OBJECTIVITY MOBILE DEVICE APIs

1. INTRODUCTION

Objectivity/DB applications generally use an API that is linked with the application code and the Objectivity/DB runtime (kernel) libraries. The exceptions are the SQL++/ODBC interface and Java RMI or JDBC code written by our users. Objectivity works best on a reliable, low latency and high bandwidth link between the application process (or thread) and remote page or lock servers. Mobile devices generally run over wireless or Infra Red links that are unreliable, high latency and low bandwidth.

The Objectivity runtime libraries have a memory footprint of between 1.2 and 3 Megabytes. The binary code needs as much as 30 Mb of disk. Single process applications that only require a local database may use our standard product if there is enough memory. This is fine on today's PCs or laptops. However, most Personal Digital Assistants [PDAs] and cellphones have small amounts of volatile and persistent memory. Nearly all of them use platforms that are currently unsupported by Objectivity/DB, e.g. PalmOS, Windows CE, pSoS, VxWorks or QNX.

The Mobile Device APIs will extend the current product to make it more suited to mobile or limited memory real-time devices working in a wide variety of wireless environments.

2. OVERVIEW

This document proposes that we add three new mobile device APIs:

- **Objectivity/Remote -** This provides a remotely invoked version of the Release 9+ C++ and Java APIs (or a subset) on the mobile or remote device.
- **Objectivity/Mobile** – This provides the same or a smaller subset of the Release 9 Java API on the mobile device using a different underlying kernel. It can be configured to access remote databases by calling Objectivity/Remote or by acting as a standard Objectivity client.
- **Objectivity/SYNCML** – This uses the emerging SYNCML protocol to transfer objects to and from an object server.

The C++ and Java interface subset could be supported by many standalone small footprint DBMSs or to provide a more portable Open Source front end to Objectivity/DB databases. SYNCML is an emerging standard for synchronizing data between devices. It

would have a totally new API to a new server built using our standard C++ or Java products.

3. Objectivity/Remote

3.1 Functionality Required

Objectivity/Remote invokes standard Objectivity C++ or Java APIs running in a server via the RMI protocol. It must be able to access at least the following functionality:

a) The equivalent of the com.objy.db package for handling exceptions and obtaining version information. Only the exceptions that can be generated by the functionality outlined in b), c) and d) need be supported.
b) The equivalent of most of the com.objy.db.app package that provides general application interfaces. DBA functionality, such as partition manipulation could be excluded as the "server" application should be able to create the partition on the mobile device.
c) A subset of the com.objy.db.iapp package that provides interfaces for persistence.
d) The equivalent of the ooMap subset of the com.objy.db.util package (which provides the collection classes). The full set of collection classes probably couldn't fit into a very small footprint machine.

Outlines for the client and server side Java RMI methods are described in Appendix A. The full list of the excluded functionality appears in Appendix B.

3.2 Platforms To Be Supported

Objectivity/Remote should initially run on all of the platforms supported by Objectivity/C++ and Objectivity for Java. Later implementations should run on VxWorks, Windows CE and LynxOS . PalmOS may be viable if a reliable Java RMI can be run there.

4. Objectivity/Mobile

4.1 Functionality Required

Objectivity/Mobile implements the functionality subset described in Section 3.1, but it runs entirely locally in the mobile device. In effect, it is a trimmed down version of our standard product. If resources permit, an application should be able to toggle transactions between: purely local invocation using Objectivity/Mobile; use of Objectivity/Remote alone; or a mixture of both depending on the target database(s).

4.2 Platforms To Be Supported

Objectivity/Mobile should run on VxWorks, Windows CE, LynxOS and Linux. It may be ported to pSoS and QNX at a later date. A PalmOS implementation may be viable if a reliable Java can be supported and if there is adequate memory.

5. Objectivity/SYNCML

5.1 Functionality Required – *[To be done]*

Objectivity/SYNCML is an implementation of the SYNCML protocol that moves objects or commands (e.g. delete this object) between devices. This requires further study as it is closer to XML than C++ or Java and it only looks suitable for transferring small amounts of ASCII data. We should consider joining the SYNCML consortium (it's free) once we have determined that it is viable to implement the interface.

