic requirements for working with a database, such as reading, updating, deleting, adding, locking, unlocking, committing, etc.
- ODBMining: is an extension of ODBConnect and provides the users with methods to perform complex data processing, such as SQL-like formulations like "select name eq Wil* and age gt 40".

ODBConnect and/or ODBMining can access different databases with only ONE connection to different ODB servers. Other APIs (ODBEvent, ODBListening and ODBListener) are used when the client app needs to listen to the ODB servers on the network. ODBIOStream is an internal API for ODBConnect and users can use it, but with caution. Example:

```
public class eDict extends Application implements ODBEventListening {
  public void start(Stage stage) {
    Application.Parameters params = getParameters();
    java.util.List<String> pl = params.getRaw();
    cluster = new ArrayList<String>();
    if (pl.size() != 5) {
      System.out.println("ServerName ServerPort PW UID dbName");
      System.exit(0);
    }
    ODBConnect odbc = null;
    String dbName = pl.get(4);
    ArrayList>String> keys = null;
    try { // instantiate ODBConnect
      odbc = new ODBConnect(pl.get(0),                    // host name or IP
                            Integer.parseInt(pl.get(1)),  // port
                            pl.get(2),                    // password
                            pl.get(3)                     // user ID
                           );
      odbc.connect(dbName, charsetName);  // connect to dbName
      odbc,register(this);   // register for ODBListener
      if (!odbc.autoCommit(true)) {
        System.out.println("Unable to set autoCommit() to Server");
        System.exit(0);
      }
      keys = odbc.getKeys(dbName)); // get all keys
    } catch (Exception ex) {
      ex.printStackTrace();
```

```
      System.exit(0);
    }
    ...
  }
  ...
  public void odbEvent(ODBEvent event) { // implement the ODBListening Interface
    String node = event.getActiveNode();
    int type = event.getEventType();
    String msg = event.getMessage();
    ...// do something
  }
  ...
}
```

If the client app is an implementation of ODBListening, it must register after instantiating ODBConnect to receive the full callback in case of events (see odbc.register(this)).

ODBListener is started and managed by ODBConnect when the connection is successful. Using this API in the client app should be done with caution to avoid unwanted disruptive events. More about these APIs and their methods: see github.com.

The implemented broadcasting commands (or types):

Type	Explanation	Format: type, node, list of information
0	Node down (offline)	0, node, list<alternative JODB>
1	Node up (online)	1, node, list<alternative JODB>
2	Node ready	2, node, list<alternative JODB>
3	Node unavailable	3, node, list<message>
4	Superuser's message (forcedFreeKey, ...)	4, node, list<message>
5	Remove Node's agents (internal)	5, node, list<message>
6	Add Node	6, node, list<alternative JODB>
7	Remove Node	7, node, list<alternative JODB>
8	Remove Node (internal)	8, node, list<alternative JODB>
9	Force to close (Superuser)	9, node, list<message>
10	Node joins Cluster (see 1)	10, node, list<message>
11	Notify for Update/Add/Delete	11, userID|dbName, list<message>
12	Client app sends message to JODB	12, node, list<message>
rest	For future use	

The primary server for client apps is the server the client apps authenticated on. It is only relevant for the users who work with the following event types: 0, 1, 2, 3, 4, 9, 10, 11, and 12 (see Fig. 4 on the next page). ODBEvent of type 11 specifies the userID and the database name, separated by a vertical bar |.

Figure 4 (sources: GITHUB)

ODBMining is, as mentioned, an extension of ODBConnect. They are not mutually exclusive, both can be used in the same application (multiple connections). However, it is sufficient to work with either ODBMining or ODBConnect, since one ODB connection can handle multiple Java ODBs (JODBs) and is thus independent of which JODB the application is currently processing. ODBMining provides users with some methods that facilitate object data mining access similar to relational databases:

- select(dbName, varName, pattern), where pattern can be wildcards (* for a string, ? for a single letter)
- select(dbName, varName, comparator, value), where a comparator can be LT, LE, EQ, GE and GT and value can be any primitive or BigDecimal/BigInteger
- SQL-like formulation like "select varName_1 EQ Joe* and varName_2 GE 100000"

If a pattern only needs to vary in, say, 3 letters, the pattern can be Pe???son for Peterson, Petteson, etc. Combinations between * and ? are allowed, but should be used in a meaningful expression. Example: J?e* for Joe, Joey, Jee, Jeep, Jeepy, etc. A SQL-like formulation must always begin with a select and can be chained by two conjunctions and and or. Each expression consists of three parts and is chained by one of the two mentioned conjunctors:

- varName: variable name
- comparators: LT/LE/EQ/GE/GT (less than, less/equal, equal, greater/equal and greater than) – case-insensitive.
- pattern (with/without wildcard) or value (primitive, BigInteger, BigDecimal)

Note: Such a SQL statement like "select ... where ..." is nothing more than a "select ... and ..." The where here can be replaced by the and conjunctor. Example: "select name eq J* where address = 'Houston' " or "select name eq J* and address eq 'Houston' ".

Example:

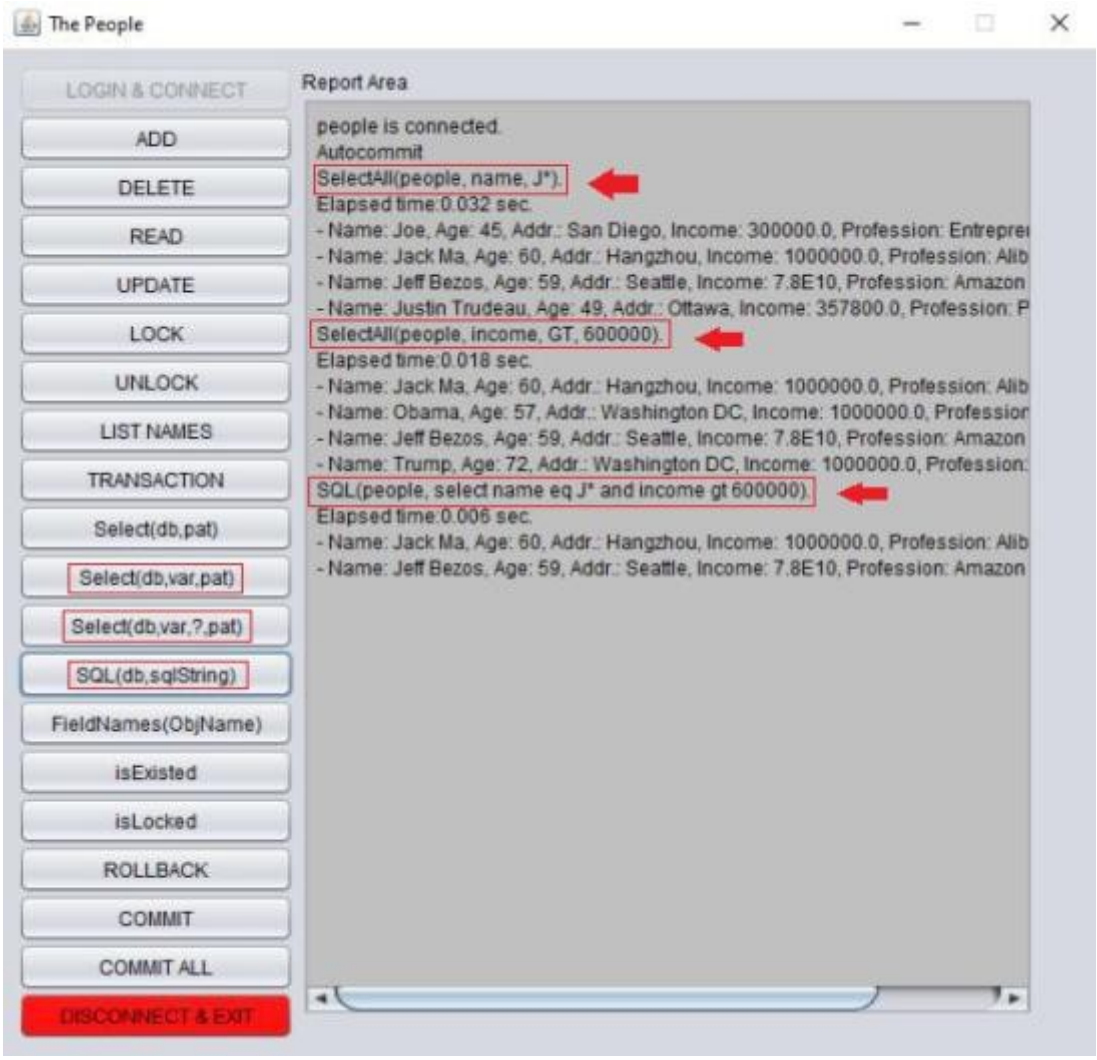


Figure 5 (sources: GITHUB)

The serialized object for JODB People is similar to the Employee object mentioned above. Besides the relevant data like name, age, address, etc., the People object (class) can have some implemented methods, for example to display a portrait of each person (see Fig. 6 on the next page. Note: all the data about the people are the product of my imagination).

Implementing methods is the greatest advantage of the object-oriented database (OODB). Since the deserialized object is based on an existing bytecode class, the classes for all serialized objects must exist at the client site - not at the server - so that the app can more meaningfully execute the methods directly on the app site. For example, the portrait representation in Fig. 6 makes the client app more attractive than a mere description of this person.

As already mentioned in the introduction, ODBObjectView is used on the server site instead of the bytecode class to extract and filter relevant data of a serialized object in order to be more efficient in data mining (see: ODBDataMining) on the server side: only the desired (filtered) objects are sent to the client for further processing. However, if the client application works directly with JAVA objects (e.g. String) that are not serialized, ODBObjectView cannot extract the name of the JAVA objects used. This means that such data mining is not possible.

