Remote ImageJ - Running macros on a distant machine

Volker Baecker[a] and Pierre Travo[b]

[a]Montpellier RIO Imaging, IFR 122, INSERM;
[b]Montpellier RIO Imaging, CRBM, CNRS

ABSTRACT

Remote-ImageJ is developed at the imaging facility Montpellier RIO Imaging. It allows to run ImageJ macros on a remote machine.

The communication between distant ImageJ-plugins is based on a custom messaging middleware called "Simple Java Message Exchange" (SIJAME). The SIJAME-server listens on a socket[1] and handles each incoming connection in a separate thread. Messages consist of serialized message-objects that can carry arbitrary data. Plugins can use the SIJAME-server in two different ways. They can either directly add themselves to the server's list of message-handlers or they can use the server's request-message queue. The queue allows an asynchronous but ordered communication. The client sends a message that is added to the server's queue of requests. Interested parties are notified when the first request in the queue changes. They can handle the request and remove it from the queue. An answer can be sent back to a client, using the server and port information carried by the message. For that purpose the client runs its own SIJAME-server on a different port. A dedicated answer-message queue can be used to receive answer-messages.

Based on the SIJAME-middleware a Remote-Macro-Runner has been written. The Remote-Macro-Runner-Server-Console allows to start and stop the server and to view log-messages. The Modal-Dialog-Killer avoids that modal dialogs, opened by a macro or by an error in a macro, block the macro execution on the server. A local macro is run on a distant machine, with the help of the remote-macro-runner-client-application. To set input files and folders, and output folders, using a `JFileChooser`-dialog and a list-editor, The `IOSettings`-plugin is used. These settings are accessible from within the macro. The `RemoteFilesystemView` implements a filesystem-view as a proxy that gets its information from a remote machine.

Keywords: macro, batch, remote, messaging, socket

1. INTRODUCTION

At the imaging facility Montpellier RIO Imaging, ImageJ macros and scripts are used to solve and automate image analysis tasks. The aim of the Remote-ImageJ project is to realize a batch-job queue-system for ImageJ macros and scripts. This batch-job queue-system will be integrated into our Cicero-Image-Database software, so that ImageJ batch-jobs can be run on images in the database from a web-interface. A number of libraries and plugins have been written, that can be used independently of the Cicero-Image-Database.

For the time being the scientists run macros either on dedicated image analysis machines of the facility or on the machines of their research group. The drawback of this approach is that long running macros and scripts will block the machines and that no centralized management is possible. We therefore decided to create a job-queue system for ImageJ macros and scripts, similar to the solution that we use for image deconvolution jobs.[2]

The idea is to have a number of dedicated ImageJ macro processing server-machines. These are known by a job queue component. A client sends job-descriptions, consisting of a script or macro, parameters and a list of input images to the queue manager. The queue manager adds the job to the job-queue. When one of the macro-processors is idle, the first job in the queue is sent to it. When the processing finished, the job is removed from the queue. Input images must be on a fileserver that is accessible with the same path from all macro-processors. Results are written to the same fileserver.

Further author information: (Send correspondence to Volker Baecker)
Volker Baecker: E-mail: volker.baecker@mri.cnrs.fr, Telephone: 33 4 67 61 22 54

Other than GridIJ,[3] Remote ImageJ does not aim at speeding up single macros by using parallel processing. Instead it allows to organize the execution of long running macros as batch jobs on dedicated machines. Some basic components needed for the realization of the job-queue system have been implemented in the form of library code and ImageJ plugins. These components are:

1. Simple Java Message Exchange middleware (SIJAME)
   This will be used for the communication between remote machines. It allows synchronous and asynchronous communication and uses its own in-memory message queue.
2. `RemoteFilesystemView`
   Implements a `FilesystemView`[4] that can be used by `JFileChooser`. The `RemoteFilesystemView` uses a proxy that gets the information from a remote machine by using SIJAME in synchronous mode. This will be used to get the paths to the images on a remote machine.
3. `ModalDialogKiller`
   Makes modal dialogs modeless and closes them. This is used by the Remote-Macro-Runner server to avoid that errors in a macro, or dialogs opened by a macro block the macro execution on the server.
4. `Macro_IO_Settings`
   This plugin provides a convenient way to select input files and folders and output folders that can be used from within a macro or script.
5. Remote-Macro-Runner server
   The Remote-Macro-Runner server uses the SIJAME middleware in asynchronous mode to receive and execute ImageJ macros. A server console allows to start and stop the server and to view log-files.
6. Remote-Macro-Runner client
   The Remote-Macro-Runner client allows to run a local macro on a remote machine. It provides a simple user interface and uses the SIJAME middleware to send the macro to a Remote-Macro-Runner server.

