The Matrex Client/Server Specification 1.1

Table of Contents

Introduction

Since Matrex was born it was clear that it had to become a client/server system, even if this feature is not present now.

You can note this from its interface: the project trees are contained in a tab called *local* or *localhost*, which makes sense only if in the future there will be other tabs with different servers addresses or names.

The fact that Matrex works with raw granularity (blocks or matrices instead of cells) makes it a natural choice to make it evolve to a client/server system: **to exchange a small amount of big packages (matrices) rather than a big amount of small packages (cell**

values) is a better use of the network and makes the applications faster and more responsive.

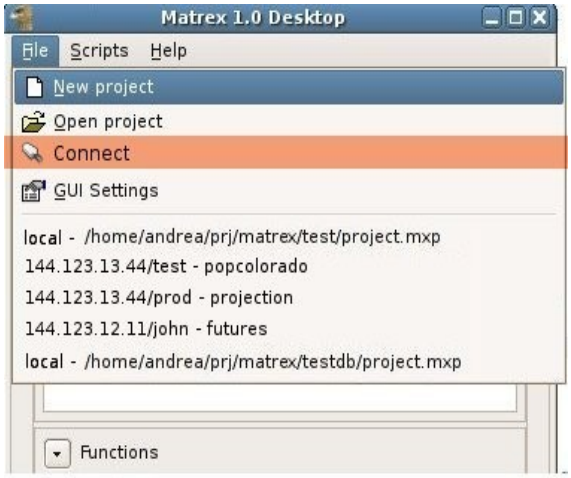
Now it is time to implement this new feature, and make it available in Matrex version 2.0.

Before we start to analyze the client/server architecture, let's give a look of how a Matrex 2.0 user will use the new feature.

A usage scenario

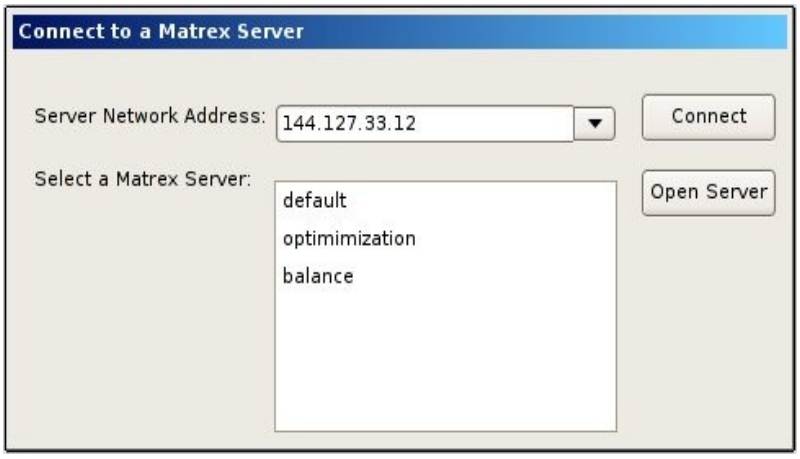
The user opens Matrex as usual. The GUI looks the same as now:

The user opens the *File* menu:



Here there is a new menu, called **Connect**.

Connect allows to connect to a Matrex server in another PC. It opens the following dialog:



The dialog has a combo box to write the **address of the PC** to which you want to connect or select this address from a list of PCs to which you connected in previous sessions.

Once the address is selected press the *Connect* button.

Since each PC can host several Matrex server, you get the list of **Matrex servers available at the selected address**.

Select one of them and press the *Open Server* button.

If the server has security activated, at this point you need to **log in**.

If the login has been successful, a the **machine tab** for the server is added to the main GUI:

The new tab works in the same way as the local tab, but opening the templates window in this tab you see the templates of its server and opening a project in this tab means opening a project on its server.

Then let's see what happens when we can create or open projects in the server.

New project

To create a project you click as usual on the *New project* menu.

The **New Project** dialog that appears is the usual one:

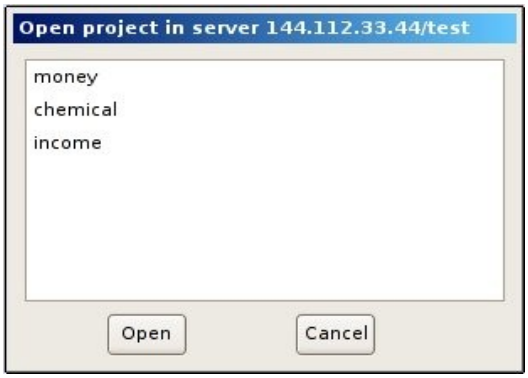
But, after you entered the name and pressed OK, the dialog to select the project base directory does not appear. This is because on a server the projects are **all saved in a specific directory**, so only the project name is needed.