Figure 6 (sources: GITHUB)

The following JODB APIs are required for client apps:

API	Explanation
EnDecrypt	Proprietary En/Decrypt algorithm
ODBConnect	Interface between Client app and Server (SocketChannel network)
ODBDataMining	Extension of ODBConnect for intensive Data Mining
ODBEvent	Optional: Event object
ODBListerning	Interface for ODBEvent implementation
ODBListener	ODBEvent listener, launched and maintained by ODBConnect
ODBIOStream	IO Stream between ODBConnect and Network - SocketChannel
ODBInputStream	Internal Input Stream used by ODBIOStream

With the exception of ODBConnect or ODBDataMining, client applications usually only deal with ODBEvent and the onEvent implementation of the ODBListening interface (see eDict example and Fig. 4 and Fig. 7 on the next page). ODBListener, ODBInputStream and ODBIOStream are more or less "invisible" to users. SocketChannel communication and the APIs are instantiated and managed by ODBConnect (see Fig. 8 on the next page).

Note: when logging into the JODB server, ODBConnect encrypts the password and user ID before sending them to the JODB server.

Figure 7

An onEvent popup implementation called back by ODBConnect's ODBListener.

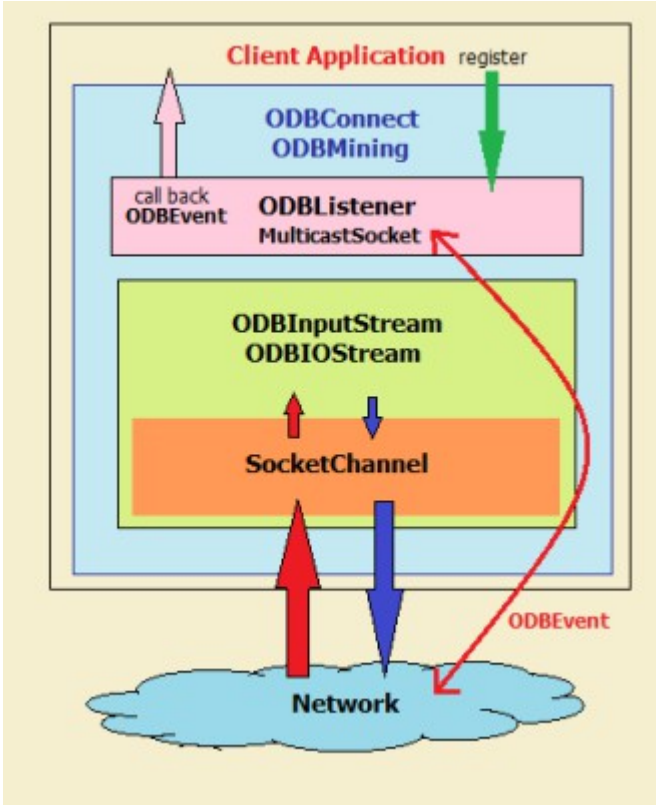


Figure 8

JODB Server:

The JODB package does not provide a fully functioning JODB server, but rather the ODBService API that allows developers to implement their own JODB server. However, I have included in this JODB package two similar JODB servers implemented in JavaFX and Java-SWING. Note that the Java SWING JODB server is implemented using my MVC-SWING package. ODBService requires a configuration file that contains the necessary data for a successfully running JODB as a server.

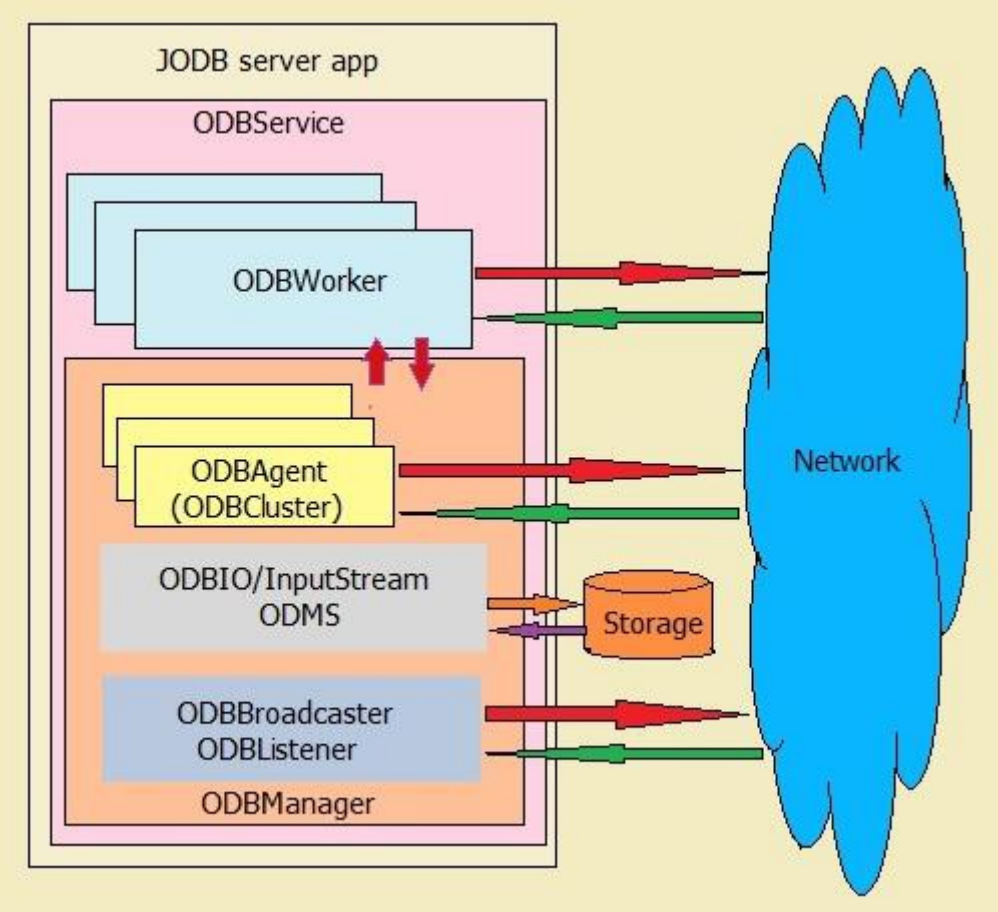


Figure 9

Figure 9 shows you how a JODB server is structured hierarchically and how to implement a customized JODB server based on ODBService. Such a JODB server only needs to provide ODBService with the necessary data required for a fully functional network server: host name or IP, port number, physical paths to various storage media, logging directory, JODB directory, etc. For more information: see a configuration script example on page 7.

ODBService starts a ServerSocketChannel in an Executors-newFixedThreadPool, which then intercepts all incoming client ODBConnects and then starts an ODBWorker associated with that client, which processes all requests between client and ODBManager to access data until the client quits. If the requested data is not local but somewhere on one of the other JODB servers, ODBManager initiates and instantiates an ODBCluster (ODB Agent) that acts like a client for that JODB server to access the data it needs and deliver that data to ODBManager, which then passes it on to ODBWorker. ODBManager, an implementation of ODBListening, manages the local "matter" between ODBWorker and the object database through ODMS (Object Data Management System), which deals directly with the

physical storage and ensures that everything is fully compliant in terms of database aspects and requirements: read, write (update and add), lock, unlock, rollback and commit. In addition, ODBManager participates in the cluster and takes care of its status (e.g. which JODB server is active or inactive) and performs some superuser requests such as removeNode, addNode, grateful shutdown, etc. Figure 10 show you the communication between Client and JODB server (and cluster) in simplified form (i.e. without lock, unlock, etc.)

