MagickBox Documentation

Release 0.0.1

Hauke Bartsch

September 14, 2015

Contents

This documentation is a work in progress. All functionality described is still experimental.

The MagickBox computer system executes processing tasks on data. Data is send to the system which reacts by performing an operation on the data optionally followed by a reply to the sender. Because of its simplicity it integrates well into existing institutions with large data processing needs. As a primary use case MagickBox was developed to facilitate image processing in a research hospital environment by supporting DICOM send/receive operations as well as distributed computations of large numbers of image studies in a heterogeneous compute environment.

CHAPTER 1

System Architecture

The system runs inside a virtual machine usually called MasterTemplate. Access to the virtual machine is provided by RDP (Microsoft Remote Desktop Connection). If you are running the virtual machines inside VirtualBox a connection will only be successful after installing the "VirtualBox Extension Pack". Use the following information to access the console of the virtual machine:

```
MasterTemplate user name: processing, password: processing
```

1.1 MasterTemplate

Inside the virtual machine runs Linux (Ubuntu). There are two partitions, one for the main system (/, 20GB) the other for data storage during processing (/data/scratch, >200GB). The dcmtk tools provide the DICOM connectivity and run as a system service (ps aux | grep store). Processing starts automatically when storescpd receives an image study. As processing is data dependent the user needs to select the appropriate processing stream using the AETitle during DICOM send. The list of available processing (AETitles) streams is displayed on the web interface (see below). After processing is done the routing will be executed. This usually just sends the newly generated images back to a listening DICOM node on your network (such as OsiriX or a PACS).

Each processing stream is implemented as a docker container and is controlled in its own directory (/data/streams/bucket<AETitle>) with a configuration file (info.json). All streams run in parallel as system services under gearman (http://gearman.org). This configuration will buffer large numbers of incoming processing requests and tunnel them through each processing stream one at a time.

All system services are monitored using monit (https://mmonit.com/monit/download/) which provides its own user interface to start/stop services. Monit makes sure that after a restart all system services are also restarted. This functionality uses process id files stored in /data/.pids/. On newer monit implementations instead of pid files the process name is used to detect if a service is running. The storescpd service may be stopped automatically by a heart-beat function if it runs but does not respond to the echoscu command. In this case monit will restart the service automatically. The monit user interface runs locally inside the MagickBox only (http://localhost:2812/).

Functionality provided by MagickBox:

- DICOM node listening to incoming connections
- Web interface which provides a user interface for configuration, log files and processed data downloads. The routing functionality can send processed data to any specified DICOM node. When configured to use routing, no continuous access to the web interface is required appart from the initial configuration.

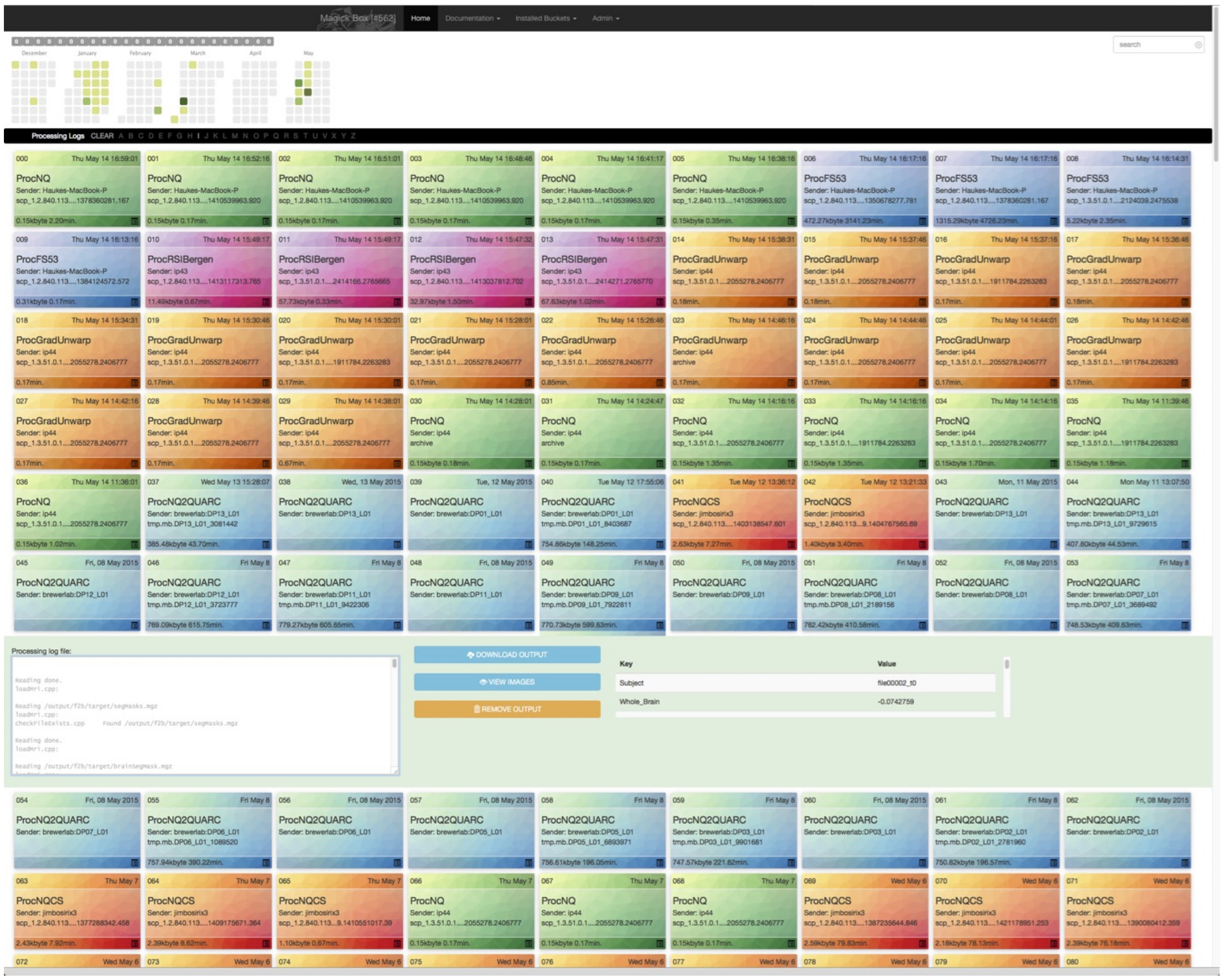
MasterTemplate runs on a Host computer using network address translation (NAT). Using this configuration no separate IP address is required. In order to be accessible to the outside the virtual machine forwards two ports to the outside world.:

```
[TCP, Host IP, port 11113] -> [TCP, Guest IP, port 1234]
[TCP, Host IP, port 2813]  -> [TCP, Guest IP, port 2813]
[TCP, Host IP, port 4321]  -> [TCP, Guest IP, port 22] (optional ssh access)
```

