Pervasync Database Synchronization Framework

User's Guide

Last revised: Jan 27th, 2023

Table of Contents

1 Installing Pervasync

1.1 Installing Java JDK

Pervasync server and the desktop version client are based on Java. Install JDK version 7 or newer on the host machines if not already installed. To check JDK version, do the following in a shell window:

```
java -version
```

OpenJDK is available at:

https://jdk.java.net/

For example, you can download JDK 17 from https://jdk.java.net/17/ and follow installation instructions on https://openjdk.java.net/install/.

Starting from Java version 17, Oracle makes Oracle JDK available for free, including all quarterly security updates. See this blog for more information. Download Oracle JDK from:

https://www.oracle.com/java/technologies/downloads/

Once JDK is installed, set the JAVA_HOME environment variable and also add the JDK "bin" folder to the PATH environment variable.

To set the JAVA_HOME and PATH on Windows:

Click Start > Control Panel > System
Click Advanced > Environment Variables.
Create or update the JAVA_HOME variable setting its value to JDK location, e.g.
`C:\Program Files\Java\jdk1.13.0\`
Add the location of the JDK bin folder to PATH in System Variables, e.g.
`C:\Program Files\Java\jdk1.13.0\bin;…`

To set the JAVA_HOME and PATH on Linux:

Create file jdk.sh in /etc/profile.d:

```
$ sudo su root
# vi /etc/profile.d/jdk.sh
```

Add the following lines to jdk.sh:

```
export JAVA_HOME=/usr/share/jdk1.13.0/
export PATH=$JAVA_HOME/bin:$PATH
```

Open a new shell window and run "`java -version`" to verify.

1.2 Installing Pervasync Server

There is a single Pervasync server package that you can download from the Pervasync web site. The server package can work with Oracle, PostgreSQL, Microsoft SQL Server and MySQL databases. Once the Pervasync server is up and running, users can download sync clients from the start page of the Pervasync server web interface.

The installation process includes un-packing the server package and setting it up.

1.2.1 Getting and Un-Packing Pervasync Server Distribution

The latest and greatest version of Pervasync server is available on the Pervasync products web page at:

https://www.pervasync.com/downloads

Download the server zip file, e.g. **pervasync_server-9.0.3.zip**, and save it on your server computer.

To install Pervasync server, choose a directory and unpack the zip file there. For example, the following commands un-pack the server distribution and create sync server home **/pervasync_server-9.0.3** on a Linux/Unix machine.

```
cd /
unzip pervasync_server-9.0.3.zip
```

On Windows platform, replace "/" with "\". By the way the sync home doesn't have to be under the top most root directory.

NOTE: Perform the setup as the user that will run the Pervasync server.

NOTE: On Windows, avoid installing Pervasync under "Program Files" folder.

A Pervasync server home has the following directory layout:

```
pervasync_server-9.0.3
   |
   +--- bin         // contains executable scripts for Pervasync server
              setup.
   |
   +--- classes     // contains Pervasync java classes
   |
   +--- config      // contains configurations files
   |
   +--- demo        // contains demo apps
   |
   +--- doc         // contains documentation
   |
   +--- lib         // contains library jars
   |
   +--- web         // contains server web app
   |
   +--- tomcat      // contains Tomcat server
   |
   +--- README.txt        // the readme file
   |
   +--- setup.bat(sh)     // command for launching the setup GUI tool
   |
   +--- shutdown.bat(sh)  // command for shutting down the Tomcat server
   |
   +--- startup.bat(sh)   // command for starting up the Tomcat server
```

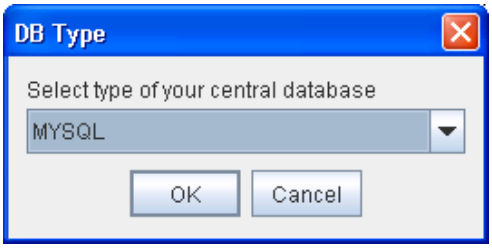
1.2.2 Setting up the Pervasync Server Using the GUI Setup Tool

If you have a hosted server and can only access the server via telnet or ssh sessions, the GUI app won't run. However, it is not difficult to set up a graphic remote desktop to the server. For example, you could use VNC. See http://www.realvnc.com/support/getting-started.html for details. Otherwise, see section 1.2.3 for steps to use the non-GUI setup scripts in folder "bin/<db type>" to set up the server.

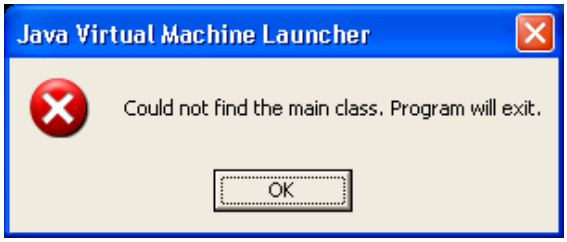
Optionally you may want to edit the conf files located in the config directory. Some of the configurations, such as pervasync.server.db.user.options and pervasync.server.db.user.grants, affect setup as well as server runtime. If you do edit the conf file, do it before you launch the Setup Tool GUI so that the new settings can take effect.

Launching the Pervasync Server GUI Setup Tool

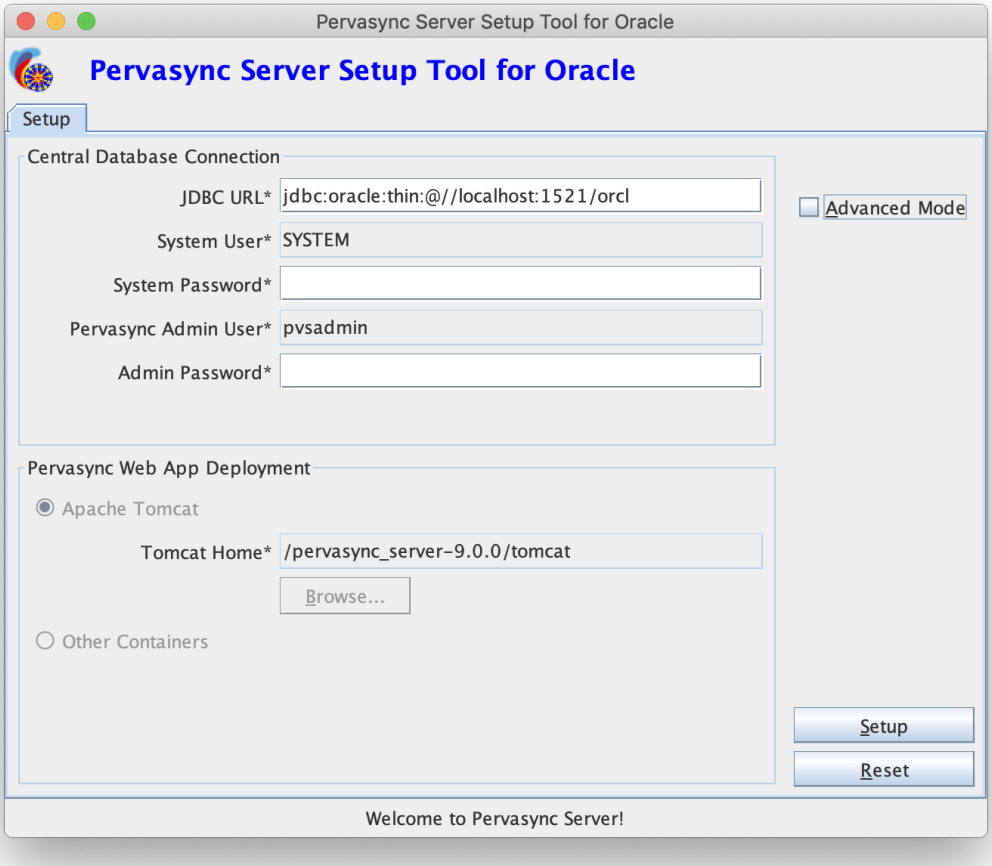
To launch the Pervasync Server Setup Tool, change directory to Pervasync server home and invoke "setup.bat" for Windows or "setup.sh" for Linux. A GUI window will pop up asking for the database type of your central DB.



NOTE: If you see the following error, most likely you are using a Java version older than JDK 7. Download and setup JDK 7 or newer.



In the DB type dialogue window, select MySQL, PostgreSQL, Microsoft SQL Server or Oracle and click OK. Then you will be presented with the main window with a Setup tab.



Use the “Setup” tab to setup the sync server. Carefully examine all the input fields and fill in proper values. Fields that have an asterisk (*) next to their names are required to be filled. Some fields are made read-only and you normally shouldn’t change them. If you do need to change their values, click “Advanced Mode” on the top right corner. See next section for descriptions of the fields.

TIP: Move mouse cursor over a field input box to show the description of the field (tooltip).

Central DB Connection Properties

Sync server needs a Pervasync admin schema to store its metadata. To create that schema, the database root/system user name and password are needed. The top panel of the setup screen is for the database connection. Listed below are the explanations of its input fields:

- **JDBC URL** – This is the URL to the Pervasync server repository database. By default, it’s *jdbc:oracle:thin:@//localhost:1521/xe* for Oracle, jdbc:sqlserver://localhost:1433 for MS SQL Server, jdbc:postgresql://localhost:5432/postgres for PostgreSQL and *jdbc:mysql://localhost:3306/* for MySQL. You may need to edit it to reflect your database’s actual host name, port number. Also for Oracle you need the service name (“xe” in the

above example) and for PostgreSQl you need the database name ("postgres" in the above example) in the JDBC URL.

- **Database Name** – This is for MS SQL Server only. A server instance may have multiple databases. Specify the name of the database that will host the sync metadata and the schema tables to be synced.

- **System Use**r – Name of a DB user with System privileges. Normally you would use *SYSTEM* for Oracle, *sa* for MS SQL Server, *postgres* for PosrgreSQL and *root* for MySQL. For Amazon AWS, use the RDS DB master user. This user/account should be pre-existent. The System user is only used to create/drop the Pervasync Admin user during setup/reset. The System password is not persisted by Pervassync for security reasons.

- **System Password** – System user's password.

- **Pervasync Admin User** – Pervasync server admin user name. You would use this username to login to the web-based Pervasync admin console after setup. If not already existent, a database user/schema with this name will be created in Pervasync DB repository during setup. At reset time, this user/schema will be dropped. Note that you should not use System User for Pervasync Admin User. Also this user should not pre-exist.

- **Admin Password** – Pervasync server admin user's password.

Pervasync Web App Deployment

The bottom panel of the setup screen is for Pervasync web app deployment. The **web** folder in Pervasync server home folder contains a standard J2EE web app for the sync servlet and admin console.

Pervasync server comes with a bundled Tomcat server which is our recommended way of serving the web app.

Use advanced mode if you would deploy the app to your own Tomcat server or other J2EE containers. If you select Tomcat as the container, you can click "**Locate Tomcat Home**", then browse and find **Tomcat Home** directory. The web app will be automatically deployed to Tomcat during setup. If you want to use a non-Tomcat container, you need to deploy it yourself following the specific web container's deployment instructions.

Performing Setup

Now, simply click "**Setup**" button to start the setup process, which will do the following:

1. Set up the sync server home.

2. Create the sync server admin user/schema in the database if it does not exist; or upgrade the schema if it already exists.

3. Deploy the sync server web application to Tomcat container.

NOTE: Do not move or remove the sync server home after setup.

NOTE: You could setup multiple sync server instances on different machines. All these sync servers share one and only one sync server DB user. To set up a new instance, repeat the setup on a new host machine.

Updating Server Settings

If you need to change Pervasync server settings after you have configured Pervasync server, you can go to the Pervasync home folder and invoke “setup.bat” for Windows or “setup.sh” for Linux. The setup main screen will pop up. You can make changes and click “Update”.

NOTE: The Pervasync admin DB and web admin console share the same set of user name (default pvsadmin) and password. To change the admin user password, follow these steps:

1. Change the Pervaync admin DB user password. Use “ALTER USER” command for Oracle and “SET PASSWORD” command for MySQL.

2. Run “setup.bat” for Windows or “setup.sh” for Linux. You will see a dialogue saying “Cannot connect to Pervasync Admin DB”. Click the “Continue” button.

3. Fill in the System password and the new Pervasync Admin password. Click the “Setup” button. You should now be able to log in to the web admin console with the new password.

1.2.3 Setting up the Sync Server Using the Non-GUI Setup Scripts

This section describes the setup steps using the non-GUI setup scripts in folders “**bin/<db type>**” of Pervasync server. Skip this section if you have finished setting up the sync server using the Pervasync Server GUI Setup Tool described in section 1.2.2.

Edit the ini File

Locate file **pervasync_server_<db type>.ini** under directory **bin/<db type>**. Use your favorite editor to edit the property values.

- **pervasync.server.db.url** – – This is the URL to the Pervasync server repository database. By default, it’s *jdbc:oracle:thin:@//localhost:1521/xe* for Oracle, jdbc:sqlserver://localhost:1433 for MS SQL Server, jdbc:postgresql://localhost:5432/postgres for PostgreSQL and *jdbc:mysql://localhost:3306/* for MySQL. You may need to edit it to reflect your database’s actual host name, port number. Also for Oracle you need the service name (“xe” in the above example) and for PostgreSQl you need the database name (“postgres” in the above example) in the JDBC URL.

- **pervasync.server.db.system.use**r – Name of a DB user with System privileges. Normally you would use *SYSTEM* for Oracle, *sa* for MS SQL Server, *postgres* for PosrgreSQL and *root* for MySQL. For Amazon AWS, use the RDS DB master user. This user/account should be pre-existent. The System user is only used to create/drop Pervasync Admin user during setup/reset. The System password is not saved.

- **pervasync.server.db.database.name** – This is for MS SQL Server only. A server instance may have multiple databases. Specify the name of the database that will host the sync metadata and the schema tables to be synced.

- **pervasync.server.db.system.password** – System user's password.

- **pervasync.server.admin.user** – Pervasync server admin user name. You would use this username to login to the web-based Pervasync admin console. If not already exists, a database user/schema with this name will be created in Pervasync DB repository during setup. At reset time, this user/schema will be dropped.

- **pervasync.server.admin.password** – Pervasync server admin user's password.

NOTE: Optionally you may want to edit the conf files located in the **config** directory. Some of the configurations, such as **pervasync.server.db.user.options** and **pervasync.server.db.user.grants**, affect setup too.

Run the Setup Scripts

In the following we use Oracle as an example. Replace *oracle* with *mysql* for MySQL DBMS.

Windows:

```
cd bin\oracle
.\pervasync_server_oracle_setup.bat
```

Linux/Unix:

```
cd bin\oracle
./pervasync_server_oracle_setup.sh
```

These setup scripts will create the sync server DB user/schema if it does not exist; or upgrade the schema if already exists.

NOTE: If you see the following error, most likely you are using a Java version older than JDK 7. Download and setup JDK 7 or newer.

```
Exception in thread "main" java.lang.UnsupportedClassVersionError: Bad
              version number in .class file
```

To drop the sync server DB user, run the drop DB scripts **pervasync_server_oracle_reset.bat** or **pervasync_server_oracle_reset.sh** depending on your OS platform**.**

NOTE: To run the batch scripts successfully**, pervasync_server_oracle.ini** must have complete and correct information. After you run the scripts you may want to erase sensitive information such as passwords from the **ini** file for security reasons.

1.2.4 Starting Up Tomcat

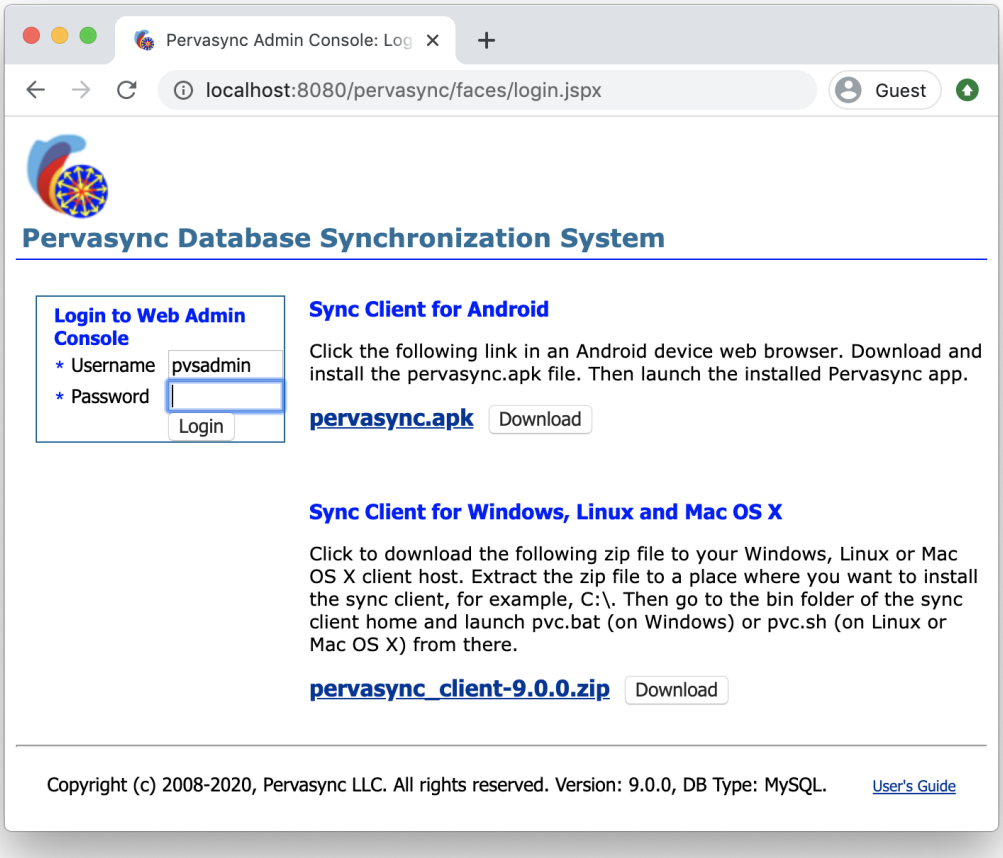
Once the setup completes successfully, restart Tomcat server:

```
cd pervasync_server-9.0.3
./shutdown.sh
./startup.sh
```

You can then access the start page of the Pervasync server web at:

http://localhost:8080/pervasync

If you see the Pervasync web app start page like that showing below, then your sync server setup has been successful. Congratulations!



Use **Pervasync Admin User** and **Admin Password** you specified in the GUI of server setup to login to the web based admin console after which you can publish central database tables and create sync users.

On the same Pervasync start page there is a link to the Java based sync client for end users to download.

NOTE: Users do not need to login to the web based admin console to download the sync client.

If your Internet Browser cannot display the web page, or it can't establish a connection to the server, then you need to check your servlet container (e.g. Tomcat) to see whether or not it is started. If your servlet container is up but Pervasync is still not available, first check the log files of your servlet container (e.g. files inside the logs folder of Tomcat home) and then the log files in the log folder of Pervasync home to see if there are error messages.

Changing the port of the Tomcat server

Tomcat runs on port 8080 by default. To change to a different port, edit Tomcat's server.xml file located at `<Pervasync server home>/tomcat/conf`. Search for

```
<Connector port="8080" protocol="HTTP/1.1"
```

and in server.xml edit the port number, then restart server.

Relaying HTTP Port 80 Connections to Tomcat Port 8080 on Linux

To have the Tomcat server itself listen on HTTP port 80 on Linux, Tomcat would have to run as `root` since only `root` can listen on ports below 1024. But for security reasons this is not recommended.

One solutionis to use the Netfilter package that already comes with Linux and is transparent to Tomcat. Execute the following commands as `root`:

```
# iptables -t nat -I PREROUTING -p tcp --dport 80 -j REDIRECT --to-ports 8080
# iptables -t nat -I OUTPUT -p tcp --dport 80 -j REDIRECT --to-ports 8080
```

The first rule redirects incoming requests on port 80 generated from other computer nodes and the second rule redirects incoming requests on port 80 generated from the local node where Tomcat is running.

Rules created with the `iptables` command are only stored in RAM. To save them so that they will still work on reboot, do the following as root:

```
# service iptables save
```

This causes the `iptables` init script to run the `/sbin/iptables-save` program and write the current `iptables` configuration to the `/etc/sysconfig/iptables` file.

There is one Tomcat configuration parameter that you may want to change, the `proxyPort` parameter in the `<Pervasync server home>/tomcat/conf/server.xml` file for port 8080:

```
<Connector port="8080" protocol="HTTP/1.1" proxyPort="80"
```

You need to restart Tomcat to make this change effective.

Running Tomcat as a Service on Windows

You would need to run Tomcat as a Service if you want it to keep running after you log out of your Windows account.

cd to <Pervasync Server Home>/tomcat/bin/ in a terminal window and execute the following command

```
service.bat install
```

The above command would make Tomcat7 a Windows service. Make sure the command succeeded. If not, you may want to try uninstalling the existing service first:

```
service.bat remove
```

After the service is installed, you would need to launch the Tomcat GUI to config and start the service:

```
Tomcat7w.exe
```

You would want to change the “Startup type” to “Automatic” under the “General” tab. Also adjust the memory pool settings under the “Java” tab. Then click on “Start” to startup the service.

1.2.5 Applying a License Key

To apply the license key, login to the web admin console, go to “Home”, locate and click “Apply License Key” under “Other Admin Tasks”. Then fill in the license key and click Apply.

Alternatively you can also apply the key by manually modifying the server config file. Locate the server config file in folder <Pervasync server home>/config, e.g. pervasync_server_mysql.conf for MySQL. Edit the following line and replace the value “0” with the license key:

```
pervasync.server.license.key=0
```

Then restart the server.

NOTE: License key could be applied before or after the server setup.

1.3 Installing Pervasync Client on Windows, Linux or Mac OS X

This section is for Java SE based sync clients supporting Oracle, MySQL, Microsoft SQL Server, PostgreSQL and SQlite on desktop/laptop hosts. See next sections for sync client setup for Android and iOS devices.

Make sure you have Java JDK 7 or newer installed. Refer to section 1.1 for more info.

1.3.1 Getting and Un-Packing Pervasync Client Distribution

When you get your server up and running, local database users can download sync client from the start page of Pervasync server web app:

http://<server>:<port>/pervasync, e.g. http://localhost:8080/pervasync

Download the client zip file, e.g. **pervasync_client-9.0.3.zip**, and save it on the computer that runs the local database.

To install a Pervasync client, choose a directory and unpack the zip file there. For example, the following commands un-pack the client distribution and create sync client home **/pervasync_client-9.0.3** on a Linux machine.

```
cd /
unzip pervasync_client-9.0.3.zip
```

On Windows platform, replace "/" with "\". By the way the sync home doesn't have to be under the top most root directory.

A Pervasync client home has the following directory layout:

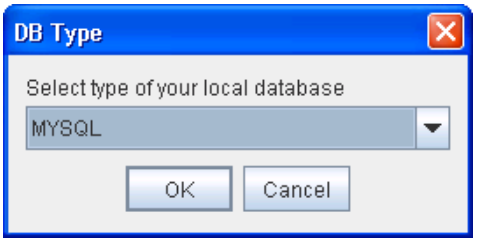
```
pervasync_client-9.0.3
   |
   +--- bin          // contains executables.
   |
   +--- classes      // contains Pervasync java classes
   |
   +--- config       // contains configurations files
   |
   +--- demo         // contains demo apps
   |
   +--- doc          // contains documentation
   |
   +--- lib          // contains library jars
   |
   +--- README.txt   // the readme file
   |
   +--- setup.bat    // the batch script for Windows to launch the setup tool
   |
   +--- setup.sh     // the batch script for Linux to launch the setup tool
```

1.3.2 Setting up the Sync Client Using the Pervasync Client Setup Tool

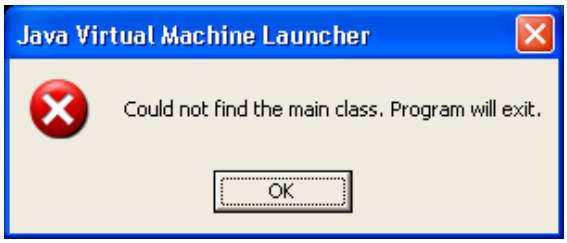
The Sync client comes with a GUI Setup Tool which is the preferred way to do client setup. Alternatively, you could also use the setup scripts located in the "**bin/<db type>**" folders to do the setup. The steps are described in the next section (1.3.3).

Launching the Sync Client Setup Tool by Invoking sertup.bat/setup.sh

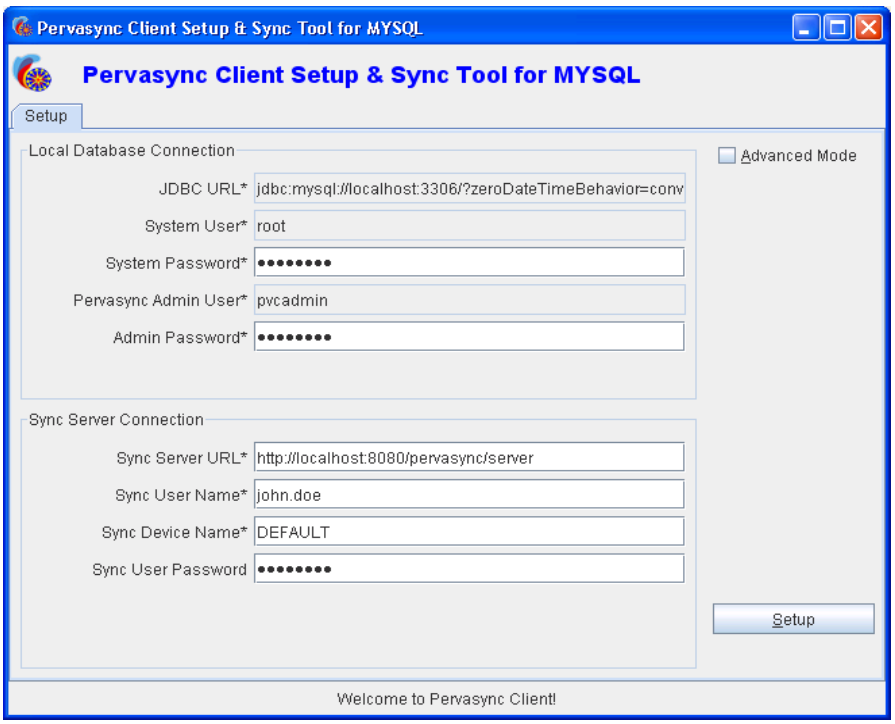
To launch the Sync Client Setup Tool, change directory to the Pervasync client home and invoke "setup.bat" for Windows or "setup.sh" for Linux. A GUI window will pop up asking the database type of local DB.



NOTE: If you see the following error, most likely you are using a Java version older than JDK 7. Download and setup JDK 6 or newer.



In the DB type dialogue window, select MySQL, SQLite, PostgreSQL, Microsoft SQL Server or Oracle and click OK. Then you will be presented with the main setup window.



Carefully examine all the input fields and fill in proper values. Fields that have an asterisk (*) next to their names are required. Some fields are made read-only. If you do need to change their values, click “Advanced Mode” on the top right corner. Read the following sections for explanations of each field.

Local Database Connection

Sync client needs a Pervasync admin schema to store its metadata. To create that schema, the database root/system user name and password are needed.

The top panel of the sync client setup tab is for local database connection info. Listed below are the explanations of text boxes of the GUI:

- **JDBC URL** – This is the URL to the Pervasync server repository database. By default, it’s *jdbc:oracle:thin:@//localhost:1521/xe* for Oracle, jdbc:sqlserver://localhost:1433 for MS

SQL Server, jdbc:postgresql://localhost:5432/postgres for PostgreSQL and *jdbc:mysql://localhost:3306/* for MySQL. You may need to edit it to reflect your database's actual host name, port number. Also for Oracle you need the SID ("xe" in the above example) and for PostgreSQl you need the database name ("postgres" in the above example) in the JDBC URL. In normal mode this field is read-only. Check "Advanced Mode" on the top-right corner to make it editable.

- **Database Name** – This is for MS SQL Server only. A server instance may have multiple databases. Specify the name of the database that will host the sync metadata and the schema tables to be synced.

- **System User** – Name of a DB user with System privileges. Normally you would use *SYSTEM* for Oracle, *sa* for MS SQL Server, *postgres* for PostgreSQL and *root* for MySQL. For Amazon AWS, use the RDS DB master user. This user/account should be pre-existent. The System user is only used to create/drop Pervasync Admin users during setup/reset. The System password is not saved.

- **System Password** – System user's password.

- **Pervasync Admin User** – Pervasync client admin user name. If not already existing, a database user/schema with this name will be created in Pervasync DB repository at setup time. At reset time, the user/schema will be dropped. Note that you should not use System User for Pervasync Admin User. Also this user should not pre-exist for the first time installation.

- **Admin Password** – Pervasync client admin user's password.

Sync Server Connection

The bottom panel is for sync server connection. Sync client needs the following information to connect to and authenticate with a sync server.

- **Sync Server Url** – Pervaync server URL. Use the host name/IP that has the sync servlet deployed. It's http://localhost:8080/pervasync/server by default.

- **Sync User Name** – Pervasync user name. The user and device should be created on sync server using server admin API or the web-based admin console. This can be done before or after client setup. The user name, device name and user password that are set here have to match those specified on the server so that the client can sync with the server.

- **Sync Device Name** – Pervasync user device name. By default, it's DEFAULT.

- **Sync User Password** – Pervasync user's password.

- **HTTP Proxy Host** – If the sync client resides inside a firewall and sync server resides outside the firewall, you need to set the HTTP proxy. Proxy host takes the host name or IP, for example: www-proxy.mycorp.com. You need to check/click "**Advanced Mode**" to be able to see and edit HTTP proxy info.

- **HTTP Proxy Port** – Proxy port takes the port number, for example: 80.

Performing Client Setup

Now, simply click on the "**Setup**" button to start the setup process, which will setup the sync client home, create the sync client DB user/schema if it does not exist, or upgrade the schema if it already exists.

Once the client setup completes successfully, you will see two more tabs, Sync and Schedule, added to the sync client main window. Before you can sync the client with the server, go ahead and follow the steps in section 2.1 to create the sync user and device for this client, a sync schema with a simple table and subscribe the client to the schema.

1.3.3 Setting up the Sync Client Using the Non-GUI Setup Scripts

This section describes the setup steps using the scripts located in the "**bin/<db type>**" folders.

NOTE: Skip this section if you have finished setting up the sync client using the Pervasync Client Setup GUI Tool described in previous section (1.3.2).

Edit the ini File

Locate file **pervasync_client_<db type>.ini** under directory **bin/<db type>/**. Use your favorite editor to edit the property values.

- **pervasync.client.db.url** – This is the URL to the Pervasync client repository database. By default, it's *jdbc:oracle:thin:@//localhost:1521/xe* for Oracle, jdbc:sqlserver://localhost:1433 for MS SQL Server, jdbc:postgresql://localhost:5432/postgres for PostgreSQL and *jdbc:mysql://localhost:3306/* for MySQL. You may need to edit it to reflect your database's actual host name, port number. Also for Oracle you need the SID ("xe" in the above example) and for PostgreSQl you need the database name ("postgres" in the above example) in the JDBC URL.

- **pervasync.client.db.system.use**r – Name of a DB user with System privileges. Normally you would use *SYSTEM* for Oracle, *sa* for MS SQL Server, *postgres* for PostgreSQL and *root* for MySQL. For Amazon AWS, use the RDS DB master user. This user/account should be pre-existent. The System user is only used to create/drop Pervasync Admin user during setup/reset. The System password is not saved.

- **pervasync.client.db.system.password** – System user's password.

- **pervasync.client.db.database.name** – This is for MS SQL Server only. A DB server instance may have multiple databases. Specify the name of the database that will host the sync metadata and the schema tables to be synced down.

- **pervasync.client.admin.user** – Pervasync client admin user name. If not already existent, a database user/schema with this name will be created in Pervasync DB repository at setup time. At reset time, the user/schema will be dropped.

- **pervasync.client.admin.password** – Pervasync client admin user's password.

- **pervasync.server.url** – Pervaync server URL. Use the host name/IP that has the sync servlet deployed. It's http://localhost:8080/pervasync/server by default.

- **pervasync.user.name** – Pervasync user name. The user and device should be created on sync server using server admin API or the web-based admin console. This can be done before or after client setup. The user name, device name and user password that are set here have to match those that are created on server so that the client can sync with the server.

- **pervasync.device.name** – Pervasync user device name. By default, it's DEFAULT.

- **pervasync.user.password** – Pervasync user's password.

- **pervasync.client.http.proxy.host** – If the sync client resides inside a firewall and sync server resides outside the firewall, you need to set the HTTP proxy. Proxy host takes the host name or IP, for example: www-proxy.mycorp.com.

- **pervasync.client.http.proxy.port** – Proxy port takes the port number, for example: 80.

NOTE: Optionally you may want to edit the conf files located in the **config** directory. Some of the configurations, such as **pervasync.client.db.user.options** and **pervasync.client.db.user.grants**, affect setup too.

Run the Setup Scripts

Then run the setup batch scripts: **pervasync_client_<db type>_setup.bat** on Windows, or **pervasync_client_<db type>_setup.sh** on Linux/Unix.

Windows example:

```
cd bin\oracle
.\pervasync_client_oracle_setup.bat
```

Linux/Unix example:

```
cd bin\oracle
./pervasync_client_oracle_setup.sh
```

These setup scripts will create the sync client DB user/schema if it does not exist; or upgrade the schema if already exists.

NOTE: If you see the following error, most likely you are using a Java version older than JDK 7. Download and setup JDK 6 or newer.

```
Exception in thread "main" java.lang.UnsupportedClassVersionError: Bad
              version number in .class file
```

To drop the sync client DB admin user, run the drop DB scripts **pervasync_client_<db type>_reset.bat** or **pervasync_client_<db type>_reset.sh** depending on the platform**.**

NOTE: To run the shell scripts successfully, **pervasync_client_<db type>.ini** must have complete and correct information. After you run the scripts you may want to erase sensitive information such as passwords from the **ini** file for security reasons.

Congratulations! You have finished server and client setup. Before you can sync the client with the server, go ahead and follow the steps in next section (2.1) to create the sync user and device for this client, a sync schema with a simple table and subscribe the client to the schema.

1.3.4 Customizing the Sync Client for Windows, Linux and Mac OS X with Pre-Seeded Configuration

For deployment of large number of sync clients, it is often desirable to customize the sync client with pre-seeded configuration so that

- it is much easier for end users to setup the client, and
- it is much harder for end users to mess up with things like local DB connection credentials.

The sync client zip file, e.g. pervasync_client-9.0.3.zip, is located on the sync server at <Pevasync server home>/web/download. You can unpack it, modify it and zip it back. Then when sync users download the client from the server web, they will get the modified version. Following are the steps.

1. Unpack the zip file, for example:

   ```
   cd \pervasync_server-9.0.3\web\pervasync\download
   unzip pervasync_client-9.0.3.zip
   ```

2. Now you have a sync client folder, `\pervasync_server-9.0.3\web\pervasync\download\pervasync_client-9.0.3`. Change directory to that folder and start the `setup.bat`/`setup.sh` program.

   ```
   cd pervasync_client-9.0.3
   setup.bat
   ```

3. Complete the client setup process. Sync client will save the local DB connection info and sync server connection info in file pervasync_install.dat under the “config” folder of sync client home.

4. Copy pervasync_install.dat to classes\pervasync\config\ under sync client home.

   ```
   copy ..\config\pervasync_install.dat ..\classes\pervasync\config\
   ```

5. Edit the copied file pervasync_install.dat under classes\pervasync\config\. Remove the properties that you don’t want to be pre-seeded. Any properties left in the file will become pre-seeded and read-only during the end user setup. You would want to leave the system and admin user/password there and remove sync user name, device name and password. Also don’t touch the encryption key.