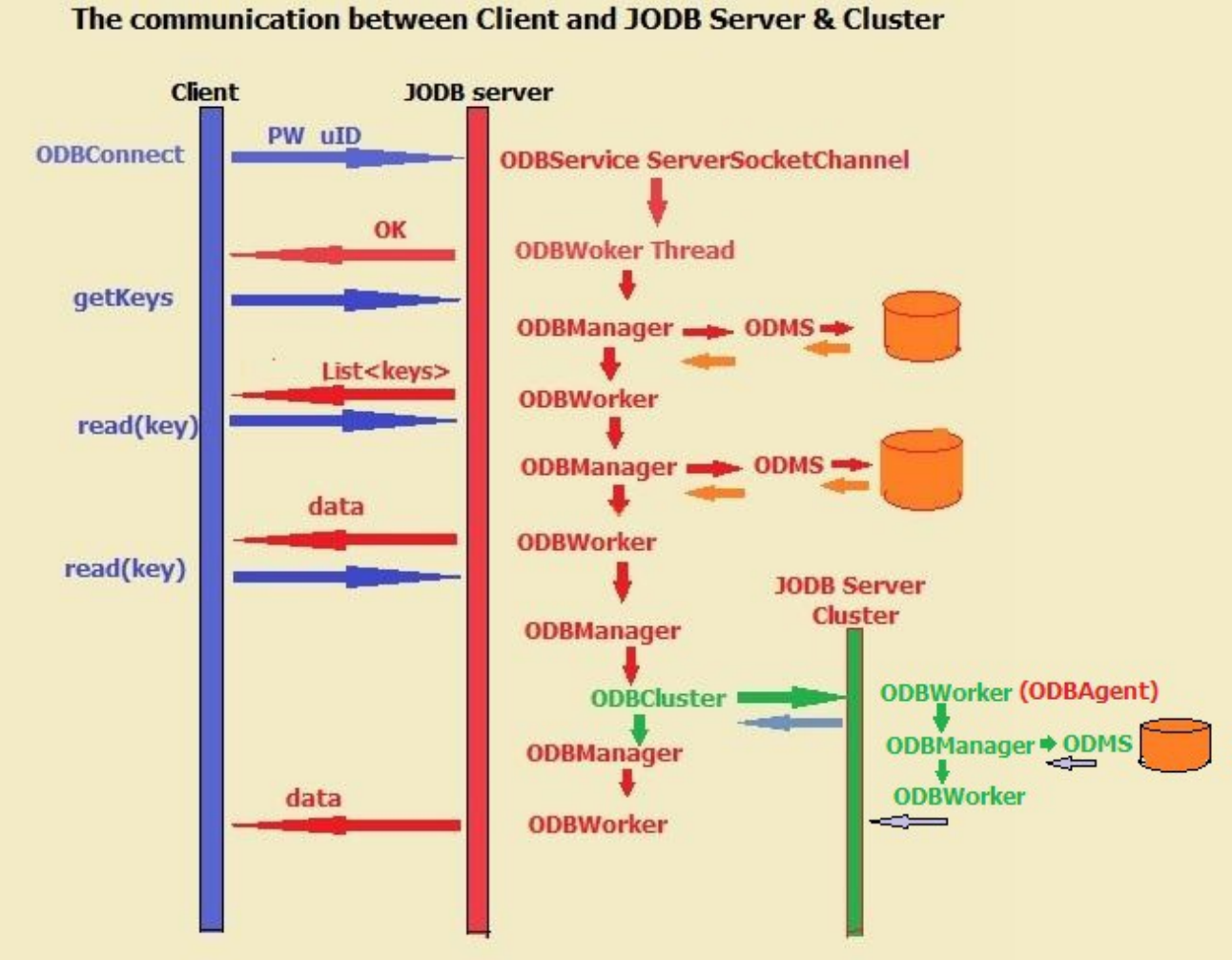


Figure 10

Figure 11 on the next page shows a JODB server implementation in Java SWING MVC. In the report pane, you can see the status message of a JODB server with port 9999, which states that this server is online (active). This can be done thanks to ODBBroadcaster and ODBListener. The JButtons control various services provided by ODBService. With the JComboBox you can ping any JODB server in the cluster. Although local and remote ODBWorkers (cluster) use the same instance class, their work is slightly different from each other. The first deals with a remote client app, the second deals with a remote ODBManager initiating a request for data. To distinguish, the first is called a worker and the second is called an agent. The UserMaintenance and ODBMaintenance tabs are the implementations of UserList (add new user, update or delete user) and ODB-Cluster (forcedUnlock, forcedCommit, removeNode, addNode, etc.). To prevent any kind of manipulation, the EXIT button only works if you are a superuser by authentication. The same applies if you want to access UserMaintenance or ODBMaintenance. Figure 12 on the next page shows you the same JODB server, but in JavaFX server implementation.

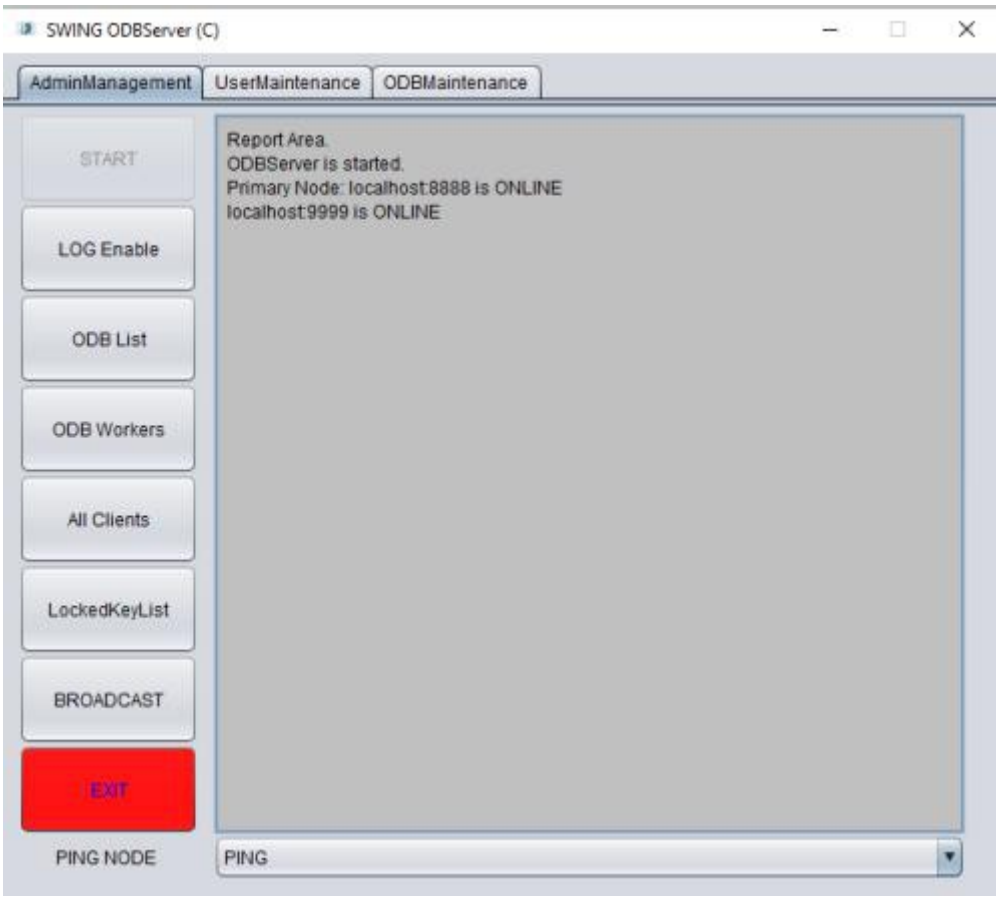


Figure 11 (sources: GITHUB)

Figure 12 (sources: GITHUB)

JODB Cluster

When several similar servers work together, they form a server cluster. The servers are either connected one after the other in a ring or in a mesh (Fig. 12) such as Ethernet or the Internet.

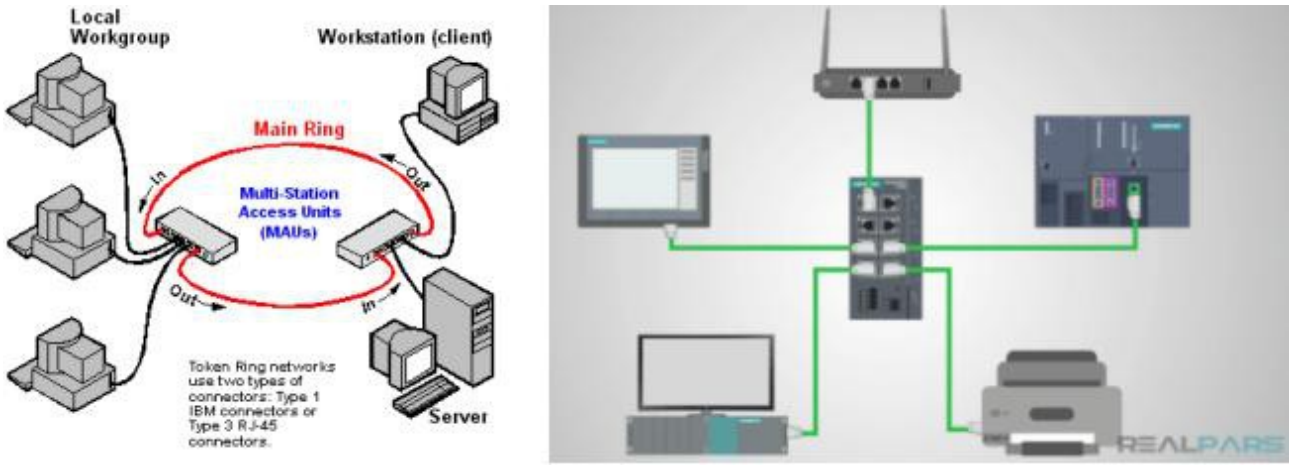


Figure 12: Token Ring (Source: computerlanguage.com) and Ethernet

It is important that these servers use the same communication protocol so that they can communicate with each other or support each other. JOBD cluster is a mesh-like cluster in which each server can reach other servers directly. Using the Broadcasting MulticastSocket technique, each JOBD server can dynamically join or leave the cluster without affecting the cluster functionality. The disadvantage of MulticastSocket-Broadcasting is the extensive synchronization to avoid message loss. Of course, the JODB cluster must rely on a specific broadcasting protocol so that each JODB server knows what to do with the received message, since the messages are not addressed to a specific server, but to all servers. The left JODB server runs on port 8888 and the right one on port 9999. Whichever comes online later than the other broadcasts an online-message (MulticastSocket) and the other confirms this with a dedicated response via ODBCluster (via SocketChannel) so that this broadcaster knows who all in the cluster are online (see Fig. 13 on the next page). The broadcasting protocol is here the ODBEvent whose content has the following format:

Type or ID	Node	List<Nodes/Messages>	comment
0	node	list<alternative JODB>	This node is down (offline)
1	node	list<alternative JODB>	This node is up (online)
2	node	list<alternative JODB>	This node is available (after up)
3	node	list<message>	This node is unavailable
4	node	list<message>	Superuser for forced to free a key, etc.
5	node	list<message>	Subperuser. Remove this node
6	node	list<alternative JODB>	Superuser. Add this node to cluster
7	node	list<alternative JODB>	Superuser. Request to remove this node
8	node	list<alternative JODB>	ODBManager. Request to remove this node
9	node	list<message>	Superuser. Force to close a DB on this node
10	node	list<message>	ODBWorker. Acknowledge ODBEvent type 1
11	userID|dbName	list<message>	ODBWorker for add/update/delete
12	node	list<message>	Sent by Client App to JODB
> 10	reserved		

Note: The format of a node: hostname:port or IP:port. For the client app, only types 0...4 and 9, 10, 11 and 12 are relevant.