A **project tab** for the new project appears and you can start to work with it. The project behaves exactly like a local one, if you don't count some performance difference because you are working remotely,

Open project

To open a project click as usual on the *Open project* menu.

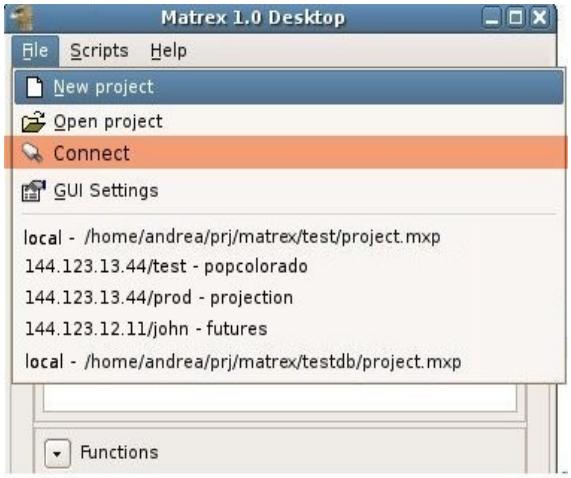
We already told that the projects on a server are all saved in a specific directory; so, instead of the dialog to select the project file in the file system, a list of the available projects in that server is displayed:



Selecting a project a pressing *Open*, the project tab is added to the server tab and you can start to work with it.

Recently used projects

We repropose here the File menu picture:



In the list of **recently used** projects there are two kinds of projects:

- **local projects**. the projects that are located on the client's PC. They are described by the word *local* followed by the name of the project.
- **remote projects**. the projects that are located on a server. They are described by the pair *server address/server name* followed by the name of the project.

Clicking on a remote project the client will:

- Connect to the project's server.
- Add the server machine tab if needed .
- Add the project tab in the server machine tab.

Motivations

Why should someone connect to a Matrex server? The reasons are:

- To **share** the projects. Two or more persons will be able to work together (in the same time or separately) with one project located in a server. Matrex guarantees consistency when working together on the same project in the same time.
- To not use your PC resources to calculate the projects functions and instead use the **resources of a server**.
- To improve the **performance** of the project functions calculation. It is possible

for several reasons that a server is faster than your PC (better CPU, java optimization for server applications).

For other purposes (publish a project on the internet, send a project to other people) other tools should be used.

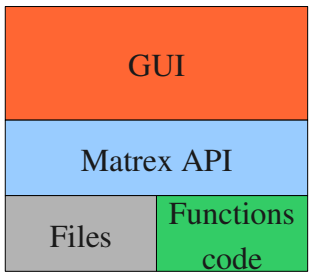
Requirements

These are the requirements that Matrex client/server must satisfy.

- **Speed**: people need to work with shared projects in the network with performances comparable with local projects.
- **Simplicity**: to install a Matrex server and to connect to it should be tasks that anyone with some minimal computer knowledge is able to do.
- **Security**: a minimal security infrastructure, but nothing that goes against the simplicity requirement. The client/server architecture is supposed to work on a a LAN, not on the internet.
- **Minimal setup file impact**: the size of the setup file must remain around the 15 MB.
- Work in a **LAN**: client and server must reside both in the same LAN. The Matrex server must not be a web service exposed in the internet. This would be against the speed requirement. Therefore firewalls and proxies issues are not handled.
- **Optional**: the fact that Matrex will be client/server does not mean that you'll need to connect to a Matrex server to work. You will still be able to use Matrex as a standalone application, in the same way and with the same performances as today.