The web interface is available at:

> http://<Host IP>:2813/

Images to be processed should be sent to port 11113 on the host IP, using OsiriX or storescu (part of the dcmtk toolkit).



000 Thu May 14 16:59:01 ProcNQ Sender: Haukes-MacBook-P scp_1.2.840.113....1378360281.167 0.15kbyte 2.20min.
001 Thu May 14 16:52:16 ProcNQ Sender: Haukes-MacBook-P scp_1.2.840.113....1410539963.920 0.15kbyte 0.17min.
002 Thu May 14 16:51:01 ProcNQ Sender: Haukes-MacBook-P scp_1.2.840.113....1410539963.920 0.15kbyte 0.17min.
003 Thu May 14 16:48:46 ProcNQ Sender: Haukes-MacBook-P scp_1.2.840.113....1410539963.920 0.15kbyte 0.17min.
004 Thu May 14 16:41:17 ProcNQ Sender: Haukes-MacBook-P scp_1.2.840.113....1410539963.920 0.15kbyte 0.17min.
005 Thu May 14 16:38:16 ProcNQ Sender: Haukes-MacBook-P scp_1.2.840.113....1410539963.920 0.15kbyte 0.35min.
006 Thu May 14 16:17:16 ProcFS53 Sender: Haukes-MacBook-P scp_1.2.840.113....1350678277.781 472.27kbyte 3141.23min.
007 Thu May 14 16:17:16 ProcFS53 Sender: Haukes-MacBook-P scp_1.2.840.113....1378360281.167 1315.29kbyte 4726.23min.
008 Thu May 14 16:14:31 ProcFS53 Sender: Haukes-MacBook-P scp_1.3.51.0.1....2124039.2475538 5.22kbyte 2.35min.
009 Thu May 14 16:13:16 ProcFS53 Sender: Haukes-MacBook-P scp_1.2.840.113....1384124572.572 0.31kbyte 0.17min.
010 Thu May 14 15:49:17 ProcRSIBergen Sender: ip43 scp_1.2.840.113....1413117313.765 11.49kbyte 0.67min.
011 Thu May 14 15:49:17 ProcRSIBergen Sender: ip43 scp_1.3.51.0.1....2414166.2765665 57.73kbyte 0.33min.
012 Thu May 14 15:47:32 ProcRSIBergen Sender: ip43 scp_1.2.840.113....1413037812.702 32.97kbyte 1.50min.
013 Thu May 14 15:47:31 ProcRSIBergen Sender: ip43 scp_1.3.51.0.1....2414271.2765770 67.63kbyte 1.02min.
014 Thu May 14 15:38:31 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.18min.
015 Thu May 14 15:37:46 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.18min.
016 Thu May 14 15:37:16 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....1911784.2263283 0.17min.
017 Thu May 14 15:36:46 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.18min.
018 Thu May 14 15:34:31 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.17min.
019 Thu May 14 15:30:46 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.17min.
020 Thu May 14 15:30:01 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....1911784.2263283 0.17min.
021 Thu May 14 15:28:01 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.17min.
022 Thu May 14 15:26:46 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.85min.
023 Thu May 14 14:46:16 ProcGradUnwarp Sender: ip44 archive 0.17min.
024 Thu May 14 14:44:46 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.17min.
025 Thu May 14 14:44:01 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.17min.