For the time being no security considerations have been taken into account. One of the risks is that the ImageJ macro language gives access to native commands on the server machine. A setting in which the usage of the software is possible would be in a local network for which the ports used are closed to the outside. The server process should run under a dedicated account that does not have root or administrator rights.

2. MATERIALS AND METHODS

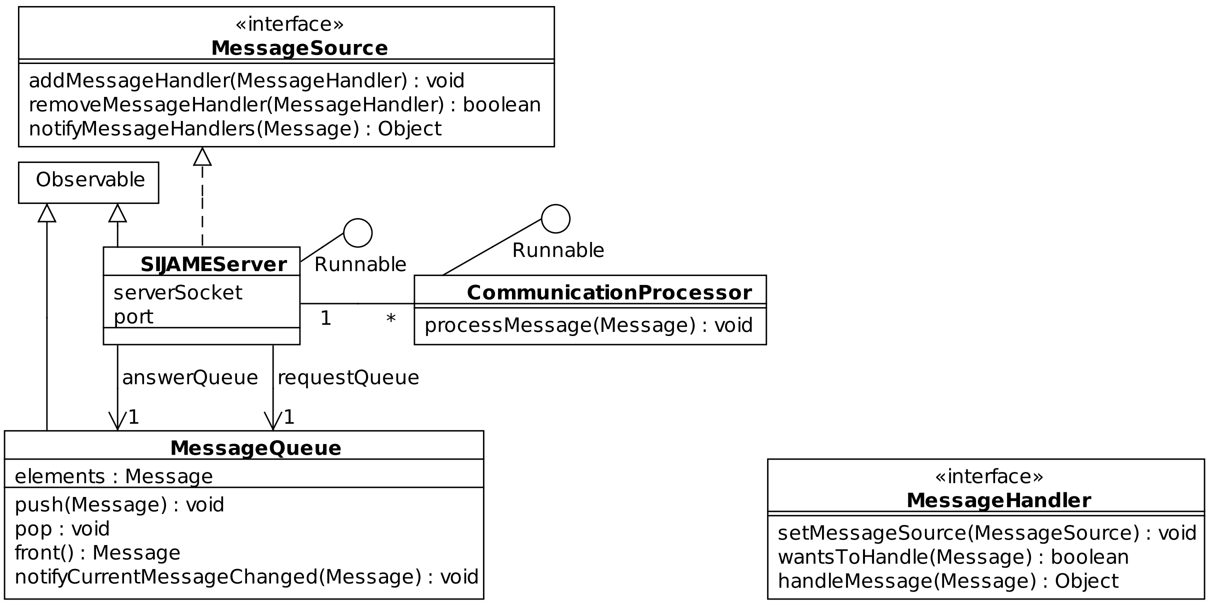
2.1 SIJAME

The Remote-Macro-Runner and the `RemoteFilesystemView` are based on the SIJAME middleware. The architecture of SIJAME will be explained in this chapter.

Figure 1 shows the main architecture of the SIJAME software. The `SIJAMEServer` is a `Runnable` and will be executed in a separate thread, so that a controlling software can communicate with it while it is listening for incoming connections. It is observable in order to notify observers about the start and the shutdown of the server. For each communication a new `CommunicationProcessor` is created. Each communication is handled in its own thread, so that the server can, in the same time, handle one communication and listen for other incoming connections.

A software that wants to use the server for synchronous communication must implement the `MessageHandler` interface, shown in figure 1b, and add itself to the server's message handlers.