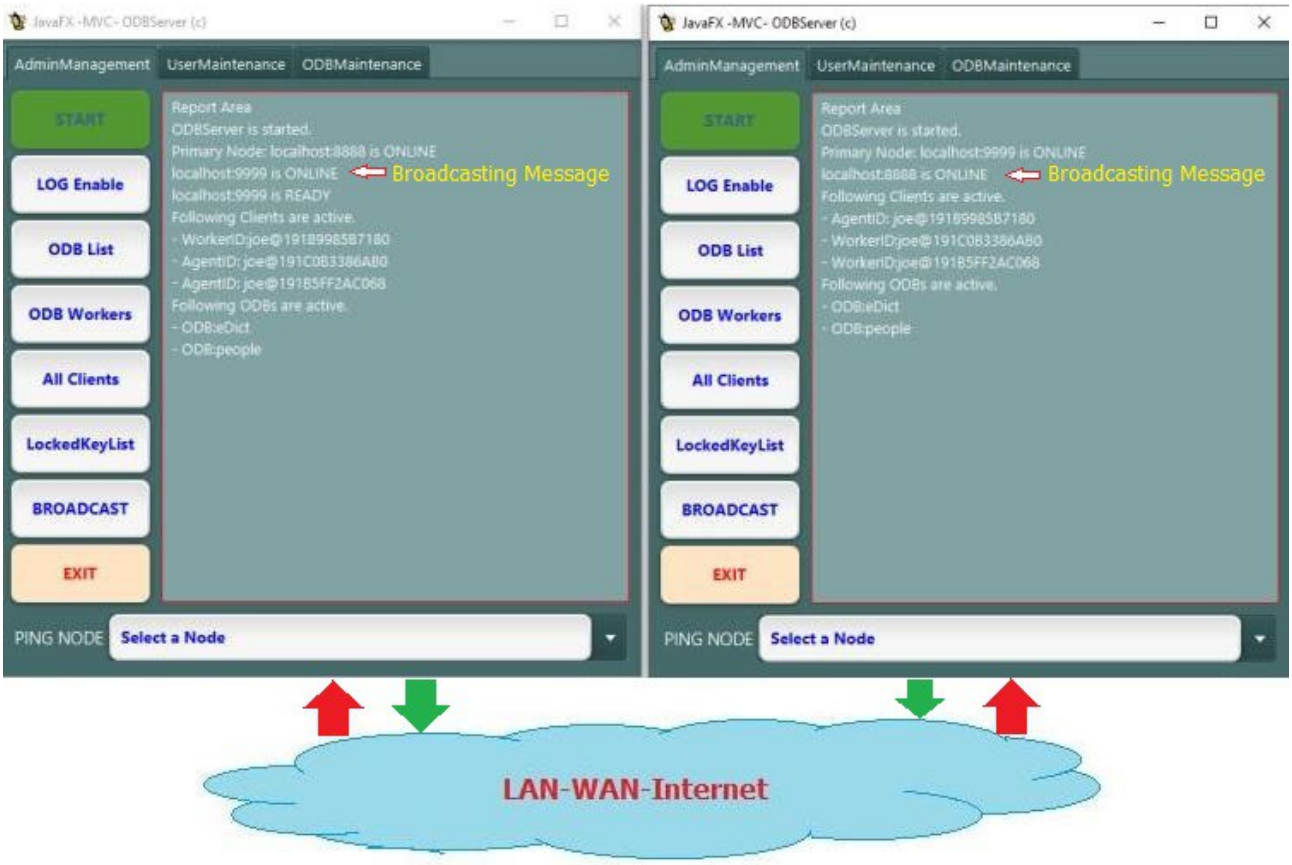


Figure 13

- The All Clients button shows all active clients. The WorkerID or AgentID is the UserID@Timestamp (in hexadecimal nanosecond format) at the time the client successfully established an ODBConnect connection to a JODB server.
- The ODB List button shows all active object databases that are in use.

The following APIs are required for a fully functional JOB server:

APIs	Explanation
EnDecrypt	Proprietary En/Decryption algorithm
JFXOptions	Optional. JFX tools for JFX application (see JFX Server)
JOptions	Optional. SWING tools for SWING application (see SWING server)
ODBBroadcaster	MulticastSocket broadcaster
ODBCluster	The counterpart of ODBConnect, used by ODBManaget to connect a server on cluster
ODBEvent	Event
ODBListener	MulticastSocket listener
ODBListening	Inteface for ODBEvent
ODBIOSstream	Internal IO stream used by ODBWorker and ODBConnect, ODBCluster
ODBInputStream	An InputStream extension. Used by ODBIOStream and ODMS
ODBManager	The manager of all ODB activities such as lauching an ODBCluster, starting ODMS, etc.
ODBObjectView	The deserialized tool without relying on serialized object class. Used by ODBWorker
ODBParms	The global internal parameters POJO used by ODB package
ODBParser	The XML-like parser used by ODBService
ODBService	The base of a JODB server. ODBService manages ODBWorker, ODBManager, etc.
ODBWorker	The dialog partner for Client ODBConnet or ODBMining
ODMS	The interface to physical DB storage
UserList	The UserList management

4. Working with ODB APIs

User Maintenance

To work with JODB databases, each user must be registered in an encrypted userlist (fixed name) managed by an administrator or superuser. The UserList API is designed for this purpose (see Fig. 14). The format of the user list entries is as follows:

password (any length)	:	UserID (any length)	@	privilege

Default PW/uID/Privilege: system:admin@3 (case sensitive). The entries are encrypted and stored as gzipped file in the default file name userlist in the given path from Server config. file: <USERLIST>c:/JoeApp/ODB/Nodes/oodb_Node1/userlist/>.

Access Privilege

The Privilege is 0 (read only), 1 (read/write), 2 (read/write/delete) and 3 as superuser (with privilege 2).

Default for administration: system as password and admin as user-ID (lower case). This default cannot be used to work with any Object Database (ODB). An implementation of UserList API in JODB server.

Figure 14 (Sources: GITHUB)

Object Database (ODB) Creation

Whenever an user logs in and starts a connect(dbName) to an ODB that does not exist on the local JODB server, following situation occurs:

- User privilege 0: Exception, or
- User privilege >0: An ODB is created with the specified dbName in the specified path from the JODB server configuration file on the LOCAL node.

In other words, only users with privilege above 0 can create an ODB. The ODB files are gzipped before being saved to the specified path in the server's configuration file. Since ODB access keys can be a string, a long value, or a big integer value, external input keys must be "normalized". The normalization process is done using this method:

key_normalized = ODBIOStream.toODBKey(key)

ODB Maintenance

The implementation of the JODB server should be done under superuser management so that any kind of inadvertence or sabotage, such as shutting down or removing a cluster server or changing a user's privilege, can be avoided. As mentioned before, the JODB server must be based on the ODBService API (Fig. 15). This API provides access to ODBManager and is itself a SocketServerChannel server that serves all incoming clients by creating an ODBWorker for each client. Each ODBWorker is then responsible for only ONE client until this client leaves (or terminates) the session.

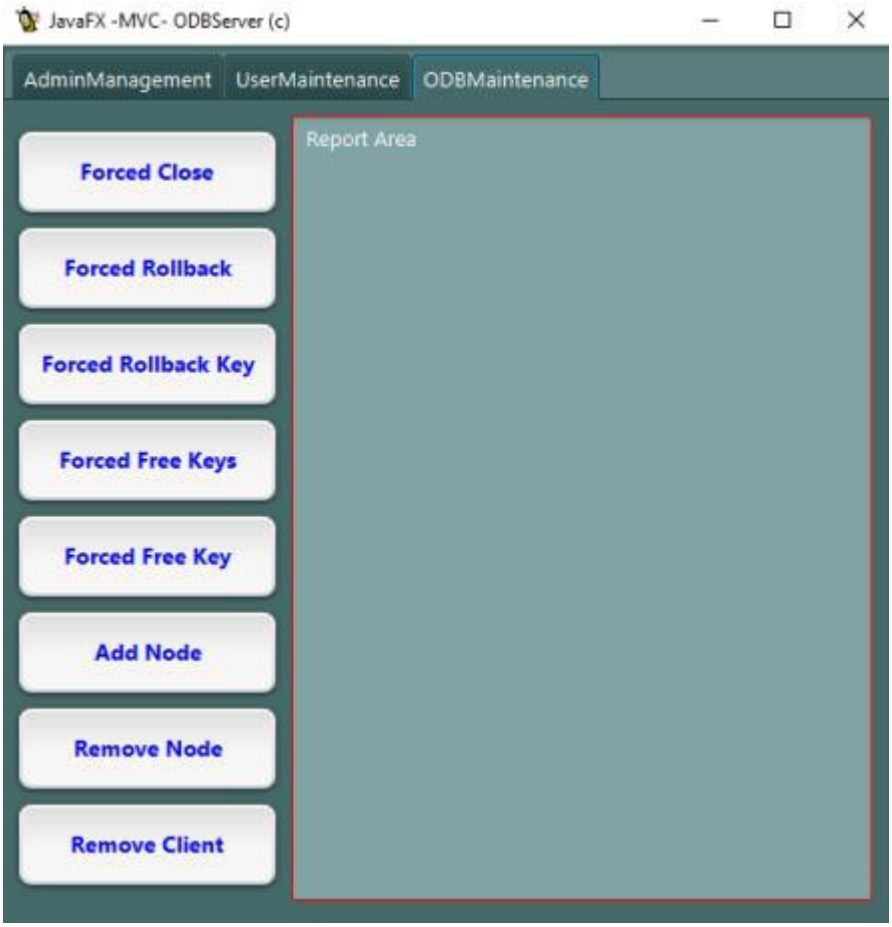


Figure 15

Client Application

A client application typically accesses data from a database server, processes it, and stores it on that server. ODBConnect or ODBMining is the API that is used by client applications for such accessing activities. That's basically all. However, if the database is distributed across multiple servers, the client application needs to know what's going on on those servers. To do this, the client application needs to be an implementation of ODBListening. Through ODBEvent, the client application can listen for events on those ODB servers in the cluster and know which servers are online or offline. Or it can respond to any activity that can affect the client application (see Fig. 16). ODBConnect/ODBMining starts with a connection to one of the servers in the cluster. If everything is OK, this server becomes the primary server for the client app. Then, with this connection, the client app starts to connect to one (or more) databases (using the connect(dbName) method). This connection stays with the client app until something happens:

- the primary (or any secondary) server is down (offline)
- the client issues the disconnect() method