026 Thu May 14 14:42:46 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....1911784.2263283 0.17min.
027 Thu May 14 14:42:16 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.17min.
028 Thu May 14 14:39:46 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.17min.
029 Thu May 14 14:38:01 ProcGradUnwarp Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.67min.
030 Thu May 14 14:28:01 ProcNQ Sender: ip44 archive 0.15kbyte 0.18min.
031 Thu May 14 14:24:47 ProcNQ Sender: ip44 archive 0.15kbyte 0.17min.
032 Thu May 14 14:16:16 ProcNQ Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.15kbyte 1.35min.
033 Thu May 14 14:16:16 ProcNQ Sender: ip44 scp_1.3.51.0.1....1911784.2263283 0.15kbyte 1.35min.
034 Thu May 14 14:14:16 ProcNQ Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.15kbyte 1.70min.
035 Thu May 14 11:39:46 ProcNQ Sender: ip44 scp_1.3.51.0.1....1911784.2263283 0.15kbyte 1.18min.
036 Thu May 14 11:36:01 ProcNQ Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.15kbyte 1.02min.
037 Wed May 13 15:28:07 ProcNQ2QUARC Sender: brewerlab:DP13_L01 tmp.mb.DP13_L01_3081442 385.48kbyte 43.70min.
038 Wed, 13 May 2015 ProcNQ2QUARC Sender: brewerlab:DP13_L01
039 Tue, 12 May 2015 ProcNQ2QUARC Sender: brewerlab:DP01_L01
040 Tue May 12 17:55:06 ProcNQ2QUARC Sender: brewerlab:DP01_L01 tmp.mb.DP01_L01_8403687 754.86kbyte 148.25min.
041 Tue May 12 13:36:12 ProcNQCS Sender: jimbosirix3 scp_1.2.840.113....1403138547.601 2.63kbyte 7.27min.
042 Tue May 12 13:21:33 ProcNQCS Sender: jimbosirix3 scp_1.2.840.113....9.1404767565.69 1.40kbyte 3.40min.
043 Mon, 11 May 2015 ProcNQ2QUARC Sender: brewerlab:DP13_L01
044 Mon May 11 13:07:50 ProcNQ2QUARC Sender: brewerlab:DP13_L01 tmp.mb.DP13_L01_9729615 407.80kbyte 44.53min.
045 Fri, 08 May 2015 ProcNQ2QUARC Sender: brewerlab:DP12_L01
046 Fri May 8 ProcNQ2QUARC Sender: brewerlab:DP12_L01 tmp.mb.DP12_L01_3723777 769.09kbyte 615.75min.
047 Fri May 8 ProcNQ2QUARC Sender: brewerlab:DP11_L01 tmp.mb.DP11_L01_9422306 779.27kbyte 605.65min.
048 Fri, 08 May 2015 ProcNQ2QUARC Sender: brewerlab:DP11_L01
049 Fri May 8 ProcNQ2QUARC Sender: brewerlab:DP09_L01 tmp.mb.DP09_L01_7922811 770.73kbyte 599.63min.
050 Fri, 08 May 2015 ProcNQ2QUARC Sender: brewerlab:DP09_L01
051 Fri May 8 ProcNQ2QUARC Sender: brewerlab:DP08_L01 tmp.mb.DP08_L01_2189156 762.42kbyte 410.58min.
052 Fri, 08 May 2015 ProcNQ2QUARC Sender: brewerlab:DP08_L01
053 Fri May 8 ProcNQ2QUARC Sender: brewerlab:DP07_L01 tmp.mb.DP07_L01_3689492 748.53kbyte 409.63min.

Processing log file:

```
Reading done.
loadMri.cpp:

Reading /output/f2b/target/segMasks.mgz
loadMri.cpp:
checkFileExists.cpp     Found /output/f2b/target/segMasks.mgz

Reading done.
loadMri.cpp:

Reading /output/f2b/target/brainSegMask.mgz
```