Client/Server architecture

The following picture shows the **current structure** of a Matrex desktop application in terms of modules:



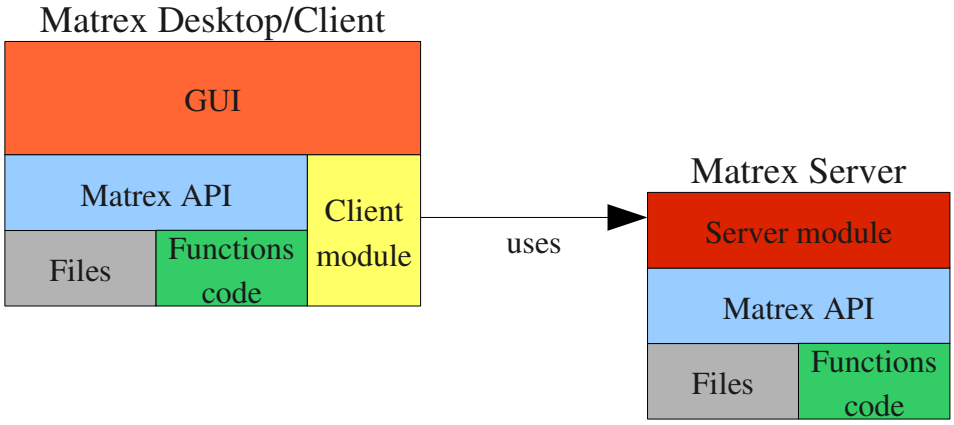
The **Matrex API** module handles all the items of Matrex (projects, machines, matrices, functions, presentations, charts, timers). It is the **engine** of Matrex (it can be used without the GUI, see the Matrex API document).

The **Functions code** (or, better said, function templates code) module contains the algorithms to calculate the functions contained in the projects.

The Matrex API module uses the **Files** module to read and write all the needed files (files describing projects, matrices, functions, presentations, charts, timers).

The **GUI** module handles **windows**, dialogs and everything that has to do with the graphical user interface. It shows and edits projects, matrices, functions, presentations, charts and timers handled by the **Matrex API** module.

With Matrex 2.0 this structure evolves: the applications are two (**Client** and **Server**), but there are only two modules more:

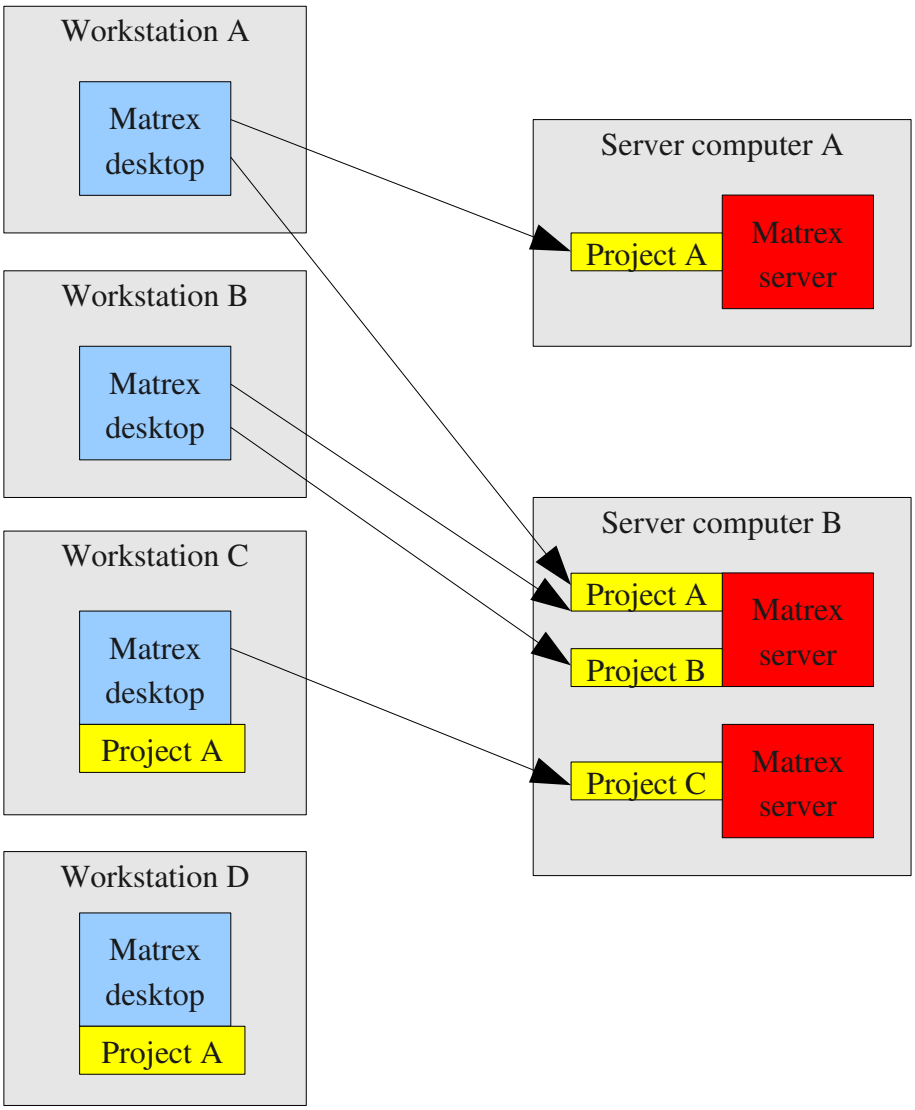


- the **Server module**, which accepts connections and publishes the projects so that they can be accessed remotely.
- the **Client module** which connects to Matrex servers and acts as a proxy, making remote projects accessible as they was local.

The other modules (Matrex API, GUI, Files, Function code) need a limited amount of changes to adapt to the new architecture. For example:

- The GUI module needs the new dialogs to connect to a server.
- The Matrex items (projects, matrices, functions...) need some abstraction to be used both locally and remotely.

The following picture shows an example of the use of Matrex desktops/clients and Matrex servers in a LAN:



Here we have examples of:

- projects used remotely by **only one client** (project A in server A, project B and C in server B).
- **shared projects** (project A in server B).
- clients that use **only remote projects** (workstations A and B)
- clients that use **only local projects** (workstation D)
- clients that use **both local and remote projects** (workstation C)

Client/Server communication

The interaction among Matrex clients and servers are of two types:

- Client's **synchronous requests** to the server. For example the client opens a project in the server, loads a matrix to view it or edit it.
- **Asynchronous messages** from the server to the client. For example the server notifies the client when the content of the project changes or when a viewed matrix changes its content, so that the client can show the changes.

Communication protocol choice

Looking at the requirement and the kind of communication between client and server it is evident that we need a communication protocol that guarantees:

- Fast objects **serialization/**deserialization.
- Fast messages **sending/**receiving.
- **Synchronous and asynchronous** communication (or at least one library for synchronous communication, one for asynchronous).
- Easy to **install**. Anyone must be able to install a Matrex server and connect to it.
- Easy to **program**.
- **Small** size library.

We don't need instead:

- Language independence. The library that implements the protocol just needs to be available in Java.
- To send messages through firewalls. The client/server connection is inside a LAN.

So, of the available protocols we discard:

- **SOAP** (or REST) because it is slower than other protocols since it is based on XML; on the other side we don't need to send messages through firewalls.
- **CORBA** because it is relatively difficult to program and we don't need language independence.
- **JMS** because it is relatively difficult to install (you must have a JMS broker) and generally the library implementing it is not small.

- **Jini** is not right for this purpose: Matrex servers are a few, in specific locations, all giving the same services (exposing projects) and don't need to synchronize their code.

The chosen protocol is **RMI**. RMI is not really the newest protocol available (it was available in Java 1.1 and did not change so much since then) but:

- it is easy to program
- it is easy to install (you just need to start the RMI registry server)
- it is included in the JRE (library size = 0)
- it is able to handle synchronous messaging and a primitive asynchronous messaging (trough callbacks).
- it is based on the Java object serialization/deserialization, which is very fast.

If I could install the servers myself I probably would choose RMI for synchronous messaging and JMS for asynchronous messaging (because it is very reliable and complete), but since the user needs to install it himself, RMI should be good enough for both purposes.

Business objects

Now that we decided the protocol, we need to model server and client to communicate through it. In other words we need to give a **structure to our client and server modules**.