6. Do a reset of the client using the Setup tab of the sync client setup tool..

7. Zip up the client home folder.

   ```
   cd \pervasync_server-9.0.3\web\pervasync\download
   zip pervasync_client-9.0.3.zip pervasync_client-9.0.3
   rmdir /S /Q pervasync_client-9.0.3
   ```

When end users download and setup your customized sync client distribution, certain fields will have pre-seeded, read-only values as you desired.

1.3.5 Cloning Pervasync Client for Windows, Linux and Mac OS X to Avoid Costly Initial Sync

Initial sync could take a long time if you have Gigabytes of application data or files to sync. If a large amount of the data or files are shared by all or groups of your clients, you could use cloning to avoid syncing them for each client.

NOTE: "Sharing the same data or files" means that clients subscribe to a schema table or a sync folder with the same parameter values. This includes no parameter subscriptions.

Let's call the client to clone from the "seed client" and the client to clone to the "target client". The idea is to sync the data and files to the seed client, make a copy of the seed client and change the client ID and user name to those of the target client. Following are the cloning steps.

1. Create publications, clients and subscriptions on Pervasync Admin Console.

2. Install and sync the seed client. For faster synchronization, install the client on the same host or network as the Pervasync server.

3. Copy the seed client to target client hosts. Includes Pervasync client home folder, local database and folders synced to client. Refer to documentation of the specific database type on how to clone the database schemas to another host. Remember to include both Pervasync admin schema (default "pvcadmin") and the application schema(s) when you clone the databases.

4. On Pervasync web Admin Console, click on the "Clients" tab and find the client ID, user name and device name of the target client. Also create subscriptions for the target client if you haven't done so.

5. On the target client host, connect to the Pervasync admin schema (default "pvcadmin") of the local database. Change the client ID, user name and device name in table pvc$sync_client_properties from seed client to target client. Assuming your target client ID is 12, client name is "user_2" and device name is "DEFAULT", do the following:

```
update pvcadmin.pvc$sync_client_properties set value='12' where name =
              'pervasync.client.id';
update pvcadmin.pvc$sync_client_properties set value='user_2' where name =
              'pervasync.user.name';
update pvcadmin.pvc$sync_client_properties set value='DEFAULT' where name =
              'pervasync.device.name';
```

6. On the target client host, open file <Pervasync Client Home>/config/pervasync_install.dat. You may need to change some parameter values. For example, if the database is on a different host from the sync client host, you will need to correct values of `pervasync.client.db.url`.

7. On the target client host, launch the sync client (setup.bat or setup.sh). You will find that the new values are reflected on the "Setup" tab. Use the same tab to update sync user password if needed.

8. Start a sync session. You will see that shared data/files are not re-synced while client specific data/file subscriptions are synced to client.

NOTE: Make sure you have all the sync folder files on the target host. Any missing files will cause delete operations sent to the server during synchronization. To avoid accidental deletions, consider using REFRESH-ONLY sync or setting the server config parameter "`pervasync.policy.check.in.deletes`" to "DISCARD" depending on your business logic.

1.4 Installing Pervasync Clients on Android

Pervasync clients for Android synchronize the on device SQLite databases with your central database. Normally you would embed the sync client inside your application. Refer to section 2.4 or Android native app integration.

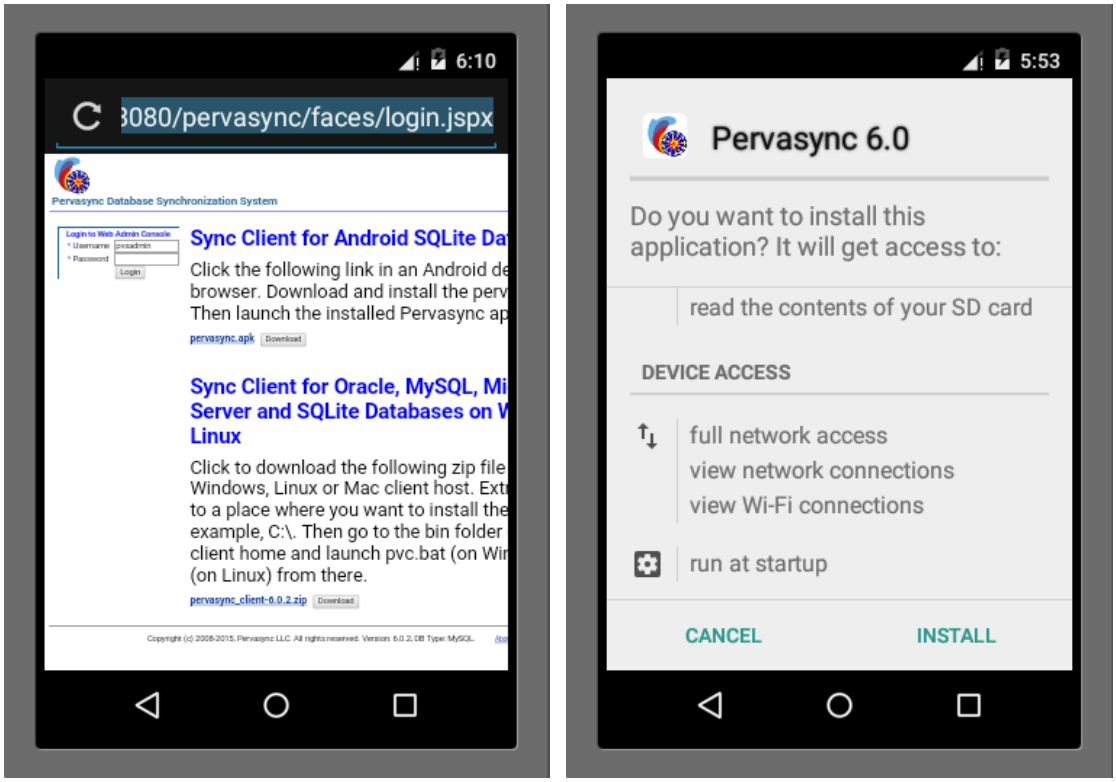
In This section we show you how to install and setup the standalone Android native app that's included out of the box.

1.4.1 Download and Install the Binaries

Enter the following URL to the web browser on your mobile device.

http://<sync server name>:<port>/pervasync

You will be presented with the Pervasync start page where you can find the links to the Pervasync client apk file for Android. Click to download and install.



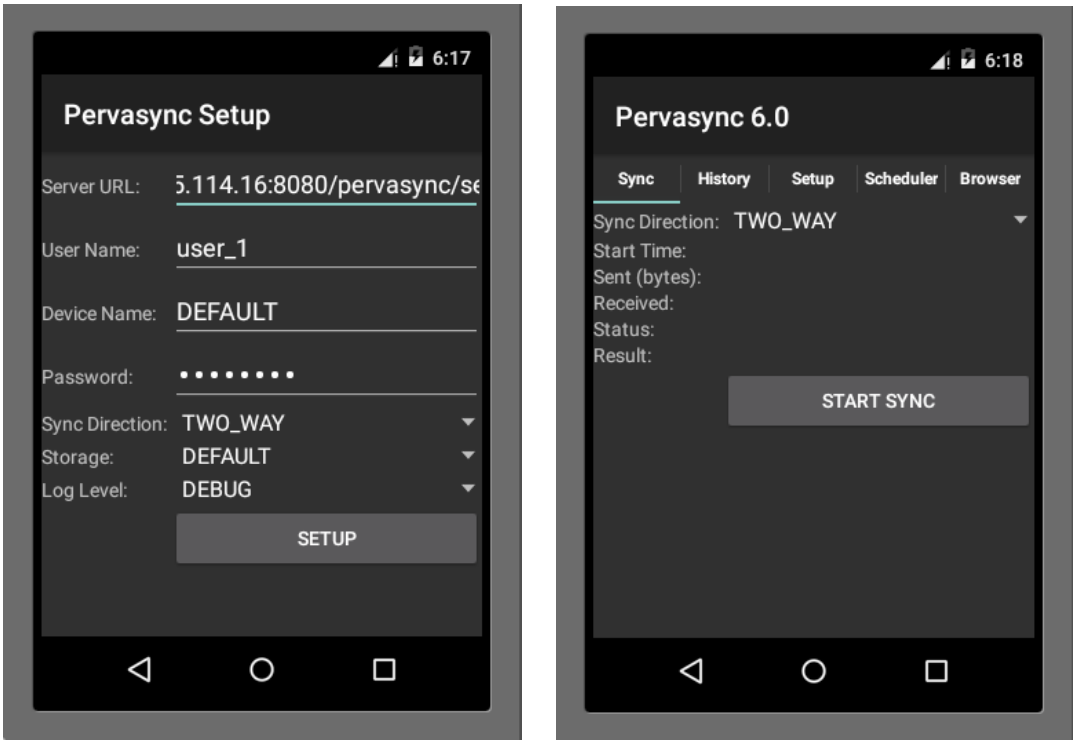
After a successful installation, you can find Pervasync app icon among other installed apps. Click to launch the client app.

1.4.2 Setup Pervasync Client

The first time you launch the Pervasync client, you will be presented with the sync client setup screen, which has the following fields to fill in.

- **Server Url** – Pervaync server url. Use the name or IP of the host that has the sync servlet deployed. Don't forget the port number if it's not 80.
- **Sync User Name** – Pervasync user name. The user and device should be created on sync server using server admin API or the web-based admin console. This can be done before or after client setup. The user name, device name and user password that are set here have to match those that are created on the server so that the client can sync with the server.
- **Device Name** – Pervasync user device name. By default, it's DEFAULT.
- **User Password** – Pervasync user password.

Optionally you can change the sync direction and log level.



Click the “SETUP” button to start the setup process. Once you are done with the setup, Pervasync client will be re-launched displaying the “Sync” tab. You are now ready to start a sync session. Jump to section 2.3 , 2.4 and 2.5 to learn how to synchronize, browse synced tables and files and how to embed the sync client into your on-device application.

1.5 Installing Pervasync Client for React Native

Find details of Pervasync client for React Native on github:

react-native-sync
react-native-sync-demo

1.6 Upgrading Pervasync

To upgrade to a new version, install the new version to a new folder and set it up following the same setup steps and use the same DB accounts as those in the previous setup. Pervasync DB repository will be automatically upgraded if needed. You may remove the old installation folder after the new installation folder is setup.

NOTE: Do not click on the RESET button during setup, which would wipe out the Pervasync DB repository among others.

NOTE: You need to apply a new license key if the new major or minor version number is different from that of the old version numbers.

1.6.1 Automatic Client Software Upgrade

NOTE: Automatic client software upgrade only works for Java SE based Pervasync desktop client. It doesn't work for Pervasync Android client.

Old versions of Pervasync clients can still work with newer Pervasync servers. However, newer servers may come with newer client software that may have new features or bug fixes. To upgrade to the latest client software, you can manually download and upgrade the clients or you can take advantage of automatic client software upgrade for desktop clients.

With automatic client software upgrade, updated client software files, e.g. Java classes, are synced to the client side and you just need to restart the client to upgrade to the latest version. On server side, the updated client software files are located at

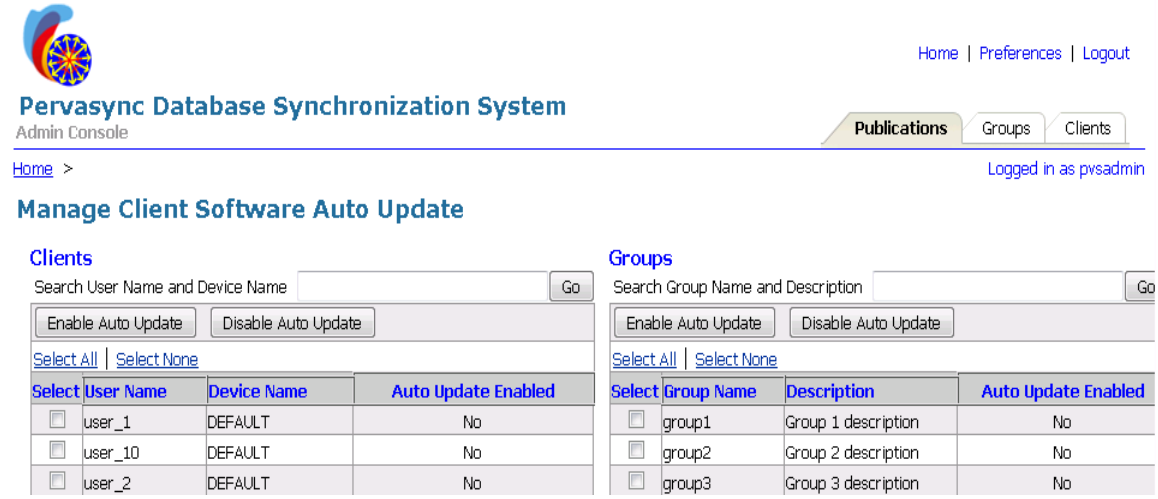
```
<Pervasync Server Home>/web/pervasync/download/pervasync_client_software_update
```

Java class files in classes sub-folder, jar files in the lib sub-folder and ".conf" files in the config sub-folder are included in the sync. Besides, classes/Pervasync/resource/pvc_gui.properties file is synced as well.

Do not touch the "pervasync_client_software_update" folder unless you want to push your own changes to the client side. For example, if you want to set some configuration parameter values for all your clients, you can modify the ".conf" files in the config sub-folder. Pervasync support also may ask you to apply one-off patches to some files to be propagated to clients. You may also be asked to turn on advanced mode by adding "pervasync.server.advanced.mode=TRUE" in server config file to make the software upgrade sync folder appear in "Sync Folders" on Pervasync web admin console. You can then update the sync folder publication to control what's included in the software update..

Pervasync clients will make a backup of the current version before downloading updates. For example, if your client folder is C:\ pervasync_client-8.0.1, a backup will be created in C:\pervasync_client-8.0.1_pvc_backup. In case the automatic software update corrupted the original client folder, you can use the backup folder to start over.

You can control which clients would get the software update and which wouldn't. Login to the web admin console, go to "Home", locate and click "Manage Client Software Auto Update" under "Other Admin Tasks". Select the client or the group to enable or disable automatic client software upgrade.


Home | Preferences | Logout
Pervasync Database Synchronization System
Admin Console
Publications
Groups
Clients
Home >
Logged in as pvsadmin
Manage Client Software Auto Update
Clients
Search User Name and Device Name
Go
Enable Auto Update
Disable Auto Update
Select All | Select None
Select
User Name
Device Name
Auto Update Enabled
user_1
DEFAULT
No
user_10
DEFAULT
No
user_2
DEFAULT
No
Groups
Search Group Name and Description
Go
Enable Auto Update
Disable Auto Update
Select All | Select None
Select
Group Name
Description
Auto Update Enabled
group1
Group 1 description
No
group2
Group 2 description
No
group3
Group 3 description
No

1.7 Importing and Exporting the Pervasync Metadata Repository

Pervasync provides a tool for you to export publication metadata to an XML file. The XML file can be used to restore the publications on another server. This is useful when you need to move your publications from a development/test environment to a production environment, assuming the two environments have the same schemas to be published.

1.7.1 Exporting to XML File

Look for the link "Export to XML File" on the Home screen of the Pervasync Admin console. Click to launch the exporting screen and perform the export.

NOTE: For security reasons, all user passwords of clients are exported as “welcome1”. Reset the passwords before or after you import the XML.

1.7.2 Importing from XML File

To import from the XML file, look for the link "Publish from XML File" on the Home screen of the admin console. Click to launch the importing screen and perform the import.

Since all user passwords of clients are exported as “welcome1”, you need to change them back to the original ones before you perform any client synchronization. To change password, you can either edit the XML file before importing or use the “Clients” tab of the admin console after the import. To edit XML file, open the file, find “welcome1”, then, replace them with your real passwords. If you want to use the admin console to change, click “Clients”, click “Update” for each user, type your real password in the textbox labeled as “New User Password”, click checkbox labeled as “Change Password”, and lastly, click “Update”.

1.8 Uninstalling Pervasync Server

To uninstall the server, you first reset it so that Pervasync admin repository is removed from the database. After that you can remove the Pervasync home to complete the uninstallation.

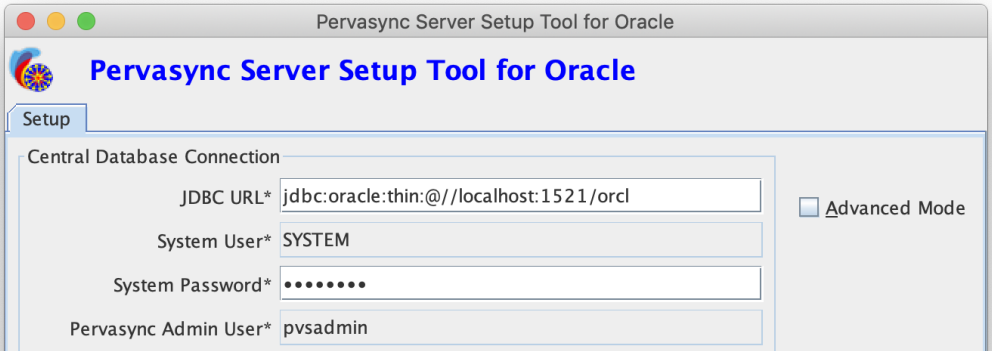
NOTE: A reset removes Pervasync objects from the database. In cases when you want to start over with your installation, you could do a reset followed by a setup.

NOTE: Un-publish all your publications before you reset the server so that the log tables and triggers created in the app schemas can be cleaned up.

1.8.1 Resetting the Server Using the Setup GUI

Just like setup, there are two ways to reset. This section describes the GUI way and the next section describes the non-GUI command-line way.

Go to the Pervasync home folder and invoke “setup.bat” for Windows or “setup.sh” for Linux. A GUI window will pop up:



On the Setup tab, fill in the required fields and click "**Reset**". What this does is removing Pervasync server admin schema from the DB and un-deploying sync server web application.

1.8.2 Resetting the Sync Server Using the Non-GUI Scripts

This section describes an alternative way to reset the server. Skip if you prefer doing it using the setup GUI described in the last section.

Remove the Server Admin Schema

To drop the sync admin schema, the database system user name and password are needed.

NOTE: Oracle is used below as an example. Replace it with the DB type you are using.

Locate file **pervasync_server_oracle.ini** under directory **bin**. Use your favorite editor to supply values to the properties.

Then run the reset batch scripts, **pervasync_server_oracle_reset.bat**: on Windows or **pervasync_server_oracle_reset.sh** on Linux/Unix.

Windows:

```
cd bin
.\pervasync_server_oracle_reset.bat
```

Linux/Unix:

```
cd bin
./pervasync_server_oracle_reset.sh
```

NOTE: To run the batch scripts successfully**, pervasync_server_oracle.ini** must have complete and correct information. After you run the scripts you may want to erase sensitive information such as passwords from the **ini** file for security reasons.

1.8.3 Removing Pervasync Server Home

To remove Pervasync server home, you can do something like the following:

Windows:

```
rmdir /S /Q C:\pervasync_server-9.0.3\
```

Linux/Unix:

```
rm –rf /pervasync_server-9.0.3/
```

1.9 Uninstalling Pervasync Client on Windows, Linux and Mac OS X

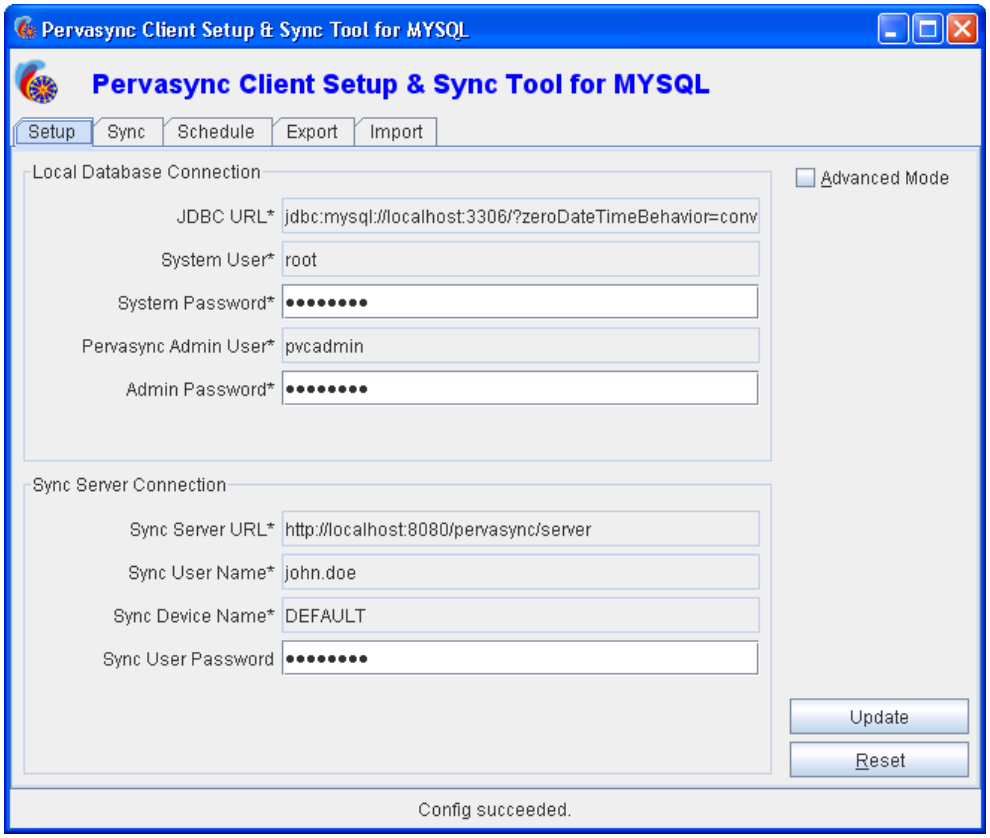
To uninstall the client, you first reset it so that Pervasync admin repository is removed from the database. After that you can remove the Pervasync client home folder.

NOTE: A reset removes Pervasync objects from the database. In cases when you want to start over with your installation, you could do a reset followed by a setup.

1.9.1 Resetting the Client Using the Setup GUI

Just like setup, there are two ways to reset. This section describes the GUI way. Next section describes the non-GUI way.

Go to the Pervasync client home folder and invoke “setup.bat” for Windows or “setup.sh” for Linux. A GUI window will pop up. Click on the “Setup” tab.



Fill in the required fields and click “**Reset**”. What this does is removing the client sync admin schema and your application schemas that have been created on your local DB.

1.9.2 Resetting the Sync Client Using the Non-GUI Scripts

This section describes an alternative way to reset the client. Skip if you prefer to do it using the setup GUI described in the last section.

Locate file **pervasync_client_oracle.ini** under directory **bin**. Use your favorite editor to edit its property values. The database system user name and password are needed.

Then run the reset batch scripts, **pervasync_client_oracle_reset.bat**: on Windows or **pervasync_client_oracle_reset.sh** on Linux/Unix.

Windows:

```
cd install
.\pervasync_client_oracle_reset.bat
```

Linux/Unix:

```
cd install
./pervasync_client_oracle_reset.sh
```

1.9.3 Removing Pervasync Client Home

To remove Pervasync client home, you can do something like the following:

Windows:

```
rmdir /S /Q C:\pervasync_client-9.0.3\
```

Linux/Unix:

```
rm -rf /pervasync_client-9.0.3/
```

1.10 Uninstalling Pervasync Client for SQLite on Android

For Android, you could just uninstall the app from the device.

1.11 Moving Pervasync Server to a New Host

NOTE: Pervasync license is based on server IP address. Obtain a new license key before you move your Pervasync server to a new host with a different IP address.

When you need to move your Pervasync server to a new host, you can always re-install from scratch and use the metadata export/import tools described in the previous sections to re-create the publications and subscriptions. However, that would force all your client devices to re-download everything from the new server.

This section describes the steps to take to maintain the server state during the move so that clients can continue to do incremental sync with the new server. The key point is that the Pervasync state is stored in a database, so make sure you move your database without losing anything.

1.11.1 Moving the Database

NOTE: Skip this if you want to move the Pervasync server without moving the database.

You need to make sure the DB is duplicated correctly before you run setup.bat/setup.sh to point the new Pervasync server to the new DB server. A test of that is to be able to log in to the database using pvsadmin user to access pvsadmin schema. To make that happen, follow these steps.

1. Dump (from old database) and restore pvsadmin schema and your app schemas on the new database.
2. Create Pervasync admin user on the new database. For MySQL, use

```
CREATE USER 'pvsadmin'@'localhost' IDENTIFIED BY 'welcome1'
```

For Oracle, use

```
CREATE USER pvsadmin IDENTIFIED BY welcome1 DEFAULT TABLESPACE
USERS quota unlimited TEMPORARY TABLESPACE TEMP
```

3. Grant privileges to Pervasync admin user. For MySQL, use

```
GRANT ALL on *.* TO 'pvsadmin'@'localhost' WITH GRANT OPTION
```

For Oracle, use

```
GRANT  connect, resource, select any table, delete any table,
insert any table, update any table, create any table, drop any
table,alter any table,lock any table, create any trigger,drop any
trigger,create any index, drop any index,SELECT_CATALOG_ROLE to
pvsadmin
```

4. Login as Pervasync admin user and check to make sure the Pervasync admin schema has the same table contents as the old DB.

1.11.2 Setting up the Pervasync Server on the New Host

Since the server state is kept in the database, you can install the Pervasync server software on the new host from scratch and then run setup.bat or setup.sh to setup the server to point to the new database.

NOTE: Make sure you didn't click on the "Reset" button in the Pervasync server setup tool. If you did, you would need to re-create the publications and subscription and clients would need to re-download everything.

2 Using Pervasync

Pervasync enables you to synchronize distributed local databases with a central database without writing a single line of code. You start with a central DB with existing schemas and schema objects. The local DB should be initially empty. You use the web based admin console to publish central database objects and subscribe clients to the publications. Then on the client machine you use a shell script (pvc.sh or pvc.bat) to invoke the sync agent to initiate a sync session. The very first sync session will create the subscribed sync schemas and schema objects in local DB. The second sync will copy the central DB schema data to client schemas. After that, synchronization will be incremental – only changes on client and server will be exchanged.

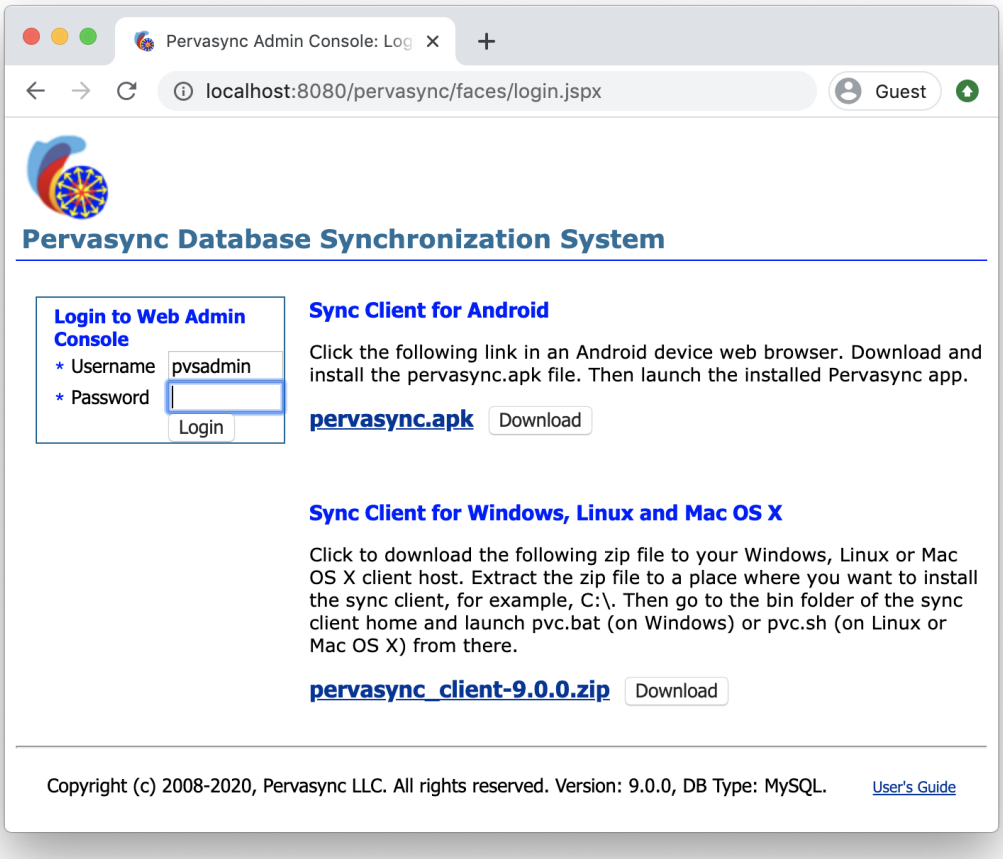
Alternatively you can use the server and client side Java API to do publication, subscription and synchronization from your Java application.

2.1 Creating Publications and Subscriptions Using the Web Admin Console

You use the web based sync server admin console to publish sync objects and create subscriptions. Besides publications and subscriptions, the admin console is also used to monitor and control the sync servlet and sync engine, manage devices, users and groups.

2.1.1 Logging in to the Admin Console

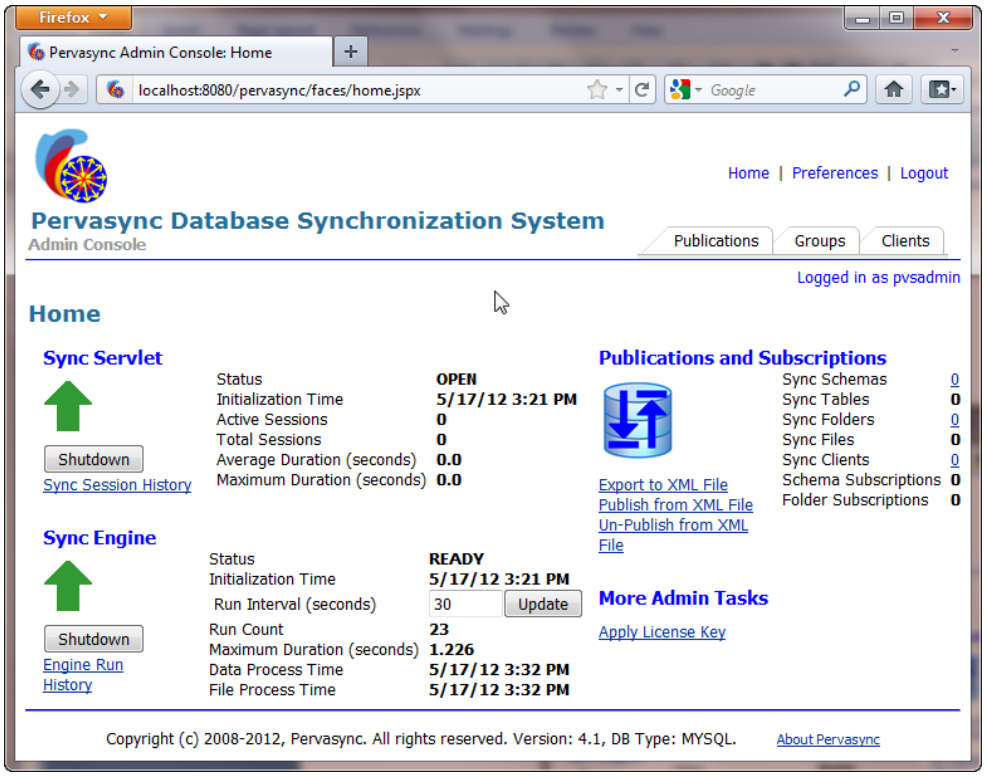
Once the Pervasync web application is deployed in the J2EE servlet container, the admin console will be available at http://<server>:<port>/pervasync, e.g. http://localhost:8080/pervasync. The following is the login page.



Use the admin username and password you specified when you setup the Pervasync server to login.

2.1.2 Admin Console Home

Once logged in, you will see the admin console home page.



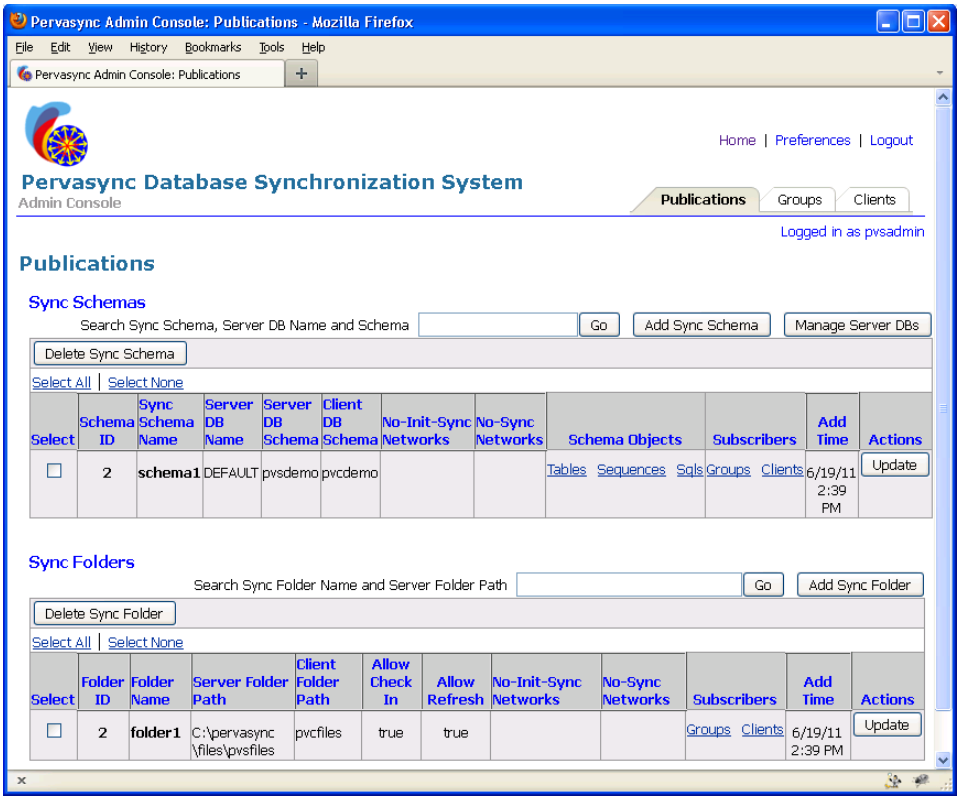
The admin console allows you to check the status of the sync servlet and the sync engine. You could also shut down or start up both of them.

The Sync Engine helps preparing server side logical transactions to be used to refresh client side databases and folders to new states. Sync Servlet serves sync clients receiving transactions (Check-Ins) from clients and sending transactions to clients. You need to have both Sync Servlet and Sync Engine up for synchronization to work. However, when you are creating publications and subscriptions, it is better to temporarily shut down both.