A software that wants to use asynchronous communication must implement the `Observer` interface and add itself to one of the server's message queues. It will be notified whenever the first message in the queue changes. The notification contains the actual message and the software can react to the message. It can decide to consume or not to consume the message. A message that is consumed will be automatically removed from the queue after all observers have been notified. If the queue is not empty the first message in the queue will then change again.


«interface»
MessageSource
addMessageHandler(MessageHandler) : void
removeMessageHandler(MessageHandler) : boolean
notifyMessageHandlers(Message) : Object
Observable
SIJAMEServer
serverSocket
port
Runnable
Runnable
CommunicationProcessor
processMessage(Message) : void
1
*
answerQueue
requestQueue
1
1
MessageQueue
elements : Message
push(Message) : void
pop : void
front() : Message
notifyCurrentMessageChanged(Message) : void
«interface»
MessageHandler
setMessageSource(MessageSource) : void
wantsToHandle(Message) : boolean
handleMessage(Message) : Object

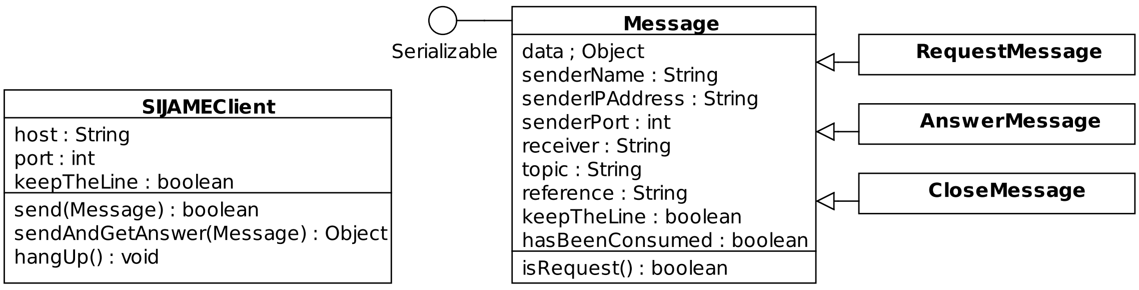
(a) The architecture of the SIJAME software. (b) The `MessageHandler` interface.

Figure 1: The SIJAME server software.

On the client side the `SIJAMEClient`, shown in figure 2a, can be used. In the case of synchronous communication, the client software uses the `sendAndGetAnswer` method. Using the `keepTheLine` attribute, it can decide to keep the same connection open for the whole communication or to close the connection and open a new connection for each message. In case of an asynchronous communication, the client software uses the `send` method. If it wants to be able to get answer message back from the server, it must run a SIJAME-server on another port itself and send the host and port information in the messages.

Figure 2b shows the classes of messages that can be exchanged within the SIJAME-framework. Request-messages are put into the server's request-queue, answer-messages are put into the answer-queue. Close-messages can be used by synchronous clients to signal the end of a communication to the server.

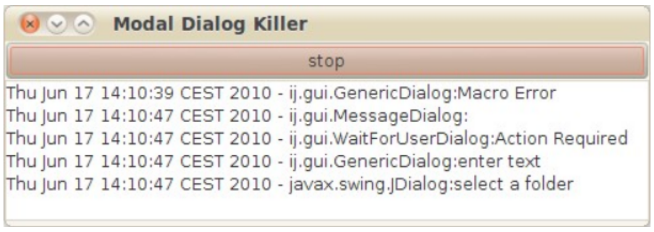
A message can carry arbitrary data in the data-attribute, including arrays or lists, as long as the objects are serializable. The attributes `senderIPAddress` and `senderPort` are used to send answer messages back. The attributes `senderName`, `receiver`, `topic` and `reference` have no fixed semantic. They can be used by the server side software to decide if and how a message should be handled.



(a) The SIJAMEClient class. (b) The message-classes used by SIJAME.

Figure 2: The SIJAMEClient and the message classes.