Key	Value
Subject	file00002_t0
Whole_Brain	-0.0742759

054 Fri, 08 May 2015 ProcNQ2QUARC Sender: brewerlab:DP07_L01
055 Fri May 8 ProcNQ2QUARC Sender: brewerlab:DP06_L01 tmp.mb.DP06_L01_1089520 757.94kbyte 390.22min.
056 Fri, 08 May 2015 ProcNQ2QUARC Sender: brewerlab:DP06_L01
057 Fri, 08 May 2015 ProcNQ2QUARC Sender: brewerlab:DP05_L01
058 Fri May 8 ProcNQ2QUARC Sender: brewerlab:DP05_L01 tmp.mb.DP05_L01_6893971 756.61kbyte 196.05min.
059 Fri May 8 ProcNQ2QUARC Sender: brewerlab:DP03_L01 tmp.mb.DP03_L01_9901681 747.57kbyte 221.62min.
060 Fri, 08 May 2015 ProcNQ2QUARC Sender: brewerlab:DP03_L01
061 Fri May 8 ProcNQ2QUARC Sender: brewerlab:DP02_L01 tmp.mb.DP02_L01_2781960 750.82kbyte 196.57min.
062 Fri, 08 May 2015 ProcNQ2QUARC Sender: brewerlab:DP02_L01
063 Thu May 7 ProcNQCS Sender: jimbosirix3 scp_1.2.840.113....1377288342.458 2.43kbyte 7.92min.
064 Thu May 7 ProcNQCS Sender: jimbosirix3 scp_1.2.840.113....1409175671.364 2.39kbyte 8.62min.
065 Thu May 7 ProcNQCS Sender: jimbosirix3 scp_1.2.840.113....9.1410551017.39 1.10kbyte 0.67min.
066 Thu May 7 ProcNQ Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.15kbyte 0.17min.
067 Thu May 7 ProcNQ Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.15kbyte 0.17min.
068 Thu May 7 ProcNQ Sender: ip44 scp_1.3.51.0.1....2055278.2406777 0.15kbyte 0.17min.
069 Wed May 6 ProcNQCS Sender: jimbosirix3 scp_1.2.840.113....1387235644.846 2.59kbyte 79.83min.
070 Wed May 6 ProcNQCS Sender: jimbosirix3 scp_1.2.840.113....1421178951.253 2.18kbyte 78.13min.
071 Wed May 6 ProcNQCS Sender: jimbosirix3 scp_1.2.840.113....1390080412.359 2.39kbyte 76.18min.
072 Wed May 6
073 Wed May 6
074 Wed May 6
075 Wed May 6
076 Wed May 6
077 Wed May 6
078 Wed May 6
079 Wed May 6
080 Wed May 6

Fig. 1.1: Web-interface of MagickBox with timeline and processing jobs..

1.2 Processing Streams

Each processing stream is stored in a separate directory in /data/streams/. This directory is controlled by the buckets script. Each stream directory contains an info file (info.json) with the following content:

```
{
  "name": "My stream",
  "description": "DICOM images are processed.",
  "version": "0.0.1",
  "AETitle": "ProcPics",
  "license": "Feel free to change the code.",
  "enabled": 1
}
```

This identifies a stream by an AETitle which has to be used by the sending DICOM node. Information about the available streams are extracted from these info files and displayed in the web-interface.

Every processing stream is linked to a system service that is responsible for scheduling the stream. Monit is used to monitor these services. The monit configuration files in /etc/monit/conf.d/processing-bucket<AETitle>.conf describes for each bucket the stream which can start/stop/restart the processing.

The script starts the processing stream using a gearman services. Each gearman deamon will wait for a processing job, or list of jobs and execute them one after another (first in - first out). Warning: This does not prevent several streams to run at the same time. For example sending DICOM images between nodes is a separate gearman service and computation will run in parallel with DICOM Send operations.

CHAPTER 2

Setup of MagickBox

The setup of a MagickBox has to be performed if the IP of the machine that hosts the service changes. In this case the user interface provides a “Setup” button at the top that allows the user to specify the IP address of the hosts machine.

In order to setup a new MagickBox follow the steps outlined below.

- install VirtualBox and the VirtualBox extension package (or another virtualization environment)
- import the MasterTemplate OVA file
- setup the virtual machines to start when the host computer starts (/Library/LaunchDaemon/ scripts)

2.1 Setup of MagickBox without VM

MagickBox can also run natively on a Linux system. This requires the following packages to be installed.

- monit - provides monitoring of system services, starts and restarts services
- gearman - provides job scheduling capabilities
- dcmtk - provides native DICOM connectivity
- apache/php5 - web-interface components and endpoints for mb-API calls
- python2.7/pydicom - optional if processSingleFiles.py is used to create alternative views for the data

2.2 DICOM connectivity

MagickBox uses dcmtk (OFFIS toolkit) for its basic DICOM Send/Receive functionality. In order to debug the connection to an existing node edit the /data/code/bin/logger.cfg file. Switch on logging by changing the line:

```
log4cplus.rootLogger = DEBUG, console, logfile
```

This option will write DEBUG, INFO, WARN, ERROR, and FATAL messages to the storescp.log files in /data/logs/. Change the option back to:

```
log4cplus.rootLogger = WARN, console, logfile
```

to reduce the number of log messages. Less log messages can improve the speed of the system.

MagickBox command shell

The MagickBox command shell is used to query, send, receive, and remove jobs from a collection of MagickBox machines. This command line tool provides a convenient way to interface with MagickBox instances for larger projects. If you have ever worked with programs like git the usage pattern should be familiar.