2.1.3 The Publish and Subscribe Model

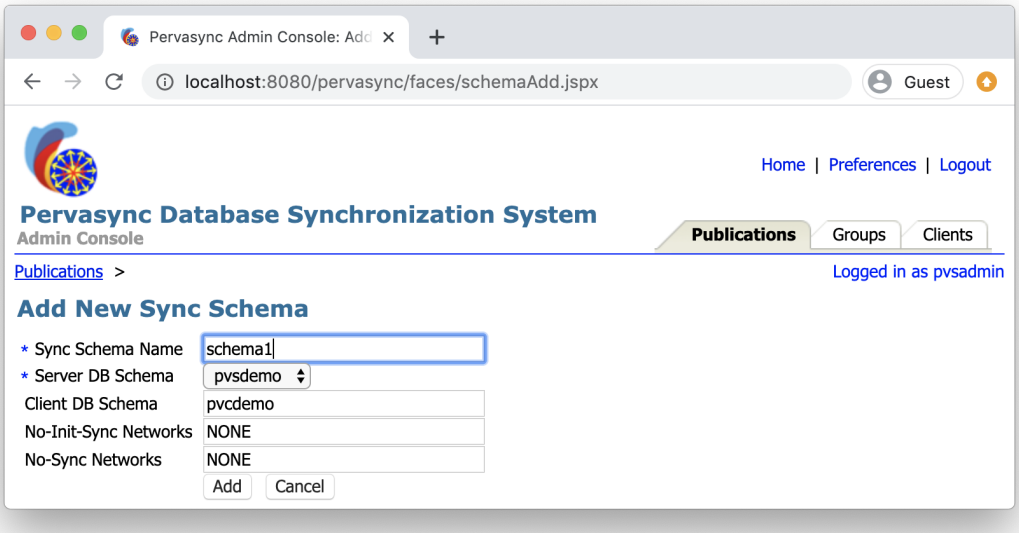
Pervasync employs a publish and subscribe model. You first define sync schemas and sync folders as publications. Then you make sync clients or groups subscribe to a set of publications. Publications may have parameters defined. At subscription time, you supply values to the parameters to customize the subscription. For example, you may define “region” as a parameter of your publications. Then at subscription time, you assign a value to region for subscribers so that they get data related to a specific region.

Click on the Publications tab to show all available publications. There are two types of publications: sync schemas and sync folders. A sync schema is a container for database tables and other objects while a sync folder is a container for files.



2.1.4 Publishing Sync Schemas

On the Publications page click on “Add Sync Schema” button under Sync Schemas header to add a schema.



Give “Sync Schema Name” a unique name. A sync schema is used to group a subset of tables belonging to a physical DB schema. You may use the physical schema name if you plan to create only one sync schema corresponding to the physical schema.

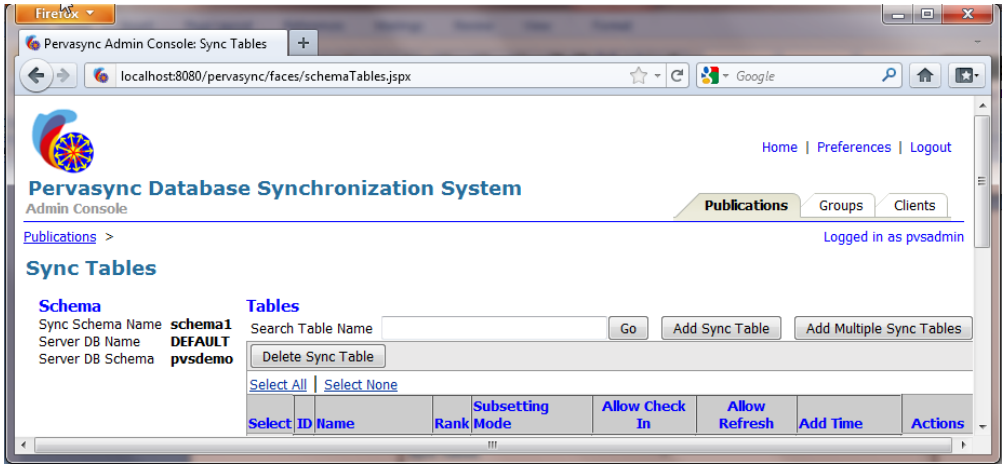
***Note**:* You may create multiple sync schemas for a single physical DB schema. However, one physical table can only belong to one sync schema.

“Client DB Schema” is the name of the physical schema to be created in the client side database. Leave it blank if you want to use the same server side physical schema name.

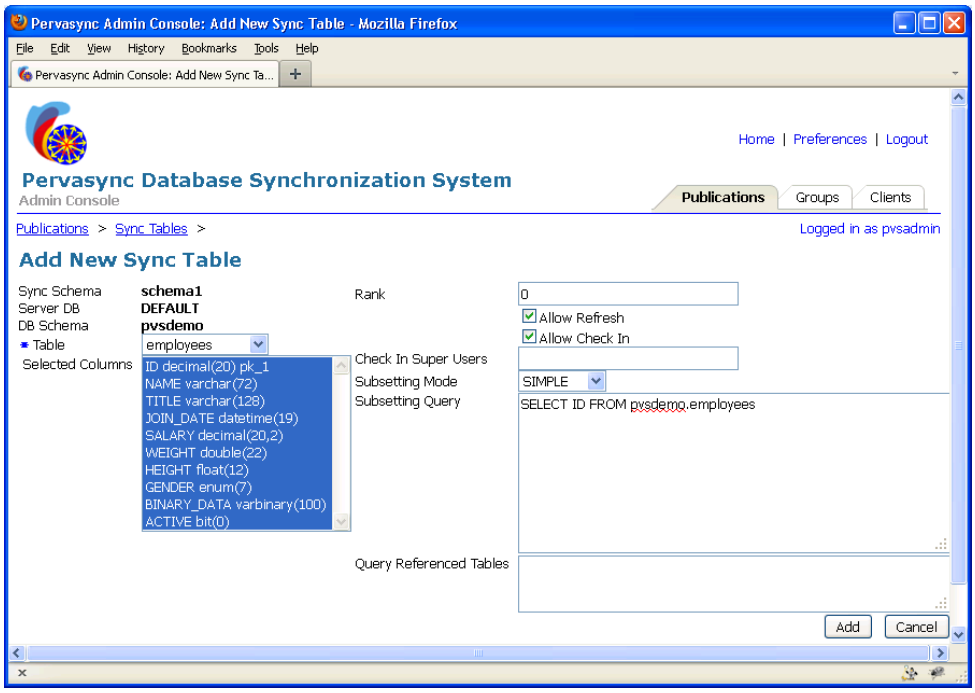
***Note**:* For desktop SQLite, a DB data file is created for each sync schema, with the name being “Client DB Schema” value and “.db” as extension. The data files are created in <Pervasync Client Home>/sqlite/ by default. If you don't like the default location, you can specify an absolute path as value of "Client DB Schema".

The No-Init-Sync Networks and No-Sync Networks fields are for the “sync based on network charactics” feature (see section 2.14 for details).

After the schema is created, you go back to the Publications page and click on the “Tables” link under “Schema Objects” to manage the tables of the schema.



Initially there would be no sync table. Click on “Add Sync Table” to select a table and specify columns and rows to sync. If you don’t have the need to select a subset of columns or rows to sync, you can use the “Add Multiple Sync Tables” to publish multiple sync tables in one click. Let’s first take a look at adding a single table.



In the above screen shot, we selected table “employees” and chose to sync all columns and rows to clients that subscribe to the schema “schema1”.

***Note**:* Table names and column names cannot contain spaces. For example, table name “xxxx table” is not supported. We suggest you replace spaces with underscores. For example, use “xxxx_table” instead of “xxxx table”.

Rank is used to indicate the referential relationship among the sync tables. Parent tables should have a smaller rank number than child tables. When you insert new records, you would insert first to a parent table and then to a child table as the child table records reference columns in the parent table.

If “Allow Refresh” is checked, Pervasync server will refresh changes to the table to Pervasync clients. If “Allow CheckIn” is checked, sync clients can check in client changes to this table on the server. “CheckIn SuperUsers” takes a comma-separated list of users who are allowed to check in even when “Allow CheckIn” is not checked.

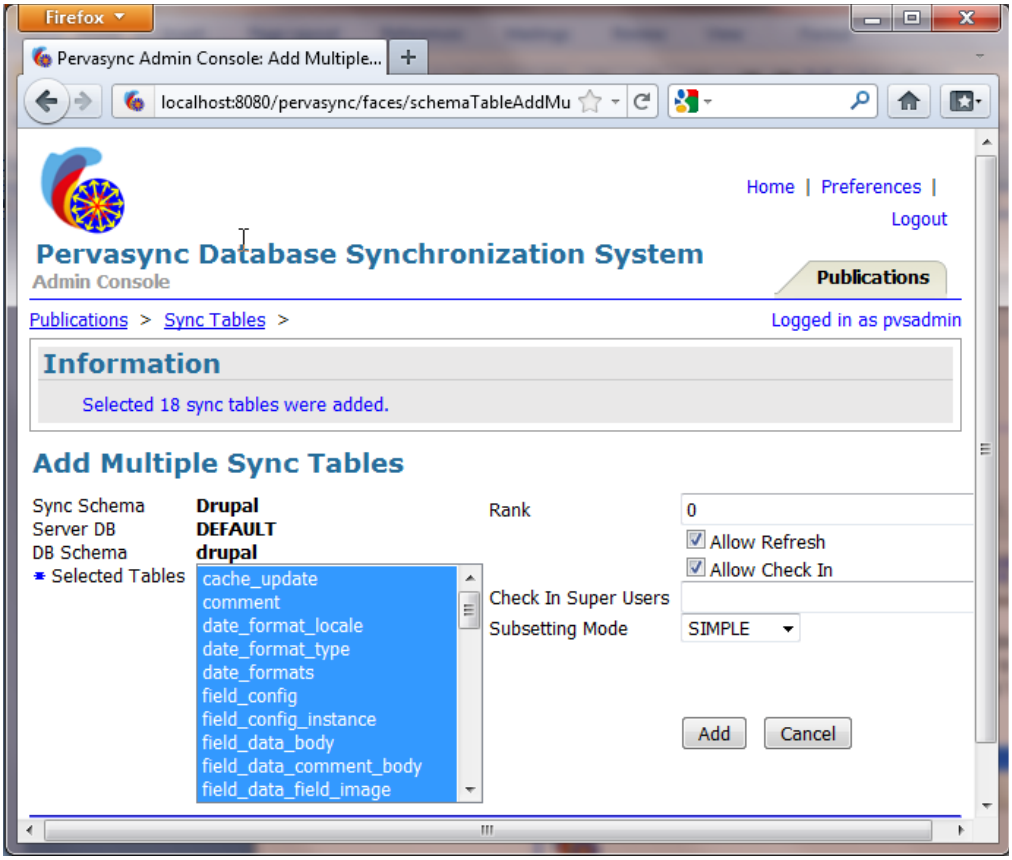
Subsetting mode and query are used to identify the rows of the table to be synced to clients. Use subsetting query to specify the primary key query as row filters. The query should return the set of the primary key values of the selected rows. The query can have parameters enclosed in “${}”. The parameter values are set at subscription time.

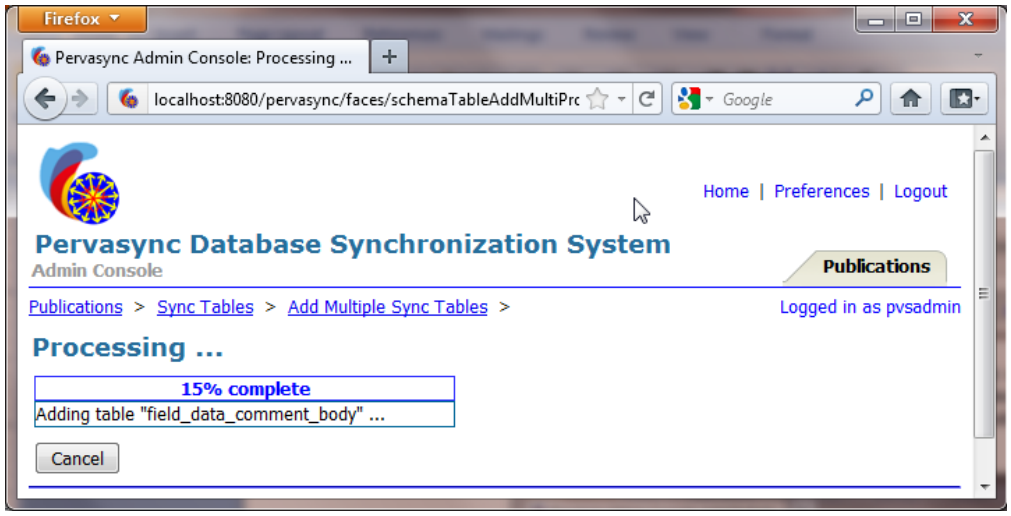
Subsetting mode SIMPLE is different from COMPLEX in that a SIMPLE query can select from only one table, which could be the sync table itself or a different table.

“Query Referenced Tables” takes a comma-separated list of tables that are referenced in the subsetting query. This is only for subsetting mode COMPLEX.

***Note**:* Subsetting is an advanced topic. If you are confused, you can leave all fields with default values. To further understand data subsetting, see section 2.6.1 A Step by Step Example of Data Subsetting and section 2.6.2 Subsetting Modes: SIMPLE and COMPLEX.

Following is the “Add Multiple Sync Tables” screen.



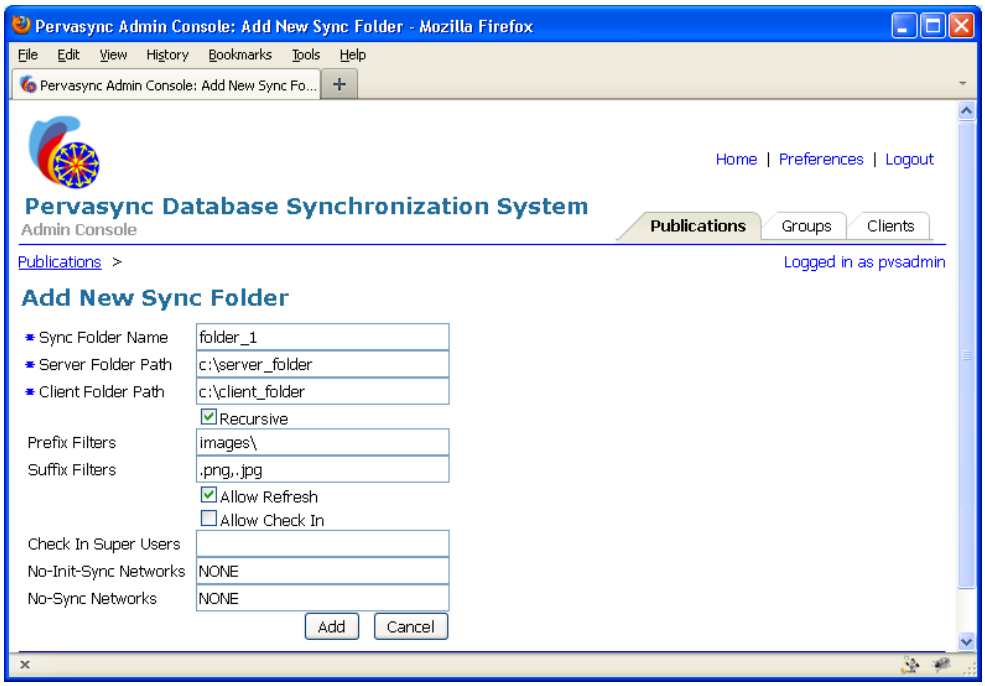


NOTE: You have to subscribe a client to the published sync folder for the folder to be synced to the client. See section 2.1.8 for details.

2.1.5 Publishing Sync Folders

If you need to sync files in folders on the server side to client hosts, you need to publish the folders as sync folders.

On the Publications page click on the “Add Sync Folder” button under “Sync Folders” header to add a folder for synchronization.



Move mouse cursor over the fields for tooltips, or see below for field descriptions.

Sync Folder Name: A unique name for the sync folder, e.g., “folder_1”.

Server Folder Path: The server side folder path, e.g., “c:\ server_folder”. If not starting with “/” or drive letter and “:”, it will be treated as a path relative to the Pervasync server home folder.

Client Folder Path: The sync client side folder path, e.g., “c:\ client_folder”. It can have parameters to bind to values at subscription time. Parameters are enclosed by “${}”. If not starting with “/” or drive letter and “:”, it will be treated as a path relative to the Pervasync client home folder. You can also have env variables in the client folder path. These env variables should be enclosed in a pair of % signs. They will be substituted with env variable values on the client side.

NOTE: Use % pairs instead of ${} for env variables regardless of whether your client machine is Windows or Linux.

Prefix Filters: Filters on file relative paths. All files in the server root folder have a relative path. For example, “c:\ server_folder\user_1\ images\file1.gif” has a relative path of

“user_1\images\file1.gif”. The prefix filters apply to the file relative path. For example, if the value of the prefix filter is set to “user_1\images\”, any file whose relative path starts with “user_1\images\” will be synced. Multiple filters can be supplied using commas as separators. A file would pass this “starts with” test if it passes any one of the prefix filters. However, a file has to pass both the “starts with” (Prefix Filter) and the “ends with” (Suffix Filter – see below) tests to be selected. It can have parameters to bind to values at subscription time. Parameters are enclosed by “${}”. If you do not need to use this, simply leave the field empty.

Suffix Filters: File filter by filename suffix or extension, which corresponds to what filename ends with. These are comma (“,”) separated file filters by file name suffix. For example: if “.gif,.jpeg,.png,.PNG,.doc” is set, any file whose name ends with “.gif” , “.jpeg”, “.png”, “.PNG”, or “.doc” will be synced. If you do not need to use this, simply leave the field empty.

Check In Super Users: Comma (“,”) separated list of users who are allowed to check in even when “Allow Check In” is set to false. If you do not need to use this, simply leave the field empty.

No-Init-Sync Networks and **No-Sync Networks**: these are for the “sync based on network charactics” feature (see section 2.14 for details).

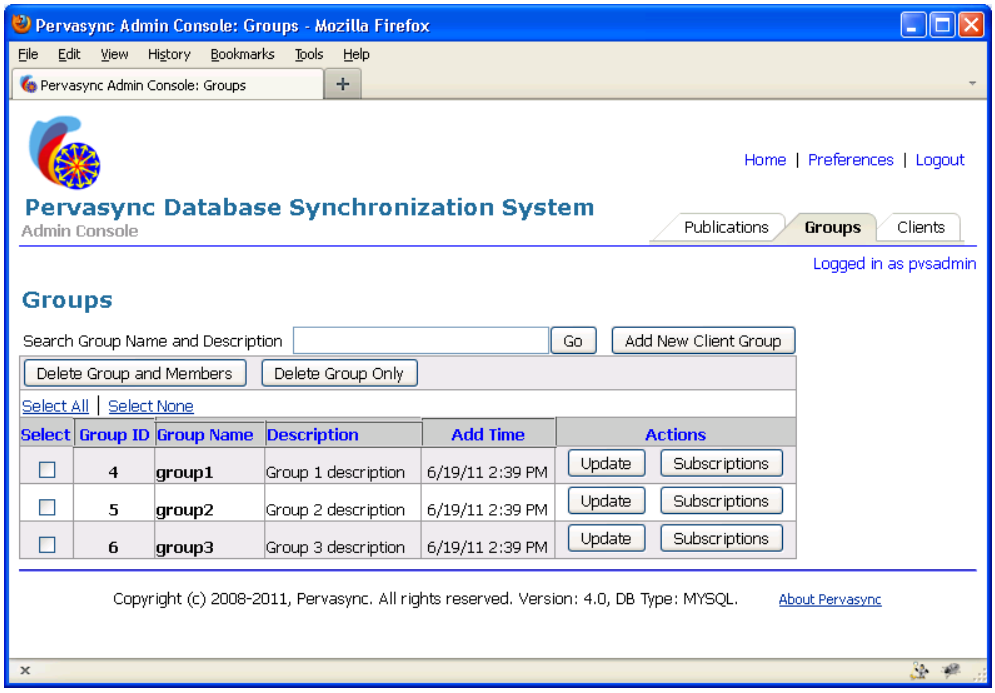
NOTE: You have to subscribe a client to the published sync folder for the folder to be synced to the client. See section 2.1.8 for details.

2.1.6 Managing Groups and Group Subscriptions

Very often you have a group of clients that share the same set of publications and even subscription parameters. In this case it is more convenient to create groups and group subscriptions first and then add/remove clients to/from the groups. Clients will inherit group subscriptions.

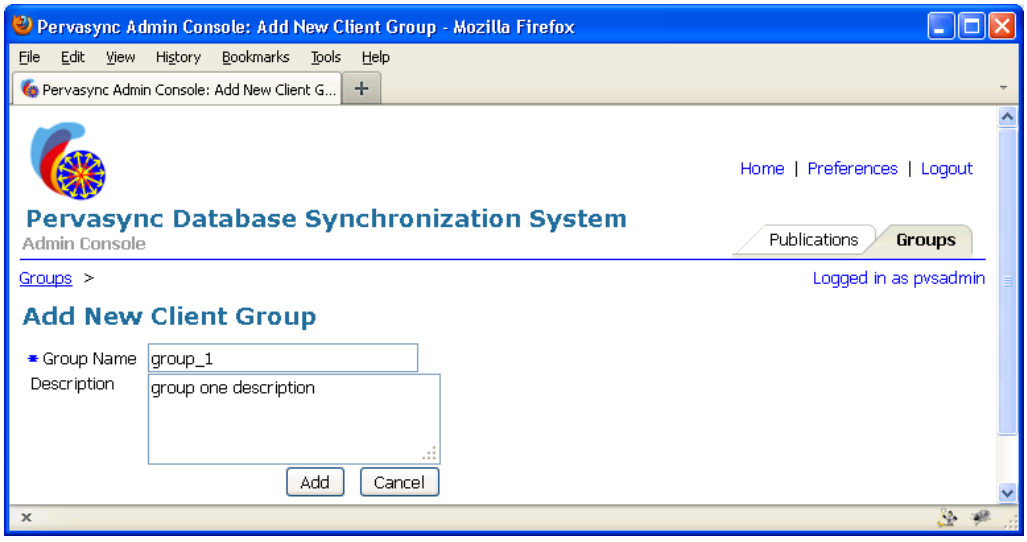
Groups are optional. If you prefer working directly with clients and subscriptions, skip this section and move to section 2.1.7 and 2.1.8.

Click on Groups tab to view existing groups.

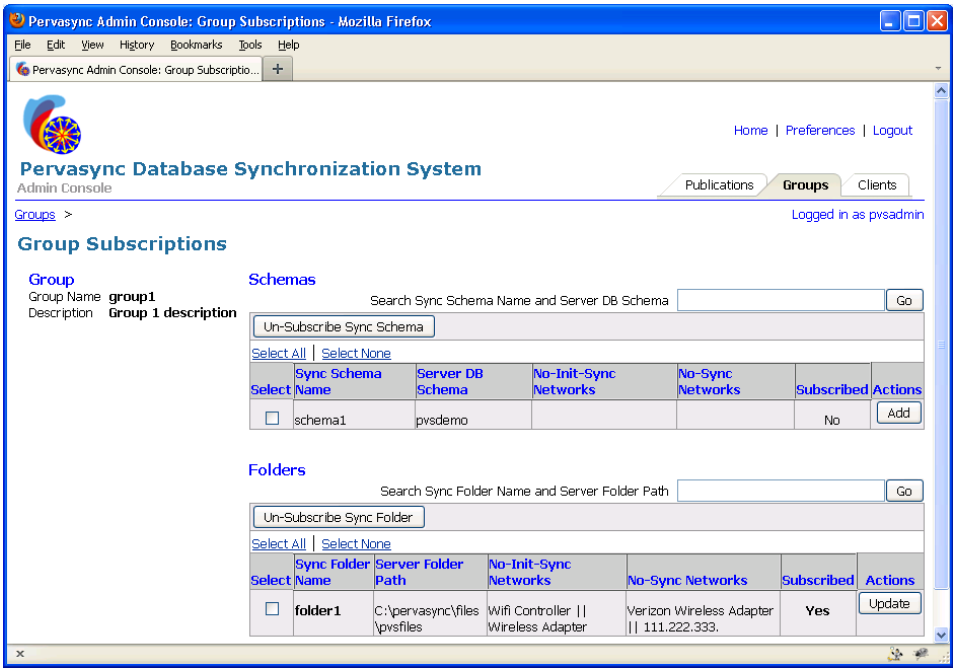


When you delete a group, you have an option to keep the group members (sync clients). If you choose “Delete Group Only”, the clients and their subscriptions will be kept intact while their “group” will be set to NULL .

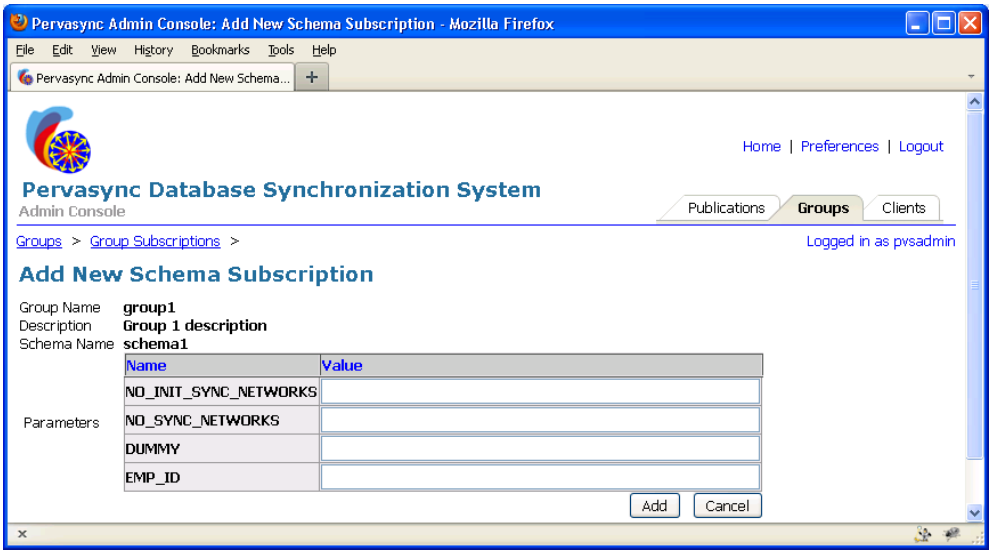
Initially you may not have any groups. Click on “Add Client Group” to add one.



Go back to the Groups page, find the group and click on button “Subscriptions”.



The Group Subscriptions page lists all the available publications, subscribed or not. Subscribed publications (including sync schemas and sync folders) have their name bold-typed and you can select and un-subscribe them. For publications that are currently not subscribed, you can click on the “Add” button to add them to the group’s subscriptions.



Before you click the Add button, fill in the values for the parameters to customize the subscription. These values will be shared by all clients that will later be added to this group.

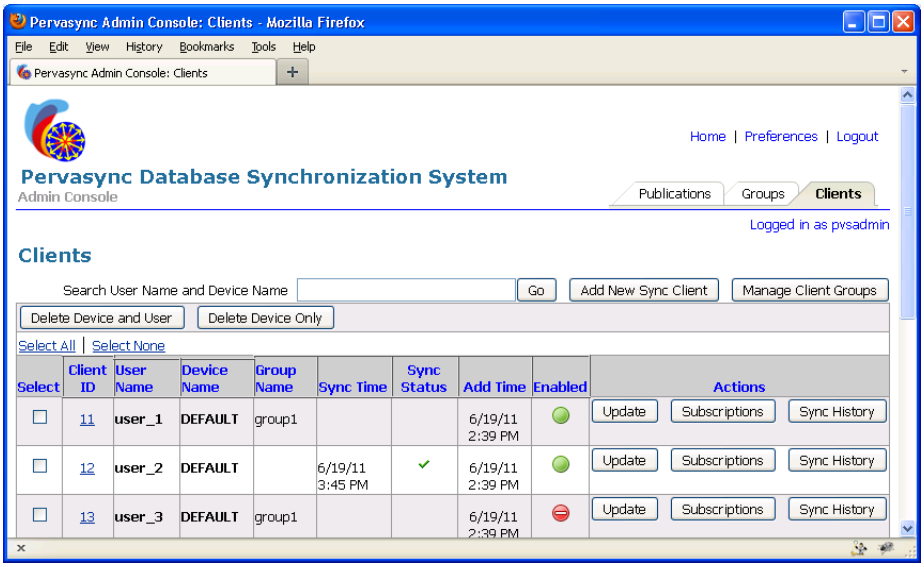
“NO_INIT_SYNC_NETYWORKS” and”NO_SYNC_NETYWORKS” are for the “sync based on network charactics” feature (see section 2.14 for details). Leave them blank if you are not using the feature.

Similarly, you can add sync folder publications to a group subscription.

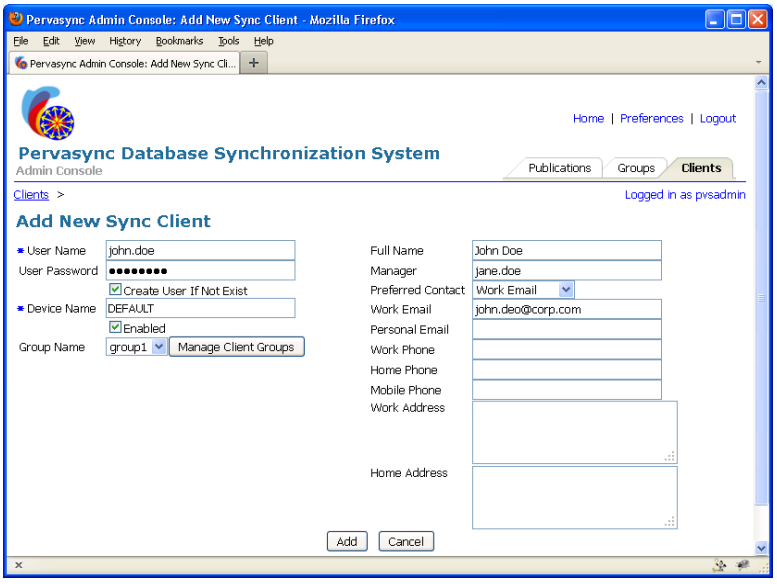
If you unsubscribe a group from a publication, all clients that belong to the group would lose their subscriptions. To add/remove clients to/from a group, see next section.

2.1.7 Managing Sync Clients

Click on the “Clients” tab.



Initially you may not have any sync clients. Click on “Add New Sync Client”.



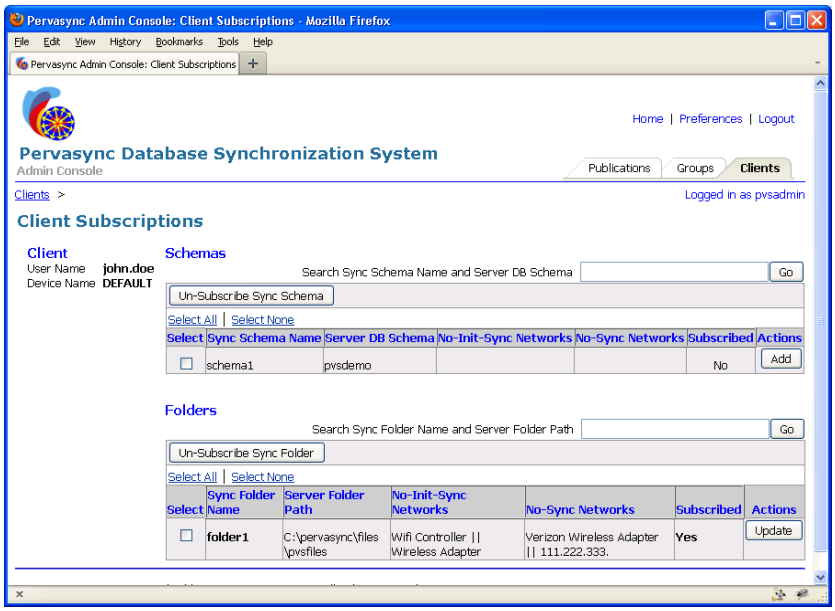
Here we create user “john.doe” and add device “DEFAULT”. We put in a password and check “Create User” to create the user in Pervasync repository. Alternatively, you could choose to manage users and their passwords yourself. See javadoc for interface

pervasync.server.UserManager for details. In that case you need and only need to specify User Name and Device name to add the sync client.

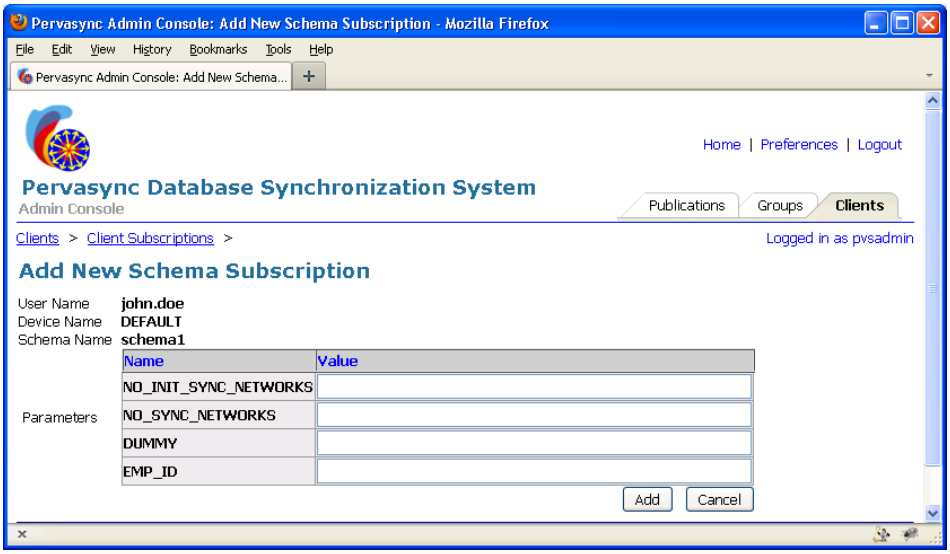
When you create a client you can assign it to a group.

2.1.8 Creating Subscriptions for Clients

Click on the “Subscriptions” button on the row of the sync client on the Clients page.



If you have assigned a group for the client, you would see the client has all the group subscriptions. Click on the Update (or Add if not subscribed) button on the row of the publication.

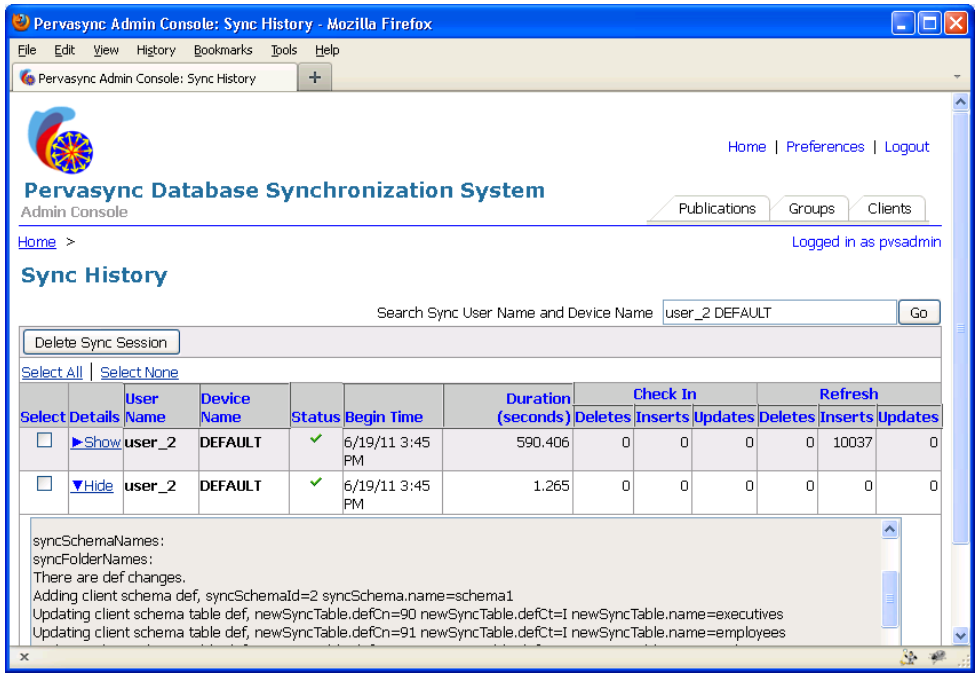


A subscription is an association of a publication with a client -- in our case, user john.doe's DEFAULT device and schema "schema1". Edit the parameter values if needed and click the "Update" to update the client subscription.