2.2 ModalDialogKiller



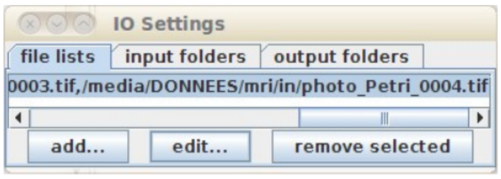
Modal Dialog Killer
stop

```
Thu Jun 17 14:10:39 CEST 2010 - ij.gui.GenericDialog:Macro Error
Thu Jun 17 14:10:47 CEST 2010 - ij.gui.MessageDialog:
Thu Jun 17 14:10:47 CEST 2010 - ij.gui.WaitForUserDialog:Action Required
Thu Jun 17 14:10:47 CEST 2010 - ij.gui.GenericDialog:enter text
Thu Jun 17 14:10:47 CEST 2010 - javax.swing.JDialog:select a folder
```

Figure 3: The modal-dialog-killer plugin.

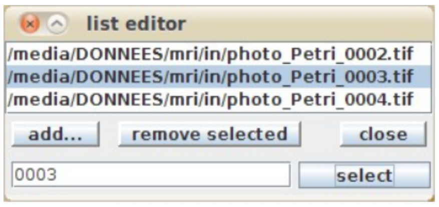
An error in a macro opens a modal dialog. If this would happen on the remote machine, the server would be blocked. This can be avoided with the help of the `ModalDialogKiller`. While it is activated it listens to changes of the active window. If the newly activated window is a `WaitForUserDialog` the dialog is closed. If it is a modal dialog the dialog is made modeless, hidden and disposed. The java logging api[5] is used to log the events when a dialog is handled. If wanted a software can get the logging messages by creating a handler for the `fr.cnrs.mri.tools.dialog` subsytem. A plugin version of the modal-dialog-killer is available. The plugin allows to start and stop the modal-dialog-killer and displays the log.

2.3 Macro_IO_Settings



IO Settings
file lists | input folders | output folders
0003.tif,/media/DONNEES/mri/in/photo_Petri_0004.tif
add... | edit... | remove selected

(a) The IO Settings plugin.



list editor

```
/media/DONNEES/mri/in/photo_Petri_0002.tif
/media/DONNEES/mri/in/photo_Petri_0003.tif
/media/DONNEES/mri/in/photo_Petri_0004.tif
```

add... | remove selected | close
0003 | select

(b) The list editor.

Figure 4: Selecting files and folders for a macro.

The plugin allows to set lists of files, input-folders and output-folders via a graphical interface. The lists are accessible from within a macro or script. Multiple lists of files are possible. The files are selected using a file dialog. If the selected file is a folder, all images in all subfolders will be added. A list-editor allows to remove images from the list. A macro that demonstrates the usage is available. An example could look like this:

```
call("fr.cnrs.mri.macro.io.IOSettings.resetFileLists");
call("fr.cnrs.mri.macro.io.IOSettings.show");
waitForUser("Please select the input files using the IO_Settings dialog and press ok");
list = call("fr.cnrs.mri.macro.io.IOSettings.getFileList");
if (list=="none") {IJ.log("No files selected! Macro stopped."); return;}
files = split(list, ",");
for (i=0; i<files.length; i++) {
    file = files[i];
...
```

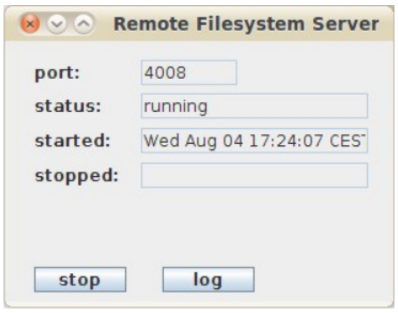
A list is returned as a string in which the single paths are separated by commas. The empty list is represented by the string `"none"`.

3. RESULTS

Based on the components described in the last chapter, the `RemoteFilesystemView` and the `Remote-Macro-Runner` allow to run macros on a distant machine.