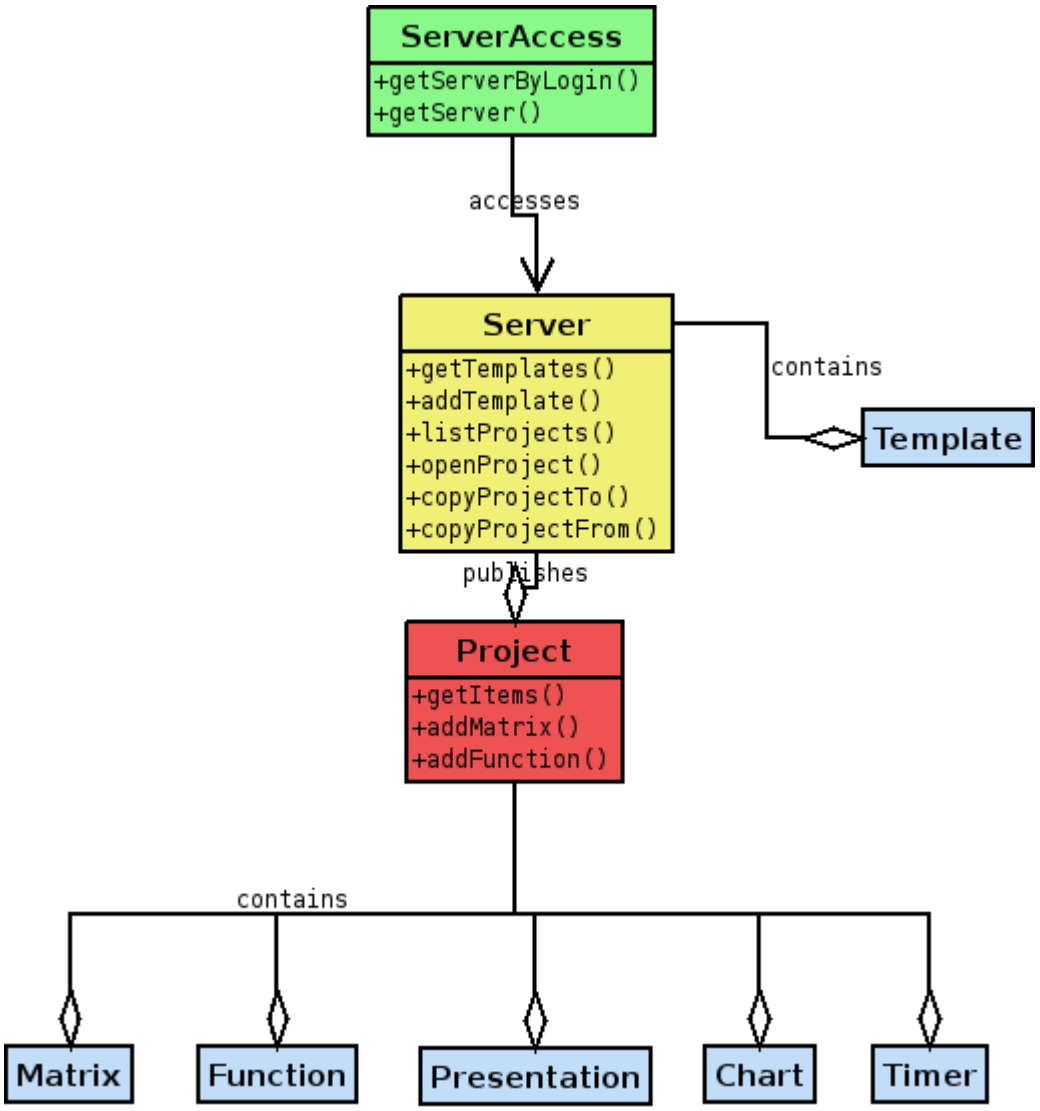
RMI, like CORBA, structures the systems in business objects (the classes that give services) and messages (the objects serialized and sent through the network).

So, let's decide:

- what are the business objects and their relationships.
- what are the messages.

As told before, Matrex has already been structured for this purpose, so the choice is not difficult. To know more about that, read the **document about the structure of Matrex**.

Let's give a look to the business objects:


ServerAccess
+getServerByLogin()
+getServer()
accesses
Server
+getTemplates()
+addTemplate()
+listProjects()
+openProject()
+copyProjectTo()
+copyProjectFrom()
contains
Template
publishes
Project
+getItems()
+addMatrix()
+addFunction()
contains
Matrix
Function
Presentation
Chart
Timer

They are very similar to the main classes we have now in the Matrex desktop (you can look at the **Matrex internal structure** document).

Clearly there must be some differences, which are given essentially by two of the business objects.

The ServerAccess business object

If the security in the server is active, the client can access the server content only after its user **logs in**.