Once you have successfully created publications, user-devices and subscriptions using the web admin console, you are ready to execute synchronizations using a sync client as described in section 2.2.

2.1.9 Checking Client Sync History

On the admin console Home click on the link "Sync Session History" under the header "Sync Servlet". This page is useful for admin to identify client sync issues in the field and track user synchronization trends.



The "Clients" page also shows each client's last sync status and provides a link to sync history for a specific client.

2.2 Doing Synchronization Using Pervasync Client for Windows, Linux and Mac OS X

NOTE: See section 1.3 for client installation instructions.

NOTE: Make sure you have subscripted the sync user to sync schemas/folders. See section 2.1.8 for instructions.

Pervasync clients can be launched in either GUI mode (see section 2.2.1) or command-line mode (see section 2.2.2).

It will take two sync sessions to bring a newly created subscription to the client. In the first sync, the schema and table definitions will be synced to device, and the schemas and tables

will be created on device database. In the second sync, the table data will be synced to the device. After the two initial syncs, data exchange will be incremental.

NOTE: If a client database is not empty, initial sync will wipe out all the schemas and tables and erase all the data in the client database.

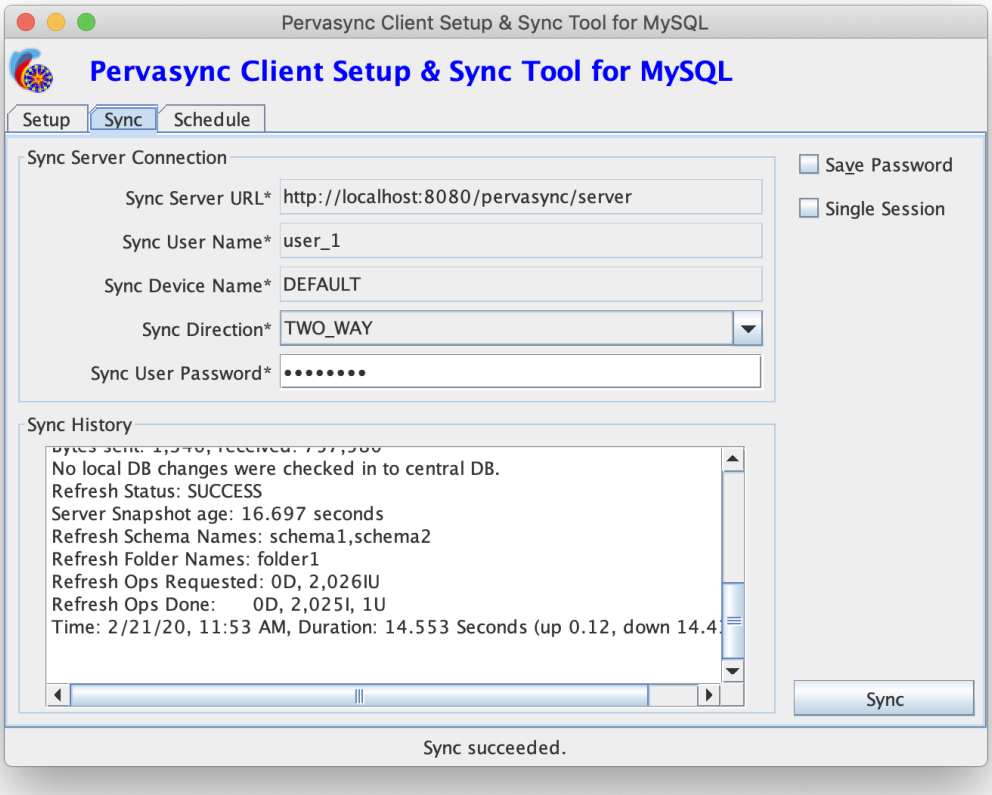
2.2.1 Run Pervasync Client in GUI Mode

Locate pvc.bat (for Windows) or pvc.sh (for Linux) in the "bin" folder of Pervasync client home. Invoke it without any arguments and the client will run in GUI mode. In GUI mode, users can start or schedule sync sessions by pointing and clicking.

NOTE: You close the GUI window to terminate the client. When you do that, you will be given an option to run the client in the system tray. There is also a command-line option "sys_tray" to "pvc.bat" that you can use to launch the client to run directly in the system tray. You may want to create a Windows task to do this upon system reboot. This is useful to run scheduled sync tasks in the background.

The Sync Tab of the Pervasync Client GUI

Click on the Sync tab and you will see the following.



There are some sync settings that you can change on this screen:

Sync Direction: By default, it's TWO_WAY sync. You can use "REFRESH_ONLY" and "CHECK_IN_ONLY" to do one-way sync.

Sync User Password: You can optionally supply the sync user password. The stored password will be used if it is absent. You can have the password saved using the "Setup" tab, or use the sync tab and click on the "Save Password" checkbox.

Single Session: When there are sync definition changes in a sync session, Pervasync will only download the definition changes and leave data changes to the next sync session. By default, Pervasync will do two sync sessions with one "Sync" button click when there are sync definition changes to make sure both sync definition and data changes are synced. Check this checkbox to perform only one sync session with one click.

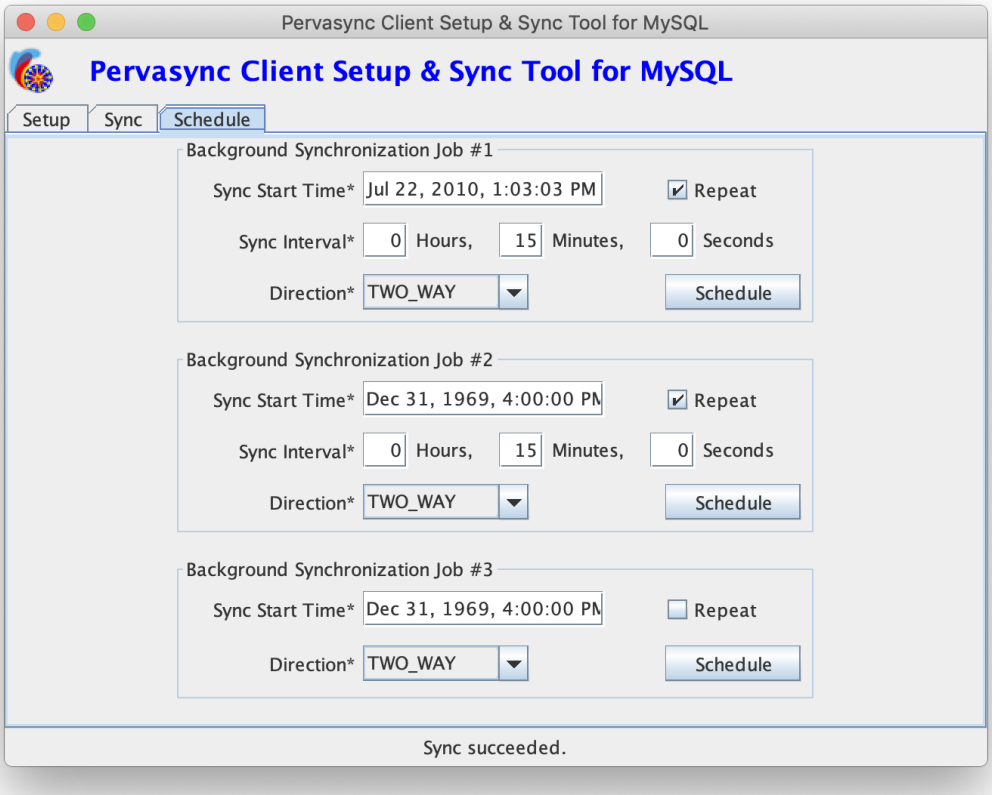
NOTE: If the sync client resides inside a firewall and the sync server resides outside the firewall, you need to set the HTTP proxy host and port using the setup tab with "Advanced Mode" turned on.

Click the "**Sync**" button to initialize a synchronization session.

Once the sync session is completed you will see an alert box telling you whether the session succeeded or failed. The "Sync History" panel on the "Sync" tab will also show you a summary of completed sessions.

If the sync fails, first check the error stack trace on the "Sync History" panel. For more detailed debugging info, check the log files under <Pervasync Client Home>/log. If you cannot resolve the issue yourself, send the log files to Pervasync support at support@pervasync.com.

The Schedule Tab of the Pervasync Client GUI



The Schedule tab enables you to schedule up to three sync jobs that do background synchronization at some predetermined time. In addition to the start time, you also can specify the sync direction and optionally, the sync interval.

Before you schedule background sync jobs, make sure you have the sync user password saved using the “Setup” or the “Sync” tabs.

Click the “Schedule” button to start a job and click on “Cancel” button to stop a job. A scheduled sync will be skipped if there is already a sync going on at the scheduled sync start time. Sync history is available on the sync tab.

NOTE: All sync jobs will be cancelled if you exit the sync client and will be resumed when you restart the sync client.

2.2.2 Run Pervasync Client with Command-line Interface

Users could also run Pervasync client in command-line mode instead of GUI mode. Locate pvc.bat (for Windows) or pvc.sh (for Linux) in the “bin” folder of Pervasync client home. If invoked with arguments, the client will run in command-line mode. Users can supply sub

commands for the client to perform synchronization on command-line. Invoke the script with "-h" option to see the usages.

```
C:\pervasync\bin>pvc -h

Usage
-----
pvc.bat [{sync|refresh|checkin|auto_sync|sys_tray} [<password>]]

    Use this command to launch the Pervasync Client Configuration and Sync
              Utility.
    If no arguments are specified, i.e.

    pvc.bat

    the sync client GUI will be launched.

    If "sys_tray" is specified as an argument, i.e.

    pvc.bat sys_tray

    the sync client GUI will be launched in system tray (minimized).

    Otherwise, it will run in non-GUI mode.
    Use the "sync" sub-command to do two-way sync, e.g.

    pvc.bat sync

    Use "refresh" and "checkin" to do one-way sync.

    You use "auto_sync" option to start the job scheduler for running sync
              jobs
    at pre-arranged times. Use the "Schedule" tab of the GUI to update job
              schedules.

    You can optionally supply the sync user password as a second argument.
    Stored passwords will be used if it is absent.

    Note that the JDBC jar file (for Oracle) or Java connector jar file
    (for MySQL) has to be available in folder ../lib to run this script.
```

2.3 Using Pervasync Standalone Client for Android

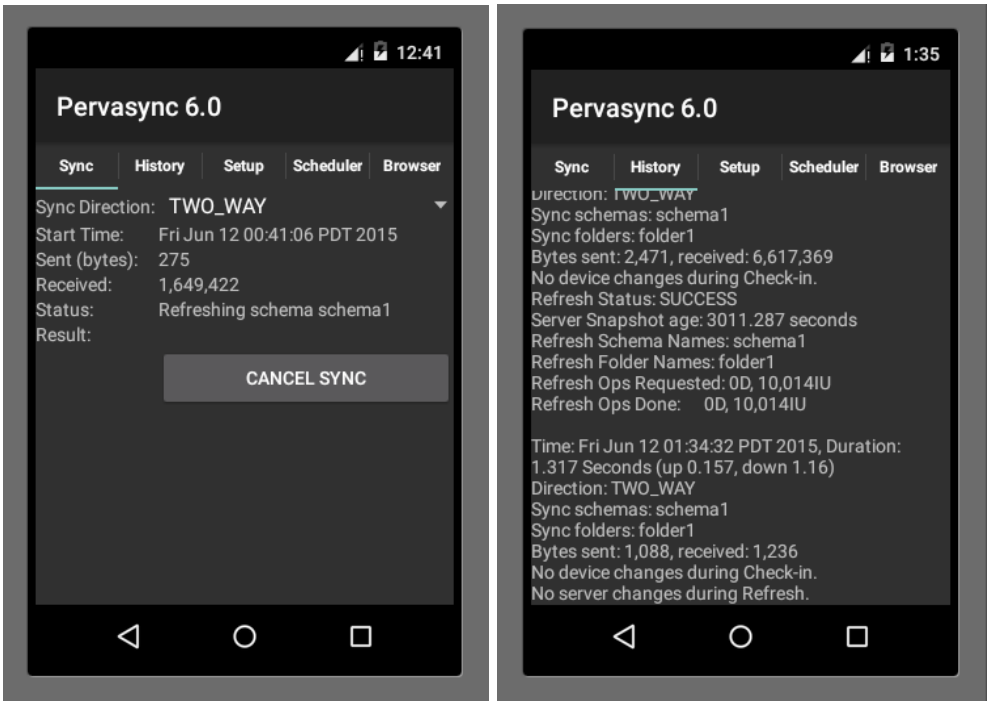
In this section we will first show the Pervasync client working in standalone mode and then describe how to embed the client into your Android app.

2.3.1 Starting Sync Sessions and Viewing Sync History

Once the sync client is installed and set up (see section 1.3 for details), you can click on the "Start Sync" button on the Sync screen of the Pervasync client.

During synchronization, the Sync screen will show you the progress. Once completed, the sync history is available on the History screen. You switch between screens by clicking on the tabs.

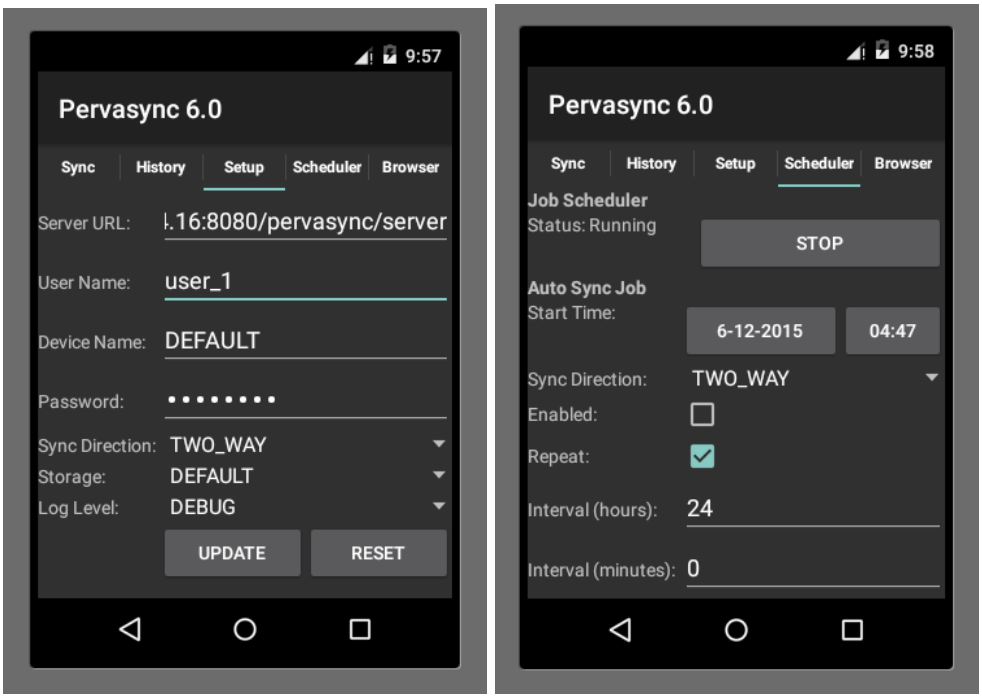
NOTE: If you see non-empty "Sent(bytes)" but empty "Received" and sync seems to get stuck, most likely you have a typo in sync URL or the sync server is not up. Use the "Setup" tab to correct the URL. Make sure you have the right IP address and port number.



2.3.2 Updating Setup Info and Configuring Auto Sync

The Setup screen allows you to update the setup info. It also allows you to reset the client.

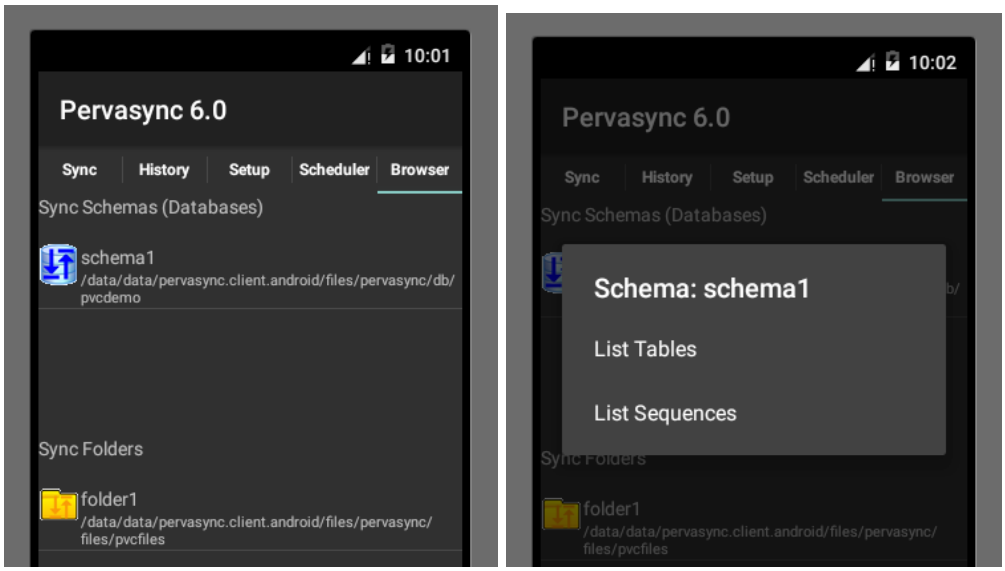
The Scheduler screen allows you to schedule sync jobs so that you don't have to manually initiate each sync session. Normally you would turn on the "Repeat" flag to make the sync happen periodically with a preset sync interval. Note that you need to have both the Job Scheduler running and the job enabled for the auto sync to be on.



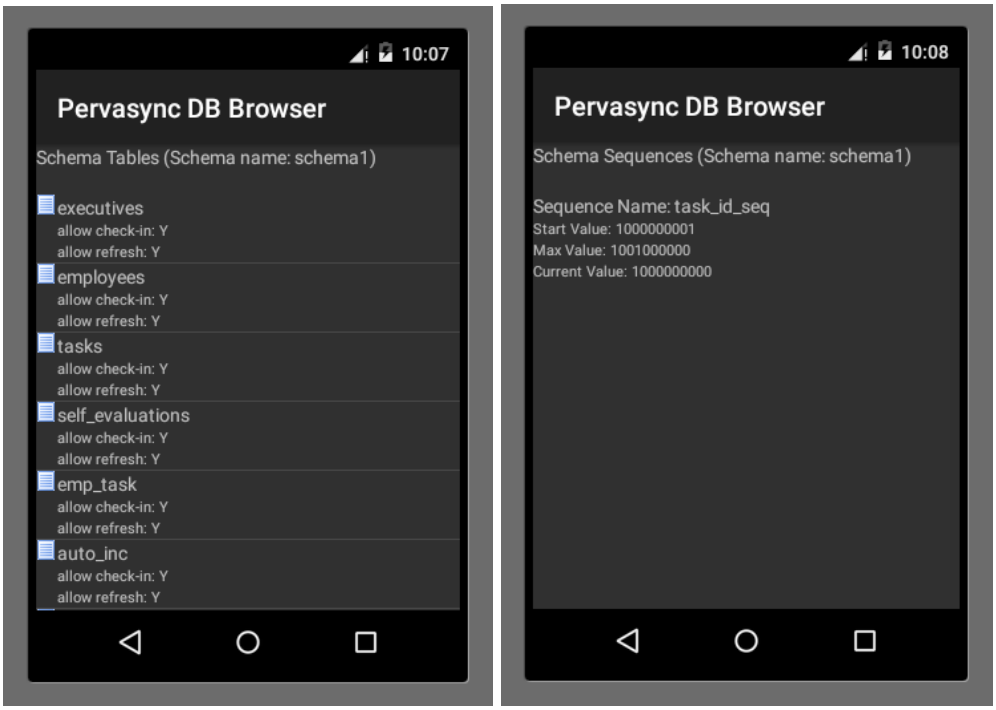
2.3.3 Database and File Browser

The Browser screen allows you to inspect the databases and files that are synced to the device. You may want to hide this functionality from end users after you compose your own user interface to the synced data. However, it's a great tool to use during development of your application that uses Pervasync for data synchronization.

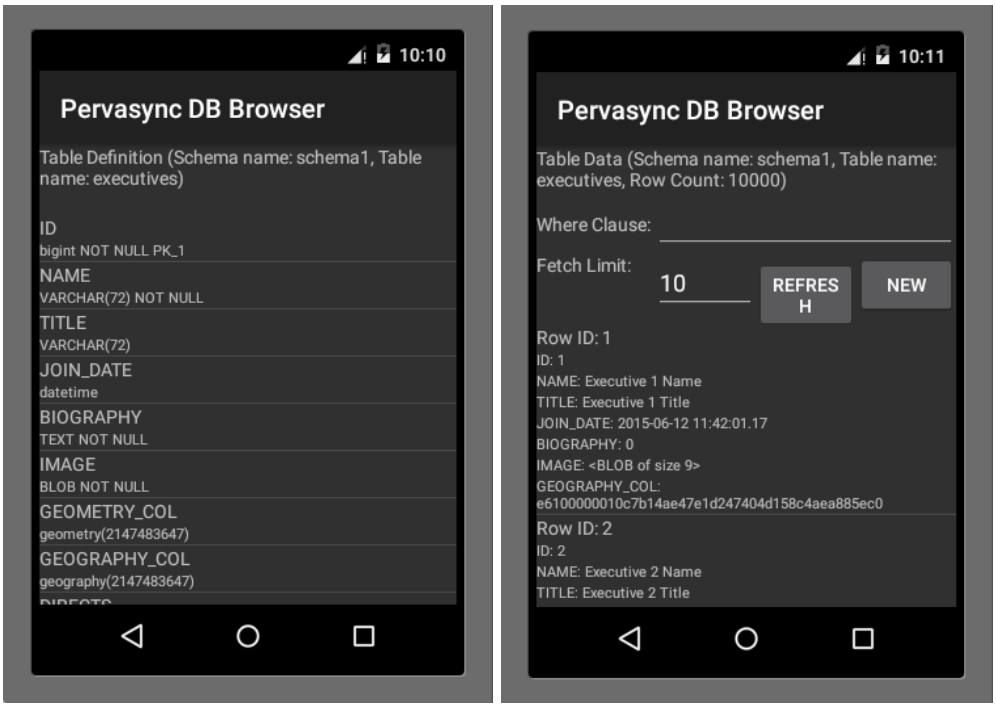
The Browser screen lists all the schemas/databases and folders. Clicking on a schema will bring up a dialogue with options to show tables or sequences that belong to the schema.



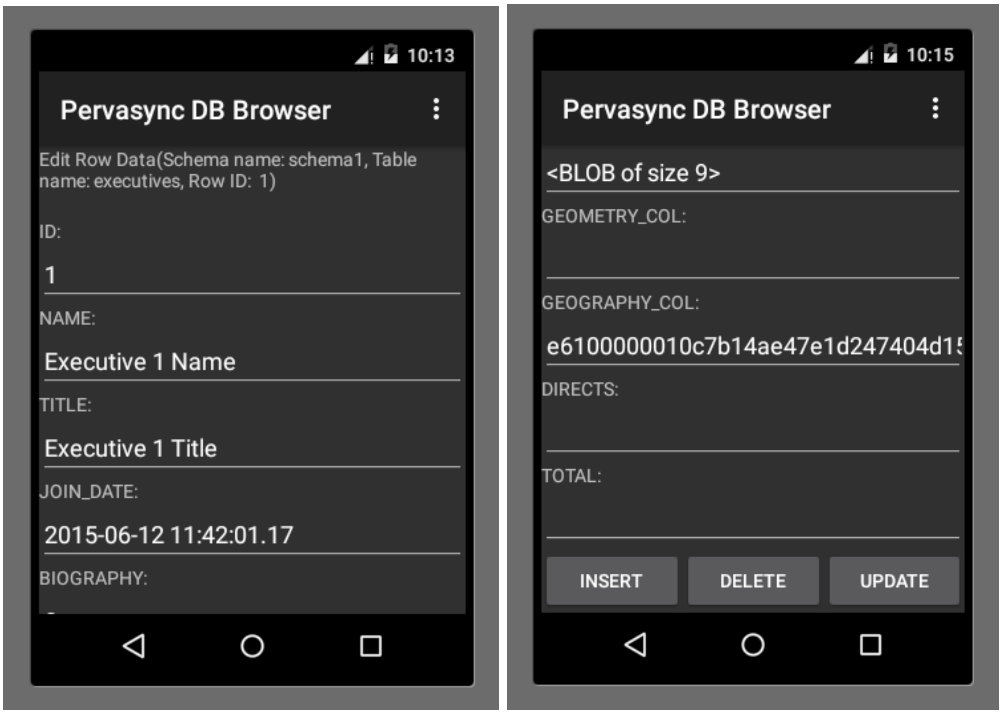
Following are the table list and sequence list.



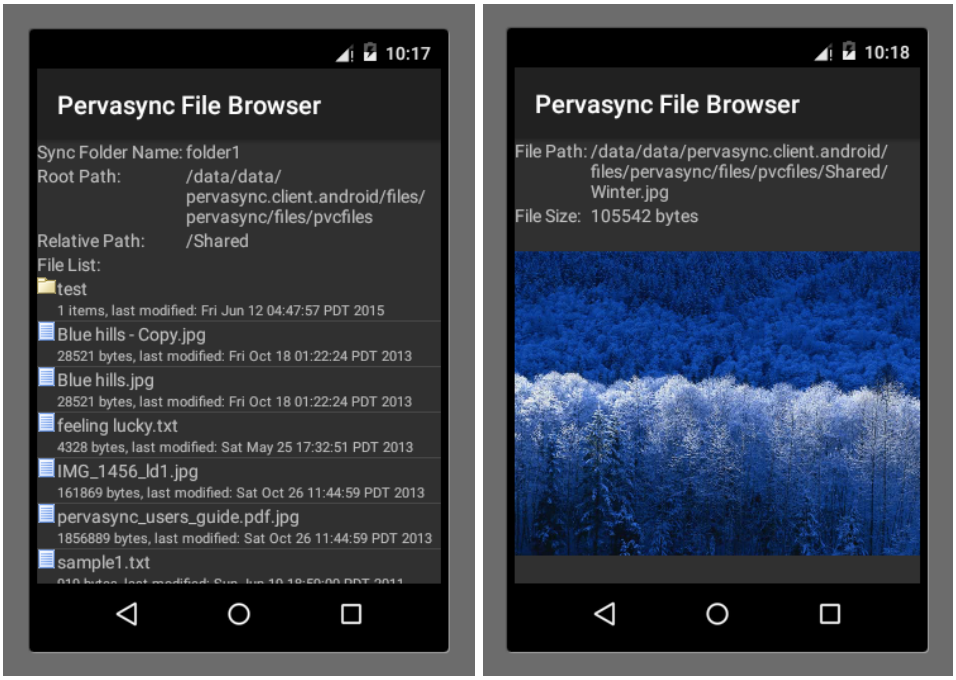
Click on the table to show table definition and table data. The table data screen will by default show 10 rows. You can change the Fetch Limit to show more or less than that. You can also put a predicate in the Where Clause box to show the desired rows, for example: `NAME like '%Jonh%'`



Click on a row to bring up the row editor screen where you can delete or update the row data. You could also insert a new row using this screen.



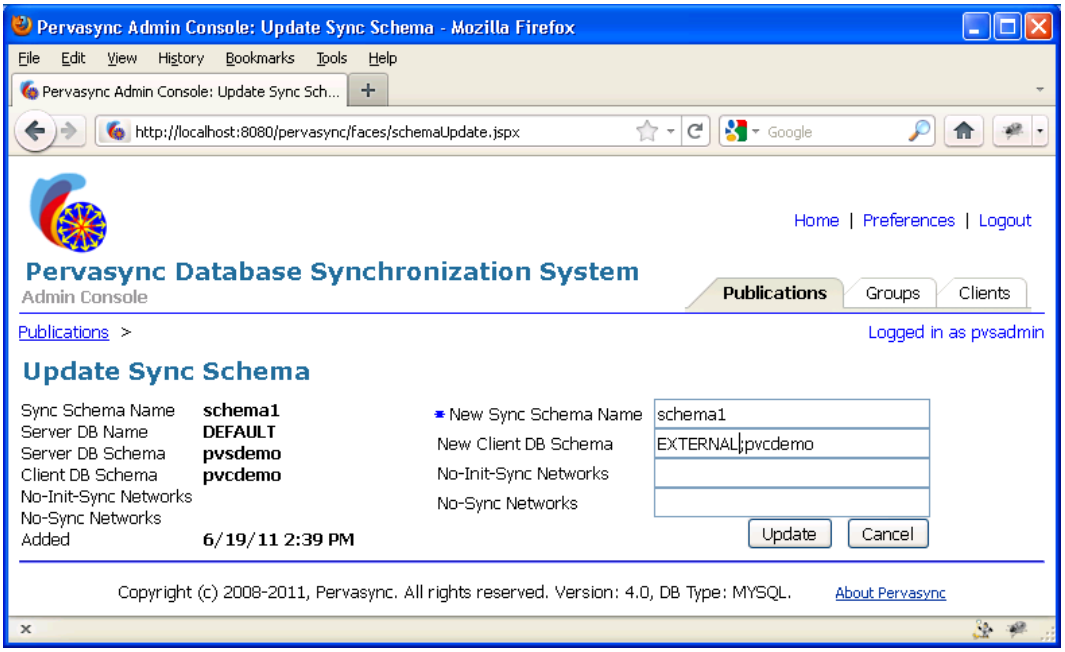
Back to the main Browser screen, you can click on a sync folder to show the sub folders and files.

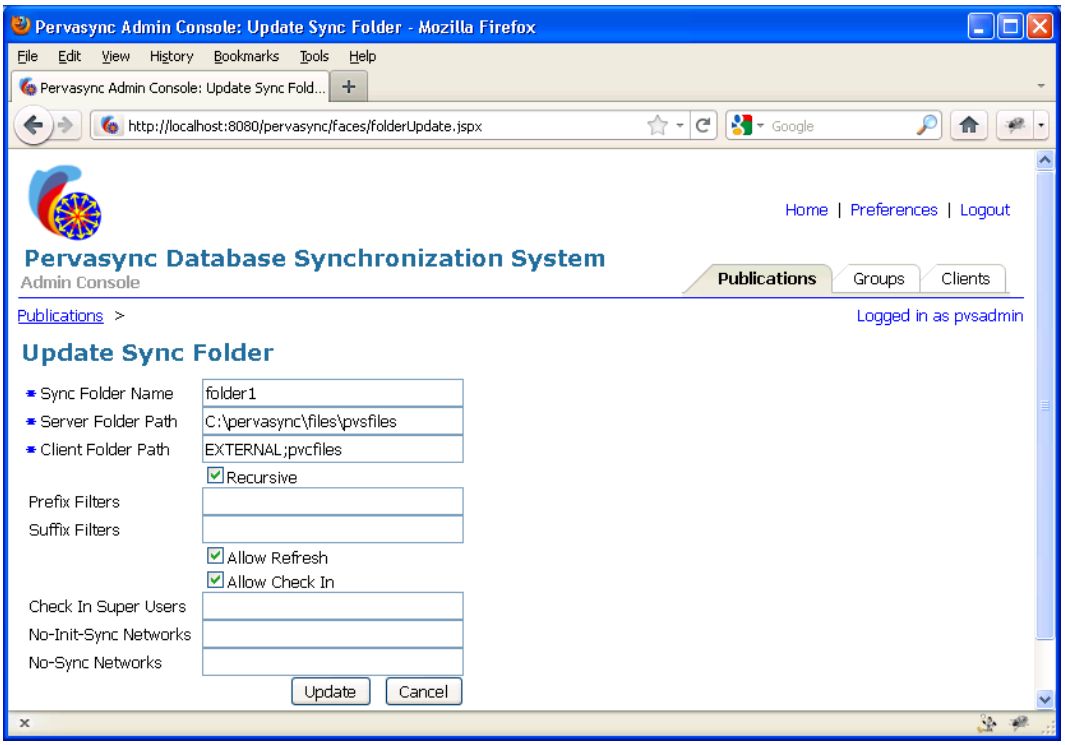


2.3.4 Syncing Data and Files to External Storage

If you run the sync client in standalone mode (as opposed to embedding it into your device app), your app usually won't be able to access the synced data unless the data is on external

storage. To sync data and files to external storage, prefix the client schema name and client folder name with “EXTERNAL;”. Note the trailing semicolon.





2.4 Embedding Pervasync Client into Android Native Applications

If you run the sync client in standalone mode, your app usually won’t be able to access the synced data unless the data is on external storage. Instead, if you embed the sync client into your own application, both the sync client code and your app code share the same

permissions to the data and you would be able to store the data on both internal and external storage.

The “android/lib” folder of your Pervasync server home has a Pervasync client library jar that you can easily include into your app following the instructions below. There is also a demo app located in

```
<Pervasync Server Home>/demo/android
```

that you can open using Android Studio.

2.4.1 Your Android Native App

Let’s assume you created an Android project “my_android_app” with package name “com.mycorp.myapp” and main Activity is “MyMainActivity”. The AndroidManifest.xml of your project would look like the following:

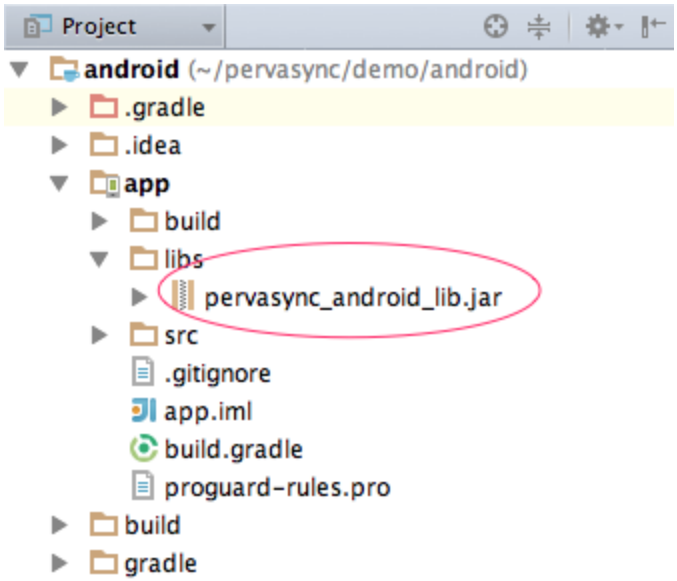
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.mycorp.myapp" >

   <application
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:theme="@style/AppTheme" >
       <activity
           android:name="com.mycorp.myapp.MyMainActivity"
           android:label="@string/app_name" >
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />

               <category android:name="android.intent.category.LAUNCHER" />
           </intent-filter>
       </activity>

   </application>

   <uses-sdk android:minSdkVersion="11" android:targetSdkVersion="22"/>


</manifest>
```

2.4.2 Adding Pervasync Android client library to Java Build Path

Android Studio

Note: If you use Android Studio to develop your app, read on. Otherwise if you use Eclipse, skip to the next section.

If it does not already exist, create “libs” directory under your Android projet’s “app” module. Then copy “pervasync_android_lib.jar” from the “android/lib” folder of your Pervasync server home to the “app/libs” folder.



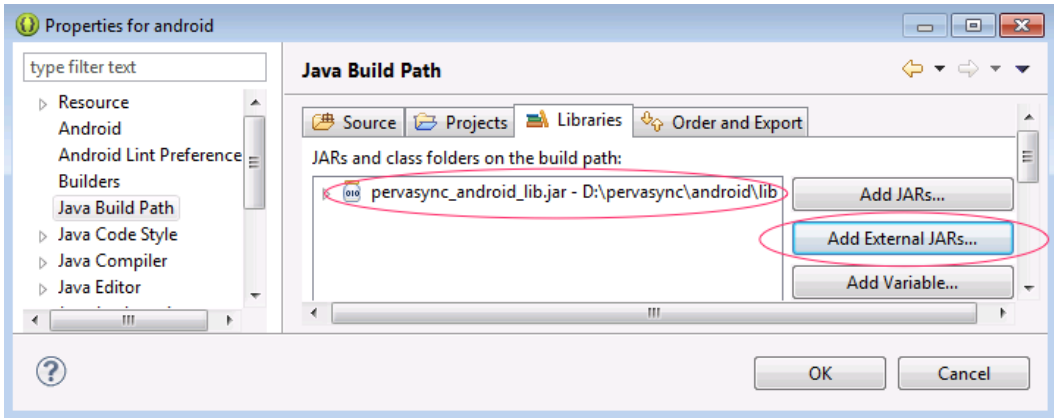
Right click on “pervasync_android_lib.jar” and hit “Add as library”. Ensure that “`compile files('libs/pervasync_android_lib.jar')`” and/or “`compile fileTree(dir: 'libs', include: ['*.jar'])`” is in your “build.gradle” file.

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.1.1'
    compile files('libs/pervasync_android_lib.jar')
}
```

Note: Use “libs” folder and “build.gradle” file under “app” module. Not those under the project.