You can download the command shell executable (mb) for your platform here:

- Linux (MD5 = 764a386128376368528cdd6e66f15487) wget https://github.com/HaukeBartsch/MagickBox/raw/master/code/mb-shell/LinuxAMD64/mb
- MacOSX (MD5 = 916df7853de9d82f6ec7a3b20a3d0c9e) wget https://github.com/HaukeBartsch/MagickBox/raw/master/code/mb shell/MacOSX/mb
- Windows (MD5 = 68b9e25fef9f351eca91be531e2f033d) wget https://github.com/HaukeBartsch/MagickBox/raw/master/code/mb-shell/Windows/mb.exe

This is the basic help page of the application:

```
NAME:
   mb - MagickBox command shell for query, send, retrieve, and deletion of data.

   Setup: Start by listing known MagickBox instances (queryMachines). Identify
     your machine and add them using 'activeMachines add'. They will be used
         for all future commands.

     Add your own identity using the setSender command. You can also add your
         projects name (see example below) to make it easier to identify your
         session later.

   Most calls return textual output in JSON format that can be processed by
   tools such as jq (http://stedolan.github.io/jq/).

   Regular expressions are used to identify individual sessions. They can
   be supplied as an additional argument to commands like list, log, push,
   pull, and remove. Only if a session matches, the command will be applied.

   If data is send (push) and there is more than 1 machine available that
   provide that type of processing one of them will be selected based on load.
   Commands such as 'list' will return the machine and port used for that session.

   Example:
     > mb setSender "hauke:testproject"
     > mb push Proc data_01/
     > mb push Proc data_02/
     > mb list hauke:testproject
     > mb pull hauke:testproject


USAGE:
   mb [global options] command [command options] [arguments...]

VERSION:
   0.0.5

AUTHOR:
  Hauke Bartsch - <HaukeBartsch@gmail.com>

COMMANDS:
   pull, g              Retrieve matching jobs output [pull <regular expression>]
   pull-input, pi       Retrieve matching jobs input [pull-input <regular expression>]
   push, p              Send a directory for processing [push <aetitle>
                                       <dicom directory> [<arguments>]]
   remove, r                 Remove data [remove <regular expression>]
   list, l                   Show list of matching jobs [list [regular expression]]
   log, l                         Show processing log of matching jobs
                                                 [log [regular expression]]
   queryMachines, q               Display list of known MagickBox instances
                                  [queryMachines]
   setSender, w                   Specify a string identifying the sender
                                                 [setSender [<sender>]]
   setSetting                     Get or overwrite a program setting [setSetting
                                             [<name> | <name> <value>]]
   computeModules, c              Get list of buckets for the active machines
   activeMachines, a              Get list of active magick box machines
   help, h                        Shows a list of commands or help for one command

GLOBAL OPTIONS:
   --config-sender      Identify yourself, value is used as AETitleCaller
                                [--config-sender <string>]
   --help, -h                    show help
   --version, -v                 print the version
```

3.1 Setup

Start by using the queryMachines command to identify your processing machines (MagickBox needs to run on those machines). You need to activate your MagickBox instances using ‘activeMachines’ once and all future calls to mb will use these machines. Also specify the ‘sender’ (your name or the name of your project for example) as it makes it easier later to identify your scans:

```
> mb queryMachines
[{ "id": "0", "machine": "137.110.172.9", "port": "2813" },
 { "id": "1", "machine": "10.193.13.181", "port": "2813" }]
> mb activeMachines add 137.110.172.9 2813
> mb activeMachines add 10.193.172.181 2813
> mb setSender hauke:project01
```

3.2 Usage

The basic workflow is to first identify some data that is locally available on your harddrive. This could be a directory with T1-weighted images in DICOM format. Send the data to a processing bucket on your MagickBox. Here an example that sends data for gradient unwarp (distortion correction for MRI data):

```
> mb push ProcGradUnwarp ~/data/testdata/DICOMS
```

Mb will zip all files in the directory and upload the zip-file to your MagickBox for processing using the 'ProcGradUnwarp' bucket. Check on the progress of the processing using the 'list' and 'log' commands:

```
> mb list hauke
[{
  "AETitleCalled": "ProcGradUnwarp",
  "AETitleCaller": "hauke:project01",
  "CallerIP": "10.0.2.2",
  "lastChangedTime": "Tue, 02 Sep 2014 00:05:57 -0700",
  "pid": "tmp.8938590",
  "processingLast": 115683,
  "processingLogSize": 1459,
  "processingTime": 387,
  "received": "Mon Sep  1 23:59:30 PDT 2014",
  "scratchdir": "tmp.cPQ1qwWqdw"
}]
```

The 'list' command on its own will list all sessions that exist on the MagickBox, specifying the sender or parts of the sender string will limit the output to entries that match. Here we have a single session returned in JSON format. As a unique key to identify this session use the value of the 'scratchdir' key which is based on a random sequence of letters and numbers.