If something unexpected happens to the client app, a graceful shutdown of ODBConnect is initiated to ensure that all open databases can be closed properly. The following example shows you how to work with ODBConnect (similar with ODBMining):

```
import joeapp.odb.*;
// an example
public class MyApp extends Application implements ODBEventListening {
  // some variables
  private ODBConnect odbc;
  private ArrayList<String> keys;
  private String dbName = "Employees";
  private String primaryNode;
  //
  public void start(Stage stage) {
    Application.Parameters params = getParameters();
    java.util.List<String> pl = params.getRaw();
    if (pl.size() != 4) {
      System.out.println("userID Password WebServerName WebServerPort");
      System.exit(0);
    }
    try {
      // create the node: HostName/IP:Port
      primaryNode = pl.get(0)+":"+pl.get(1);
      int port = Integer.parseInt(pl.get(1));
      //                    host       port  password   userID
      odbc = new ODBConnect(pl.get(0), port, pl.get(2), pl.get(3));
      odbc.connect(dict, odbc.charsetNameOf("Tiếng Việt")); // connect to DB Employees
      odbc.autoCommit(true); // set autoCommit
      odbc.register(this); // register Listener
      keys = odbc.getKeys(); // get all keys
    } catch (Exception ex) {
      ex.printStackTrace();
      System.exit(0);
    }
    …
    // as Object and must be cast to Employee
    Employee joe = (Employee)odbc.read(dbName, "Joe");
    // lock, update and unlock key "Joe"
    if (!odbc.isLocked(dbName, "Joe")) {
      odbc.lock(dbName, "Joe");
      joe.income += 100.0;
      odbc.update(dbName, "Joe", joe);
    }
    ...
    odbc.save(dbName);
```

```
    odbc.close(dbName);
    odbc.disconnect();
  }
  ...
  // implement the ODBEvent-----------------------------------------------------
  // only check for:
  // 0: a node is down
  // 2: a node is ready.
  // 4: forcedFreeKey, forcedRollabck, etc.
  // 9: forcedClose
  public void odbEvent(ODBEvent event) {
    // if not JFX: this work should be done by a Runnable in an Executors Pool,
    // or for SWING: SwingUtilities.invokeLater(() -> {...});
    Platform.runLater(() -> {
      String node = event.getActiveNode();
      int type = event.getEventType();
      String msg = event.getMessage();
      if (type == 0 && (node.equals(primaryNode)) { // is primary server down ?
        ... // do something
      } else if (type == 2) { // node is ready
        keys = odbc.getKeys(); // update key-list
        ...
      } else if (type == 4) { // forcedFreeKey, etc.
        ... // display the message
      } else if (type == 9 && msg.equals(dbName)) { // forcedClose
        ... // panic
      }
      ...
    });
  }
  ...
}
```

Note: If autoCommit(true) is NOT set, unlock() must be called after update (or delete). Otherwise, this key to this object remains locked and no other client app can access this object using this key.

Figure 16 (sources: GITHUB)

5. ODB APIs Programming

ODBConnect

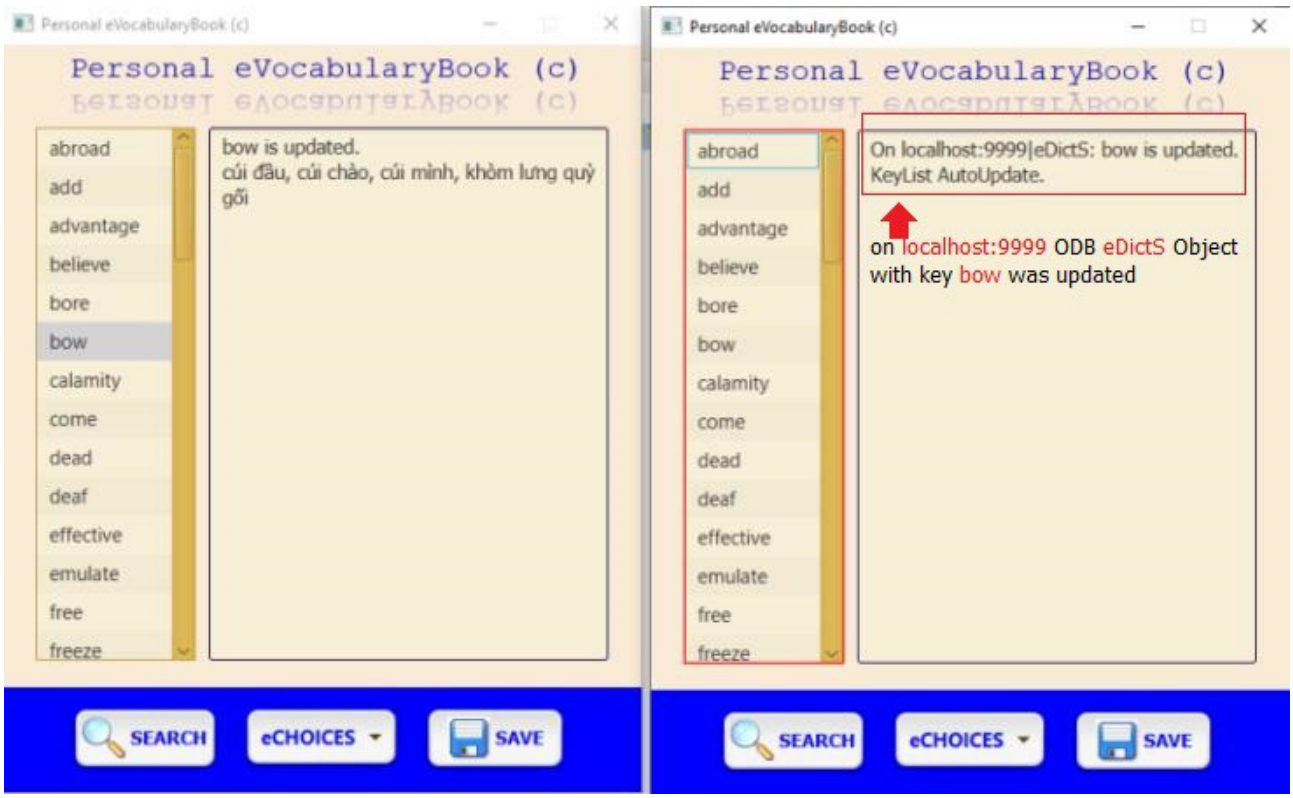
ODBConnect is the interface from client applications to the JODB server. You probably noticed that the connect() method has two parameters. The first is the dbName and the second is the character set name (returned value of the odbc.charsetNameOf("Tiếng Việt") method). This charsetNameOf(String) returns the character set name used to represent the given text (here: Tiếng Việt). Each ODB (via its dbName) is associated with ONE character set. If connect is called with only one parameter (dbName), the UTF-8 character set is used by default. To ensure that the correct character set name is returned, the text should be typical for the language. For example, Tiếng Việt for Vietnamese, le cœur français for French or rötlicher Käfer for German. Since ODBConnect can handle multiple ODBs and each ODB is associated with one character set, client applications can work with different ODBs with different character sets. However, mixed character sets within ONE ODB will lead to unpredictability. When a client application works directly with Java objects that are not implemented by Serializable interface, for example Java String, ODBConnect does not deserialize the result but returns an object byte[] or String and you must cast the object to byte[] and covert it to the class you are working with. Example:

```
// directly used String
public String read(String key) {
  done = true;
  try {
    Object obj = odbc.read(dict, key);
    return (obj instanceof String? (String)obj : new String(obj, charsetName));
  } catch (Exception ex) { }
  done = false;
  return "Can't read. Probably "+key+" is locked.";
}
```

```
// serialized POJO
public eParms read(String key) {
  done = true;
  try {
    return ((eParms)odbc.read(dict, key));
  } catch (Exception ex) {
    ex.printStackTrace();
  }
  done = false;
  return null;
}
```

You can use the charsetNameOf(text) and charsetOf(text) methods of ODBConnect to determine which character set is used to represent the language you are currently working with. The charsetOf(text) returns a Charset instance and charsetNameOf(text) returns the name of the character set used. With the notify(dbName, enabled) method you can turn on (enabled:true) or off (enabled:false) the event that informs you about an object (from the specified ODB dbName) that is just now updated, deleted, or added (see Fig. 17 on the next page). The notification of Object modification can be turned on or off at anytime. By using non-serialized objects directly, you can theoretically store and access all kinds of objects (as byte[] objects) easily. However, it is your responsibility how to work with the byte[] and how to process it correctly. The byte[] is the raw form of an object. Data mining with ODBMining will not work because the object's name and field names are missing (see ODBObjectView in chapter 2). In case of unexpected emergency (e.g. CTRL-C), ODBConnect automatically intercepts this SIGINT and gracefully disconnects the app connection to the JODB server.

As mentioned, you can work directly with different ODBs using a single ODBConnect (or ODBMining). Fig. 18 shows you how the modified eDict (Fig. 17) can access three different ODBs: English - Vietnamese, English - Japanese and English – Chinese.