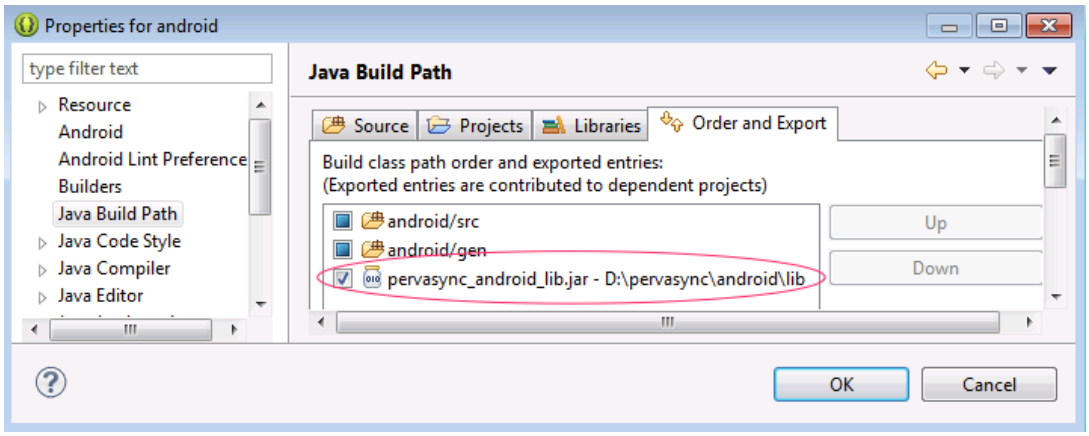
Eclipse

Right click on your Eclipse project for your Android app and open the “Properties” window. Click on “Java Build Path” on the left panel and then click the “Libraries” tab on the right panel. Click on “Add External JARs…” button to add Pervasync Android client library “pervasync_android_lib.jar” which is located in the “android/lib” folder of Pervasync server home.



NOTE: If your Pervasync server and your Eclipse IDE are on different hosts, just copy the jar to your Eclipse host.

After adding the jar, click the "Order and Export" tab. Select the check box in front of "pervasync_android_lib.jar".



2.4.3 Adding Pervasync Activities and Permissions to AndroidManifest.xml

Copy-paste the "activity" and "uses-permission" elements from the following sample AndroidManifest.xml (see also android/AndroidManifest_sample.xml) to your AndroidManifest.xml.

NOTE: Only copy the highted parts. Your AndroidManifest.xml should already have the rest. Also adding the "`android:largeHeap="true"`" option to the application tag will increase the heap memory limit.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.mycorp.myapp" >

   <application
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:theme="@style/AppTheme"
       android:largeHeap="true">
```

```
        <activity
            android:name="com.mycorp.myapp.MyMainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                 <category android:name="android.intent.category.LAUNCHER"
/>
            </intent-filter>
        </activity>

 <activity android:name="pervasync.ui.android.PervasyncClientActivity"
            android:label="Pervasync">
        </activity>

        <service android:name="pervasync.ui.android.PervasyncService">
            <intent-filter>
                                                                    <action
android:name="pervasync.ui.android.PervasyncService" />
            </intent-filter>
        </service>

                                                                    <activity
android:name="pervasync.ui.android.PervasyncSyncActivity"
            android:label="Pervasync Sync">
        </activity>

                                                                    <activity
android:name="pervasync.ui.android.AutoSyncSchedulerActivity"
            android:label="Pervasync Scheduler">
        </activity>

                                                                    <activity
android:name="pervasync.ui.android.PervasyncHistoryActivity"
            android:label="Pervasync Sync History">
        </activity>

                                                                    <activity
android:name="pervasync.ui.android.PervasyncSetupActivity"
            android:label="Pervasync Setup">
        </activity>

                                                                    <activity
android:name="pervasync.ui.android.PervasyncBrowserActivity"
            android:label="Pervasync Browser">
        </activity>

        <activity android:name="pervasync.ui.android.SchemaTablesActivity"
            android:label="Schema Tables">
        </activity>

        <activity android:name="pervasync.ui.android.FolderFilesActivity"
            android:label="File Browser">
        </activity>

                                                                    <activity
android:name="pervasync.ui.android.SchemaSequencesActivity"
            android:label="Schema Sequences">
        </activity>

        <activity android:name="pervasync.ui.android.TableRowsActivity"
            android:label="Table Data">
```

```
        </activity>
        <activity android:name="pervasync.ui.android.TableColumnsActivity"
            android:label="Table Column Types">
        </activity>

        <activity android:name="pervasync.ui.android.RowEditActivity"
            android:label="Row Edit">
        </activity>

        <activity android:name="pervasync.ui.android.ImageFileActivity"
            android:label="Image File">
        </activity>

        <activity android:name="pervasync.ui.android.TextFileActivity"
            android:label="TextFile">
        </activity>

                                        <receiver     android:enabled="true"
android:name="pervasync.ui.android.BootUpReceiver"

android:permission="android.permission.RECEIVE_BOOT_COMPLETED">

            <intent-filter>
                                                                    <action
android:name="android.intent.action.BOOT_COMPLETED" />
                    <category android:name="android.intent.category.DEFAULT"
/>
            </intent-filter>
        </receiver>

    </application>

    <uses-sdk android:minSdkVersion="11" android:targetSdkVersion="22"/>

                                                        <uses-permission
android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE"></uses-permission>
<uses-permission
android:name="android.permission.ACCESS_WIFI_STATE"></uses-permission>
<uses-permission
android:name="android.permission.INTERNET"></uses-permission>
<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"></uses-permissio
n>

</manifest>
```

2.4.4 Invoking Pervasync Client from Your App Code

With the inclusion of the library and the additions to the manifest file, Pervasync is now part of your application. You have two options in invoking Pervasync. One is to provide your own UI and call the SyncClient API Java API to do setup and manage synchronization. The other is to show the build-in Pervasync client UI, i.e. load the PervasyncClientActivity via an Intent.

Invoke SyncClient API

Use this option if you prefer to use your own UI.

Pervasync supports multiple user accounts on the same device. Just assign a unique ID to each account and call SyncClient.setAccountId to switch between the accounts. You can omit this step if you only have one account.

```
SyncClient.setAccountId(accountId);
```

Call SyncClient.setup to setup Pervasync client with account info. For one account, you only need to set it up once.

```
SyncClient.setup(userName, deviceName, serverUrl, password,
progressCallback);
```

Call SyncClient.synchronize to start a sync session:

```
SyncClient.synchronize(syncDirectionStr, syncProgressCallback);
```

You can find Javadoc of the Pervasync client interface in <Pervasync Server Home>/android/javadoc_for_android. Open the index.html file in a browser.

Show PervasyncClientActivity UI

Use this option if you prefer to use the Pervasync setup and sync UI.

In the main activity of your app, you should start the Pervasync service so that it can perform auto-sync in the background:

```
if (!PervasyncService.isStarted()) {
   startService(new Intent(this, PervasyncService.class));
}
```

To show the Pervasync UI, start the Pervasync client activity:

```
Intent intent = new Intent(myMainActivity,
       PervasyncClientActivity.class);
startActivity(intent);
```

Following is a sample main activity:

```
package com.mycorp.myapp;

import pervasync.ui.android.PervasyncService;
import pervasync.ui.android.PervasyncClientActivity;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.Gravity;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.LinearLayout;
import android.widget.TextView;
import android.widget.LinearLayout.LayoutParams;

/*
```

```
* This is the main screen of your app
*/
public class MyMainActivity extends Activity {
   private static final int SYNC_CLIENT_ID = 0;
   private final MyMainActivity myMainActivity = this;

   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);

       // start Pervasync service if not already started.
       if (!PervasyncService.isStarted()) {
           startService(new Intent(this, PervasyncService.class));
       }

       // set title
       setTitle("Pervasync Sample App");

       // contentView is a LinearLayout
       LinearLayout contentView = new LinearLayout(this);
       contentView.setOrientation(LinearLayout.VERTICAL);
       this.setContentView(contentView);

       // empty line
       contentView.addView(new TextView(myMainActivity));

       // description
        String description = "This is a place holder main screen for your
application. "+ "You can replace the title, add UI components to the
screen and add " + "child screens that can be linked from this main
screen.\n\n" + "We added a button below that invokes the Pervasync Client
main screen where users can setup " + "Pervasync and perform
synchronization.\n\n"+ "The Pervasync Client also includes a database
browser for you to explore the "+ "synced databases during development
time. You could hide the browser in the production " + "version of your
app.";

       TextView textView = new TextView(this);
       textView.setText(description);
       contentView.addView(textView);

       // empty line
       contentView.addView(new TextView(myMainActivity));

       // buttonBar
       LinearLayout buttonBar = new LinearLayout(myMainActivity);
       buttonBar.setGravity(Gravity.CENTER); // How button bar aligns its
       // children
       buttonBar.setOrientation(LinearLayout.HORIZONTAL);
       contentView.addView(buttonBar, new LinearLayout.LayoutParams(
               LayoutParams.FILL_PARENT, LayoutParams.WRAP_CONTENT));

       // Launch Pervasync Client button
       Button pervasyncButton = new Button(this);
       LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(
               LayoutParams.WRAP_CONTENT, LayoutParams.WRAP_CONTENT);
       pervasyncButton.setLayoutParams(params);

       pervasyncButton.setText("Launch Pervasync Client");
       pervasyncButton.setTag("Pervasync");
```

```
        pervasyncButton.setOnClickListener(new OnClickListener() {
            public void onClick(View view) {
                Button button = (Button) view;
                                if  ("Pervasync".equalsIgnoreCase((String)
button.getTag())) {
                    Intent intent = new Intent(myMainActivity,
                            PervasyncClientActivity.class);
                    startActivity(intent);
                }
            }
        });
        buttonBar.addView(pervasyncButton);
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        menu.add(Menu.NONE, SYNC_CLIENT_ID, Menu.CATEGORY_CONTAINER,
                "Launch Pervasync Client");
        return (super.onCreateOptionsMenu(menu));
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case SYNC_CLIENT_ID:
                startActivity(new Intent(myMainActivity,
                        PervasyncClientActivity.class));
                return (true);
            default:
                ;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

You can now build and run your Android app.

The Pervasync client UI has five tabs. You can hide some of the tabs by setting the “enable” properties of PervasyncClientActivity to false before you invoke PervasyncClientActivity.

```
public class PervasyncClientActivity extends TabActivity {

  public static boolean enableSync = true;
  public static boolean enableHistory = true;
  public static boolean enableSetup = true;
  public static boolean enableScheduler = true;
  public static boolean enableBrowser = true;
```

For example, to hide the “Browser” tab, doe this:

```
PervasyncClientActivity.enableBrowser = false;
Intent intent = new Intent(myMainActivity,
                           PervasyncClientActivity.class);
   startActivity(intent);
```

2.4.5 Advanced Topics

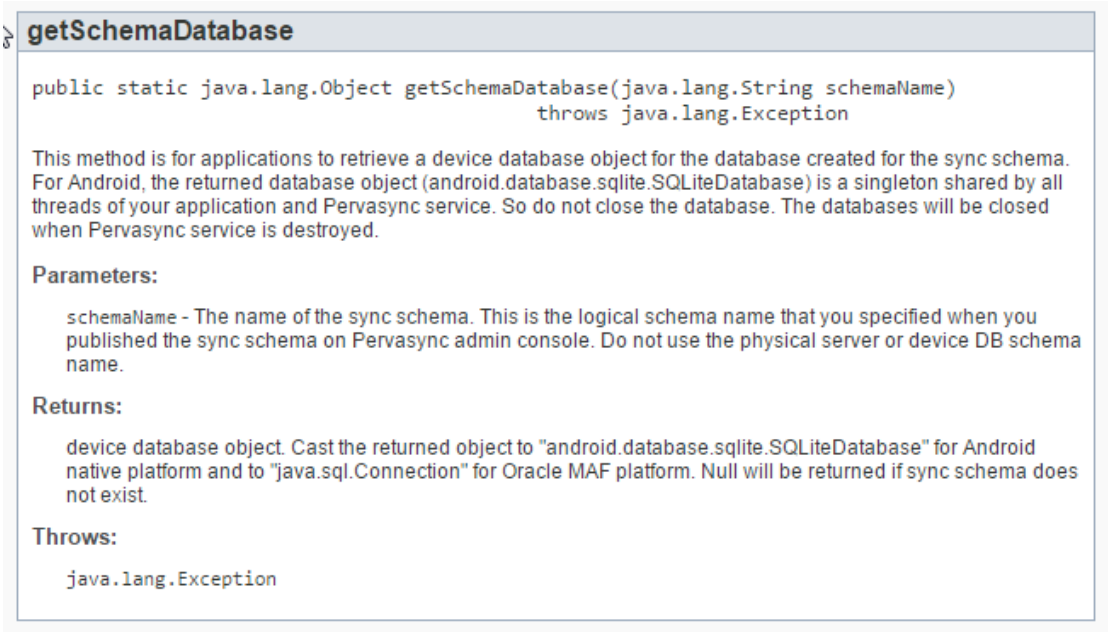
SQLite Locking and Concurrency on Android

SQLite stores one schema (AKA database) in one OS file. When a thread or process needs to commit changes to the database, the database file is locked until the commit is completed. If two or more threads try to commit at the same time, only one would succeed and others would fail with an android.database.sqlite.SQLiteDatabaseLockedException.

This file level locking for the entire schema is in contrast to the table row level locking on desktop and server databases. The latter apparently provides better concurrency. Another thing to note is that Android SQLite would throw an exception instead of waiting for the resource to be available, which is the server DB behavior.

The Android developer community has made it a best practice to maintain a singleton SQLiteDatabase object (essentially a DB connection) for each DB schema (corresponding to a file). This, combined with setting setLockingEnabled to true (the default), would serialize access to the schema database to avoid the database lock exception and other related issues.

Pervasync follows this best practice. When you embed Pervasync in your application, you should use the static method getSchemaDatabase of class pervasync.client.SyncClient to get a handle to the singleton SQLiteDatabase object.



getSchemaDatabase

```
public static java.lang.Object getSchemaDatabase(java.lang.String schemaName)
                                          throws java.lang.Exception
```

This method is for applications to retrieve a device database object for the database created for the sync schema. For Android, the returned database object (android.database.sqlite.SQLiteDatabase) is a singleton shared by all threads of your application and Pervasync service. So do not close the database. The databases will be closed when Pervasync service is destroyed.

Parameters:

`schemaName` - The name of the sync schema. This is the logical schema name that you specified when you published the sync schema on Pervasync admin console. Do not use the physical server or device DB schema name.

Returns:

device database object. Cast the returned object to "android.database.sqlite.SQLiteDatabase" for Android native platform and to "java.sql.Connection" for Oracle MAF platform. Null will be returned if sync schema does not exist.

Throws:

`java.lang.Exception`

Encryption

To encrypt the database, prefix the client schema name with “ENCRYPT;” when you publish the sync schema. If you want it be both encrypted and stored on external storage, use the prefix “EXTERNAL; ENCRYPT;”.
Note that other apps on the device can still access the databases when it’s encrypted. To further secure your DB so that only your app can access it, use prefix “PROTECT;”.

OTA Deployment

Once you package and sign your app with Android signature tool, you will have an apk file generated. Copy the apk file, e.g. my_android_app.apk, to <Pervasync Server Home>/web/pervasync/download and your users can install the app over-the-air (OTA) by entering the following URL in their mobile browser:

http://<Pervasync_Host>:<port>/pervasync/download/my_android_app.apk

2.5 Sync Realmjs DB for React Native Applications

Get Pervasync client for React Native on github: react-native-sync and react-native-sync-demo

2.6 Data Subsetting Using a SQL Query with Parameters

2.6.1 A Step by Step Example of Data Subsetting

Let's use the following table EMP of schema SCOTT as an example. Suppose you have your entire employee data stored in the EMP table of your central DB. The EMP table has the following definition. Each employee has an employ ID (column EMPNO) and belongs to a department (column DEPTNO):

```
CREATE TABLE "SCOTT"."EMP"
 (           "EMPNO" NUMBER(4,0),
             "ENAME" VARCHAR2(10),
             "JOB" VARCHAR2(9),
             "MGR" NUMBER(4,0),
             "HIREDATE" DATE,
             "SAL" NUMBER(7,2),
             "COMM" NUMBER(7,2),
             "DEPTNO" NUMBER(2,0),
              CONSTRAINT "PK_EMP" PRIMARY KEY ("EMPNO") ENABLE,
              CONSTRAINT "FK_DEPTNO" FOREIGN KEY ("DEPTNO")
               REFERENCES "SCOTT"."DEPT" ("DEPTNO") ENABLE
 );
```

Your company has grown very big and become geographically distributed. Employees started to have problems accessing the EMP table because the central DB is under heavy load and the network is not always fast and stable. So you decide to create a local DB for each department. The EMP table in a local DB only contains data for employees belonging to that department. Here we are subsetting by DEPTNO. In real world applications, people may do subsetting by country, region, branch office name/ID, and clinic name/ID etc.

Following are the steps to take to use Pervasync to synchronize the EMP table in a local DB with the EMP table in the central DB.

1. **Prepare the central DB schema.**

You may need to make some adjustments to the structure of your existing table to make it ready to be published as a sync table.

Pervasync requires that all sync tables have a primary key. The EMP table already has a primary key: column EMPNO. So, we are fine.

Assume we want to do the subsetting by DEPTNO. Then, we should have a table that defines the association between DEPTNO and EMPNO. One way to do this is to add a DEPTNO column to the EMP table. EMP table already has a DEPTNO column, so we don't need to do anything. Another way is to create a separate table that has two columns EMPNO and DEPTNO with EMPNO being the primary key column. You know what I mean if you learned about table normalization.

Now we have a decision to make: do we want to keep EMPNO as the sole primary key column or do we want to make DEPTNO and EMPNO together a composite primary key? Both have pros and cons. You make the decision based on your business requirements. If we keep EMPNO as the sole primary key column, we have to make sure that local DB do not

assign conflicting EMPNO values when adding new employees locally. In Pervasync, you can create a sync sequence for the EMPNO column and draw globally unique values from the sync sequence locally (For more information and explanation of sync sequence, see section 2.8.2 Sync Sequence). If you make DEPTNO and EMPNO together a composite primary key, you only need to make sure EMPNO is locally unique when you add a new employee. If your company does not require every employee to have a company wide unique ID (value of EMPNO), you can choose the latter way.

In summary, to make a table ready to be published as a sync table, it has to have a primary key defined and an association between the primary key and the subsetting column defined.

2. **Publish the sync table and add it to the sync schema**

When you define the sync table via the Pervasync web admin console at Sync Schemas-> Sync Tables-> Add new Sync Table, or via the admin Java API using method addSyncTable of class SyncServerAdmin, you have a chance to supply a data subsetting query. If you already have a sync table that you want to use for subsetting, you can click “Update”, and then edit “Subsetting Query”. Data subsetting query determines which rows go to which local database. There is no restriction on the query as long as it returns the Ids of the rows to be synced to a local DB. If we keep EMPNO as the sole primary key column we would use the following query:

```
SELECT EMPNO FROM SCOTT.EMP WHERE DEPTNO=${DEPTNO_PARAM}
```

If we make DEPTNO and EMPNO together a composite primary key we would use the following query:

```
SELECT DEPTNO, EMPNO FROM SCOTT.EMP WHERE DEPTNO=${DEPTNO_PARAM}
```

In the above queries, we have a placeholder, ${DEPTNO_PARAM}. The variables enclosed in ${} are subscription parameters. Queries for other tables could share the same parameter names. The collection of unique table level parameters becomes parameters for the published sync schema.

The queries in our example are among the simplest as they have only one parameter and reference only one table. One could have multiple parameters and use table joins in the query. Pervasync employs different algorithms for simple and complex queries to achieve maximum efficiency for both cases. You need to specify a subsetting mode when doing the subsetting: SIMPLE or COMPLEX. Subsetting mode SIMPLE is different from COMPLEX in that a SIMPLE query can select from only one table, which could be the sync table itself or a different table. For more information and explanation of SIMPLE or COMPLEX subsetting modes, see section 2.6.2: “Subsetting Modes: SIMPLE and COMPLEX”.

3. **Subscribe a sync user to a sync schema**

For each local DB, you create a sync client – a combination of user and device. When you subscribe a sync client to a sync schema via the Pervasync web admin console at Sync Schemas-> Schema Subscriptions-> Add new Schema Subscription, or via the admin Java API method addSchemaSubscription of class SyncServerAdmin, you have a chance to supply a value for each of the subscription parameters. If you already have a Schema Subscription that you want to use for subsetting, you can click “Update Subscription”, and then enter the value of the parameter, which you specify as a placeholder in subsetting query at step 2. In our case, we will see DEPTNO_PARAM listed as a parameter name and we can

assign a value. For example, if we want a sync client to sync data for DEPTNO=1, we put 1 as the value for DEPTNO_PARAM.

4. **Sync**

During synchronization for the sync client we just subscribed, effectively the following query is executed:

```
SELECT EMPNO FROM SCOTT.EMP WHERE DEPTNO=1
```

So that only the rows with DEPTNO=1 are synced to this client.

2.6.2 Subsetting Modes: SIMPLE and COMPLEX

The parameterized SQL query is at the heart of data subsetting. There is virtually no restriction to the query as long as it returns the primary key values of the desired data subsets. This gives users great flexibility but at the same time, it poses great challenges to the sync engine, which has to figure out physical and logical changes to the data subsets. Algorithms that apply to generic queries may not be scalable. On the other hand, algorithms that perform well for simple queries may not apply for complex queries.

Pervasync asks users to indicate the complexities of their queries via subsetting modes so that different algorithms can be applied to achieve best possible performance and scalability.

Criteria for SIMPLE query

There are two subsetting modes, SIMPLE and COMPLEX. Criteria for SIMPLE query are as follows.

1. The query can reference only one table. This table can be the table you are subsetting or a different table.

   The query predicates that involve the table columns can use not only “=”, but also other conditional operators like “>”, “<”, and “like” etc. For example, the following queries qualify as SIMPLE queries,

```
SELECT EMPNO FROM SCOTT.EMP WHERE DEPTNO=${DEPTNO_PARAM} AND TITLE =
              ${TITLE_PARAM}


SELECT EMPNO FROM SCOTT.EMP WHERE DEPTNO=${DEPTNO_PARAM} AND SAL <
              ${SAL_PARAM}


SELECT EMPNO FROM SCOTT.EMP WHERE SAL > ${SAL_MIN} AND SAL < ${SAL_MAX}


SELECT EMPNO FROM SCOTT.EMP WHERE JOB like '%Engineer'
```

   NOTE: You set the values of the Subsetting parameters when you subscribe a client to the sync schema. At runtime, the values are set to the queries as Strings (Char values).

Most database engines would implicitly convert the values to desired types, e.g. Numbers, when needed. However, for PostgreSQL, you have to do explicit casting. For example, if DEPTNO column is of NUMERIC type and TITLE is of VARCHAR type, you would change the first query above to the following:

```
SELECT EMPNO FROM SCOTT.EMP WHERE DEPTNO=CAST(${DEPTNO_PARAM} AS NUMERIC(20))
              AND TITLE = ${TITLE_PARAM}
```

2. The query cannot use functions, expressions, joins, set operations like “union”, “intersect” etc.

3. Query referenced columns (e.g. DEPTNO, MGR, SAL and JOB in the above examples) cannot have NULL values and the column data type cannot be too long. The reason is that these columns will become part of a primary key for a Pervasync internal table so they have to satisfy primary key column criteria of the database. If you cannot make these columns “NOT NULL” and have shorter length, mark the subsetting query COMPLEX.

Scalability implications

In general, SIMPLE queries are much more scalable than COMPLEX queries in terms of database space and sync engine processing time. For SIMPLE queries, sync engine performance is virtually independent of the number of sync clients you have.

COMPLEX queries can be scalable if a large number of clients share the same set of parameter values. For example, suppose you have 1000 clients and there are only 10 unique values for the subscription parameter DEPTNO_PARAM, the resources needed by sync engine are about the same as 10 clients each having a unique value for the parameter. Still, if at all possible, we recommend you use SIMPLE queries as opposed to COMPLEX queries.

2.6.3 Converting a COMPLEX Query to a SIMPLE Query

If you mark a SIMPLE query as COMPLEX, sync will still work but just not as efficiently. However, if you simply mark a COMPLEX query as SIMPLE, the sync may not work – you may find that client is not getting the changes you expect.

Fortunately, there are ways to convert a COMPLEX query to a SIMPLE query. For example, let’s say you want to sync users’ emails to their mobile devices and you use email recipient as subsetting parameter. The query qualifies as a SIMPLE query:

```
SELECT EMAIL_ID FROM USER_EMAILS WHERE RECEIPIENT = ${EMAIL_ADDRESS_1} OR
              RECEIPIENT = ${EMAIL_ADDRESS_2}
```

Now, let’s say you only want to sync last 30 days’ emails to save space on device. The query becomes COMPLEX:

```
SELECT EMAIL_ID FROM USER_EMAILS WHERE (RECEIPIENT = ${EMAIL_ADDRESS_1} OR
              RECEIPIENT = ${EMAIL_ADDRESS_2}) AND RECEIVE_DATE > (SYSDATE –
              INTERVAL ‘30’ DAY(5))
```

To convert this query to a SIMPLE query, you can add a column INCLUDE_IN_SYNC, which takes values of “Y” for yes and “N” for no and defaults to “N”. You create a DB job to update the INCLUDE_IN_SYNC values every night:

```
UPDATE USER_EMAILS SET INCLUDE_IN_SYNC=”Y” WHERE INCLUDE_IN_SYNC=”N” AND
              RECEIVE_DATE > (SYSDATE – INTERVAL ‘30’ DAY(5))
```

With this in place, you would be able to use the following SIMPLE query to achieve the same result as the original COMPLEX query.

```
SELECT EMAIL_ID FROM USER_EMAILS WHERE (RECEIPIENT = ${EMAIL_ADDRESS_1} OR
              RECEIPIENT = ${EMAIL_ADDRESS_2}) AND INCLUDE_IN_SYNC=”Y”
```

2.7 Schema Evolution and Sync Table Reload

Sometimes it is inevitable that new business requirements come up after the system has gone production and you have to change table structure or drop/add table from/to schemas. Pervasync supports schema evolution so that you don’t have to re-build the whole system from scratch.

2.7.1 Schema Evolution Steps

1. **Shutdown sync server**

Before you make any changes to the system, it is recommended to first shut down sync engine and sync servlet using the web admin console. It is also recommended to sync all the clients before the server shutdown so that all client side changes are uploaded to the server.

2. **Alter DB table**

Then, you make physical changes to the central database schemas/tables, such as create/drop tables, alter tables to add/remove columns or change column data types.

3. **Update sync table**

Go to the web admin console and locate the table you just changed. Delete the sync table if you dropped the corresponding DB table. Update the sync table if you altered the corresponding DB table.

4. **Start sync server**

Re-start sync engine and re-open sync servlet using the web admin console.

2.7.2 Propagation of Table Definition to Clients and Table Reload

The first sync after the server schema change will propagate the new schema definition to the client. If a sync table was removed, the corresponding client DB table will be dropped. Also in general, if a sync table was updated, the corresponding client DB table will be altered.

Some updates to the sync table would cause the client DB table to be dropped, re-created and re-populated with data from the server. We call this a table reload.

If you change the subsetting query, clients may be assigned a different subset of data. Therefore, Pervasync will do a table reload for all the clients. Altering a table's primary key will cause the same.

If you change the values of subsetting parameters for a particular client, that client would get a reload for the affected tables. Other tables and clients won't get a reload.

2.8 Generating Unique Key Values in Distributed Databases

A synchronization system contains distributed databases. How do you ensure that new records created on these databases have unique key values? In a single DB system, people usually use AUTO_INCREMENT (in MySQL), IDENTITY (in Microsoft SQL Server) columns or sequence objects (e.g. in Oracle, PostgreSQL and Microsoft SQL Server 2012) that produce unique sequence numbers. In a multi-DB environment, special treatments have to be done to AUTO_INCREMENT columns and sequences otherwise different DBs might generate same primary key values that would conflict with each other when new records are synced from one DB to another.

NOTE: The IDENTITY columns in Microsoft SQL Server are not distributed databases friendly. We recommend you use Sequences to generate unique key values. SQL Server 2012 and newer has sequence support. For older versions of SQL Server, you can use Pervasync stored procedures for sequence management.

2.8.1 Non-Conflicting AUTO_INCREMENT for MySQL Databases

NOTE: AUTO_INCREMENT does not apply to Oracle, which uses sequences. See next section for using sequences in a sync system.

AUTO_INCREMENT is the standard way for MySQL applications to generate unique key values. MySQL 5 introduces a couple of server variables, **auto_increment_increment** and **auto_increment_offset**, which make it possible to keep using AUTO_INCREMENT columns in a multi-DB system.

By default, **auto_increment_increment** and **auto_increment_offset** both have a value of 1. The inserted value is the least in the series [1, 2, 3…] that is greater than the maximum existing value of the column. When **auto_increment_increment** and **auto_increment_offset** take on non-default values, the series is no longer [1, 2, 3…], but calculated using the following formula:

auto_increment_offset + N × auto_increment_increment

where N is a non-negative integer value in the series [0, 1, 2, 3, …].

For example, with **auto_increment_increment** = 10 and **auto_increment_offset** = 5, the values are drawn from series [5, 15, 25…].

Following are the steps to take for using AUTO_INCREMENT in Pervasync.

1. Choose an **auto_increment_increment** value. The value should be at least one greater than the number of local DBs you plan to deploy. For example, you may choose 1000 if your system is expected to have hundreds of local DBs.

2. On central DB and all local DBs, edit your MySQL Server configuration file (usually called my.cnf or my.ini) to set **auto_increment_increment** and **auto_increment_offset** values. All MySQL servers should use a same **auto_increment_increment** value and a different **auto_increment_offset value**. For example, on central DB you may put

```
auto_increment_increment=1000
auto_increment_offset = 1
```

   while on the first local DB you may put

```
auto_increment_increment=1000
auto_increment_offset = 2
```

On central Db, keep the AUTO_INCREMENT column definitions intact. The AUTO_INCREMENT flag of the columns will be synced to local DB.

That's it! From now on, do inserts as you normally do on a single DB without worrying about unique key violations. Nevertheless, if for some reason you prefer not to use the AUTO_INCREMENT feature, you can use the Pervasync "Sync Sequence" feature described in the next section.

2.8.2 Sync Sequence

The idea of the Pervasync "sync sequence" feature is to divide a sequence into contiguous partitions/windows and distribute the partitions to central DB and local DBs. Database applications can draw globally unique values for unique keys from the sequence partitions. When a local DB is about to run out of the numbers, a new range is synced to the local DB.

Following are the steps to use Sync Sequences.

1. Determine central DB sequence partition range. Normally you create a sequence for each unique key column. The central DB sequence has to be modified or re-created so that it will only occupy a partition. The START WITH value should be 1 greater than the maximum existing value of the column. The MAXVALUE of the sequence should be big enough so that central DB app would not easily run out of numbers, while at the same time it shouldn't be too big so that local DBs have more room for their partitions. Manually create (or modify) the central DB sequence, for example, on Oracle server:

```
CREATE SEQUENCE pvsdemo.task_id_seq START WITH 1000 MAXVALUE 1000000000;
```

   and on MySQL server

```
CALL pvsadmin.create_sequence('pvsdemo', 'task_id_seq', 1000,
             1000000000);
```

2. Publish the sync sequence on the web admin console. Here you need to supply a sequence name, **task_id_seq** in our example and add it to a schema, **pvsdemo** in our example. In addition, you need to supply a "Start Value" and a "Window Size". The start value should be one greater than the central DB sequence partition MAXVALUE. In our case, it should be 1000000001. The window size determines the number of values available in one local DB sequence partition. We use 1000000 for our example.

3. After the first sync, the sequence partitions will be automatically created on local DBs. This is in contrast to central DB, where you create the sequences manually. To use the

sequences, call their NEXTVAL methods. For example, to insert into a table with column col1 as primary key, for Oracle database you do the following in your client application.

```
INSERT INTO pvcdemo.table1(c1, c2) VALUES
(pvcadmin.task_id_seq.nextval, 'hello world');
```

For MySQL database you do the following in your client application.

```
INSERT INTO pvcdemo.table1(c1, c2) VALUES
(pvcadmin.sequence_nextval('schema1', 'task_id_seq'), 'hello world');
```

For SQLite databases on Android or iOS, use the SyncClient method to retrieve the sequence next value and use it as key value for a new record. Note that the first argument is the logical sync schema name, not the client schema/database name. See API Javadoc for more details.

```
long idValue = pevasync.client.SyncClient.sequenceNextval("schema1",
               "task_id_seq");
```

2.8.3 Other Options

We believe that AUTO_INCREMENT and sync sequence are the best choices for most situations. Still, there are other options that may fit your specific needs. We list them below.

1. Only allow transactions to happen on central DB. Device local DBs are made read-only, i.e. for queries only. Believe it or not, there are systems that adopt this model.

2. Use randomly generated numbers for key values. The length of the random numbers has to be long to reduce the possibility of collisions.

3. Use a composite key. The sync client API has a method that returns the sync client ID. Apparently this is a globally unique number. You can use the client ID as the first column of a primary key. You then use a second column that takes locally unique values. Let's call it the LUID column. You define the client ID column and the LUID column together as your composite primary key. The values of the composite key are globally unique.

4. Compute a GUID using client ID and LUID. For example, you could use this formula:

   ```
   GUID = <client ID> * 1000000000 + LUID
   ```

5. Map local UID with GUID. This is the technique used by OMA DS (aka SyncML), and Activesync. Locally created records are assigned an LUID. During Activesync synchronization, a GUID for the same record is generated on server by mapping the LUID and synced back to client. For OMA DS, the client sends an LUID back to the server for every server sent record. A map table of LUID and GUID is maintained on the server.

   The mapping methods may be OK for simple PIM (Personal Information Management) sync. However, for enterprise applications that have large amounts of data, a lot of tables and complex referential relationships between tables, the mapping would cause performance and maintenance problems. Hence, Pervasync does not support mapping method.

2.9 Avoiding Sync Errors and Conflicts

Bad things happen. Synchronization systems are no exception. However, Pervasync has measures to help you keep things under control.