Use any other string as a search term instead of the sender. You could specify "Sep" and all session that contain "Sep" will be listed. The specified string can also be a regular expression.

A command that works very similar to 'list' is 'log'. Additionally to the information listed by 'list', 'log' will also contain the processing log. Getting the processing log is more time consuming, therefore 'log' is a separate command. You can use it for example to search for error messages in the log files.

Once you have identified your session and processing finished you can download them using 'pull' with the same search term:

```
> mb pull hauke
```

The output of your processing will be downloaded as a zip file into your current directory. The name of the zip file will contain the 'scratchdir'.

How to process a large number of subjects

There are two ways to process large numbers of cases. Use the new command line utility ‘mb’ or follow the description below.

Lets assume that we have a directory with a larger number of DICOM datasets that we want to process. We will send the data to the MB and download the results. First load the data into OsiriX (MacOS) or use the dcmtk toolkit (http://dicom.offis.de) to send the DICOM data to the processing system:

```
storescu +sd +r -nh -aet MySelf -aec ProcBucket \
  <ip of MagickBox> <port of MagickBox> <single DICOM directory>
```

The call above will descend into sub-directories and keep going even if non-DICOM files are encountered. The receiving AE title “ProcBucket” will be used to select the processing stream. Only single subject data should be send to the system in this way as each send operation is interpreted as a command for processing the received data. The above line should therefore be used in a loop with some sleep interval between send operations if more than one session needs to be send.

In order to find out the status of the processing we can either use the web-interface provided by MB or we can use ‘curl’ together with ‘jq’ to automate the procedure. Lets assume that curl and jq are installed. We can get a list of sessions send from a specific machine (here ip44) by:

```
curl http://<ip of MB>:2813/code/php/getScratch.php | jq '.[] | \
  select(.AETitleCaller=="ip44" and .processingTime!="0")'
```

In order to count how many sessions are still in the pipeline for processing use:

```
curl http://<ip of MB>:2813/code/php/getScratch.php | jq -c -M '.[] | \
  select(.AETitleCaller=="ip44" and .processingTime=="0")' | wc -l
```

In order to download the OUTPUT directory of a processed session we need to get the scratchdir and pid information for each finished session:

```
fileList=`curl <ip of MB>:2813/code/php/getScratch.php | jq '.[] | \
  select(.AETitleCaller=="ip44" and .processingTime!="0")' | \
  jq '{"scratchdir": .scratchdir, "pid": .pid}'`
```

We can now download each finished session as a zip file into a separate directory:

```
echo $fileList | jq -c -M . | while read line; do sc=`echo $line | \
  cut -d'"' -f4`; d=`echo $line | cut -d'"' -f8`; mkdir -p "$d"; cd $d; \
  curl -o ${d}.zip http://<ip of MB>:2813/code/php/getOutputZip.php?folder=$sc; \
  cd ..; done
```

In order to delete processed scans use the same mechanism as above:

```
echo $fileList | jq -c -M . | while read line; do sc=`echo $line | \
  cut -d'"' -f4`; d=`echo $line | cut -d'"' -f8`; mkdir -p "$d"; cd $d; \
  curl http://<ip of MB>:2813/code/php/deleteStudy.php?scratchdir=$sc; \
  cd ..; done
```

Routing DICOM

The routing function sends processed DICOM files to other DICOM aware systems. It can react differently in response to success or failure of the computations. Here are some use cases:

- specify the destination for sending result images
- setup a dedicated system that collects copies of partially generated result images
- try to send to a specific PACS system, if send fails try to send to an alternative system (see "break" option)

Routing is performed after a computation is finished. Computations are performed on an INPUT folder and results are placed in an OUTPUT folder. Routing only works on DICOM files found in the OUTPUT folder.

At the level of the parent directory (directory that contains INPUT/ and OUTPUT/ folders) is one file initially called info.json. This file contains information that describes the input connection the data comes from. The computation should place a new file called 'proc.json' next to info.json. This file is evaluated to obtain the information required to start routing. Here is an example content:

```
[{ "success": "failed", "message": "today is Monday" }]
```

This file specifies that the computation failed and provides a reason. In a later version of the program more than one success entry will be read. Currently only the first entry is evaluated.