Fugure 17

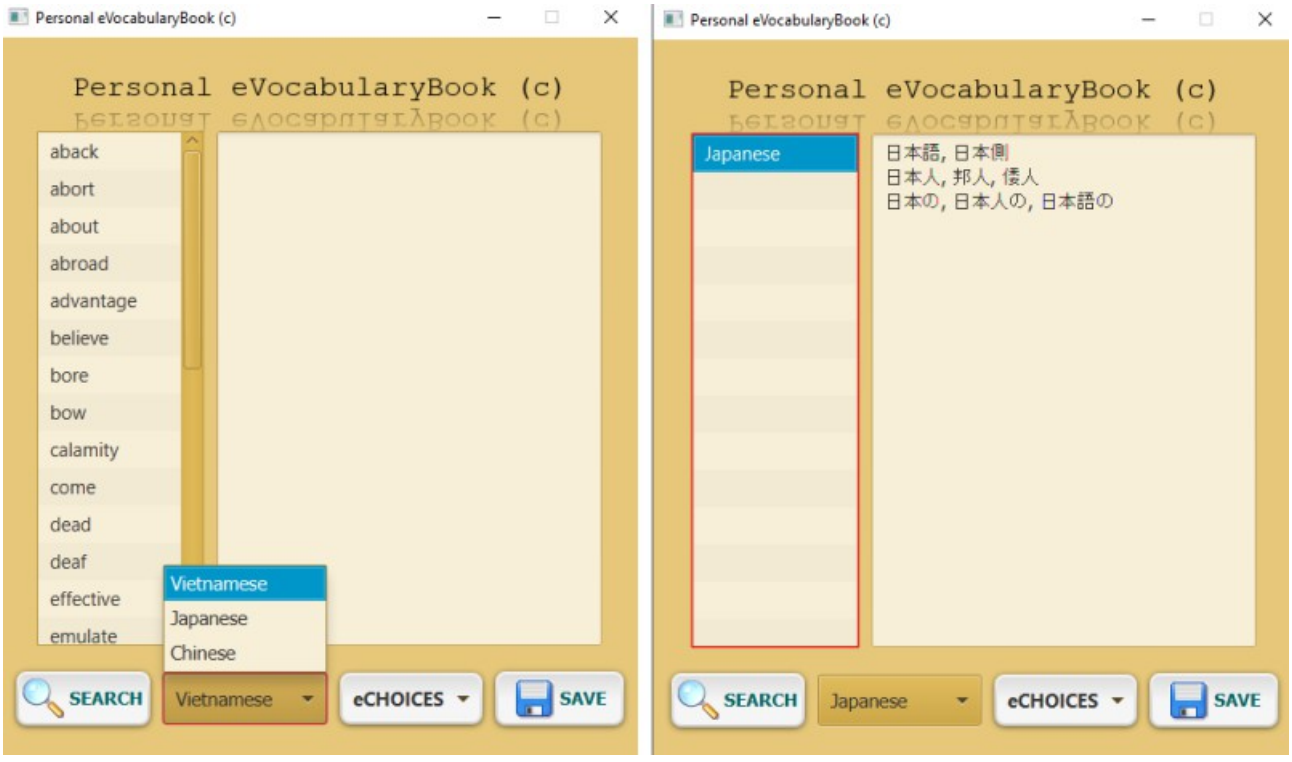


Figure 18

If the user privilege is 2 (R/W/D), an empty ODB is created only in the local JODB under the specified dbName. The same with the same dbName when the user moves to another JODB server in the cluster, but the user can access the ODB on the previous node (as a

distributed ODB on two or more nodes). This code snippet shows you how it works:

```
    odbc = new ODBConnect(host, port, pw, uid);
    cset = new String[] { odbc.charsetNameOf("Tiếng Việt"),
                          odbc.charsetNameOf("日本語"),
                          odbc.charsetNameOf("中国人")
                        };
    if (!odbc.autoCommit(true)) {
      System.out.println("Unable to set autoCommit() to Server");
      System.exit(0);
    }
    // starting with ODB vdDict
    odb = odbs[0];
    csName = cset[0];
    odbc.connect(odb, csName);
    odbc.notify(odb, true); // enable notifying
    // establish a connection to ODB jdDict (Japanese)
    odbc.connect(odbs[1], cset[1]);
    odbc.notify(odbs[1], true);
    // and ODB cdDict (Chinese)
    odbc.connect(odbs[2], cset[2]);
    odbc.notify(odbs[2], true);

    odbc.notify(odb, true);
    ...
    // direct with String
    Object obj = odbc.read(odb, "aback");
    String meaning = new String((byte[])obj, csName); // cast obj as byte[]
    ...
    odbc.update(odb, "aback", "ngạc nhiên"); // Vietnamese update "aback"
    odb = odbs[1]; // switch to Japanese
    odbc.update(odb, "aback", "びっくり");   // Japanese update "aback"
    ...
  }
  ...
  private String odb, cset[];
  private String[] odbs = { "vdDict", "jdDict", "cdDict" };
  ...
}
```

An ODB is a database of objects. The word object means anything that can be considered an object: human, animal, tree, thing, etc. The following example shows how to work with an ODB that consists of two different objects: Member with all the necessarities (name, age, address and salary) and Animal with a name and a link to the animal's picture (Fig. 19 on the next page).

```
public class Member implements java.io.Serializable {
  private static final long serialVersionUID = 1234567890L;
  private int age;
  private double salary;
  private String name, address;
  public Member(String name, int age, String address, double salary) {
    this.age = age;
    this.name = name;
    this.address = address;
    this.salary = salary;
  }
  ...
}
```

```
public class Animal implements java.io.Serializable {
  private static final long serialVersionUID = 1234L;
  private ImageIcon img;
  private double weight;
  private String name, pic;
  public Animal(String name, String pic, double weight) {
    this.pic = pic;
```

```
    this.name = name;
    this.weight = weight;
  }
  ...
}
```

```
    String dbName = "OODBmix"; // ODB with two different Objects
    try {
      ODBConnect dbcon = new ODBConnect("localhost", 8888, "password", "Joe");
      dbcon.connect(dbName); // create ODBmix if needed
      if (!dbcon.autoCommit(true)) {
        System.out.println("Unable to set autoCommit() to OODB-Server@8888");
        System.exit(0);
      }
      // Object Member
      dbcon.add(dbName, "Joe", new Member("Joe", 40, "San Diego", 100000.0));
      // Object Animal
      dbcon.add(dbName, "bear", new Animal("bear", „https://....“, 500.0));
      ...
      if (dbcon.lock(dbName, key)) {
        Object obj = dbcon.read(dbName, key);
        if (obj instanceof Member) dbcon.update(dbName, key, (Member)obj);
        else if (obj instanceof Animal) dbcon.update(dbName, key, (Animal)obj);
        else dbcon.update(dbName, key, obj);
        // commit and free key after Thinking time
        dbcon.commit(dbName, key);
      }
      ...
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    ...
```

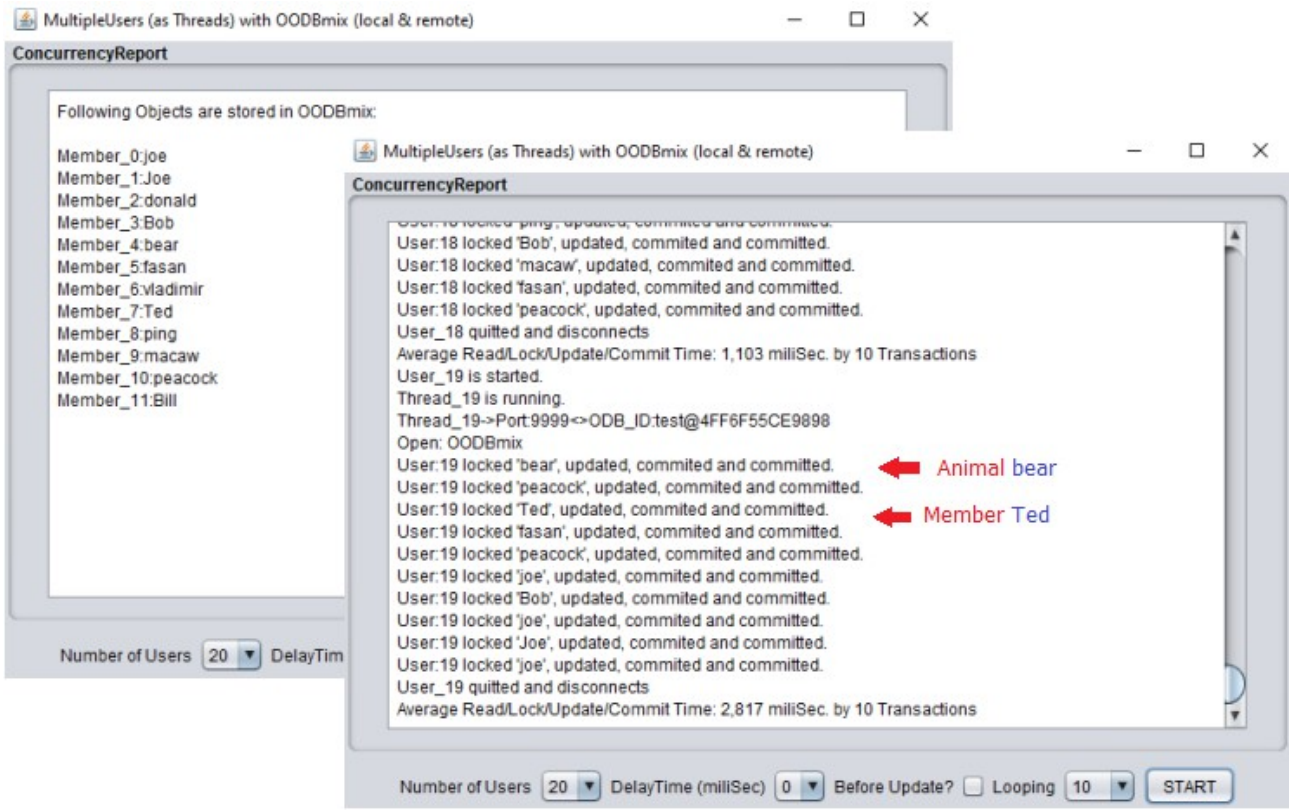


Figure 19

ODBService

ODBService is the interface for customized Java ODB servers. Figures 11 and 12 show you how to use ODBService as SWING JODB and JavaFX JODB (both developed and compiled with JDK 9.0.4 - reason: JavaFX is still part of JDK). ODBService gives you direct access to ODBManager and other internal ODB APIs (e.g. ODBParms, ODBWorker, etc.). In addition, ODBService manages various threads such as ODBBroadcaster, ODBListener, ODBWorker and an ODB server in ServerSocketChannel technology. The mentioned servers are divided into three Tabs:

- **ServerTab**: Fig. 20. the home window where you can view information about clients, workers of specific ODBs or pinging a JODB in the cluster, etc. The so-called clients are either workers (interfaces to client apps) or agents (interfaces to other JODBs as client proxies),
- **UserTab**: Fig. 21-left. This tab is only accessible if you are superuser. In this tab you can view, delete and update a JODB user or add a new JODB user.
- **ODBTab**: Fig. 21-right. Like UserTab, this tab is only accessible to superusers. This tab gives you control over the active JODBs. Here you can force close an ODB, free (commit) or rollback a locked key, adding or removing a JODB in the cluster, or removing unresponsive (dead) clients (Workers or Agents).