As you know a sync session has two phases, Check-in and Refresh. Refresh should rarely go wrong, as it is a forced application of central DB changes to local DB. If it does go wrong, you have to manually fix the error and try Refresh again. As a last resort, the local DB has to be re-loaded by either re-installing the client or re-subscribe the client from the server side.

Most of the errors happen in the Check-in phase where the client tries to apply local DB transactions to central DB. In the following we will talk about errors due to DB constraint violations and conflicts due to con-current updates of same records from different clients. The emphasis here is prevention.

2.9.1 Errors Due to DB Constraint Violations

The sync client check-in is just like other central DB transactions. It has to meet all the DB constraints to be committed in central DB.

Create Local DB Constraints via Sync SQL

First thing to do to avoid the constraint violations during Check-in is to ensure the local transactions meet the constraints on the client side. Pervasync does not sync central DB constraints to local DB except for NOT NULL and primary key constraints. However, Pervasync has a “Sync SQL” feature that allows you publish SQL statements to be applied to client DB during synchronization. On web admin console, locate the sync schema and click on “Add Sync Sql” to define the desired UNIQUE constraints, CHECK constraints or referential constraints, for example,

```
ALTER TABLE pvcdemo.tasks ADD CONSTRAINT tasks_unique UNIQUE (NAME)
```

Instead of local DB constraints, we recommend you ensure the constraints in the client application layer where you can give end users direct, meaningful feedback when a constraint is violated.

Sync Table Ranks: Check-in Order

Even though your local changes meet the referential constraints, during Check-in, you still may encounter referential constraint violations like “Parent Key Not Found” and “Child Record Found”. This has to do with the order of the DML (Data Manipulation Language) operations in the Check-in transaction.

Pervasync uses the sync table ranks to determine the check-in order. When you publish a sync table, you need to assign a rank number that indicates the referential relationship among the sync tables. Parent tables should have a smaller rank number than child tables. For Insert operations, tables with a lower rank number are done first so that parent table is inserted first to avoid the “Parent Key Not Found” error when child table is inserted. Deletes are done in the reverse order to avoid the “Child Record Found” error.

2.9.2 Conflict Detection and Resolution

In an online system, a client could lock a DB row (record) when making changes to it so that the client is sure that it is modifying an up-to-date version of the row. In a sync system, sync

client keeps a local copy of the row and makes offline changes to it without locking the central DB row. While the client makes these offline changes, other sync clients or central DB users may be making changes to the same row in central DB. During check-in time, if we do not do anything to handle conflicts, the newly checked-in row will overwrite changes from other sources. This may break your business logic since essentially the client was modifying an out-of-date copy of the row and changes from other clients were not taken into consideration.

Pervasync provides conflict detection and resolution to help you handle the conflicts so that your business requirements are satisfied.

There are three types of conflicts:

- Insert conflict – Insert causes primary key violation. Likely others have inserted a row with the same primary key value. To avoid insert conflicts, refer to section 2.8 Generating Unique Key Values in Distributed Databases.

- Update Conflict – Record to be updated has a higher version number indicating it was updated or has been deleted.

- Delete conflict – Record to be deleted has a higher version number indicating it was updated or has been deleted.

For each type of conflict, we provide three resolution methods:

- FORCE_CHECK_IN – Apply client transaction ignoring the conflict. For delete operations, records are deleted regardless of their versions. If records to be inserted already exist, update the records. If records to be updated have been deleted, insert the records.

- DISCARD: Discard the conflicting changes from the client and check in non-conflicting changes.

- REPORT_ERROR: Treat conflicts as errors and notify the client. This will cause the sync session to fail. In this case the client could do a REFRESH_ONLY sync to let the server side changes overwrite the conflicting client side changes. This effectively modified the client transaction so that it won't cause conflicts in next check-in assuming others won't make new changes on server again. Before check-in, the client application could give end users a chance to inspect and update the new version of the records. End users should be clear that until the check-in succeeds, all changes are temporary although they are committed locally.

To configure the resolution methods, edit the following parameters in <PERVASYNC SERVER HOME>/config/pervasync_server_<db>.conf.

```
pervasync.conflict.resolution.InsertExisting=REPORT_ERROR
pervasync.conflict.resolution.UpdateChanged=FORCE_CHECK_IN
pervasync.conflict.resolution.DeleteChanged=DISCARD
```

2.9.3 REFRESH-ONLY to the Rescue

Pervasync supports three sync directions, TWO_WAY, REFRESH_ONLY and CHECK_IN_ONLY. You can pass in sync direction to the sync() method of client SyncAgent

object, or you can use the corresponding pvc.bat/pvc.sh sub commands sync, refresh and checkin.

In a two-way sync session, Check-in is always done before Refresh. The reason is that conflict detection is done on the server side during Check-in. Refresh force-applies changes on local DB without any conflict detection. If Refresh is done before a successful Check-in, some local changes may be silently overwritten.

However, when Check-in encounters some errors or conflicts and cannot proceed, it is often useful to do a REFESH-ONLY sync to intentionally overwrite client changes with server changes so that a subsequent Check-in can go through. Examples are that the server has added or dropped a column, or the server sets the conflict resolution method to be REPORT_ERROR and a conflict is found. The algorithm is as follows.

1. Do two-way sync.

2. Catch sync error, present the error to end-user and ask for permission to do a REFESH-ONLY sync.

3. Do a REFESH-ONLY sync and give users a chance to edit the refreshed data.

4. Do two-way sync.

By the way, even a lot of online applications do not lock the records while end users are editing the data. Instead, when the user is done with the editing and submits the data, they do the conflict check and if a conflict is found, a new version of the data is returned for the user to edit and re-submit. The algorithm is essentially the same as above.

2.10 Configuration and Logging

Pervasync server and client have similar ways of configuration and logging.

Configuration is mainly done by editing the **.conf** files, which is located in the **config** directory. Property values are needed to customize runtime behaviors. Server or client has to be re-started for any changes to take effect.

NOTE: The **.dat** file in the same folder is for Pervasync internal use. So never modify it.

2.10.1 The Server Configuration File

The content of file **pervasync_server_oracle.conf** is located in the server **config** directory:

```
#====================================================================
#==                 Pervasync server for Oracle                    ==
#==                 conf file for runtime options                  ==
#==                                                                ==
#==   Property values are needed to customize runtime behaviors.   ==
#==   Server has to be re-started for any changes to take effect.  ==
#====================================================================

# This is the pervasync license key that you get from Pervasync sales
pervasync.server.license.key=0

# encryptor class name. Class needs to implement interface
              pervasync.Encryptor
pervasync.encryptor=pervasync.security.DefaultEncryptor
```

```
# UserManager class name. Class needs to implement interface
              pervasync.server.UserManager
pervasync.server.user.manager=pervasync.server.PervasyncUserManager

# Values: SEPARATELY, TOGETHER; default TOGETHER
# This will affect only client check in, not refresh engine
# SEPARATELY (for table groups) for maximum concurrency, TOGETHER for maximum
              consistency
pervasync.check.in.commit.mode=TOGETHER

# client subscription metadata cache limit (number of clients whose sub
              metadata are cached)
pervasync.server.cached.client.subs=1000

# Sync Server DB create user, table options
pervasync.server.admin.user.options=DEFAULT TABLESPACE USERS TEMPORARY
              TABLESPACE TEMP
pervasync.server.admin.user.grants=connect, resource, select any table,
              delete any table, insert any table, update any table, create
              any table, drop any table,alter any table,lock any table,create
              any sequence, drop any sequence, alter any sequence, select any
              sequence, create any view, drop any view,create any
              trigger,drop any trigger,create any index, drop any
              index,create role,drop any role,grant any role,create profile,
              drop profile,create any synonym, create user, drop user,grant
              any privilege,SELECT_CATALOG_ROLE
pervasync.server.db.table.options=

# acceptable values: true, false
pervasync.server.db.conn.cache.enable=true

# acceptable values: integer; default 0; don't set to a value other than 0
pervasync.server.db.conn.cache.InitialLimit=0

# acceptable values: integer; default 0
pervasync.server.db.conn.cache.MinLimit=0

# acceptable values: integer; default no limit
pervasync.server.db.conn.cache.MaxLimit=10

# acceptable values: integer seconds?; default 0, no timeout
pervasync.server.db.conn.CacheInactivityTimeout=0

# acceptable values: integer; default 0
pervasync.server.db.conn.cache.MaxStatementsLimit=0

# When client uses JSON transport, a sync session could consist multiple sync
# messages. This sets an upper limit for the message size.
Pervasync.max.message.size=10000000

# BLOBs are transferred in chunks. This is the chunk size (number of bytes).
Pervasync.blob.buffer.size=4000

# CLOBs are transferred in chunks. This is the chunk size (number of chars).
Pervasync.clob.buffer.size=4000

# If set to true, use one query to select non-lob columns and a separate
# query to select lob columns. Use true for Mysql Db and false for Oracle Db.
Pervasync.separate.lob.query=false

# Sleep this amount of milliseconds between trigger creation to work around a
              mysql issue.
Pervasync.create.trigger.sleep.ms=0

# For multi-message session, client has to come back within this
```

```
# amount of time, default 30 minutes
pervasync.session.timeout.seconds=180

# Max message processing time, default 100 hours. Need a big value for
# object transport where there is only one request and response message
# for the whole session
pervasync.message.timeout.seconds=360000

#
# Sync engine run interval (seconds).
# Sync engine will start another run after this amount of time
pervasync.engine.run.interval=30

#
# Sync engine retry interval (seconds).
# When sync engine encounters a runtime error, it will retry after this
              amount of time
pervasync.engine.retry.interval=300

#
# Sync engine retry limit.
# When sync engine encounters a runtime error, it will retry this many times
# before it shuts down itself.
Pervasync.engine.retry.limit=48

# Number of days that delete operations are kept in the user's subscriptions.
# older deletes will be purged. Sync client would get “snapshot too old”
# error if the deletes were not synced down before they were purged.
Pervasync.server.deletes.keep.days=365
pervasync.server.ignore.snapshot.too.old=false

# Which sync sessions are logged. Valid values are NONE, ALL and NON_EMPTY.
# NON_EMPTY sessions have at least one data op or an exception.
Pervasync.server.sync.history.log.mode=NON_EMPTY

# Number of days that sync session logs are kept before they are purged.
Pervasync.server.sync.history.keep.days=30

#
# Conflict detection and Conflict resolution. Default REPORT_ERROR.
# For each type of conflict, valid resolution methods are
#    -- FORCE_CHECK_IN: no conflict detection; Delete as is; If Insert fails
              Update; If update fails insert.
#    -- DISCARD: Ignore conflicting changes; check in non-conflicting changes
#    -- REPORT_ERROR: Treat conflicts as errors and notify client
#
# Insert causes primary key violation
pervasync.conflict.resolution.InsertExisting=REPORT_ERROR

# Record to be updated has a higher version number (was updated or has been
              deleted)
pervasync.conflict.resolution.UpdateChanged=FORCE_CHECK_IN

# Record to be deleted has a higher version number (was updated or has been
              deleted)
pervasync.conflict.resolution.DeleteChanged=DISCARD

# If TRUE/YES, enable rollback of file sync
pervasync.file.check.in.rollback.enabled=TRUE

# SQl statements to be executed before check in.
pervasync.server.before.check.in.sql=

# SQl statements to be executed after check in.
pervasync.server.after.check.in.sql=
```

```
# SQl statements to be executed before refresh.
Pervasync.server.before.refresh.sql=

# SQl statements to be executed after refresh.
Pervasync.server.after.refresh.sql=

# If TRUE/YES, ignore errors in executing the before/after sqls
pervasync.server.before.after.sql.ignore.error=TRUE

# The level of the logger named “pervasync.server.oracle” and appenders
# Acceptable values for level and threshold are:
# DEBUG, INFO, WARN, ERROR and FATAL
log4j.logger.pervasync.server.oracle=DEBUG, CONSOLE, FILE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=INFO
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=[%d{MM/dd,HH:mm:ss} %t] %-5p
              %x- %m%n

# Appender FILE writes to the log files in Pervasync log folder.
Log4j.appender.FILE=org.apache.log4j.RollingFileAppender
log4j.appender.FILE.Threshold=DEBUG
log4j.appender.FILE.File=${pervasync.home}/log/pervasync_server_oracle.log
log4j.appender.FILE.MaxBackupIndex=3
log4j.appender.FILE.MaxFileSize=1MB

# Truncate file or not
log4j.appender.FILE.Append=true

# Appender A2 uses the PatternLayout.
Log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=[%d{MM/dd,HH:mm:ss} %t] %-5p %x-
              %m%n

#
# Oracle JDBC Logging
#
# acceptable values: OFF, SEVERE, WARNING,INFO, CONFIG, FINE, FINER, FINEST,
              ALL
# This will override <JAVA_HOME>/jre/lib/logging.properties
# oracle.jdbc.driver.OracleLog.level value
# oracle.jdbc.driver.OracleLog.level=WARNING
oracle.jdbc.level=OFF

#
# Email notification settings
#
# Use true to turn on email notifications of server failures like
# sync engine errors and/or sync session errors
pervasync.server.enable.notification = false
# Number of seconds before the Notification engine wakes up to do its job
pervasync.server.notification.process.interval = 600
# smtp mail server host name. Defaults to localhost
pervasync.server.mail.smtp.host =
# smtp mail server user name. Defaults to current login user.
Pervasync.server.mail.smtp.user =
# comma-separated notification recipient email addresses
pervasync.server.mail.recipients =
#
# admin console preferences
#
pervasync.console.rows.to.display=25
pervasync.console.rows.to.retrieve=1000
pervasync.console.look.and.feel=beach
pervasync.console.accessibility.mode=inaccessible
pervasync.console.date.format=short
```

```
pervasync.console.time.format=short
```

2.10.2 The Client Configuration File

The content of file **pervasync_client_oracle.conf** is located in the client **config** directory:

```
#=====================================================================
#==                 Pervasync client for Oracle                      ==
#==                 conf file for runtime options                   ==
#==                                                                 ==
#==   Property values are needed to customize runtime behaviors.    ==
#==   Client has to be re-started for any changes to take effect.    ==
#=====================================================================

# General

# Sync Client DB create user table options
pervasync.client.admin.user.options=DEFAULT TABLESPACE USERS TEMPORARY
              TABLESPACE TEMP
pervasync.client.admin.user.grants=connect, resource, select any table,
              delete any table, insert any table, update any table, create
              any table, drop any table, alter any table, lock any table,
              create any sequence, drop any sequence, alter any sequence,
              select any sequence, create any view, drop any view, create any
              trigger,drop any trigger, create any index, drop any index,
              create role,drop any role,grant any role,create profile, drop
              profile,create any synonym, create user, drop user, grant any
              privilege,SELECT_CATALOG_ROLE
pervasync.client.db.table.options=

# Sync sequence options
# A sequence is considered full if (MaxValue - CurrentValue) < threshold
# A full sequence will get refreshed with a new range in next sync
pervasync.client.sync.sequence.threshold=10

# Transport serialization, use Object or Json
pervasync.client.transport.serialization=Json

# Http transport content encoding. Use gzip to compress traffic; else leave
              empty
pervasync.client.transport.content.encoding=gzip

# When client uses JSON transport, a sync session could consist multiple sync
# messages. This sets an upper limit for the message size.
Pervasync.max.message.size=4000000

# BLOBs are transferred in chunks. This is the chunk size (number of bytes).
Pervasync.blob.buffer.size=4000

# CLOBs are transferred in chunks. This is the chunk size (number of chars).
Pervasync.clob.buffer.size=4000

# Values: SEPARATELY, TOGETHER; default TOGETHER
# This will affect only client refresh
# SEPARATELY (for table groups) for maximum concurrency, TOGETHER for maximum
              consistency
pervasync.client.refresh.commit.mode=TOGETHER

# If set to true, use one query to select non-lob columns and a separate
# query to select lob columns. Use true for Mysql Db and false for Oracle Db.
Pervasync.separate.lob.query=false

# Sleep this amount of milliseconds between trigger creation to work around a
              mysql issue.
```

```
Pervasync.create.trigger.sleep.ms=0

# Whether to send the server the client sync summary. Valid values are
              ALWAYS,
# NEVER, NON_EMPTY and ON_ERROR.
Pervasync.client.send.sync.summary=NON_EMPTY

# Number of days that sync session logs are kept before they are purged.
Pervasync.client.sync.history.keep.days=30

# If TRUE/YES, enable rollback of file sync
pervasync.file.refresh.rollback.enabled=TRUE

# SQl statements to be executed before check in.
pervasync.client.before.check.in.sql=

# SQl statements to be executed after check in.
pervasync.client.after.check.in.sql=

# SQl statements to be executed before refresh.
Pervasync.client.before.refresh.sql=

# SQl statements to be executed after refresh.
Pervasync.client.after.refresh.sql=

# If TRUE/YES, ignore errors in executing the before/after sqls
pervasync.client.before.after.sql.ignore.error=TRUE

#Log4j properties

# The level of the logger named “pervasync.client.oracle” and appenders
# Acceptable values for level and threshold are:
# DEBUG, INFO, WARN, ERROR and FATAL
log4j.logger.pervasync.client.oracle=DEBUG, CONSOLE, FILE

log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=INFO

log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=[%d{MM/dd,HH:mm:ss} %t] %-5p
              %x- %m%n

# Appender FILE writes to the log files in Pervasync log folder.
Log4j.appender.FILE=org.apache.log4j.RollingFileAppender
log4j.appender.FILE.Threshold=DEBUG
log4j.appender.FILE.File=${pervasync.home}/log/pervasync_client_oracle.log
log4j.appender.FILE.MaxBackupIndex=2
log4j.appender.FILE.MaxFileSize=1MB

# Truncate FILE or not
log4j.appender.FILE.Append=true

# Appender CONSOLE uses the PatternLayout.
Log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=[%d{MM/dd,HH:mm:ss} %t] %-5p %x-
              %m%n

# Oracle JDBC Logging
# acceptable values: OFF, SEVERE, WARNING,INFO, CONFIG, FINE, FINER, FINEST,
              ALL
# This will override <JAVA_HOME>/jre/lib/logging.properties
              oracle.jdbc.driver.OracleLog.level value
#oracle.jdbc.driver.OracleLog.level=WARNING
oracle.jdbc.level=OFF

# Whether to allow dropping tables in initial sync
```

```
pervasync.client.allow.dropping.tables=true

# Auto sync schedules
# Schedule1
pervasync.client.schedule1.enabled=false
pervasync.client.schedule1.sync.direction=TWO_WAY
pervasync.client.schedule1.start.time=1264208583000
pervasync.client.schedule1.repeat=true
pervasync.client.schedule1.interval.hours=0
pervasync.client.schedule1.interval.minutes=15
pervasync.client.schedule1.interval.seconds=0

# Schedule2
pervasync.client.schedule2.enabled=false
pervasync.client.schedule2.sync.direction=TWO_WAY
pervasync.client.schedule2.start.time=0
pervasync.client.schedule2.repeat=false
pervasync.client.schedule2.interval.hours=0
pervasync.client.schedule2.interval.minutes=0
pervasync.client.schedule2.interval.seconds=0

# Schedule3
pervasync.client.schedule3.enabled=false
pervasync.client.schedule3.sync.direction=TWO_WAY
pervasync.client.schedule3.start.time=0
pervasync.client.schedule3.repeat=false
pervasync.client.schedule3.interval.hours=0
pervasync.client.schedule3.interval.minutes=0
pervasync.client.schedule3.interval.seconds=0
```

2.10.3 Email notification settings

To turn on email notifications of server failures like sync engine errors and/or sync session errors, assign a “true” value for the sync server parameter “pervasync.server.enable.notification”.

In addition, you need to specify the email recipients. If the sync server host does not have a local SMTP setup, you need to specify a SMTP host.

```
#
# Email notification settings
#
# Use true to turn on email notifications of server failures like
# sync engine errors and/or sync session errors
pervasync.server.enable.notification = false
# Number of seconds before the Notification engine wakes up to do its job
pervasync.server.notification.process.interval = 600
# smtp mail server host name. Defaults to localhost
pervasync.server.mail.smtp.host =
# smtp mail server user name. Defaults to current login user.
Pervasync.server.mail.smtp.user =
# comma-separated notification recipient email addresses
pervasync.server.mail.recipients =
```

If notifications were failed to be sent because of the following exception,

```
Util.sendMail exception: javax.mail.MessagingException: Can't send command to
              SMTP host;
nested exception is:
javax.net.ssl.SSLHandshakeException: No appropriate protocol (protocol is
              disabled or cipher suites are inappropriate)
```

It could be that the Java version used had some default security settings that were not compatible with the SMTP server. Find the following file(s),

Windows:

$JAVA_HOME/conf/security/java.security

$JAVA_HOME/lib/security/java.security

Linux:

/etc/java*/security/java.security

and comment out this line:

```
jdk.tls.disabledAlgorithms=SSLv3, TLSv1, RC4, DES, MD5withRSA, DH keySize <
1024, \
EC keySize < 224, 3DES_EDE_CBC, anon, NULL
```

$JAVA_HOME is that of the Java used by Tomcat server. Restart Tomcat server after.

Another possible fix is to change the default TLS version of Tomcat from 1.0 to 1.2.

2.10.4 Logging

You can change the logging behavior by editing the related properties in the configuration files. By default, the Pervasync logging outputs go to log files in folder **log** and the console/stdout.

NOTE: Depending on the running environment, the console/stdout logs might be redirected to files too. For example, the Tomcat console/stdout logs on Linux is redirected to <Catalina_Home>/logs/Catalina.out.

2.11 Pervasync Server Java API

2.11.1 Server API Javadoc

To access the server API javadoc, open the following page in a browser:

```
<PERVASYNC_HOME>/doc/server_javadoc/index.html
```

The Pervasync server API includes a SyncServerAdmin class for interacting with the server DB repository. There are also a UserManager interface and Encryptor interface for users to customize user management and encryption.

The SyncServerAdmin class in package pervasync.server contains methods for users to publish sync objects, as well as create subscriptions.

The UserManager interface in package pervasync.server enables a user to plug in his own user management system for Pervasync's use.

The Encryptor interface in package pervasync.security enables a user to plug in his own encryption implementations for Pervasync's use.

The SyncServerAdmin class contains methods for users to publish sync objects, as well as create subscriptions. To call its methods, a SyncServerAdmin instance needs to be created, e.g., syncServerAdmin = SyncServerAdmin.getInstance(). The constructor will open a JDBC connection to the Pervasync server repository database. The destroy() method should be

called when you are done with the SyncServerAdmin object so that the database connection could be released.

The methods that start with “add” would add sync objects to the sync repository. Those that start with “remove” would remove and those start with “update” would update.

A sync schema is a container that contains other sync objects, including sync tables, sequences and sql statements to be applied to client DB. Typically, you create a schema and add other sync objects to it. That makes a publication. Then you create a sync client and subscribe the client to the publication.

A sync schema corresponds to a physical DB schema that by default resides in the same DB as the sync repository. To publish a schema from a different DB, you need to use the “addServerDb” method to define that DB.

In addition to publishing sync schemas for synchronizing DB data to devices, you can also publish a file folder, which is not DB related, and sync its files to devices.

When you publish sync objects, you can optionally define some sync parameters. At subscription time, you would specify values to the parameters so that different clients could get different sub-sets of data.

The xmlExport() method would export the sync object definitions into an XML doc. Then you can use xmlUnpublish() to remove all the sync objects and use xmlPublish() to re-create all the sync objects.

2.11.2 Setting Up Environment for Server Applications

To make the sync server admin API available to server applications, include the **classes** directory and the necessary jar files in the CLASSPATH environment variable.

NOTE: We assume that you have already included the appropriate JDBC driver (or Java connector for MySQL) jar in CLASSPATH. The driver (or connector) jars shipped with Pervasync are located at <Pervasync Server Home>/lib.

Windows**:**

```
SET CLASSPATH=C:\pervasync_server-9.0.3\classes;
              C:\pervasync_server-9.0.3\lib\commons-codec-1.3.jar;C:\pervasyn
              c_server-9.0.3\lib\log4j-1.2.9.jar;%CLASSPATH%
```

Linux/Unix bash:

```
export CLASSPATH=/pervasync_server-9.0.3/classes:
              /pervasync_server-9.0.3/lib/commons-codec-1.3.jar:
              /pervasync_server-9.0.3/lib/log4j-1.2.9.jar :${CLASSPATH}
```

Linux/Unix csh:

```
setenv CLASSPATH /pervasync_server-9.0.3/classes:
              /pervasync_server-9.0.3/lib/commons-codec-1.3.jar:
              /pervasync_server-9.0.3/lib/log4j-1.2.9.jar :${CLASSPATH}
```

2.12 Pervasync Client Java API

NOTE: If sync client resides inside a firewall and sync server resides outside the firewall, you need to set the HTTP proxy host and port in the client configuration file. See section 2.10.2 for details.

2.12.1 Client API Javadoc

To access the client API javadoc, open the following page in a browser:

```
<PERVASYNC_HOME>/doc/client_javadoc/index.html
```

The Pervasync client API includes a SyncClientAdmin class for interacting with the client DB repository and a SyncAgent class for interacting with the sync agent.

The SyncClientAdmin class in package pervasync.client contains methods for users to do admin work, such as setting the sync user's password etc.

The SyncAgent class in package pervasync.client is for client applications to start a sync session.

The Encryptor interface in package pervasync.security enables a user to plug in his own encryption implementations for Pervasync's use.

Package pervasync contains classes used by the above-mentioned classes.

The SyncAgent class is for client applications to trigger a sync session. The sync agent synchronizes application data on device with that on server on behalf of the client application. In your client application, you should call the static method getInstance() to get a SyncAgent object and then call its instance method sync() to trigger a synchronization. The method sync() can be called multiple times and each call would return a SyncSummary object for you to check the sync result. The instance method destroy() should be called when you are done with the SyncAgent object.

2.12.2 Setting Up Environment for Client Applications

To make the sync client admin API available to client applications, include the **classes** directory and the necessary jar files in the CLASSPATH environment variable.

NOTE: We assume that you have already included the appropriate JDBC driver (or Java connector for MySQL) jar in CLASSPATH. The driver (or connector) jars shipped with Pervasync are located at <Pervasync Client Home>/lib.

Windows**:**

```
SET CLASSPATH=C:\pervasync_ client-9.0.3\classes;
              C:\pervasync_client-9.0.3\lib\commons-codec-1.3.jar;C:\pervasyn
              c_client-9.0.3\lib\log4j-1.2.9.jar;%CLASSPATH%
```

Linux/Unix bash:

```
export CLASSPATH=/pervasync_client-9.0.3/classes:
              /pervasync_client-9.0.3/lib/commons-codec-1.3.jar:
              /pervasync_client-9.0.3/lib/log4j-1.2.9.jar :${CLASSPATH}
```

Linux/Unix csh:

```
setenv CLASSPATH /pervasync_client-9.0.3/classes:
              /pervasync_client-9.0.3/lib/commons-codec-1.3.jar:
              /pervasync_client-9.0.3/lib/log4j-1.2.9.jar :${CLASSPATH}
```

2.12.3 Connecting to the Local DB Schema from Your Client Application

For client deployment, we recommend you embed the sync client in your client application so that they are installed at the same time. The sync client keeps the local DB and central DB in sync while the client app connects to the local DB schema and does its job. The client DB schema name is what you have specified when you created the sync schema on the server. The password is the same as the Pervasync client **Admin Password** you specified in the GUI during sync client install.

2.13 Using Sync SQL to Create Indexes and Constraints on Local DBs

2.13.1 Sync SQL

Pervasync does not automatically sync indexes (except for Primary key indexes) and constraints (except for NOT NULL) to local databases. The reason is to make the system less complex and to make synchronization between different types of databases possible.

In addition to indexes and constraints, there are other database objects, such as views, functions, triggers and PL/SQL packages (for Oracle) that users may or may not want to sync to local DBs. If they do, they may not want the exact same objects as central Db to be created on local DBs.

To assist users in managing these generic DB objects, Pervasync implemented a feature called “Sync SQL” that enables users to publish SQL statements on the web admin console. The SQL statements will be downloaded and executed on local DB during synchronization. In the SQL statement, users could create, alter and delete any DB objects that are appropriate for the local databases.

Sync SQL belongs to a sync schema. To get started, locate a sync schema on the web admin console “Sync Schemas” page, click on “Sqls” under “Sync Objects” and then you will get to a page where you can add, remove and update sync SQLs.

The Sync SQLs are executed with the client side sync schema as the default schema.

NOTE: How to include multiple SQL statements in one Sync SQL so that you don’t have to create too many Sync SQLs? For Oracle databases each Sync SQL has to be a single execution unit to be executed by the database. However you could enclose multiple SQL statements by “BEGIN” and “END;” to make it a PLSQL block so that it can be executed as a single unit. For MySQL databases, you could put multiple statements, separated by

semicolon, in one Sync SQL as long as you have the “allowMultiQueries=true” in your JDBC URL, for example:

```
jdbc:mysql://localhost:3306/?zeroDateTimeBehavior=convertToNull&allowMultiQue
              ries=true
```

To update the JDBC URL, go to the “Setup” tab of the sync client UI, enable “Advanced Mode”, edit and submit.

NOTE: For Oracle databases, a simple SQL statement cannot have a trailing “;” while a PLSQL block has to have a trailing “;”.

2.13.2 An Example of Foreign Key Creation via Sync SQL

Foreign key constraints on client DB can be tricky as they may interfere with the synchronization process. For example, in some situations like subscription parameter values being updated, Pervasync would need to re-create and re-populate affected tables on device. If a table has a foreign key constraint, the operation may fail.

A solution to this problem is to disable the foreign key constraints during sync: drop the foreign key constraints before sync and re-create the constraints after sync. In <Pervasync Client Home>/config/ pervasync_client_mysql.conf, you can set the "drop constraint" sql as value of parameter pervasync.client.before.refresh.sql and set the "create constraint" sql as value of pervasync.client.after.refresh.sql. To make the “create/drop” sql easier to manage, you can put them in stored procedures and use Sync SQL to create the stored procedures. This way you can make changes to the constraints any time you want from the admin console.

Following is an example of the stored procedures that you can modify and then publish as a Sync SQL of your publication schema.

```
DROP PROCEDURE IF EXISTS `CreateFK`;

CREATE DEFINER=`root`@`localhost` PROCEDURE `CreateFK`(
    IN parm_table_name VARCHAR(100),
    IN parm_key_name VARCHAR(100),
    IN parm_cascade_on_delete BOOL,
    IN parm_child_field_name VARCHAR(100),
    IN parm_parent_table_name VARCHAR(100),
    IN parm_parent_field_name VARCHAR(100)
)
BEGIN
    set @ParmTable = parm_table_name ;
    set @ParmKey = parm_key_name ;
    set @ParmChildFieldName = parm_child_field_name;
    set @ParmParentTableName = parm_parent_table_name;
    set @ParmParentFieldName = parm_parent_field_name;

    IF EXISTS (SELECT NULL FROM information_schema.TABLE_CONSTRAINTS WHERE
              CONSTRAINT_SCHEMA = DATABASE() AND CONSTRAINT_NAME =
              parm_key_name) THEN
        set @StatementToExecute = concat('ALTER TABLE ',@ParmTable,' DROP
              FOREIGN KEY ',@ParmKey);
        prepare DynamicStatement from @StatementToExecute ;
        execute DynamicStatement ;
        deallocate prepare DynamicStatement ;
    END IF;

    IF (parm_cascade_on_delete = false) THEN
```

```
        set @AddFKStatement1 = concat('ALTER TABLE `',@ParmTable,'` ADD
              CONSTRAINT `',@ParmKey, '` FOREIGN KEY (`',
              @ParmChildFieldName, '`) REFERENCES `', @ParmParentTableName,
              '` (`', parm_parent_field_name, '`) ON UPDATE NO ACTION');
        prepare DynamicStatement1 from @AddFKStatement1;
        execute DynamicStatement1 ;
        deallocate prepare DynamicStatement1 ;
    ELSE
        set @AddFKStatement2 = concat('ALTER TABLE `',@ParmTable,'` ADD
              CONSTRAINT `',@ParmKey, '` FOREIGN KEY (`',
              @ParmChildFieldName, '`) REFERENCES `', @ParmParentTableName,
              '` (`', parm_parent_field_name, '`) ON DELETE CASCADE ON UPDATE
              NO ACTION');
        prepare DynamicStatement2 from @AddFKStatement2;
        execute DynamicStatement2 ;
        deallocate prepare DynamicStatement2 ;
    END IF;
END


DROP PROCEDURE IF EXISTS `DropFK`;

CREATE DEFINER=`root`@`localhost` PROCEDURE `DropFK`(
    IN parm_table_name VARCHAR(100),
    IN parm_key_name VARCHAR(100),
    IN parm_cascade_on_delete BOOL,
    IN parm_child_field_name VARCHAR(100),
    IN parm_parent_table_name VARCHAR(100),
    IN parm_parent_field_name VARCHAR(100)
)
BEGIN
    set @ParmTable = parm_table_name ;
    set @ParmKey = parm_key_name ;
    set @ParmChildFieldName = parm_child_field_name;
    set @ParmParentTableName = parm_parent_table_name;
    set @ParmParentFieldName = parm_parent_field_name;

    IF EXISTS (SELECT NULL FROM information_schema.TABLE_CONSTRAINTS WHERE
              CONSTRAINT_SCHEMA = DATABASE() AND CONSTRAINT_NAME =
              parm_key_name) THEN
        set @StatementToExecute = concat('ALTER TABLE ',@ParmTable,' DROP
              FOREIGN KEY ',@ParmKey);
        prepare DynamicStatement from @StatementToExecute ;
        execute DynamicStatement ;
        deallocate prepare DynamicStatement ;
    END IF;
END


DROP PROCEDURE IF EXISTS `CreateForeignKeys`;

CREATE DEFINER=`root`@`localhost` PROCEDURE `CreateForeignKeys`( )
BEGIN

CALL YourDBName.CreateFK('announcementattachments',
              'fk_AnnouncementAttachments_Announcements1', true,
              'AnnouncementID', 'announcements', 'AnnouncementID');

CALL YourDBName.CreateFK('announcementattachments',
              'fk_AnnouncementAttachments_Companies', false, 'CompanyID',
              'companies', 'CompanyID');

.... (add the other call to CreateFK here for other foreign keys)

END
```

```
DROP PROCEDURE IF EXISTS `DropForeignKeys`;

CREATE DEFINER=`root`@`localhost` PROCEDURE `DropForeignKeys`( )
BEGIN

CALL YourDBName.DropFK('announcementattachments',
              'fk_AnnouncementAttachments_Announcements1', true,
              'AnnouncementID', 'announcements', 'AnnouncementID');

CALL YourDBName.DropFK('announcementattachments',
              'fk_AnnouncementAttachments_Companies', false, 'CompanyID',
              'companies', 'CompanyID');

.... (add the other call to DropFK here for other foreign keys)

END
```

After that, call the stored procedures DropForeignKeys and CreateForeignKeys in the client config file:

```
pervasync.client.before.refresh.sql = call YourDBName.DropForeignKeys()
pervasync.client.after.refresh.sql = call YourDBName.CreateForeignKeys()
```

2.14 Sync Based on Network Characteristics

A mobile computer usually has multiple network interfaces, e.g. cable, Wi-Fi and cellular wireless (mobile data), to connect to the Internet, through which to connect to the central sync server. Some of the network interfaces are slower or more expensive compared with others. On the other hand, of all the subscribed publications for synchronization, some may involve much larger amounts of data than others. One example is a folder of media files (photos or videos) or a database table of blobs of data items. When you are on a slow or expensive network, you may want to exclude these publications from synchronization. The “Sync Based on Network Characteristics” feature was implemented for this purpose.