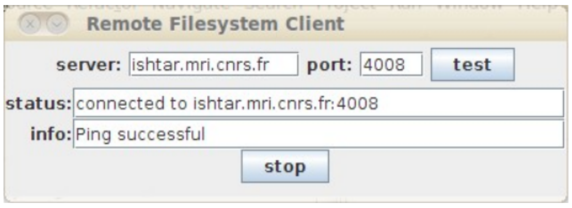
3.1 RemoteFilesystemView

To use the `RemoteFilesystemView` the remote-filesystem-server must run on the remote machine. It can be started as a standalone program or as an ImageJ-plugin. A configuration file contains a default port, the root-folder of the fileserver and a flag that specifies if user folders will be used. The root folder of the fileserver is the folder that will be visible as the root of the filesystem for clients, if the `use_user_folders`-flag is set to false. If the flag is set to true, clients will see two folders as root-folders of the filesystem: a folder with the login-name of the user and a folder `common`. These are automatically created under the fileserver root folder, if they do not exist. The user's folder will be accessible only by the user himself, while the folder `common` will be accessible to all users.


Remote Filesystem Server

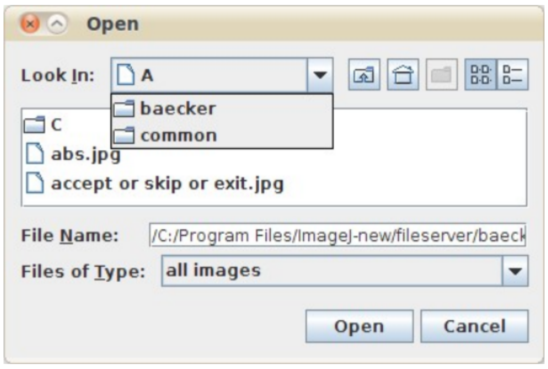
port: 4008
status: running
started: Wed Aug 04 17:24:07 CES
stopped:

Figure 5: The console of the remote-filesystem-server.


Remote Filesystem Client

server: ishtar.mri.cnrs.fr port: 4008
status: connected to ishtar.mri.cnrs.fr:4008
info: Ping successful

(a) The remote-filesystem-client


Open

Look In: A
baecker
common
C
abs.jpg
accept or skip or exit.jpg
File Name: /C:/Program Files/ImageJ-new/fileserver/baeck
Files of Type: all images

(b) A JFileChooser with remote-filessystem-view.

Figure 6: Accessing remote file paths.

If the remote-filesystem-client is started and connected to a server, the io-settings plugin will use the remote-filesystem-view. The user will see the accessible folders on the remote machine, in the JFileChooser-dialog. The dialog will answer the paths on the remote machine.

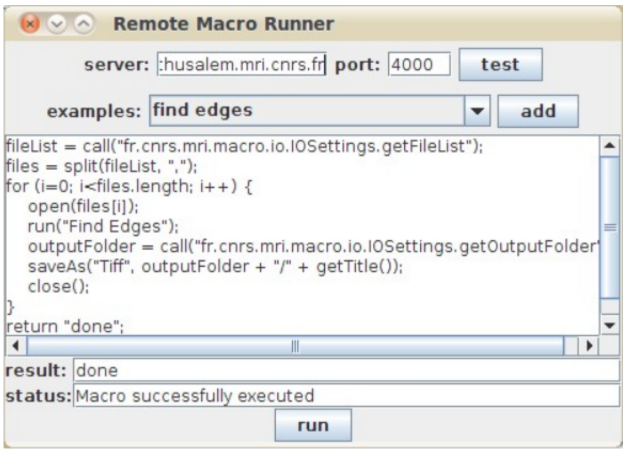
3.2 Remote-Macro-Runner

The remote-macro-runner uses SIJAME to send macros from a client to a server. A macro is sent together with the current io-settings in one SIJAME-message. The remote-macro-runner server puts the incoming messages into the message-queue. It handles one message after the other. The modal-dialog-killer is started by the server, so that modal dialogs opened from macros will not block execution. The server extracts the macro and the io-settings from the message. It sets the io-settings and runs the macro. The macro must have a return-value. The return value is sent back as an answer-message to the client.

A user can open the remote-macro-runner client and connect to a server. He can paste a macro into the text area of the client and run the macro on the server. He can use the io-settings together with the remote-filesystem-view to specify multiple lists of input images, an input folder and an output folder with paths that exist on the server. Multiple clients can use the same server in the same time. The remote-macro-runner will handle incoming macros one after the other.