This is guaranteed by the **ServerAccess** business object, which returns the reference to the Server business object as result of a method call.

- If the security in the server is active the client needs to call **getServerByLogin** with userid and password to obtain the reference to Server.
- If the security is not active a call to **getServer** without parameters is enough. Calling getServer when the security is active gives an error.

ServerAccess is the only business object of the server that is registered in the RMI registry. So, if you want to use the server, you need to pass through ServerAccess.

The Server business object

ServerAccess returns a reference to the **Server** business object.

Server is a specialization of **Machine**, which has a special method, called **listProjects**.

ListProjects returns the list of the **available projects** in the server (which are stored in a specific directory in the server computer).

Since we specialized Machine with Server, Machine will become an abstract class. The Machine used locally in the Matrex desktop will be another specialization of Machine called **LocalMachine**.

All the other business objects (project, template, matrix, function, presentation, chart, timer) are the same as today in terms of functionalities.

The usage scenario as client/server interaction

So, if we go back to the usage scenario, in which the client opens a project in a server, this is what happens from the point of view of the client and server processes;

- When the user gives the ip address of the server computer, the client contacts the

RMI registry server on that machine.

- The RMI registry returns a list of the **registered Matrex servers** (in general the Matrex servers available on that server computer)
- When the user selects one of the Matrex server, the client obtains from the registry the RMI reference to the (unique) **ServerAccess business object** of that Matrex server.
- The client obtains a reference to the **Server** business object either directly or by logging in (if the security is active),
- The client gets the list of templates available in the server (getTemplates) as RMI references.
- From the Server business object the client is able to get the template tree and a **list of the projects** available in that Matrex server (as names).
- When the user selects one of the projects from the list, the client calls the Server openProject method with the selected project name, obtaining the reference to the **Project** business object for that project.
- From the Project business object the client gets the **list of RMI references to all the project items** (matrices, functions...). From these the client builds the trees in the project tab.

Editing

We have to be careful about the editing functionality.

It is not possible to have two clients **editing the same item** (matrix, presentation, function...) of a project **at the same time**. To allow this would cause inconsistencies, for example at the end of the editing the item would be as one of the client saved it, with all the changes entered by the other client discarded.

Therefore only one client can edit an item of a project at one time. Only when the client terminates the editing, the item can be edited by another client.

To enforce this feature, the server keeps track of who is editing what. This feature can be maintained by the **edit table** [item id -> client that is using it]. In this way:

- When a client starts to edit one item, every client is notified. In the desktop application the message appears in the messages view.

- When the client closes the editor for an item the server notifies every client. The item is removed from the edit table and becomes available for editing.
- When a client tries to edit one item that is already in edit state, it gets an error.
- To avoid that, when a client crashes, the items it started to edit remain unavailable for the other clients, the system uses a **lease** protocol, available in RMI.
  A client that implements the lease protocol has to periodically notify the server that it is alive. If the server does not receive this notification for a certain period, it assumes that the client crashed and removes all the references to it from the edit table, so that the items it was editing become free.

Transactions

There are cases in which the client has to do a **series of operations on a project** that cannot be interrupted.

Here are examples of these cases:

- Creation of a function. The creation of a function implies also the creation of all the matrices result of the new function.
- Creation of functions/matrices with the Expression Parser. At the end of the expression parser wizard several functions and matrices are created together. They are dependent by each other.
- Simple refactoring of the project. Operations on the project that imply for example multiple items movements.

These are typical **transactional** operations: either the whole series of operations is applied to the project or, if an error or any kind of problem occurs in doing it, nothing has to be done.

To have a transactional behavior, when:

- the function with its result matrices is saved
- the functions and matrices created by the Expression Parser are saved
- the project items are moved

and in general when the client start saving any kind of item in the project (also simple items like matrices), **the client acquires the control on the project** in the server. This means that the server adds to the edit table a pair [project id -> client that wants to apply the transactional operation].

Once the client is finished to save the items, it gives back the control on the project and

the server removes the pair from the edit table.

Note that:

- If a client tries to acquire the control on a project that is already in edit state it has to wait until the project becomes available.
- The save operations should be always structured in the following way:
    - acquire the control on the project
    - check if it is possible to save (for example there are no matrices with the same name/path of any that is supposed to be saved in this operation). If not, try to save in a different way (other name/paths?). If nothing works, abort the save operation and release the control on the project.
    - If everything is fine, save.
    - release the control on the project

Messages: local and remote

The business objects remain **always in the server**: they are never sent as RMI messages. The clients access their content through their methods.