2.14.1 Defining Network Characteristics Using a Matching String

First, we provide a way to identify the network interfaces of your client machines. After you install Pervasync client, you can use the utility listnetworks.bat (or listnetworks.sh for Linux) in the bin folder of Pervasync client home to list attributes of all the network interfaces on client machines. Pay attention to the display names, hardware addresses and IP addresses of your network cards. Following is a sample output.

```
C:\pervasync\bin>listnetworks.bat

Network Name:     lo
Display Name:     MS TCP Loopback interface
Hardware Address:
IP Address:       127.0.0.1
IP Address:       0:0:0:0:0:0:0:1
IP Address:       fe80:0:0:0:0:0:0:1%1

Network Name:     eth0
Display Name:     Intel(R) PRO/Wireless 3945ABG Network Connection - McAfee
                NDIS Intermediate Filter Miniport
Hardware Address: 0018DE157BA9
IP Address:       182.138.0.102
IP Address:       fe80:0:0:0:218:deff:fe15:7ba9%4
```

According to the network attributes, admin comes up with strings that match the display name, hardware address and/or IP address. The matching string defines a specific network

card or a type of network. The matching string may contain operators “&&” and “||”. For example, the following matching string will match network eth0 in the above listing:

```
Intel && Wireless && Network Connection
```

The following matching string will match network cards that have an IP address that starts with “182.138” or “193.111.2”:

```
182.138. || 193.111.2
```

2.14.2 Specifying No-Sync Lists Associated with Sync Schemas and Sync Folders

After you have gathered information about network characteristics of your client machines, then, on Pervasync admin console, you can specify No-Sync Lists Associated with Sync Schemas and Sync Folders.

Pervasync adds two no-sync-list attributes to the sync schema and sync folder definition: NO_INIT_SYNC_NETWORKS and NO_SYNC_NETWORKS. The values of the attributes are the matching strings that identify a specific network card or a type of network. To enable this feature, you need to change the value of the No-Init-Sync Networks and No-Sync Networks fields from the default value “NONE” into matching string that identifies the network cards. You may use full strings or sub-strings of the following network card attributes: display name, hardware address and IP address. You may also have “&&” and/or “||” between the strings to indicate whether it’s “match all” or “match any” respectively.

When you add a new publication (sync schema or folder), you specify global values of the no-sync lists that apply to all user devices. At subscription time you have a chance to overwrite the values for a specific user device or group. That is to say, values can be specified at publication level, group subscription level and client subscription level. Publication level values are inherited and can be overridden by group subscription values, which in turn are inherited and can be overridden by client subscription values.

Pervasync client will match the active network with the no-sync list matching strings before starting a synchronization. If there is a match, the sync schema/folder will be excluded from synchronization. For example, if schema “schema1” has a NO_SYNC_NETWORKS value of “182.138. || 193.111.2” and the current active network card has an IP of 193.111.222.111, “schema1” will be excluded from the synchronization.

NOTE: NO_INIT_SYNC_NETWORKS only applies to initial sync of a publication while NO_SYNC_NETWORKS applies to both initial and incremental syncs.
On the Pervasync admin console’s publications and subscriptions UI, sync folders and sync schemas are displayed on the same pages. Admins have a complete view of all the available schema and folder publications. Admin could easily make sure related folders and schemas are either included or excluded together for a certain network type. At subscription time, admin again could easily make sure related folders and schema are either included or excluded together in a group or client subscription.

No-sync lists are part of the sync metadata that admin can update at any time. When a sync session detects new metadata changes, it will postpone data and file syncs and will instead sync these metadata changes to the device. The device will then use these new metadata in following syncs.

2.15 Supporting Dynamic Sync Users

Sync users are normally static. That is to say, one user is associated with a specific local host, mobile or fixed. You setup the sync client once, often manually, on the local host. The sync client will then download the user's data and keep it in sync with a central server. However, there are situations where the sync users are dynamic, meaning they move from host to host. How do you handle that?

2.15.1 Use Case

Say you are a health organization with many clinics and doctors. There is a central database server containing all the data. In each clinic you have a local database server serving your local application. A doctor may work at any clinic at a given time. However, you cannot afford synchronizing all your central DB data to each local DB. When a doctor logs in to your local app on a local host, you want this doctor's data to be synchronized to that local DB so that the doctor can use your app to do his or her work. You want the same to happen when the doctor logs on to another host, or another doctor logs on to this host.

2.15.2 Solution

The solution is to dynamically setup a sync client when a new user logs on to your app on a local host. The sync client will synchronize this user's data to a user-specific schema in the local DB. The same local DB could have data for multiple users in separate schemas which can be synced separately. The same user could have a sync client on multiple hosts. When a user logs on to any of the hosts, data can be fast refreshed and presented to the user through your local app.

Following are the implementation steps.

Step 1. On the Pervasync server admin console, create publications. When you define a sync schema, include a parameter in the value of "Client DB Schema", for example, use "schema1_${USER_ID}". This is to sync the schema to a user specific schema on local DB.

Step 2. On the Pervasync server admin console, create a sync client and subscriptions for each user. At this time, assign a value to the USER_ID parameter.

Step 3. On each local host, setup the local DB and your client application. Your client app should include a copy of Pervasync client zip file. Do not setup the sync client yet.

Step 4. When a user logs in to your app, your app detects that this is the first time for this user so it needs to programmatically set up a sync client for this user:

- Unzip the Pervasync client zip file to a user specific folder
- Edit file pervasync_client_oracle.ini under the "bin/oracle" folder of the user-specific sync client. You need to assign a user-specific value for parameter "pervasync.client.admin.user", e.g. pvcadmin_user1. Also assign value for "pervasync.user.name". Then invoke the pervasync_client_oracle_setup.bat script in this "bin/oracle" folder to setup the client. Refer to section 1.2.3 "Setting up the Sync Client Using the Non-GUI Configuration Scripts", for more details.

- Once the client is setup, invoke pvc.bat script under the “bin” folder to do synchronization.

Step 5. After synchronization, the user’s data will be synced to a user-specific schema. Your app can then retrieve/modify data in that schema. Repeat step 4 and 5 when another user logs on to your local app.

2.16 Adding Sync Tables from Client Side

Normally you have your schema tables created and published on the server side on Perasync Admin Console by system administrator. Users are subscribed to the schemas and get the tables synced to their devices. However, in some use cases, you may need to define the tables on the client side based on user input. Once you have the table definition, you want the table to be created on both server and client side and have them synchronized.

To facilitate this, we added the functionality of adding sync tables from the client side. You call a client side API and in the background, the table metadata will be sent to the server and the table will be created and published on the server. The table will be synchronized to the client in the next sync.

The API is a static method named “addSyncTable” in class SyncClient. See the javadoc for SyncClient.addSyncTable at <Pervasync Client Home>/doc/javadoc_for_client/ for more info.

To access the API, you need to have <Pervasync Client Home>/classes and the JDBC jar in <Pervasync Client Home>/lib on your CLASSPATH.

Following is a sample code.

```
public void addSyncTable(){
    String syncSchemaName = "schema1";
    String tableName = "employees_table_created_from_client";
    String createTableSql =
          "DROP TABLE IF EXISTS "  + tableName + ";\n"
        + "CREATE TABLE " + tableName + "(\n"
        + "    ID NUMERIC(20) PRIMARY KEY,\n"
        + "    NAME VARCHAR(72),\n"
        + "    TITLE VARCHAR(72) DEFAULT 'default title',\n"
        + "    JOIN_DATE DATETIME DEFAULT '2009-06-07 00:00:00',\n"
        + "    SALARY NUMERIC(20,2) DEFAULT 100000,\n"
        + "    WEIGHT DOUBLE,\n"
        + "    HEIGHT FLOAT DEFAULT 6.0,\n"
        + "    GENDER ENUM('M','F','UNKNOWN') DEFAULT 'UNKNOWN',\n"
        + "    BINARY_DATA VARBINARY(100) DEFAULT
           x'C9CBBBCCCEB9C8CABCCCCEB9C9CBBB',\n"
        + "    ACTIVE TINYINT(1) DEFAULT 1\n"
        + ") ";

    try{
         SyncClient.addSyncTable(
         syncSchemaName,  tableName,
         createTableSql);
         System.out.println("SyncClient.addSyncTable succeeded.");
    }catch(Throwable t){
        t.printStackTrace();
        System.out.println("SyncClient.addSyncTable failed with
          exception: " + t);
    }
}
```

2.17 Customizing the Check-In Process

In mission critical applications you often need to apply strict business rules in the client check-in process. For example, you may want to control who-can-modify-what, or you may want to trigger some operations when a certain object is modified.

The mechanism to plug in your own business logic to the client check-in process is to create a customization Java class. The Java class has to implement the pervasync.ReceivePlugin Java interface. Copy the Java class to <Pervasync Server Home>/web/pervasync/WEB-INF/class or your jar file to <Pervasync Server Home>/pervasync/WEB-INF/lib. Then in the configuration file of the Pervasync server (e.g. config/pervasync_server_mysql.conf), you set the value of parameter "pervasync.server.receive.plugin" to the fully qualified name of the Java class.

At runtime during check in, the methods of the classes will be called by the sync server.

Methods beginUpload and endUpload will be called at the beginning and ending of the check in. During check in, each schema will be received. For each schema, delete/insert/update of tables will be received one-by-one.

The receiveRow method will only receive non-LOB columns. So it followed by the LOB receive operations

Exceptions thrown by the methods will fail the sync. So if you don't want the sync to fail, catch the exceptions in your method implementations.

See javadoc of interface pervasync.ReceivePlugin for more details. Following is a sample implementation.

NOTE: While the plug-in mechanism allows you to take over the check-in process completely, you don’t have to.. Normally you would want to do some verification and notifications in the plug-in and let Pervasync do the normal check-in. Only if you let the method receiveRow() returns null, Pervasync will not do the check-in for you.

```
package pervasync.server;

import java.io.File;
import java.io.OutputStream;
import java.sql.Connection;
import java.util.List;
import org.apache.log4j.Logger;
import pervasync.ReceivePlugin;
import pervasync.config.Config;
import pervasync.object.SyncColumn;

/**
 * This is the mechanism to plug in your own business logic to the client
 * check-in process. You create a Java class that implements this
 * pervasync.ReceivePlugin interface. Copy the Java class to &lt;Pervasync
 * Server Home>/class or your jar file to &lt;Pervasync Server Home>/lib.
             Then
 * in the configuration file of the Pervasync server (e.g.
 * config/pervasync_server_mysql.conf), you set the value of parameter
 * "pervasync.server.receive.plugin" to the fully qualified name of the Java
```

```
 * class. At runtime during check in, the methods of the classes will be
             called
 * by sync server.
 * <p> Methods beginUpload and endUpload will be called at the
 * beginning and ending of the check in. During check in, each schema will be
 * received. For each schema, delete/insert/update of tables will be received
 * one-by-one.
 * <p> The receiveRow method will only receive non-LOB columns. So it
 * followed by the LOB receive operations.
 * <p> Exceptions thrown by the methods
 * will fail the sync. So if you don't want the sync to fail, catch the
 * exceptions in your method implementations.
 * <p> To implement your own conflict detection and resolution, first set
             "FORCE_CHECK_IN"
 * as the conflict resolution method. In receiveVersion() method, detect
             conflict by comparing
 * the versions. You can record the info in instance variables of your
             ReceivePlugin
 * class. Then in the receiveRow method you can do conflict resolution:
             return null to discard
 * client changes; return passed in values to force check in client changes;
             return modified values;
 * or throw an exception to fail the sync.
 *
 */
public class MyServerReceivePlugin implements ReceivePlugin {
    // instance variables

    Logger logger = null;
    String syncUserName = null;
    String syncDeviceName = null;
    Connection serverAdminDbConn = null;
    String syncSchemaName = null;
    String dbSchema = null;
    Connection serverDbConn = null;
    String tableName = null;
    SyncColumn[] columnMetaData = null;
    String dml = null;

    /**
     * Constructor
     */
    public MyServerReceivePlugin() {
        // You can use Pervasync's logger
        logger = Config.getInstance().getLogger();
    }

    /**
     * Called at the beginning of check in.
     *
     * @param syncUserName Sync user login name
     * @param syncDeviceName Sync user device name
     * @param serverAdminDbConn Database connection to the Pervasync admin
     * database. Do not commit or roll-back the connection in the methods.
     * @throws Exception
     */
    public void beginUpload(String syncUserName, String syncDeviceName,
            Connection serverAdminDbConn) throws Exception {
        logger.debug("In beginUpload, syncUserName=" + syncUserName
                + ", syncDeviceName=" + syncDeviceName);
        this.syncUserName = syncUserName;
        this.syncDeviceName = syncDeviceName;
        this.serverAdminDbConn = serverAdminDbConn;
    }

    /**
```

```
 * Called at the ending of check in.
 *
 * @throws Exception
 */
public void endUpload() throws Exception {
    logger.debug("endUpload");
}

/**
 * Called at the beginning of each schema processing
 *
 * @param syncSchemaName The logical sync schema name
 * @param dbSchema The physical schema name
 * @param serverDbConn Database connection to the schema database. Do not
 * commit or roll-back the connection in the methods.
 * @throws Exception
 */
public void beginSchema(String syncSchemaName, String dbSchema,
        Connection serverDbConn) throws Exception {
    logger.debug("beginSchema, syncSchemaName=" + syncSchemaName
            + ", dbSchema=" + dbSchema);
    this.syncSchemaName = syncSchemaName;
    this.dbSchema = dbSchema;
    this.serverDbConn = serverDbConn;
}

/**
 * Called at the ending of each schema processing
 *
 * @throws Exception
 */
public void endSchema() throws Exception {
    logger.debug("endSchema");
}

/**
 * Called at the beginning of DELETE operations of each table
 *
 * @param tableName Table name
 * @param columnMetaData Column meta data. Column data values received in
 * receiveRow will be in the same order of the columnMetaData columns.
 * SyncColumn object has public variables like columnName, typeName that
          you
 * can use.
 * @throws Exception
 */
public void beginDeleteTable(String tableName, SyncColumn[]
          columnMetaData)
        throws Exception {
    logger.debug("beginDeleteTable, tableName=" + tableName);
    this.tableName = tableName;
    this.columnMetaData = columnMetaData;
    this.dml = "DELETE";
}

/**
 * Called at the ending of DELETE operations of each table
 *
 * @throws Exception
 */
public void endDeleteTable() throws Exception {
    logger.debug("endDeleteTable");
}

/**
 * Called at the beginning of INSERT operations of each table
```

```
 *
 * @param tableName Table name
 * @param columnMetaData Column meta data. Column data values received in
 * receiveRow will be in the same order of the columnMetaData columns.
 * SyncColumn object has public variables like columnName, typeName that
          you
 * can use.
 * @throws Exception
 */
public void beginInsertTable(String tableName, SyncColumn[]
          columnMetaData)
        throws Exception {
    logger.debug("beginInsertTable, tableName=" + tableName);
    this.tableName = tableName;
    this.columnMetaData = columnMetaData;
    this.dml = "INSERT";
}

/**
 * Called at the ending of INSERT operations of each table
 *
 * @throws Exception
 */
public void endInsertTable() throws Exception {
    logger.debug("endInsertTable");
}

/**
 * Called at the beginning of UPDATE operations of each table
 *
 * @param tableName Table name
 * @param columnMetaData Column meta data. Column data values received in
 * receiveRow will be in the same order of the columnMetaData columns.
 * SyncColumn object has public variables like columnName, typeName that
          you
 * can use.
 * @throws Exception
 */
public void beginUpdateTable(String tableName, SyncColumn[]
          columnMetaData)
        throws Exception {
    logger.debug("beginUpdateTable, tableName=" + tableName);
    this.tableName = tableName;
    this.columnMetaData = columnMetaData;
    this.dml = "UPDATE";
}

/**
 * Called at the ending of UPDATE operations of each table
 *
 * @throws Exception
 */
public void endUpdateTable() throws Exception {
    logger.debug("endUpdateTable");
}

/**
 * Receive row versions. If clientRowVersion and serverRowVersion match,
          there is no conflict.
 * @param clientRowVersion The version of the row on client side.
 * @param serverRowVersion The version of the row on server side.
 * @param serverRowOwner The client ID which edited the current server
          row.
 * @throws Exception
 */
```

```
    public void receiveVersion(long clientRowVersion, long serverRowVersion,
            long serverRowOwner) throws Exception {
        /*logger.debug("receiveVersion");
        logger.debug("clientRowVersion=" + clientRowVersion);
        logger.debug("serverRowVersion=" + serverRowVersion);
        logger.debug("serverRowOwner=" + serverRowOwner);*/
    }

    /**
     * Receive the non-LOB column values of a row.
     *
     * @param columnValues Column data values received will be in the same
            order
     * of the columnMetaData columns.For DELETE operation, only primary key
     * columns are included. For INSERT and UPDATE, primary key columns are
     * followed by regular columns.
     * @return a new list of column values. Return the original list if you
            want
     * the original values be applied. Return null if you want this row be
     * skipped during check-in. You can also return a list with modified
            values,
     * which will be used by the check-in process.
     * @throws Exception
     */
    public List receiveRow(List columnValues) throws Exception {

//        logger.debug("receiveRow");
//        for (int i = 0; i < columnValues.size(); i++) {
//            String columnName = columnMetaData[i].columnName;
//            String typeName = columnMetaData[i].typeName;
//            Object value = columnValues.get(i);
//            logger.debug("columnName=" + columnName + ",typeName="
//                    + typeName + ", value=" + value);
//        }

        return columnValues;
    }

    /**
     * Called at the beginning of receiving a BLOB column
     *
     * @param columnName Name of the column
     * @throws Exception
     */
    public void beginBlob(String columnName) throws Exception {
        //logger.debug("beginBlob, columnName=" + columnName);
    }

    /**
     * Called at the ending of receiving a BLOB column
     *
     * @throws Exception
     */
    public void endBlob() throws Exception {
        //logger.debug("endBlob");
    }

    /**
     * Called at the beginning of receiving a CLOB column
     *
     * @param columnName Name of the column
     * @throws Exception
     */
    public void beginClob(String columnName) throws Exception {
        //logger.debug("beginClob, columnName=" + columnName);
    }
```

```
/**
 * Called at the ending of receiving a CLOB column
 *
 * @throws Exception
 */
public void endClob() throws Exception {
    //logger.debug("endClob");
}

/**
 * Receives a chunk of the BLOB data
 *
 * @param blobChunk a chunk of the BLOB data
 * @throws Exception
 */
public void receiveBlobChunk(byte[] blobChunk) throws Exception {
    //logger.debug("receiveBlobChunk, blobChunk.length=" +
           blobChunk.length);
}

/**
 * Receives a chunk of the CLOB data
 *
 * @param blobChunk a chunk of the CLOB data
 * @throws Exception
 */
public void receiveClobChunk(char[] clobChunk) throws Exception {
    //logger.debug("receiveBlobChunk, clobChunk.length=" +
           clobChunk.length);
}

/**
 * Called at the beginning of Big Data Lob uploading
 *
 * @param syncSchemaName The logical sync schema name
 * @param dbSchema The physical schema name
 * @param tableName The physical table name
 * @param columnName The table column name
 * @param primaryKey String array of primary key values
 * @param serverDbConn Database connection to the schema database. Do not
 * commit or roll-back the connection in the methods.
 * @return Return null to let Pervasync process the uploaded LOB. Return
           an
 * OutputStream for Pervasync to write the LOB to. To ignore the LOB,
           return
 * an OutputStream that discards writes, e.g. new
 * FileOutputStream("/dev/null") for Linux or new FileOutputStream("NUL")
 * for Windows.
 * @throws Exception
 */
public OutputStream beginReceiveBigDataLob(String syncSchemaName, String
           dbSchema,
        String tableName, String columnName, String[] primaryKey,
        Connection serverDbConn) throws Exception {
    logger.debug("beginReceiveBigDataLob, syncSchemaName=" +
           syncSchemaName
             + ", dbSchema=" + dbSchema + ", tableName=" + tableName + ",
           columnName=" + columnName);
    return null;
}

/**
 * Called at the end of Big Data Lob uploading
 */
public void endReceiveBigDataLob() throws Exception {
```

```
        logger.debug("endReceiveBigDataLob");
    }

    /**
     * Called at the beginning of Big File Lob uploading
     *
     * @param syncFolderName The logical sync folder name
     * @param serverFolder The physical server folder
     * @param file The file
     * @return Return null to let Pervasync process the uploaded LOB. Return
               an
     * OutputStream for Pervasync to write the LOB to. To ignore the LOB,
               return
     * an OutputStream that discards writes, e.g. new
     * FileOutputStream("/dev/null") for Linux or new FileOutputStream("NUL")
     * for Windows.
     * @throws Exception
     */
    public OutputStream beginReceiveBigFileLob(String syncFolderName,
            File serverFolder, File file) throws Exception {
        logger.debug("beginReceiveBigFileLob, syncFolderName=" +
               syncFolderName
                 + ", serverFolder=" + serverFolder.getPath() + ", file=" +
               file.getPath() );
        return null;
    }

    /**
     * Called at the end of Big File Lob uploading
     */
    public void endReceiveBigFileLob() throws Exception {
        logger.debug("endReceiveBigFileLob");
    }
}
```

3 The Demo Application

There is a server piece and a client piece to the demo application. They are in the **demo** directory of the sync server home and sync client home respectively. In the **demo** directory you will find a java file, an ini file and two shell script files with suffix bat and sh. The java file reads the ini file for site-specific configurations. To facilitate the compilation and execution of the java file, we created the two shell script files, which will take care of things like setting the CLASSPATH. However, you still need to include the JDBC driver or Java connector jar in CLASSPATH first. The driver (or connector) jars shipped with Pervasync are located at <Pervasync Home>/lib.

Tasks except for **compile** require the parameters in the ini file to be set to appropriate values.

NOTE: On both server and client, there are a demo app for Oracle database and a demo app for MySQL database. In the following we use Oracle database as an example.

3.1 The Application Scenario

This application is for a virtual company to synchronize a manager's device DB with the company central DB. The company has multiple departments and each department has several managers. The central DB schema contains the following tables: EXECUTIVES, DEPARTMENTS, MANAGERS, EMPLOYEES and TASKS. The device DB schemas are

expected to have the same tables and data except for EMPLOYEES and TASKS tables, which would contain a subset of data, only belong to the manager's department.

3.2 The Server Piece of the Application

On Windows machines, open a shell window, go to **C:\pervasync_server-9.0.3\demo** and execute **server_app_oracle.bat** you will get the usage:

```
server_app_oracle.bat {compile | createschema | dropschema  | publish | unpublish | insert | delete | update}
```

On Linux/Unix machines, Open a shell window, go to **/pervasync_server-9.0.3/demo** and execute **server_app_oracle.sh** you will get the usage:

```
server_app_oracle.sh {compile | createschema | dropschema  | publish | unpublish | insert | delete | update}
```

You can see that the shell scripts take one command-line argument, each of which would carry out a sub-task.

3.2.1 The parameter File: server_app_oracle.ini

Make sure the parameter file contains complete and correct information before you execute any of the sub-tasks.

3.2.2 Compile the Application

Note: To compile the application, you need to have Java Development Kit (JDK) installed. Make sure the "bin" folder of JDK is included in the value of environment variable "Path". Also, you need to include the JDBC driver or Java connector jar in CLASSPATH. The driver (or connector) jars shipped with Pervasync are located at <Pervasync Home>/lib.

To compile ServerAppOradb.java to ServerAppOradb.class, execute the **compile** sub-command.

Windows:

```
server_app_oracle.bat compile
```

Linux/Unix bash:

```
server_app_oracle.sh compile
```

3.2.3 Create and Drop the Application Schema

The **createschema** sub-command is used to create the server side application schema. **Dropschema** will do the opposite.

Windows:

```
server_app_oracle.bat createschema
server_app_oracle.bat dropschema
```

Linux/Unix bash:

```
server_app_oracle.sh createschema
server_app_oracle.sh dropschema
```

3.2.4 Publish and Unpublish

The **publish** sub-command is used to define the sync objects so that they can be synchronized to device DBs. **Unpublish** would do the opposite.

Windows:

```
server_app_oracle.bat publish
server_app_oracle.bat unpublish
```

Linux/Unix bash:

```
server_app_oracle.sh publish
server_app_oracle.sh unpublish
```

publish invokes the method **publish()** in **ServerAppOradb.java**. The sync server API is in a class named **pervasync.server.SyncServerAdmin**. Refer to the javadoc for the usages of the API. We will describe here briefly how we use it in the demo app. First we create an object out of **SyncServerAdmin**:

```
SyncServerAdmin syncServerAdmin = SyncServerAdmin.getInstance();
```

To publish the application schema and tables for synchronization, we call the **addSchema()** and **addTable()** methods of **SyncServerAdmin**. To define a sync table, the API takes a primary key query as an argument to select all or a sub-set of table records for synchronization. The primary key query could have parameters. For example, the query for table **TASKS** has a parameter named **DEPT_ID**. You need to supply values for the parameters at subscription time.

```
// add sync schema
syncServerAdmin.addSyncSchema( "schema1",
syncServerDbAppSchema, syncClientDbAppSchema, null);
// add sync tables
syncServerAdmin.addSyncTable("schema1", "DEPARTMENTS", 1, null, "SELECT ID
               FROM " + syncServerDbAppSchema + ".DEPARTMENTS");
…
syncServerAdmin.addSyncTable("schema1", "TASKS", 3, null, "SELECT EMP_ID,ID
               FROM " + syncServerDbAppSchema + ".TASKS WHERE " + "EMP_ID IN
               (SELECT ID FROM " + syncServerDbAppSchema + ".EMPLOYEES WHERE
               DEPT_ID=:DEPT_ID)");
```

A user can then subscribe to the published sync schema so that the user's device DB can have a synchronized DB schema.

Now let's create a user. We have a flexible mechanism for customers to plug in their own user management system. Refer to Javadoc for sync.UserManager and the sync server configuration sections for how to plug in your user manager. In the demo, we use the built-in user manager: sync.server.oracle.DefaultUserManager to create users:

```
defaultUserManager = new DefaultUserManager();
defaultUserManager.createUser(syncUserName, syncUserPassword);
```

Each user could have multiple device DBs. Here we add one:

```
syncServerAdmin.addSyncClient(syncUserName, "DEFAULT");
```

The subscription is an association of a sync schema with a user device DB. As mentioned earlier, you need to supply values for the primary key query parameters at subscription time:

```
HashMap paramNameValMap = new HashMap();
paramNameValMap.put("DEPT_ID", iDept + "");
syncServerAdmin.addSubscription("schema1",
syncUserName, "DEFAULT", paramNameValMap);
```

After **publish**, you are ready to initiate synchronization from the client side.

To un-subscribe and unpublish, you call the **remove** methods.

3.2.5 DML Operations on the Server App Schema

The insert, delete and update tasks can be used to do DML changes to the central app schema. You can use this to test server to client incremental sync.

3.3 The Client Piece of the Application

On Windows machines, open a shell window, go to **C:\pervasync_client-9.0.3\demo** and execute **client_app_oracle.bat** you will get the usage:

```
client_app_oracle.bat {compile | dosync | insert | delete | update}
```

On Linux/Unix machines, Open a shell window, go to **/pervasync_client-9.0.3/demo** and execute **client_app_oracle.sh** you will get the usage:

```
client _app_oracle.sh {compile | dosync | insert | delete | update}
```

You can see that the shell scripts take one command-line argument, each of which would carry out a sub-task.

3.3.1 The parameter File client_app_oracle.ini

Make sure the parameter file contains complete and correct information before you execute any of the sub-tasks.

3.3.2 Compile the Application

To compile ClientAppOradb.java to ClientAppOradb.class, execute the **compile** sub-command.

Windows:

```
client_app_oracle.bat compile
```

Linux/Unix bash:

```
client_app_oracle.sh compile
```

3.3.3 Synchronize with Server

The **dosync** sub-command is used to invoke a sync with the sync server.

Windows:

```
server_app_oracle.bat dosync
```

Linux/Unix bash:

```
server_app_oracle.sh dosync
```

sync would create a **ClientSyncSession** object and invoke its sync() method. Following is what is done in ClientAppOracle.java:

```
ClientSyncSession syncSession = new ClientSyncSession(null);
syncSession.sync();
```

3.3.4 DML Operations on the Device App Schema

The insert, delete and update tasks can be used to do DML changes to the device app schema. You can use this to test client to server incremental sync.

4 Trouble Shooting

4.1 Out of Memory Errors

When you sync large amount of data, you may encounter some memory exceptions, for example on Pervasync Java SE server and client,

```
java.lang.OutOfMemoryError: Java heap space
```

and on Android device,

Out of memory on a 4088016-byte allocation

Fortunately there are ways to resolve these issues by adjusting memory settings.

4.1.1 Increasing Android Sync Client Heap Memory Size

If you sync large BLOBs you may have out-of-memory issues since SQLite does not support streaming of BLOB data. You can add the "`android:largeHeap="true"`" option to the application tag of your Android app manifest file. This will increase the heap memory limit on Android version 3.0 and newer.

```
<manifest package="com.mycorp.myapp" android:versionCode="1"
        android:versionName="1.0"
        xmlns:android="http://schemas.android.com/apk/res/android">
        <application android:icon="@drawable/icon"
                android:label="My Android App" android:largeHeap="true">
```

4.1.2 Adjusting Sync Client Heap Memory Size

If the exception was thrown by sync client, you can edit the pvc.sh/pvc.bat script file to adjust the maximum heap memory size, i.e. change the value of option –Xmx.

```
Start javaw -Xmx512m -classpath %myclasspath% pervasync.client.gui.SyncClient
```

4.1.3 Adjusting Sync Server Heap Memory Size

If the exception was thrown by the server, try adjusting the server heap space size. Consult your servlet container or application server documentation for how to adjust the maximum heap space size. For example, for Tomcat you can put the following at the beginning of <Tomcat Home>/bin/catalina.bat on Windows:

```
set JAVA_OPTS=%JAVA_OPTS% -Xmx1024m
```

or for Linux put the following at the beginning of <Tomcat Home>/bin/catalina.sh after `#!/bin/sh`,

```
JAVA_OPTS="$JAVA_OPTS -Xmx1024m "
```

If you install Tomcat as a Windows service, these settings are kept in the Windows Registry. If you have the start menu items, the easiest way to change them is to go to
Start -> Programs -> Apache Tomcat -> Tomcat Configuration -> Java (tab).
There are fields for your initial and max heap sizes.

Setting maximum heap memory (Xmx) too small may cause Pervasync server to fail with `OutOfMemoryError: Java heap space`. Setting it too big may cause other native programs, including MySQL server and Tomcat itself(!) to crash with `OutOfMemoryError`.

If you still get memory exceptions on server, locate the config file, e.g. pervasync_client_mysql.conf, under <Pervasync Client Home>/config and try reducing the max message size from 10MB to, say 2 MB:

pervasync.max.message.size=2000000

4.2 MySQL Innodb Lock Wait Timeout Error

You may occasionally see the following error:

java.sql.SQLException: Lock wait timeout exceeded; try restarting transaction

When there is more than one process trying to access the same resource exclusively, one has to wait for another. If the waiting time is too long, the process may throw this exception and abort the waiting. The process just needs to re-try at a later time.

MySQL has a default transaction isolation level of “REPEATABLE-READ”. REPEATABLE-READ has a higher level of locking as compared to READ-COMMITTED, which is Oracle default.

If you see a lot of “Lock wait timeout” errors, try changing the transaction isolation to “READ-COMMITTED”. Also try increasing the lock wait timeout threshold. In my.ini or my.conf, under section [mysqld], add or edit parameter innodb_lock_wait_timeout and transaction-isolation.

```
 [mysqld]
# How long an InnoDB transaction should wait for a lock to be granted
# before being rolled back. Default 50 seconds.
Innodb_lock_wait_timeout=600

# MySQL has a default transaction isolation level of “REPEATABLE-READ”.
# REPEATABLE-READ has a higher level of locking as compared to
# READ-COMMITTED, which is Oracle default.
transaction-isolation=READ-COMMITTED
```

Instead of manually editing the MySQL configuration file, you can also use MySQL Workbench->Server Admin->Options File to find and modify the parameters.

Restart MySQL DB and Tomcat for the new values to take effect.

Another thing to consider is to speed up the sync engine. Sync engine takes more time to process COMPLEX sync tables than SIMPLE sync tables. Try converting the COMPLEX sync tables to SIMPLE ones.

4.3 Sync Didn't Bring Down Newly Added/Updated Records

The scenario:

Update the server database (add new record). Immediately (within a few seconds), do a SYNC operation from the client. Look at the client database – the new record is not there. Wait a little bit and do a SYNC operation again. Now the new record is in the client database.

The explanation:

This is normal. There is a sync engine running on the server to periodically take snapshots of the DB. When a client syncs with a server, data is retrieved using the latest snapshot. The default sync engine run interval is 30 seconds (you can check and/or change this value on web admin console), so it is possible that during sync, a client is using a 30-second old snapshot that does not include changes in the last 30 seconds.

The sync engine and snapshot mechanism is the basis of the consistency, performance and scalability of the sync system. The lag normally is not an issue. Sync clients are most times offline so the client data is hours or days old depending on the sync interval. So it doesn't matter whether it's 30-second older.

Also the sync engine interval can be set smaller to achieve a near real-time result.

5 Tutorials

5.1 Running Pervasync Client on Windows as a Scheduled Task

Pervasync client GUI has a tab named "Schedule" , where you can schedule sync jobs to run at specific times and/or intervals. The job schedules are persisted and if you do not hit the "Cancel" button, the jobs would resume after you restart the client.

The problem is, you still need to start the sync client in order for the synchronizations to happen. This section describes how to automatically start the synchronizations after Windows reboots or user logs in.

First create a simple Windows batch file and name it, say, "auto_sync.bat":

```
cd C:\pervasync_client-9.0.3\bin
pvc.bat sync
```

If you run it one time, It will invoke the "sync" sub-command of "pvc.bat" to start a single sync session. Now you just need to use Windows Task Scheduler to schedule running "auto_sync.bat" periodically and/or on fixed times.

Normally you would want the sync to run in background, without showing the shell window. You have at least two options. One is to run "auto_sync.bat" as "SYSTEM" user and check `"Run whether the user is logged on or not"`. See

http://stackoverflow.com/questions/6568736/how-do-i-set-a-windows-scheduled-task-to-run-in-the-background

for more details.

The other option is to invoke “auto_sync.bat” in a “vbs” file. For example put the following in “auto_sync.vbs” and let the task scheduler run the “auto_sync.vbs” file.

```
Dim WinScriptHost
Set WinScriptHost = CreateObject("WScript.Shell")
WinScriptHost.Run Chr(34) &
"C:\pervasync_client-9.0.3\bin\auto_sync.bat" & Chr(34), 0
Set WinScriptHost = Nothing
```

See

http://serverfault.com/questions/9038/run-a-bat-file-in-a-scheduled-task-without-a-window

for more details.

NOTE: When sync sessions are running in the background, you can check the log files in <Pervasync Client Home>/log to monitor the progress.

5.2 Pervasync in the Cloud – Setting up Pervasync with Amazon EC2 & RDS

Following is a tutorial for setting up a Pervasync server in the Amazon cloud so that you can sync your mobile devices with Oracle, MySQL, PostgreSQL or SQL Server databases hosted in Amazon cloud.

5.2.1 Preparing the AWS Env

1. Create an AWS account at http://aws.amazon.com/ec2/

2. Create a Beanstalk env to have a running Tomcat instance and a RDS database instance: http://aws.amazon.com/elasticbeanstalk/.

Setup the DB security group so that the DB instance can be accessed from the EC2 instance where Tomcat is hosted.