Figure 20 - ServerTab

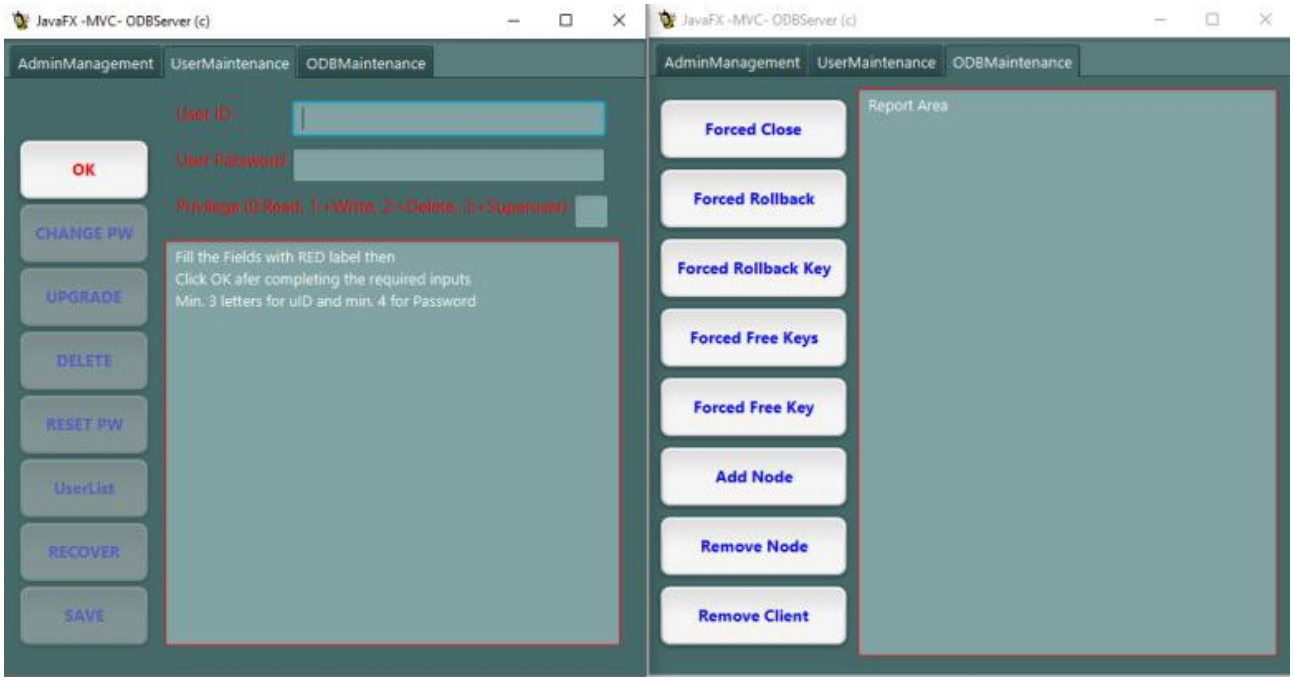


Figure 21. Left: UserTab, right: ODBTab

ODBService requires 9 parameters to run. If you want to customize your own JODB server, you need to create a file with any name that contains these 9 parameters:

Parameter	example
WebHostName or IP	<WEB_HOST/IP> localhost />
Port number	<PORT> 9999 />
Max. running Threads	<MAX_THREADS> 4096 />
Delay time in milliseconds	<DELAY> 2 />
MultiCasting IP:Port	<MULTICASTING> 224.0.1.3:7777 />
Logging (0: no, 1: yes)	<LOGGING> 0 />
Directory Path to ODBs	<ODB_PATH>C:/JoeApp/ODB/Nodes/oodb_Node1 />
Directory Path to logging files	<LOG_PATH>c:/JoeApp/ODB/Nodes/log_Node1/ />
Path to userlist	<USERLIST>c:/JoeApp/ODB/Nodes/oodb_Node1/userlist />

The parameter names in RED are fixed (unchangeable) and in uppercase. The format is based on XML:

- <keyword> value />
- <! any comment !>

The keyword is surrounded by < and >, its value follows and is terminated by />. Keyword is a string of any length, value can be a string of any length or an integer. The comment always starts with <! and ends with !>, regardless of how long the comment is.

To facilitate GUI JODB customization, two utilities are available:

- JFXOptions: This is an API for JavaFX with some useful methods like login with ID and password by replacing each character with a symbol like the asterisk * or a dot •. JFXOptions needs to be instantiated before use.
- JOptions: This contains some useful static methods (like JFXOptions).

These two APIs are included in the ODB package. Similar to ODBConnect, ODBService automatically intercepts the CTRL-C SIGINT and gracefully closes all open resources:

- Rollback or commit ODBs,
- Close all workers and agents,
- Sends a "down" message to other JODB servers in the cluster,
- Shuts down the local thread pool

An example of JODB Server in SWING (see Fig. 22)

```
import javax.swing.*;
import joeapp.odb.*;
// Joe (C)
public class SimpleJODB extends JFrame {
  private boolean boo = true;
  private ODBService odbs;
  public static void main(String[] args) throws Exception {
    if (args.length == 0) {
      System.err.println("Usage: java SimpleJODB configFile");
      System.exit(0);
    }
    UIManager.setLookAndFeel("com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel");
    new SimpleJODB(args[0]);
  }
  //
  public SimpleJODB(final String config) throws Exception {
    setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
    JButton start = new JButton("START JODB");
    start.addActionListener(e -> {
      if (!boo){
        odbs.shutdown();
        System.exit(0);
      } else try {
        odbs = new ODBService(config);
        start.setText("STOP JODB");
        boo = false;
      } catch (Exception ex) {
        System.err.println("Unable to start ODBService. Check:"+config);
        System.exit(0);
      }
    });
    add("Center", start);
    setSize(200, 200);
    setVisible(true);
  }
}
```

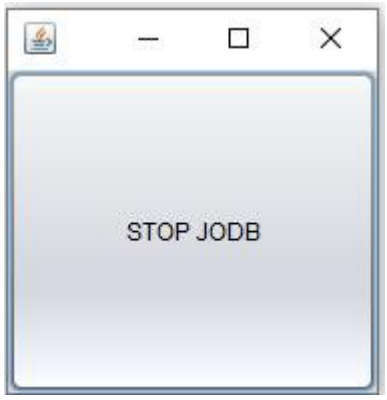


Figure 22

If this SimpleJODB applies JOptions for the START button, the login window looks like this:

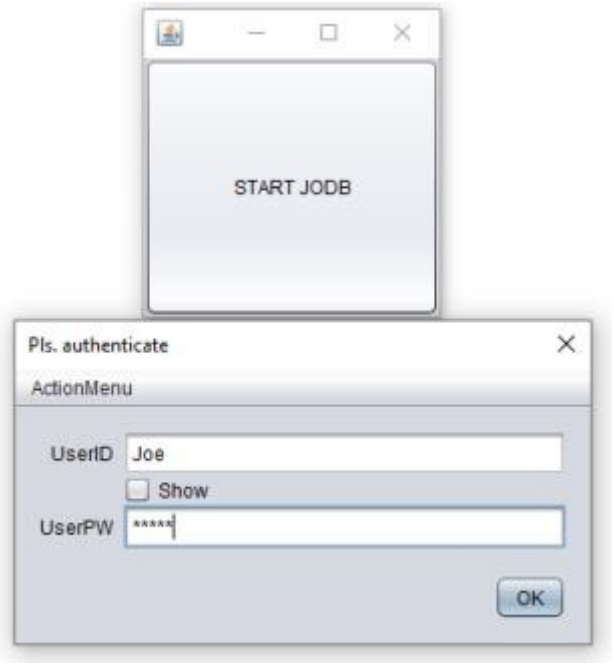


Figure 23

and the modification

```
start.addActionListener(e -> {
  if (!boo){
    odbs.shutdown();
    System.exit(0);
  } else {
    try {
      java.util.HashMap<String, String> map = ODBParser.odbProperty(config);
      UserList userList = new UserList(map.get("USERLIST"));
      String[] aut = JOptions.login(this, "Pls. authenticate");
      if (!userList.isSuperuser(aut[1], aut[0])) {
        JOptionPane.showMessageDialog(this,
                                      "Invalid",
                                      "Invalid Passowrd or UserID",
                                      JOptionPane.ERROR_MESSAGE);
        return; // do NOTHING
      }
      odbs = new ODBService(config);
      start.setText("STOP JODB");
      boo = false;
    } catch (Exception ex) {
      System.err.println("Unable to start ODBService. Check:"+config);
      System.exit(0);
    }
  }
});
```

ODBService allows a JODB server to send messages to other JODB servers in the cluster. As mentioned before, the format of a message is: Type, Node, List<?>.

- Type is a number between 0...12 (the rest is reserved for future extensions).
- Node is usually a string of hostname/IP and port separated by a semicolon (:). Usually the node is the JODB that causes the event (e.g. up, down, etc.). For type 11 (notiffy): an UserID and an dbName with the vertical bar (|) in between. For type 12: the node is the JODB server where the client app opens the connection.
- List<?> contains either some nodes (hostname/IP:port) as alternative JODB servers in case the main JODB is shut down unexpectedly (e.g. CTRL-C), or a message of any length.

A JODB server can only start successfully if

1. ODB ServerSocketChannel is started successfully and
2. ODBBroadcaster MulticastSocket is started successfully and
3. ODBListener MulticastSocket is started successfully.