This means that all the messages and calls among various Matrex items (projects, matrices, functions...) are **local to the server**.

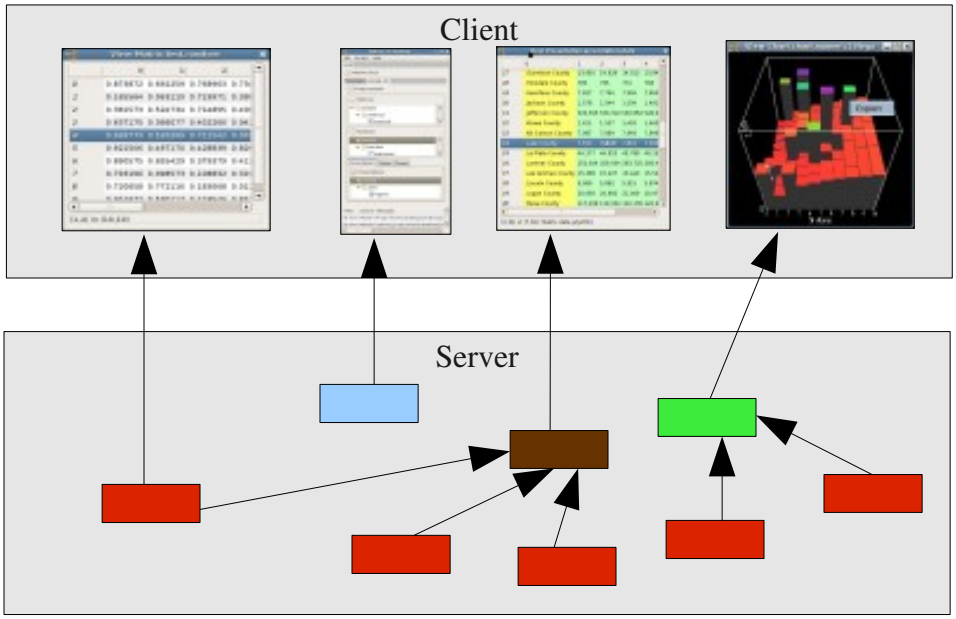
For example, when the content of a matrix changes and the matrix is part of a presentation, the matrix notifies the change to the presentation locally in the server.

The client is notified of the changes **only if it has a viewer open**, showing a matrix or a presentation contained in the server.

Here we show a client with 3 viewers open (matrix, presentation, chart). The viewers are notified when the related matrix, presentation, chart content are changed. Also the trees of the project tab are notified when an item (matrix, function...) is added, deleted or updated.

On the other side, you can see that all the other notifications are **performed internally in the server**, which means that the network traffic is limited: in general only a few viewers are open in the same time in a client.

In the picture the red boxes in the server are matrices, the brown is a presentation, the green is a chart and the blue is the project.



Remote asynchronous messaging

RMI handles the asynchronous messaging using **callbacks**: the client object that needs to receive asynchronous messages **subscribes** to the server object using its **own RMI reference**. From that moment the server object calls a specific method of the client object each time it has to notify something to it.

This means two things:

- The client object becomes an RMI business object and each time the server object notifies something the two objects in a certain way invert their roles (client becomes server, server become client).
- The RMI callback mechanism is perfectly compatible with the notifier/listener pattern used by Matrex for the internal asynchronous messaging.

Now the questions are:

- Which kind of messages the server sends asynchronously to the client?
- Which are the objects in the client that are notified by the server through callbacks?
- How we implement the callback subscription mechanism in the server objects?

The messages are the same sent internally in the Matrix desktop:

- **matrex.item.events.matrix.MatrixEvent** when the content of a matrix changes.

- **matrex.item.presentation.attributes.PresentedAttributes , matrex.item.presentation.object.IPresentedObjectData** when the content of a presentation changes.
- **matrex.item.events.chart.ChartEvent** when the content of a matrix changes.
- **matrex.item.events.project.ProjectEvent** when the content of the project changes (items have been added, deleted, updated).
- **matrex.item.events.machine.MachineEvent** when the content of the server changes (templates have been added, deleted, updated).