3. Create your app database, schema and tables. For MS SQL Server, connect to the DB server and create a database for your application’s DB schemas and tables. The DB name will be used as value of pervasync.server.db.database.name during Pervasync setup. For Oracle you supply the DB name during DB instance creation. The DB name will be the Oracle SID to be used in DB JDBC URL. For MySQL the RDS DB name is really the schema name.

4. For MySQL, create a DB Parameter Group and set the value of parameterlog_bin_trust_function_creators to 1. Modify the MySQL instance and set its parameter group to the one with log_bin_trust_function_creators set to 1. Reboot the DB instance.

Alternately to Beanstalk, you could also directly create an EC2 instance and RDS instance. To install Java & Tomcat on your EC2 instance, see

http://www.excelsior-usa.com/articles/tomcat-amazon-ec2-basic.html

5.2.2 Installing Pervasync in the AWS Env

Note: You need to substitute the host name, password etc. with yours in the following.

1. Download pervasync_server-9.0.3.zip from

   https://www.dropbox.com/s/4uwhvo99j7ra2xc/pervasync_server-9.0.3.zip

2. Copy the zip file to your Amazon EC2 instance home folder:

```
scp -i keypair1.pem pervasync_server-9.0.3.zip
              ec2-user@ec2-54-214-122-102.us-west-2.compute.amazonaws.com:/ho
              me/ec2-user
```

3. Log on to your Amazon EC2 instance

```
ssh -i keypair1.pem
              ec2-user@ec2-54-214-122-102.us-west-2.compute.amazonaws.com
```

4. From now on act as root user. Un-package the zip in /usr/share/:

```
sudo su root
cd /usr/share/
unzip /home/ec2-user/pervasync_server-9.0.3.zip
```

5. Go to non-GUI install folder for your DB type, for example for SQL Server:

```
cd pervasync_server-9.0.3/bin/mssql/
```

Edit pervasync_server_mssql.ini to supply your DB connection info. Pay special attention to the DB endpoint and system user.

```
#============================================================================
#==                 Pervasync server for MS SQL Server                     ==
#==                 ini file for setup/uninstall                           ==
#==                                                                        ==
#==   Property values are needed and only needed when you run              ==
#==   the setup/uninstall scripts. After you are done with                 ==
#==   setup/uninstall, you can erase sensitive info from this file.        ==
#============================================================================

# The JDBC URL to the Pervasync server repository database.
# Replace "localhost" with the DB host name or IP if DB is not on the same
              host
# For AWS RDS, replace "localhost" with the DB Endpoint

pervasync.server.db.url=jdbc:sqlserver://db1.cinuuz0ecpzz.us-west-2.rds.amazo
              naws.com:1433

# The database name. Replace "master" with the database name you created for
              your app

pervasync.server.db.database.name=myappdb

# Name and password of a DB user with root privileges.
# This user/account should be pre-existent.
# For AWS RDS, use the master user

pervasync.server.db.system.user=master
```

```
pervasync.server.db.system.password=welcome1234!

# PervaSync server admin user name and password. You would need to use this
# user/password pair to login to the web-based Pervasync admin console.
# If not already exist, a database user/schema with this name will be created
# in Pervasync server DB repository at setup time. At un-install time, the
              user/schema
# will be dropped.

pervasync.server.admin.user=pvsadmin
pervasync.server.admin.password=welcome1234!
```

6. Run the setup

```
./pervasync_server_mssql_setup.sh
```

7. Install Pervasync server web app to Tomcat

```
chown -R tomcat:tomcat /usr/share/pervasync_server-9.0.3
cp /usr/share/pervasync_server-9.0.3/config/Catalina/localhost/pervasync.xml
              /usr/share/tomcat7/conf/Catalina/localhost/
vi /usr/share/tomcat7/conf/Catalina/localhost/pervasync.xml
```

Make sure the “docBase” value to point to Pervasync server home:

```
<?xml version="1.0" encoding="UTF-8"?>
<Context docBase="/usr/share/pervasync_server-9.0.3/web/pervasync/"
              path="/pervasync">
  <Logger className="org.apache.catalina.logger.FileLogger"
              prefix="pervasync." suffix=".log" timestamp="true"/>
</Context>
```

8. Log on to Pervasync web admin console

http://default-environment-ddffdffd.elasticbeanstalk.com/pervasync

If not working, check Tomcat log:

```
vi /usr/share/tomcat7/logs/catalina.out
```

Check Pervasync log:

```
vi /usr/share/pervasync_server-9.0.3/log/pervasync_server_mssql.log
```

NOTE: For PostgreSQL, before you publish a schema for synchronization, you may need to grant the role that owns the schema to pvsadmin. This is because pvsadmin is not superuser on RDS instance. However, note that pvsadmin has already been granted the role of the master user during setup. Therefore no re-grant is needed if the schema owner is the master user.

6 Database Synchronization Under the Hood

6.1 Overview

Founded in 2008 in the Silicon Valley of USA, Pervasync specializes in database synchronization for Oracle, MySQL, PostgreSQL, SQLite and Microsoft SQL Server

databases. The name “Pervasync” was derived from the words “Pervasive” and “Synchronization” indicating that our products enable you keep your smaller databases on pervasively available devices in sync with your central databases. Based on years of research and development, we created the innovative sync framework that has brought pleasant surprises to enterprise database users who had been treading water with their previous third party or homemade sync solutions.

The signature feature of the framework is that it allows you to sub-set the central server data using free form SQL queries with parameters. Each user-device gets shared and private data that are synchronized with central databases bi-directionally.

One current trend in computing is the proliferation of smart devices. As the devices become more and more capable and pervasive, it is getting both feasible and necessary to run a database system on the devices for the management of the large amount of data that could be stored locally on devices. The device databases would need to exchange data with central databases for data distribution, data collection and data backup purposes. The exchange of data between the local and central databases is called database synchronization.

Another trend in computing is the globalization of economy in which an enterprise’s geographically distributed employees, partners and customers access central servers via the Internet. All is good until there is a poor Internet connection or a network outage. More and more businesses are adding local servers to resolve this issue. Here again, the databases of the local servers and those of the central servers need to be synchronized in order to have the best of both worlds: fast, reliable access to the information and consolidation of the information.

Pervasync software provides a data synchronization framework for synchronizing distributed local databases with a central database. It supports SQLite, MySQL, PostgreSQL, Microsoft SQL Server and Oracle databases (including Express, Standard and Enterprise editions). Typically you would run MySQL, PostgreSQL, Microsoft SQL Server or Oracle Enterprise Edition on the server side (in your data center or in the Cloud), while on the client side (Linux or Windows desktops, laptops, tablets, and smartphones), you may choose SQLite, MySQL, PostgreSQL, Microsoft SQL Express, Oracle Express Edition or Oracle Standard Edition.

The Pervasync data synchronization server and client for SQLite, MySQL, PostgreSQL, Microsoft SQL Server and Oracle are written in pure Java. They can run on any platforms that support Java SE. This includes Windows, Linux and Mac OS X. The sync clients for Android and iOS devices are written in Java. Sync clients could run in standalone mode or be embedded into your applications on devices.

Pervasync server comes with a web based admin console for administrators and developers to create publications and subscriptions of the sync objects. On the device side there is a sync client GUI application for end users to start or schedule synchronization sessions. The sync system can be set up quickly and you can get your central DB synchronized with your local databases without writing a single line of code. Nevertheless, we also provide client and server Java API that can perform the same tasks in case you need to embed the sync functionality seamlessly into your applications.

From ground up, the framework is designed to support data subsetting, so that each client can synchronize its private data as well as shared data with a central server. We introduced the concept of logical transaction for synchronization of the data subsets. Server side logical transactions are computed using the state comparisons of the data subsets. They are used to refresh client side data. Client side logical transactions are tentative and local until they are checked-in on the server. The logical transaction based synchronization scheme contrasts

with traditional physical transaction based replication schemes, which do not apply to subsets of data and cannot scale due to divergences.

Last but not least, one of the differentiators of Pervasync compared with other sync solutions are performance and scalability. The number of sync clients and amount of data in your organization tend to grow rapidly over time. So it is important to test the scalability of the sync solutions before you commit to one of them. You will find that Pervasync stands out in your performance and scalability tests.

6.2 Pervasive Computing and Data Synchronization

Whatever you call it, Pervasive Computing, Ubiquitous Computing, Mobile and Embedded Computing, Internet of Things or Edge Computing, there is a trend that more and more smart and connected processing devices are pervasively embedded in the environment we live and serve us anywhere, anytime [1,2].

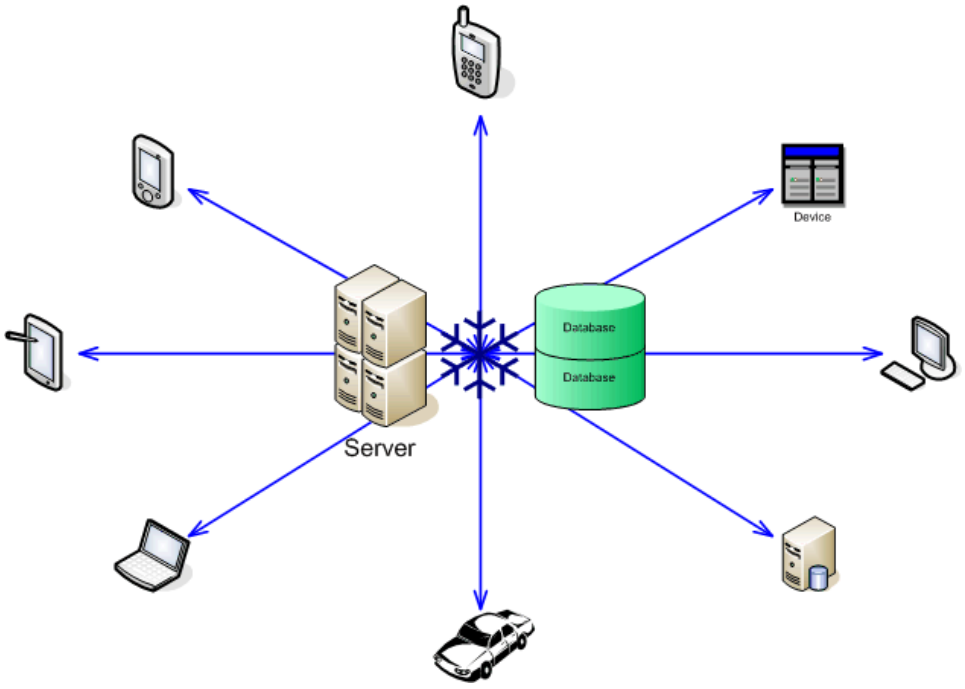


Figure 1 Server Centric Pervasive Computing

The computing devices may be able to perform tasks on their own, or by interacting with human and other devices. However, more often than not, central servers are needed to manage and serve these devices. We call this Server Centric Pervasive Computing as illustrated in Fig. 1. The devices are connected with the servers via wired or wireless networks. Applications on the devices provide an interface to end users while all data is stored on a central server. Applications on the devices communicate with the central server to do data collection, data distribution and device management.

There are basically two models for how the servers serve the devices. One is an online model, where a device has to keep a connection with a server to be able to perform any functions. The other is an offline model, where the device has its own cache of data and is

able to function even when it's disconnected from the server. The device only needs to connect to the server when it needs to synchronize its cache with the server database.

Intrinsically the offline model is a more scalable approach since it offloads a big chunk of the server work to the devices, which are getting more and more capable. The offline model also helps in reducing network congestion and improving responsiveness of devices. Still, the success of the offline model largely hinges on the efficiency and robustness of the underlying data synchronization framework.

Today, there are all kinds of data synchronization frameworks available, with or without relational databases being involved. We believe that more and more data synchronization applications will be standardized on database synchronization frameworks; just like information management applications are now mostly based on database systems.

Globalization represents another aspect of Pervasive Computing. Typically a business' geographically distributed employees, partners and customers access central servers through the Internet. All is fine until there is a poor Internet connection or a network outage. More and more enterprises are adding local servers to resolve this issue. Here again, the databases of the local servers and those of the central servers need to be synchronized in order to have the best of both worlds: fast, reliable access of the information and consolidation of the information.

Pervasync database synchronization framework provides a data synchronization infrastructure for your pervasive business, supporting popular databases such as Oracle, MySQL, PostgreSQL, Microsoft SQL Server and SQLite. Traditional replication techniques have failed to address the large scale and personalized characteristics of Pervasive Computing. Pervasync's innovative logical transaction based approach guarantees the convergence and stability of the system. From ground up, the framework is designed to support data subsetting, so that each device can synchronize its private data as well as the shared data.

6.3 Data Subsetting in Database Synchronization

In a database synchronization system, typically you have a big central database on the server side and a large number of small databases each residing on a device. The central database contains data for all the devices while each device's local database only contains the device's private data and some shared data.

In Fig. 2 we illustrate the subsetting of the data of a big central DB table. A table has columns and rows. One or more columns form a primary key to identify a row. To define the data subset, you can first select columns of the table. Only primary key columns and selected columns will be synchronized to the devices. This is called vertical subsetting. All devices share the same vertical subsetting, meaning they all get the same set of columns.

In the more interesting horizontal data subsetting, you select a subset of the rows by specifying a set of primary key values. In the figure, the rows identified by primary keys <PK 1> and <PK 2> are chosen to be distributed to all devices. These rows contain shared data. The rows identified by primary keys <PK 3> and <PK 4> are chosen to be distributed to device # 1 only. These rows contain private data for device #1. Similarly, the rows identified by primary keys <PK 5> and <PK 6> are synchronized to device # 2.

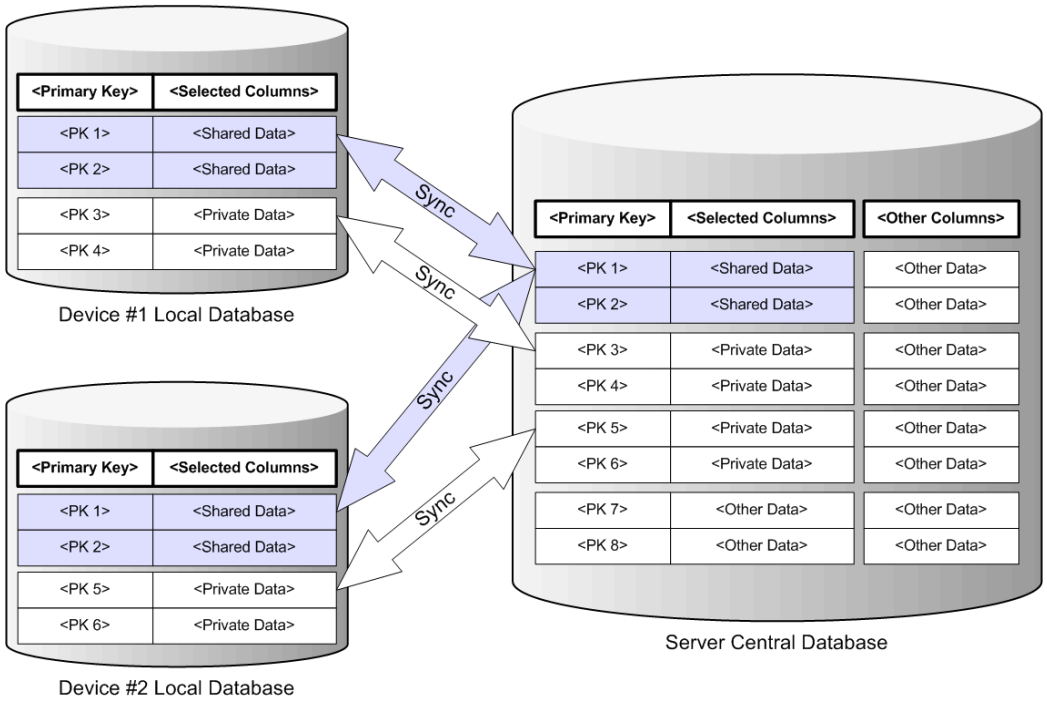


Figure 2 Data Subsetting in Database Synchronization

The horizontal subsetting can be carried out through a SQL query that returns the primary key values of the row sub-set. When a table is published, the SQL query is defined with embedded parameters (optional). During subscription of a client to the sync schema containing the table, concrete values for the parameters are specified. Different values cause the SQL query to return a different subset of data to the clients.

6.4 Replication versus Synchronization

Database synchronization is closely related to database replication. In fact, sometimes people use the terms interchangeably. However, there are big differences between them. Understanding the differences will help you understand the different approaches used for solving replication and synchronization problems.

6.4.1 Database Synchronization Is Not Replication

Replication is mostly used in situations where identical replicas of the complete data set are maintained for high availability and performance. Replicas can often work independently as backups for each other. On the other hand, synchronization is often between a more temporal sub-set of data and a more persistent full-set of data, both of which are integral parts of a system. For instance, parts of a file could be buffered in-memory by an operating system and are “synchronized” with the file on hard disk. Another example is the synchronization of the data in a CPU cache memory with the data in the main memory. In both cases people use the term “synchronization”, not “replication”.

In Pervasive Computing, devices maintain a data cache that is a small subset of the data stored on centralized servers. Changes to the cache are temporary and should eventually be

propagated to the servers and the server should refresh the caches with up-to-date data in central databases. Clearly, this is a synchronization process, not a replication process.

6.4.2 Replication Techniques Won't Work for Synchronization

In traditional database replication schemes, physical transactions on each node are recorded and played back on all the other nodes. This technique would only work if each node has a replica of the full-set data.

There is also a stability issue with physical transaction based replications when the number of nodes goes up. Transactions on different replicas may conflict with each other. To handle this, cross system locking or complicated conflict resolution schemes are needed. In fact, they are used in eager replication and lazy replication respectively [3,5].

Eager replication synchronously updates all replicas as part of one atomic transaction. This is also called synchronous replication or pessimistic replication as it incurs global locking and waiting. This scheme is not suitable for Pervasive Computing since the locking and waiting are simply not feasible in an environment where there are lots of nodes/devices. All the devices may not even be connected to the network at the same time, let alone be locked at the same time to let the transaction go through.

In contrast to eager replication, lazy replication allows updates of any replicas without locking others. The local transactions then propagate to other replicas in the background. This scheme is also called asynchronous replication or optimistic replication since it is based on the assumption that conflicts will occur rarely. Here each host must apply transactions generated by the rest of the hosts. Conflicts must be detected and resolved. Since the transactions cannot be simply un-done at their origin nodes, usually manual or complicated ad hoc conflict resolutions are required at the destination nodes.

Gray et al. [3] showed that the traditional transactional replication has unstable behaviors as the workload scales up: a ten-fold increase in node number or data traffic gives a thousand fold increase in deadlocks or reconciliation. A system that performs well on a few nodes may become unstable as the system scales up to even a dozen nodes.

The traditional database replication schemes are clearly not suited for Pervasive Computing which involves hundreds or even thousands of nodes in one system. On the other hand, we can exploit the unique characteristics of Pervasive Computing to construct a synchronization scheme that may not be suitable for traditional replication situations but works well for Pervasive Computing.

Sync solutions usually employ a client-server model, instead of the peer-to-peer model as used in replication. Clients all communicate with the server directly. Clients do not sync with each other directly.

In synchronization, client and server usually exchange accumulated record level changes, instead of physical transactions as used in replication. Change tracking is the first thing you have to face in synchronization.

6.4.3 The Dangers of Timestamp Based Change Tracking

All sync solutions have to somehow track the changes to records/rows on the originating DB, which we call logical transactions, and apply the changes to the destination DB.

The most popular change tracking method is based on timestamps. The approach is very straightforward. You add a timestamp column to your table and update the timestamp column whenever you change a row. This can be done in your SQL statements or through triggers. By the way, deletions have to be tracked separately, e.g. using a companion table. You track the changes on server this way and then at sync time, the sync client would come in with a last sync timestamp and you select all the server changes that have a newer timestamp than the last sync timestamp.

A lot of sync solutions put the burden of change tracking on app developers and the timestamp approach is the number one recommended technique for change tracking. This is also a widely used technique when people have to implement their own sync logic. However, be aware of its pitfalls.

One obvious pitfall is system time. Timestamps are generated using system time so messing up system time would cause problems. Don't adjust system time even if it is off. Do not sync during time shifts between daylight saving time and standard time.

There is a more serious problem with this technique which could cause change loss. We know that the default isolation level for MySQL with Innodb and Oracle is "Read Committed", which means that others cannot see the changes before a transaction is committed. Let's say at time T1 you modified a record R1. The timestamp column would have a value of T1. Then before you commit, someone synced at T2 (T2 > T1). This sync could not pick up the changes to record R1 since the transaction was not committed yet and the change was invisible. At T3 you committed the transaction and at T4 (T4>T3>T2>T1) you do another sync. Guess what, you still won't get the changes to R1! It was committed but it has a timestamp T1 that is older than last sync time T2. The client will forever miss the change no matter how many times you sync.

This problem is not so well known and is very hard to work around in a production environment where you can't control when a sync and transaction would occur, and how long they would last etc.

Fortunately Pervasync has an innovatively designed sync engine that can take care of this kind of sync issues. You don't need to have any timestamp column to worry about. Just do your normal DB DML (Data Manipulation Language) operations and the Pervasync system would track the changes for you and guarantee no change loss.

6.5 Logical Transaction Based Pervasync Sync Engine

A traditional replication system is symmetric and each node contains a full-set data, while a synchronization system is asymmetric and the client nodes contain sub-sets of the data on central server. The asymmetry, together with the instability introduced with large number of nodes, calls for a different approach than propagating physical transactions among all nodes, as used in replication.

Instead of using physical transactions, database synchronization can be carried out by exchanging logical transactions between the devices and the central server. Devices do not interact with each other directly. Instead, devices only interact with the server. The logical transaction is the result of the physical transactions. At the heart of a successful sync solution is a sync engine that computes the logical transaction in a fast and scalable way.

6.5.1 Logical Transactions

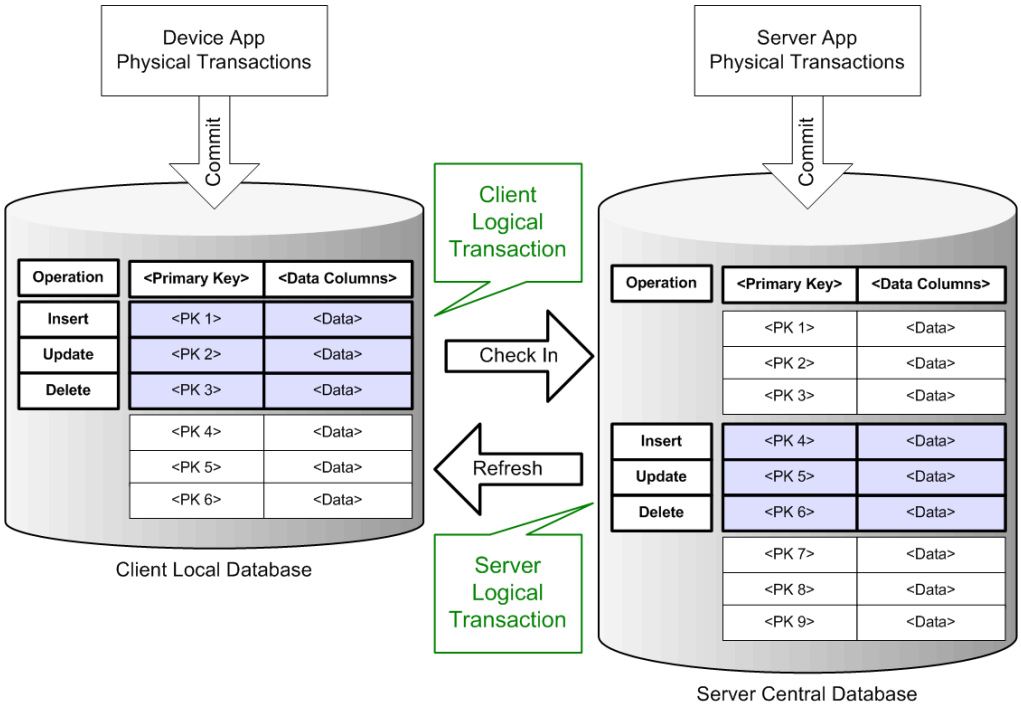


Figure 3 Synchronization Is Composed Of the Check In And Refresh Of Logical Transactions

In a server centric synchronization system, the server database is the single source of truth. Physical transactions applied on the central database are final. In contrast, device databases serve as a cache of a subset of central database data. Physical transactions applied on the device database are tentative until they are checked in to the central database. At check in time, the changes committed by all the local physical transactions form a single logical transaction, which is applied directly to only the central database, not directly to other devices. The checking in of logical transactions in the sync system is just like the committing of user physical transactions in a single DB system.

In Fig. 3 we show the synchronization process of the client and server databases. During the initialization, the row-subsetting SQL query is executed on the server and the result set is downloaded to the device. Now the client data and the corresponding server subset of data are consistent with each other. In the figure the subset includes <PK_2>, <PK_3>, <PK_5> and <PK_6>. <PK_1> and <PK_4> do not exist yet and <PK_7> through <PK_9> are outside of the subset.

Then the device application makes changes to the local database by committing physical transactions. The local changes are recorded in the form of Data Manipulation Language (DML) operations on the table rows, which include inserts, deletes and updates. At check-in time, these DML operations form a client side logical transaction, which has the following properties.

- **State based**. It is the result of one or more local physical transactions. The physical transactions bring the subset data from an old state to a new state so the logical

transaction can be seen as computed from the differences between the new and old states.

- **DML merged**. The original DML operations in the physical transaction may be merged in the logical transaction. For example, an insert followed by an update will be merged as an insert.

- **DML re-ordered**. The order of the DML operations in the logical transaction may be different from the original ones. We arrange the DML order in a way to minimize the chances of database integrity violations at check in time, e.g. referential constraints.

- **Tentative**. The logical transaction may fail to apply on the server due to conflicts or other reasons. It can be modified by new client local transactions, by conflict resolution rules on server, or by force refreshing it with server changes, so that it can be checked-in successfully and become permanent. Before checked-in, the affected rows are in a temporary (tentative) state.

NOTE: The device application needs to handle not only the success/failure of the local physical transactions, but also that of the logical transactions. For example, if a check-in fails, it should try to modify the logical transaction and retry. A good synchronization infrastructure should give application the access to the logical transaction. Application can in turn expose the logical transaction and check-in process to end-users or handle it using some pre-existing logic.

The server can also generate logical transactions to refresh device data from an old state to a new state so that the whole system will be consistent. The refreshing transactions are logical transactions too, which have the following properties.

- **State based**. It is the result of the one or more physical transactions that occurred on the server, which can be transactions committed by server side applications or transactions committed by other devices through the “check-in” process. The logical transaction is computed from the differences between the new and old states of the subsets.

- **DML merged**. The original DML operations in the physical transaction may be merged or changed in the logical transaction.

- **DML re-ordered**. The order of the DML operations in the logical transaction may be different from the original ones.

- **Preemptive**. The server side logical transaction should be applied to device database forcefully. That is to say, if there are any conflicts, the changes on device will be overwritten by server side changes.

6.5.2 Conflict Resolution during Check-In

When a device is generating changes to the local subset data, other devices or server side applications may be changing copies of the same data on the server. This could result in conflicts. To help detect conflicts, we assign version numbers to rows. If a server row still has the same version number since last sync, then we know it hasn’t been modified by others and there is no conflict.

If a conflict is detected, there are configurable resolution schemes available to ignore client transactions or ignore the conflict.

6.5.3 Computation of Server Logical Transaction in Refresh

The data exchange between the server and the device is two-way. Check-in is the uploading of changes from client to server and Refresh is the downloading of changes from server to client.

Figuring out the server side changes for a device is a complicated process. It could become a bottleneck for sync performance. We cannot simply log the transactions on the whole table as done in replication schemes. What we need is the changes to the subset of data for a device, which is defined by the subsetting SQL query. Each unique set of the query parameter values corresponds to a subset of records. Changes made to the server side tables bring a subset from an old state to a new state. We need to compute the differences between the new and old states and wrap them into a logical transaction, then send them to the device and refresh the subset data on device to the new state.

In our implementation, for each base table, we create a companion log table to keep the DML operations made to the base table. We also create a companion state table to keep the states of all the data subsets. Each subset is identified by a unique parameter-value combination, and the state table contains the logical DML operations on the data subsets.

A sync engine that runs periodically in the background does the work of updating the state table. In each run, the refresh engine would identify the changed records and the associated DML operations. It then updates the state table with this information and assigns the changed records a new state number. The algorithm is as follows.

For each subset

- **Delete**: if a record is in the state table subset but not in the current SQL query result set, mark it as “delete” in the state table.

- **Insert**: if a record is in the current SQL query result set but not in the state table subset, insert it to the state table and mark it as “insert”.

- **Update**: if a record is in all of the following three tables: the state table subset, the current SQL query result set and the base table’s new logs, mark it as “update” in state table.

Note that our algorithm produces logical insert and delete operations. For example, suppose we have a TASKS table that has a user column for task owners. We could use user column in the subsetting of the tasks table so that each user only gets the tasks he or she owns. Now, if we update a task record to change the user from user1 to user2, user1 would get a logical delete operation to his subset and user2 would get a logical insert operation to his subset. This reassignment type of operations is quite common in real-world applications and our algorithm handles it very well. Traditional replication schemes, which use original physical transactions, wouldn’t be able to handle this.

Note that data subsetting is essential to Pervasive Computing, which is based on user, location and device dependent data management. At sync time, the device brings in the state number of its subset and downloads the records that have a newer state number. For records marked with a delete operation, we would only download the primary keys.

6.6 Pervasync System Architecture

When you build an application for a server centric Pervasive Computing system, you typically would have a server piece and a client piece of the application (Fig. 4). The server application runs on the central server interacting with the data stored in the server application schema. The client application runs on the devices interacting with the data stored in the client application schema.

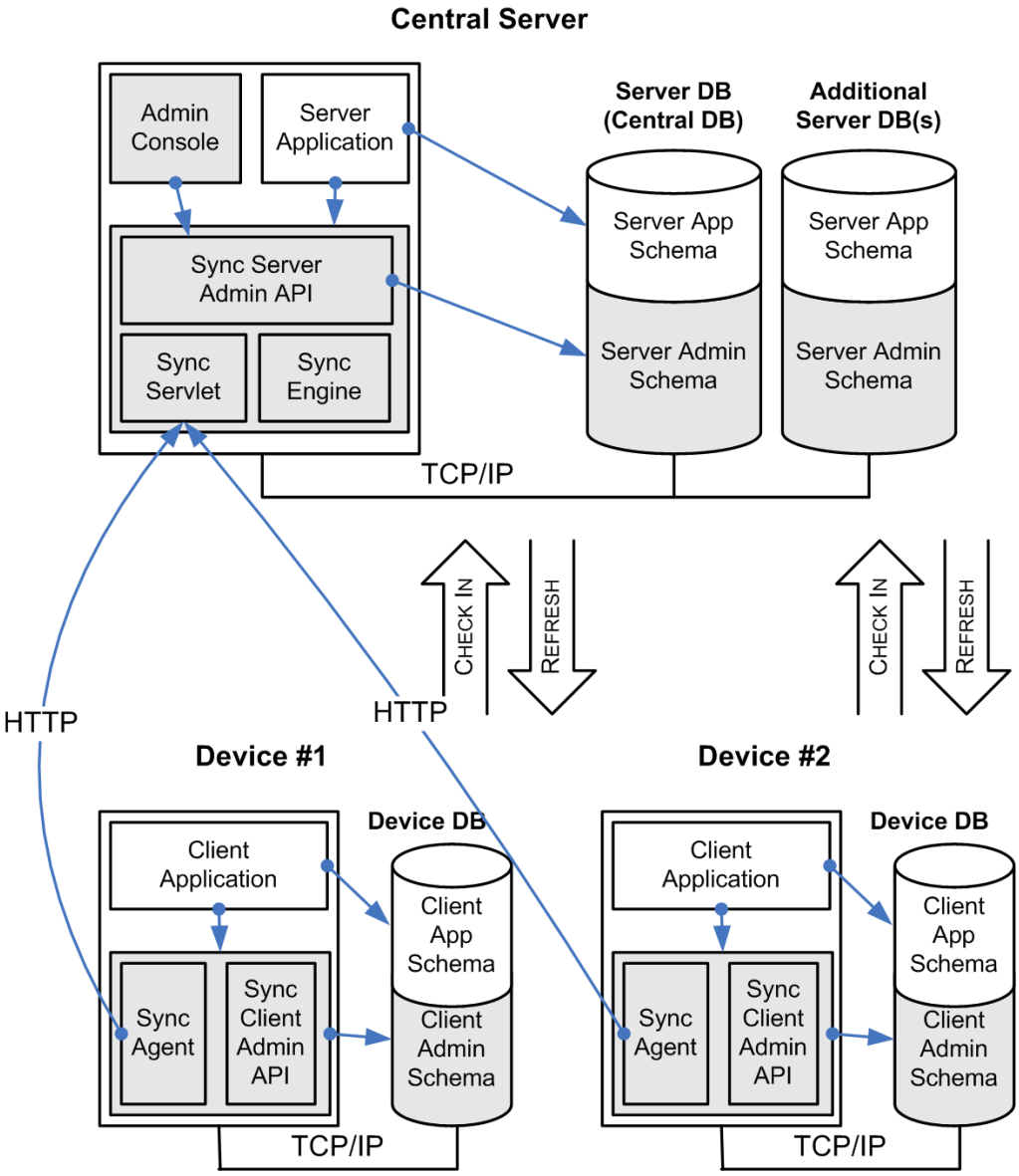


Figure 4 Pervasync Database Synchronization Framework Architecture

Your applications can rely on the Pervasync Database Synchronization Framework (shaded parts of Fig. 4) to make the client app schema and the server app schema in-sync. On the server side, the server application interacts with the sync server via the sync server admin API to define the sync behavior, e.g. data subsetting. On the client side, the client application

interacts with the sync client. Each sync client has a sync agent API and a sync client admin API.

The Pervasync synchronization infrastructure has a server piece and a client piece. The sync server connects with the server admin schemas, which contain the sync server metadata. Multiple server DBs can be supported for scaling out. There is one server admin schema on each server DB. The sync server itself also can have multiple instances to meet scalability requirements.

The sync client connects with a client admin schema, which contains sync client metadata. Client application can initiate synchronization by invoking the sync agent. The sync client admin API can be used by client application to manage the sync client metadata.

Sync clients communicate with sync server via HTTP. Logical transactions from the client DB are wrapped in the HTTP requests and sent to the server to be checked-in to server DB by the sync servlet. Check-in is the first half of the sync session. The second half of the sync session is Refresh, in which the logical transaction from server is wrapped in the HTTP response and sent to client.

In addition to the sync servlet component, which interacts directly with sync clients, the sync server also has a sync engine. Sync engine is a background process that updates data subset state tables by processing table logs so that sync servlet can quickly figure out server side changes for a client.

7 Bibliography

[1] Weiser M. The computer for the twenty-first century. In *Scientific American*, September 1991, pp. 94-104.

[2] Franklin M J. Challenges in Ubiquitous Data Management. In *Informatics - 10 Years Back. 10 Years Ahead. Lecture Notes In Computer Science 2000*, Wilhelm R (eds.), Springer-Verlag, 2001, pp. 24-33.

[3] Gray J, Helland P, O'Neil P, Shasha D. The dangers of replication and a solution. *SIGMOD Rec.,* 1996, 25(2): 173-182.

[4] Ding Z, Meng X, Wang S. A Transactional asynchronous replication scheme for mobile database systems. *J. Comput. Sci. Technol.,* 2002, 17(4): 389-396.

[5] Saito Y, Shapiro M. Optimistic replication. *ACM Comput. Surv.,* 2005, 37(1): 42-81.