If either of these fails, the JODB server will terminate. The reason of failure will be printed with the System.err stream. The UserTab allows a superuser to assign a new user who can then change the temporary password to his own. To do this, this new user simply needs to create an ODBConnect instance to the JODB server to which he or she is assigned and then uses the changePassword(oldPW, newPW) method to make the change. Example

```
import joeapp.odb.*;
import java.util.*;
public class ChangePW {
  public static void main(String[] args) throws Exception {
    ODBConnect odbc = new ODBConnect("localhost", 9999, "OldDummy", "dummy");
    if (odbc.changePassword("OldDummy", "Newdummy")) { // change OldDummy to Newdummy
      System.out.println("Password successufully changed.");
    } else System.out.println("Cannot change password.");
    odbc.disconnect();
  }
}
```

When LOG is enabled, a NEW log file named "DayName_Day_Month_Year_Timestamp.txt" will be created. Example: WED_30_10_2023_15H02Min17Sec.txt. And this file will remain active until LOG is disabled. The log content looks like this:

```
User joe successfully signs in. Assigned ID: joe@678590DAAC208
joe@678590DAAC208 is connected.
people (charset:UTF-8) connected with joe@678590DAAC208
autoCommit(true) set by joe@678590DAAC208
Object with key:  Obama of people read by joe@678590DAAC208
Key:  Obama of people locked by joe@678590DAAC208
Key:  Obama of people unlocked by joe@678590DAAC208
select(people,income GT 400000) by joe@678590DAAC208
sql(people,select name eq J* and age ge 40) by joe@678590DAAC208
joe disconnected.
ODBService is down.
```

ODBManager

ODBManager is started by ODBService and manages all activities around the open ODBs:

- ODB synchronization such as locking, unlocking, committing, etc.
- ODMS for the physical data accesses,
- ODBClusters on behalf of ODBWorkers to access remote JODBs in the cluster,
- Monitoring ODBWorkers (as workers) and ODBClusters (as agents) so that there is no orphaned worker or agent.
- Propagating (broadcasting) events (JODB up, down, etc.) in the cluster,
- Graceful shutdown of ODBs in case of emergency

When a client connects to ODBService via ODBConnect or ODBMining, ODBService creates an ODBWorker which then represents the client to cooperate with ODBManager to access the ODBs (read, add, delete, update, lock, etc.). ODBManager executes the requested requests: either directly or indirectly via ODBCluster

ODBWorker and ODBCluster

ODBWorker, as mentioned, is created by ODBService on behalf of a client app. ODBCluster is created by ODBManager on behalf of ODBWorker to access remote ODBs in the cluster. From the perspective of a local ODBService, there is no difference between client apps and ODBManagers over ODBClusters. Both are clients and both share the same corresponding ODBWorker to work with. And in this sense, ODBCluster is a special form of ODBConnect. To distinguish between the workers, the ODBWorker that works for clients is called worker the other is called a agent. Moreover, only workers of clients with higher privileges (2 or 3) can create an empty ODB upon request from the client app if the ODB with the specified database name does not exist, while agents cannot.

Figure 24 – Workers and Agents

UserList

The userlist is automatically created with ONE superuser (password:system, ID:admin, both lowercase) from UserList API if this userlist file does not exist in the specified path from the config-file. As mentioned, this superuser can add new users (with any privilege), but it cannot be used to access ODBs in JODBs. For the newly created users the userlist must be saved before these users can start working with ODBs. If the userlist is modified, the list must be saved explicitely, otherwise the modification will be lost. UserList entry format: UserPassword**:**UserID**@**Privilege where UserPassword and userID cannot contain a colon (:) or an at (@). Privilege is between 0 and 3.

EnDecrypt

Passwords, IDs and also the userlist of users are encrypted when sent over the network and decrypted by the local recipients. An encryption/decryption algorithm is only secure if as few people as possible know about it. In this ODB package I include a dummy algorithm and my own EnDecrypt.class (no source). You can use it or exclude it and use your own algorithm. However, the method names must be preserved and match the parameters. The dummy EnDecrypt.java

```
public class EnDecrypt {
  /**
    proprietary decrypt algorithm
    @param inp String to be decrypted
    @return decrypted String (null if invalid inp)
    @exception Exception thrown if inp is NOT encrypted by encrypt()
  */
  public static String decrypt(String inp) throws Exception {
    if (inp != null) try {
      StringBuilder sb = new StringBuilder(inp);
      // your algorithm here
      return sb.toString();
    } catch (Exception ex) { }
    throw new Exception("Unknown or corrupted Encrypted String:"+inp);
  }
 /**
    proprietary encrypt algorithm
    @param inp String to be encrypted
    @return encrypted String
    @exception Exception thrown if the inp is less/equal than 5 letters
  */
  public static String encrypt(String inp) throws Exception {
    if (inp != null && inp.length() > 5) try {
      StringBuilder sb = new StringBuilder(inp);
      // your algorithm here
      return sb.toString();
    } catch (Exception ex) { }
    throw new Exception("Input String:"+inp+" is too small.");
  }
}
```

ODMS

ODMS or Object Data Management System is responsible for the logical and physical IOs between the physical storage and the ODBManager. The object data is compressed in GZIP and stored without a suffix in the specified path from the config file. Each object is stored together with its corresponding key in the following format as an Object Entry:

2 bytes Key-Length	4 bytes Object length	Key in bytes	Object in bytes

This format allows storing keys and objects of variable size. A 2-byte key length can be a maximum of 32736 bytes for a key and a 4-byte object length can be a maximum of 2147483647 bytes for an object. However, the larger an Object Entry is, the slower the access is.

6. The ODB package and some hints

As with traditional RDB programming, ODB's serialized objects must be locked before being updated or deleted, then committed, or they must be rolled back for some reason and then unlocked so that others can access them. And all these steps can be done with ODBConnect (or its extension ODBMining).The order is as usual:

- Lock
- Update or Delete
- Commit or Rollback
- Unlock (only necessary with update)

If the sequence is too cumbersome for you in some cases, you can set autoCommit(true) - the default value is autoCommit(false). With this autoCommit(true) ODBManager executes the commit and unlocks the object key. However, a rollback is no longer possible.Therefore autoCommit(true) should be used with caution or in tandem with autoCommit(false).

The two ODBConnect methods xUpdate() and xDelete() are the extended update and extended delete methods. xUpdate performs locking, updating, committing and unlocking. xDelete performs locking, deleting and committing (no unlocking because the key and object are deleted afterwards).

Modified ODB objects should be saved (with the save() method) occasionally to ensure that everything is persistent. Even ODBConnect takes care of unexpected emergencies (such as shutdown) and automatically issues a disconnect so that ODBManager can close the open ODBs gracefully. But it is best practice for ODBs to be explicitly closed (with the close() method). The difference between emergency closing and graceful closing is that graceful closing rolls back or commits the modifications depending on the autoCommit() setting.

The ODB package contains two JODB servers. One is in SWING, the other in JavaFX (MVC technology). The SWING JODB server is based on my proprietary MVC SWING technology. Therefore, this SWING server requires an additional SWING MVC package: joemvc.jar. And besides, the example People package (see Fig. 5 and Fig. 6) is also based on this MVC SWING (see here: GITHUB and the PDF document).

Note: EnDecrypt.java contains two empty methods that should be implemented by your own encryption/decryption algorithm. The EnDecrpyt.class file is intended for use in case you do not want to implement an alternative algorithm or do not have one.

Directory	APIs or Sources	Comment
ODBCore	- EnDecrypt.class	My proprietary EnDecrypt algorithm for common use
	- EnDecrypt.java	Dummy source. Client and JODB server
	- JFXOptions.java	Client and JODB server (optional)
	- JOptions.java	Client and JODB server (optional)
	- ODBBroadcaster.java	JODB Server
	- ODBCluster.java	JODB Server
	- ODBConnect.java	Client
	- ODBEvent.java	Client and server
	- ODBEventListener.java	Client and server
	- ODBEventListening.java	Client and server
	- ODBInputStream.java	Client and server
	- ODBIOStream.java	Client and server
	- ODBManager.java	JODB Server

	- ODBMining.java - ODBObjectView.java - ODBParms.java - ODBParser.java - ODBService.java - ODBWorker.java - ODMS.java	Client (optional) JODB Server JODB Server JODB Server JODB Server JODB Server JODB Server
Nodes - JFX	 - JFXController.java - JFXODBServer.java - ODBTab.java - ServerTab.java - UserTab.java - odbConfig_Node1.txt - odbConfig_Node2.txt	JFX: Subdirectory of Nodes JODB server in JavaFX JODB server JODB server JODB server JODB server Config file example for Node_1 Config file example for Node_2
	icons - bee.gif	subdirectory of JFX
	resources - jfxServer.fxml - joe.css - manifest.mf - ODBTab.fxml - ServerTab.fxml - UserTab.fxml	subdirectory of JFX fxml for JFXODBServer the css file the manifest file for JAR fxml for ODBTab fxml for ServerTab fxml for UserTab
- SWING	 - JOODBServer.java - ServerController.java - ODBTab.java - ServerTab.java - UserTab.java - odbConfig_Node1.txt - odbConfig_Node2.txt	SWING: subdirectory of Nodes JODB Server in SWING
	icons - oodb.jpg	subdirectory of SWING
	resources - manifest.mf - joodbServer.txt - serverTab.txt - userTab.txt - odbTab.txt	subdirectory of SWING Manifest file for JAR SWING-MVC for JODBServer SWING-MVC for ServerTab SWING-MVC for ODBTab SWING-MVC for UserTab
- oodb_Node1	userlist	the userlist
- oodb_Node2	. . . . . .	some ODBs some ODBs
- log_Node1 - log_Node2	 log files of node 1 log files of node 2	 subdirectory of Nodes subdirectory of nodes
Examples - datamining	 - ODBmix.java - Member.java - Animal.java - MultipleUsers.java	 App to create ODBmix of serialized Member object and serialized Animal object Simulation of multiple users on ODBmix
- people	 - CreatePeoleODB.java - people.txt - People.java - PeopleController.java - ThePeople.java	 App to create ODB people based on the file people.txt serialized People object SWING MVC controller the main App
	resources - manifest.mf - people.txt	subdirectory of people the manifest file for jar the SWING MVC view text file
- eDict	 - eDict.java - eOpenDict.java - eDialog.java	Direct access without serialized object the JFX app API interface to ODBConnect the usual dialog
	icons	subdirectory of eDict

	- find.png - save.png	
	resources - style1.css	subdirectory of eDict
- dDict	- eDict.java - eOpenDict.java - eDialog.java	Direct access with multiple Character sets same subdirectories icons and resources
- pDict	- eParms.java - eDict.java - eOpenDict.java - eDialog.java	dDict with serialized object: eParms Serialized object. Multiple Character sets same subdirectories icons and resources

Joe T. Schwarz