5.1 Configuration

The configuration of the routing function is done in the user interface. Here an example:

```
{ "routing": [
  {
       "name": "Default Rule for OUTPUT to OsiriX",
       "AETitleIn": ".*",
       "status": 0,
       "send": [
               {
                       ".*": {
                               "IP": "$me",
                               "PORT": "$port",
                               "AETitleSender": "ProcDefault",
                               "AETitleTo": "OsiriX"
                       }
               }
       ],
       "break": 0
  },
  {
    "name": "ProcRSI bucket routing of results",
    "AETitleIn": "ProcRSI",
    "AETitleFrom": "PACS",
    "send": [
        { "success": {
            "IP": "192.168.0.1",
            "PORT": "403",
            "AETitleSender": "me",
            "AETitleTo": "PACS",
            "break": 1,
            "which": [
               { "0008,103e": ".*" }
            ]
          },
          "failed": {
            "IP": "192.168.0.1",
            "PORT": "403",
            "AETitleSender": "me",
            "AETitleTo": "PACS",
            "break": 1
          },
          "partial": {
            "IP": "192.168.0.1",
            "PORT": "403",
            "AETitleSender": "me",
            "AETitleTo": "PACS",
            "break": 1
          }
        }
    ],
    "break": 0
  },
  {
       "name": "Route Input to PACS",
       "RouteDirectory": "INPUT", // default value is "OUTPUT"
       "AETitleIn": ".*",
       "enabled": "F",
       "send": [
               {
                       ".*": {
                               "IP": "$me",
                               "PORT": "$port",
                               "AETitleSender": "ProcDefault",
                               "AETitleTo": "PACS1"
                       }
               }
       ],
       "break": 0
  }
 ]
}
```

A DICOM connection from a station A (PACS) that sends DICOM data to station B (MagickBox) is specified by three types of information for both the sender and the receiver of the information. The Application Entity (AE) title of A and B, the internet protocol (IP) numbers of both stations and the port number that A called on the IP of B. MagickBox uses a single port for all its incoming connections, therefore routing depends on the AETitles and the status (success) returned by the computation.

The default rule above specifies “AETitleIn” which is the application entity title of our MagickBox (B). Additionally, or as an alternative one can also specify “AETitleFrom” as the AETitle that was used by the sending station (A). These two entries, AETitleIn and AETitleFrom are used by the routing function to find out if a specifc routing rule should be applied. A status entry, if present, is used to activate (1, default) or de-active a route (0).

> Currently we do not support the IP-address as a possible filter. This is because the MagickBox runs as a virtual machine using NAT and port forwarding. Therefore the IP address of the incoming DICOM connection is not the IP of the sending machine but of the interface that forwards the packages (host computer running the VM).

For example, the default rule above applies if the AETitle called on B by A matches the pattern ”.*”. This is a regular expression that reads as some character (.) and there can be none, one or more of those. As this rules matches any string, the rule will always apply (default rule) regardless of where the data comes from.

The “send” section contains one or more destinations for sending. Each of the entries is matched one at a time against the processing result (returned proc.json “success” value string). The default rule matches any value of “success” whereas the rule named “ProcRSI bucket routing of results” matches specific strings like “success”, “failed”, or “partial”. If the “success” string matches one of these entries the corresponding destination is chosen to receive the OUTPUT data.

If the “break” entry of a successful sending operation has the value 1 sending stops without evaluating if other send entries would match as well. This allows for a fail-back send destination.

If a “which” statement is set DICOM files are tested before they are send. This filtering step allows you to select DICOM images based on DICOM tags. The value of each tag is filtered by a regular expression and only files that fullfil at least one of the “which” array entries are send to the corresponding destination.

Input data can also be routed. This will happen only after processing and requires a route with a “RouteDirectory”: “INPUT” entry.

A routing rule can be disabled if “enabled” is set to “F”. By default routing rules are executed.

Two placeholders are available “$me” references the IP of the MagickBox and “$port” the port specified in the Setup interface. Both usually refer to a default PACS to send images to.

5.2 Logging

A log file for routing (/data/logs/routing.log) contains routing related messages.

CHAPTER 6

Bucket development

A bucket is a portable light-weight container (based on docker) for MR image analysis pipelines. A bucket runs in the same way a program is executed. If data is available the bucket will start, perform its function and quit. This page explains the 'buckets' development program that is used to create, run and install buckets.

Note: The instructions to develop a bucket work on Linux, MacOS and Windows/cygwin systems only. You need to have docker or boot2docker installed.

6.1 Create

Name your bucket using lower-case characters and numbers only. Create one by downloading the "buckets" script:

```
wget https://raw.githubusercontent.com/HaukeBartsch/MagickBox/master/code/bin/buckets
```

Make the script executable:

```
chmod +x buckets
```

And run its 'create' command:

```
./buckets create mytestbucket
```

This will create "mytestbucket" as an almost empty bucket with some MagickBox special sauce. The bucket contains already its own documentation and a basic configuration. Edit the setup by starting the bucket using the "open" command. It should start an editor (github.com/HaukeBartsch/editor) inside the bucket open a web-browser that points it it (http://localhost:9090):

```
./buckets open mytestbucket
```

Set the application entity title of info.json (AETitle) to be something short and unique. This string is later used to address this processing bucket. Store your changes inside the bucket using a second terminal. Remember to commit your changes after you edited any file inside the bucket.

> docker ps docker commit <running image id> mytestbucket

Now it is time to install your program into the bucket. The program will receive input data in an /input directory and it can produce output data in /output. The bucket is based on ubuntu so you can install a large number of existing programs simply by calling ubuntu's package manager apt-get.