5.2 Platforms To Be Supported

Objectivity/SYNCML should eventually be supported on all platforms that support our standard product or Objectivity/Mobile. The initial platforms should be Solaris, NT and Linux as these are likely to be the most popular "servers" for the next few years.

[Leon] Addendum: We currently favor XMLrpc over SyncML for a number of reasons.

APPENDIX A – RMI Conventions To Be Used By Objectivity/Remote.

A1. Declaring the methods to be called from the remote device

```
// Each method extends the RMI Remote class and must throw a RemoteException.

import java.rmi.*;

public interface ooRemoteMethods extends Remote{

    public ooDBObj getDB() throws RemoteException;   // As an example.
    // declare the other methods similarly.
}
```

A2. The ooRemoteServer class that implements the methods and performs remote service registration

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
public class ooRemoteServer extends UnicastRemoteObject
                implements ooRemoteMethods{

    // A default constructor is needed, but keep it empty
    public ooRemoteServer(){
        // this will automatically call super();
    }

    // Now implement the methods in the remote interface, for example…

    public ooDBObj getDB() throws RemoteException {
        // Put the code that calls the regular API here
    }

    // …Repeat similar implementations as necessary

    // Perform registration in the main method.
    public static void main(String args[]){
        try{
            // Register using a port. The default port is 1099.
```

```
                // This step can be omitted if you start rmiregistry
                // using the command line:  start rmiregistry port_number;

                LocateRegistry.createRegistry(port_number);


                /* next bind an object of this class to a URL of form
                rmi://machine_name:port_number/class_name
                current machine is the default machine and 1099 is
                the default port */

                ooRemoteServer rsObj = new ooRemoteServer();
                Naming.bind("rmi://:port_number/ooRemoteServer", rsObj);
                System.out.println("RMI Registration successful");
            }
            catch(Exception e){
                System.out.println("RMI Registration failed");
            }
        }
    }  // end Main. A real server would need multiple threads, of course.
```

A3. An Objectivity/Remote Client Program

The following client code should be linked with some of the files that the RMI tool produces when the above classes are processed.

```
import java.rmi.*;
public class ooRemoteClient{
    public static void main(String args[]){
        ooRemoteMethods rm;
        try{
            // get a reference to remote methods from registry
            // Naming.lookup() returns Remote object which must be
            // cast to our interface

            rm= (ooRemoteMethods)Naming.lookup(
                "rmi://server_name:port_number/ooRemoteServer");
            ....
            // call remote methods as follows
            ooDBObj db;
            db= rm.getDB("MyRemoteDB");
            ...
```

```
            }
            catch(Exception e){
                System.out.println("Remote call failed");
            }
        } // end main
    } // end ooRemoteClient
```

APPENDIX B – FUNCTIONALITY THAT MAY BE EXCLUDED

B1. Features that may be omitted in Objectivity/Remote

B1.1 com.objy.db package for handling exceptions and obtaining version information

- Only the exceptions that can be generated by the functionality outlined in B1.2-4 are supported.

B1.2 com.objy.db.app package that provides general application interfaces

- Interface ClusterReason
  - Only the Application field value and the getreason method are supported
- Interface ClusterStrategy
  - Only the Explicit Clustering strategy is supported.
- Interface oo
  - Not supported as the explicit values may be used instead.
- Interface ooId
  - Not supported.
- Interface SchemaPolicy
  - Not supported
- Class Connection
  - Schema Management is not supported
  - IPLS is not supported
- Class DefaultClusterStrategy
  - Not Supported

B1.3 com.objy.db.iapp package that provides interfaces for persistence

- The items that are reserved for internal use are not implemented.

B1.4 com.objy.db.util package that provides persistence capable collection classes etc.

- Only ooMap is supported

B2. Features that may be omitted in Objectivity/Mobile

The following features of the regular products may be omitted or may be configured out by the user to achieve a footprint of 250 Kb or less:

- Versioning.
- Creation, use and maintenance of indices (but affected indices must be marked as out of date at most once per session).
- DBA interfaces.
- User access to type information.
- Propagation of delete() and lock() operators.
- Schema migration and Active Schema
- Data replication.
- Predicate query.
- Legacy interfaces.
- (Possibly) Creation of new databases and containers.
- Automatic recovery.