4. CONCLUSIONS AND FUTURE WORK

A simple messaging middleware SIJAME has been implemented. It allows synchronous and asynchronous communication between remote java programs. It has been used in synchronous mode to implement a remote-filesystem-view and in asynchronous mode to implement a remote-macro-runner for ImageJ. The `Macro_IO_Settings` plugin provides a convenient way to select multiple lists of image paths, one or more input-folders and one or more output-folders. The selected paths are available via static class methods and can therefore be used in macros and scripts. They can be used for local paths and, in combination with the remote-filesystem-view, for paths on a remote machine. The remote-macro-runner allows to queue and run macros on a remote machine. It uses the modal-dialog-killer to automatically close modal dialogs, so that errors will not block the server. All these components are available as ImageJ-plugins. No third party software is needed.



server: :husalem.mri.cnrs.fr port: 4000

examples: find edges

```
fileList = call("fr.cnrs.mri.macro.io.IOSettings.getFileList");
files = split(fileList, ",");
for (i=0; i<files.length; i++) {
   open(files[i]);
   run("Find Edges");
   outputFolder = call("fr.cnrs.mri.macro.io.IOSettings.getOutputFolder"
   saveAs("Tiff", outputFolder + "/" + getTitle());
   close();
}
return "done";
```

result: done

status: Macro successfully executed

Figure 7: The remote-macro-runner client.

With the help of the system in its current form, ImageJ users can run macros from their workstations on dedicated ImageJ processing machines. They need to be able to access a disk that is reachable from these machines, for example via a common mount or ftp. Advantages are that the workstation will not be blocked and that powerful images processing machines can be used from all clients to run the jobs.

The next step will be to implements a queue-manager. Clients will then send macros to the queue manger that will queue them and distribute them to the processing machines.

For the time being, only macros are handled. Besides macros, ImageJ supports Javascript and the FIJI[6] distribution of ImageJ supports different other scripting languages. Remote-ImageJ can easily be extend to allow the remote execution of theses scripts as well as the remote execution of visual-scripts used in the MRI-Cell-Image-Analyzer.[7]

Currently only paths are sent as parameters along with the macros. Other parameters must be set within the macro itself. Dialogs can not be used for this, since they would be opened on the server. A mechanism to provide parameters with a graphical user interface and to send them to the server, would further simplify the usage, as well as an option to directly send local images from within ImageJ to the server.

To allow the usage across internet a future version of the software would need the possibility to use password-authentication, secure sockets and certificates.

At Montpellier RIO Imaging, Remote-ImageJ will be used from the web-interface of our image-database application Cicero, to run prepared batch-jobs directly on images referenced in the image-database.

REFERENCES

[1] Harold, E. R., [*Java Network Programming*], ch. 9 and 10, O'Reilly Media, third ed. (2004).

[2] Ponti, A., Gulati, A., Baecker, V., and Schwarb, P., "Huygens remote manager, a web interface for high-volume batch deconvolution," *Imaging & Microscopy* **9**(2), 57–58 (2007).

[3] Frank, A., Stotzka, R., Jejkal, T., Hartmann, V., Sutter, M., and Gemmeke, H., "GridIJ - a dynamic grid service architecture for scientific image processing.," in [*EUROMICRO-SEAA*], 375–384, IEEE Computer Society (2007).

[4] Flanagan, D., [*Java Foundation Classes in a Nutshell*], ch. 27, 552, O'Reilly Media, Inc. (1999).

[5] Gupta, S., [*Logging in Java with the JDK 1.4 Logging API and Apache log4j*], Apress (2003).

[6] Schindelin, J. E., "Fiji is just imagej - batteries included," in [*Proceedings of the ImageJ User and Developer Conference*], Jahnen, A. and Moll, C., eds., 99–104, Centre de Recherche Public Henri Tudor (2008).

[7] Baecker, V. and Travo, P., "Cell image analyzer - a visual scripting interface for imagej and its usage at the microscopy facility montpellier rio imaging," in [*Proceedings of the ImageJ User and Developer Conference*], Jahnen, A. and Moll, C., eds., 105–110, Centre de Recherche Public Henri Tudor (2006).