The client items notified asynchronously by the server are all the client viewers and the trees in the main GUI. So, at a first look we will have:

- **matrex.gui.viewer.matrix.MatrixViewerCache** for the matrix messages
- **matrex.gui.viewer.presentation.PresentationCache** for the presentation messages
- The various classes implementing **matrex.gui.viewer.chart.dataIDatasetAdapter** for the chart messages
- **matrex.gui.TemplateGUI** for the machine messages (changes in the template structure).
- **matrex.gui.ProjectGUI** for the project messages.

The callback subscription mechanism in the server business object is exactly the same that we use now internally in Matrex. The **addListener/removeListener** methods can accept both local and remote listeners.

Messages format

At least in a first stage, when an item changes, its **complete content** will be sent to the subscribing clients. This means that, for example, if only one cell of a matrix changes, the whole matrix content is sent in the message.

The reason of this decision are:

- maintain the implementation **simple** and therefore reduce the problems.
- it is not told that sending only the part that changes is faster: the code to verify what changed could be **slow** and the **size** of the “delta” message (that contain the coordinates of the changed cells together with their values) could be not so much smaller than the

size of the “complete” message.

- in general the **cases** in which only a small portion of a matrix or presentation is changed are not common. The Matrex algorithms act generally on the whole matrix and the use of Matrex is really convenient when big portions of the matrices are changed.

Since we are using RMI, the messages will consist of the event objects that we saw previously, serialized using the common **Java binary serialization**.

The general alternative, to use XML messages, is discarded because of the performance penalization involved in parsing XML; for the same reason we discarded SOAP as a protocol.

Copying projects in the network

Projects are in the server, the client accesses them through RMI, everything works fine. Yes, but how did the projects end up in the server?

Every server machine tab in the GUI has a menu to copy project to or from that server.

When the user decides to **copy a project to a server**:

- The user clicks on the **copy to** menu of the server tab.
- He selects the project using the *Open project* dialog. He cannot copy to the server a project that is currently open in the client.
- The client **zips** the whole project directory in one single byte array (Matrex is already able to zip projects).
- The client calls the server **copyProjectTo** method with the zipped project.
- The server unzips the project in its own projects directory.
- The project is saved and can be used remotely.

It is also possible to copy a project **from a server** to the client machine, so that it can be used locally by the client. To do that:

- The user clicks on the **copy from** menu on the server tab.
- He selects the project from the projects list dialog seen in the usage scenario.
- The client calls the server **copyProjectFrom** method for the selected project.

- If the project is currently used by one or more clients the method returns an error.
- Otherwise, the server **zips** the project in a single byte array and returns it.
- The user needs to select the directory that will contain the project using the *Save Project dialog*.
- The project is saved and can be used locally.

Security

We decided to keep the system as simple as possible, and that has a cost, especially in terms of security.

Since Matrex is supposed to work in a LAN, the assumption is that the level of security can be low. This guarantees that the system is easy to install and maintain.

A Matrex server can have the security active or inactive.

- If the security is inactive anyone can access that server without login.
- If the security is active each user has to login to access that server.

Once you can access a server you can do **what ever you want**: open a project, copy a project from and to the server.

To make the security active, the server is started with the *passwordfile* option:

*java -cp matrex_server.jar;... matrex.server.**ServerStart -passwordfile=password.xml** ...*

This file password.xml contains pairs user-password, where the password are encrypted.

To add a new pair user-password in the password.xml we make a special class/command, for example PasswordAdd:

*java -cp matrex_server.jar;... matrex.server.**PasswordAdd** -passwordfile=password.xml <user> <password>*

where <user> and <password> are the user and password to add.

At least in a first stage, it will not be possible for a user to change his password from the client.

A last word about security: the messages exchanged in the network are not encrypted.

Configuration

To start the server, as we saw in part speaking about the security, the command should be more or less that one:

java -cp matrex_server.jar;matrex_api.jar;matrex_fun.jar matrex.server.ServerStart -projectdir=<project directory> [-passwordfile=<xml password file>]

where:

- matrex_server.jar is the library containing all the server classes, included ServerStart
- matrex_api.jar and matrex_fun.jar are the two other libraries needed for the server to work.
- matrex.server.ServerStart is the class that contains the static main method.
- <project directory> is the directory containing all the server's projects.
- <xml password file> (optional) is the xml file containing the user-password pairs if the server security is active.