To link up your program and the MagickBox processing entry point work.sh edit that script using the web-editor and have it execute your program given the input and output directories:

```
./bucket open mytestbucket
```

The 'open' command will also create a link to your local directory inside the bucket. If your program is not avialable from one of the apt repositories you can copy the files into your local directory. That directory is available inside the bucket as /local/. Only use this connection to copy data into the bucket during installation of your program. The directory will not be available if the bucket runs inside the MagickBox environment.

6.2 Test

Test your bucket by specifying an input and output directory for the 'run' command (this will emulate tbe work done by MagickBox later):

```
./bucket run mytestbucket <input directory> <output directory>
```

After a successful run you should see your results appear in the directory you specified as the output folder. This is also a great way to locally run a computation on a limited number of cases.

Optionally you can include a plugin that extracts measurements from your output files. Measures exported this way will be available to MagickBox. Useful measures include demographic information or for example measures for volumes of interest. An example plugin file is included with your bucket and listed in the web editor page.

6.3 Install

After developing a bucket and successful local testing using run it can be integrated into a MagickBox machine. If you don't do your development inside MagickBox start by creating a tar-file that represents the content of your bucket:

```
docker export <id of running docker container> > mytestbucket.tar
```

Copy this file to your MagickBox machine and import the bucket using docker followed by "buckets install mytest-bucket":

```
cat mytestbucket.tar | docker import - mytestbucket
/data/code/bin/buckets install mytestbucket
```

The buckets-script will query your mytestbucket container and read the AETitle from the /root/storage/info.json file. The AETitle is used to create a MagickBox bucket directory in /data/streams/bucket<AETitle>. If successful the installer will also start a worker process for your processing bucket. If you now send data to the system using your AETitle to address your bucket the processing should start (see the mb tool). A log-file /data/logs/bucket<AETitle>.log will contain the output of MagickBox when it tries to call your bucket. If your bucket produces some outputs they will be displayed as the processing.log.

6.4 Advanced: Persistent memory for buckets

Buckets have a fixed environment, they receive input and output directories but otherwise the same files will be present during each run of the bucket. Files that are created inside the bucket - but not stored in /input and /output directories are deleted after the bucket stops. If your program requires space that is presistent between runs, like a database, use the /root/storage/memory directory inside the bucket. Place initial versions of your files into this directory. Once the bucket is installed and runs this directory is available inside the bucket as /memory/. Changes to files inside this directory will be available the next time the bucket is started.

CHAPTER 7

Classification of DICOM Files

This module is specific to DICOM file import via storescp. It is not available if DICOM files are send using mb.

The DICOM import is build to support high-volume DICOM ingestion with advanced DICOM series classification. As DICOM files arrive they are copied to a /data/scratch/archive/<Study Instance UID>/ directory for permanent storage.

7.1 Views/raw

Views are alternative directory structures that contain versions of the input data suitable for particular purposes such as quality control and processing. Such directory structures (/data/scratch/views/raw) provide sub-directory structures on the series level and extract DICOM tags from the data. Views are created in parallel with the data import using a daemon process (processSingleFile.py). This allows for accelerated processing buckets with access to series level information before a secondary DICOM parse operation.

The views/raw structure contains a folder named after the StudyInstanceUID which is unique for each study. Inside this folder are folders for each series named using the SeriesInstanceUID. Together with the series directory a <SeriesInstanceUID>.json contains the following DICOM tags derived from the imported series (series level json):

```
{
  "ClassifyType": [
      "GE",
      "sag",
      "T1"
  ],
  "EchoTime": "2.984",
  "InstanceNumber": "3",
  "Manufacturer": "GE MEDICAL SYSTEMS",
  "NumFiles": "166",
  "PatientID": "P0979_03_001",
  "PatientName": "P0979_03_001",
  "Private0019_10BB": "1.000000",
  "Private0043_1039": [
      400,
      0,
      0,
      0
  ],
  "RepetitionTime": "7.38",
  "SeriesDescription": "IRSPGR_PROMO",
  "SeriesInstanceUID": "1.2.840.113619.2.283.6945.3146400.18515.1404745836.841",
  "SeriesNumber": "3",
  "SliceSpacing": "1.2",
  "SliceThickness": "1.2",
  "StudyDate": "20140711",
  "StudyDescription": "PLING",
  "StudyInstanceUID": "1.2.840.113619.6.283.4.679947340.3258.1405103835.996"
}
```

The content of this structure is likely to change in the future. Most of the entries reflect DICOM tags on the series level. The "NumFiles" tag is added to reflect the current number of files in the series directory. The series level directories contain symbolic links to the data stored in the archive folder to limit the number of copy operations and file duplications.

7.2 ClassifyType

The tag called "ClassifyType" is derived from rules that specify how a particular class of scans can be detected from the availble DICOM tags. The test is executed for each incoming DICOM file in the series.

The rule file classifyRules.json stores the control structure for classification and has the following structure:

```
[
  { "type" : "GE",
    "id" : "GEBYMANUFACTURER",
    "description" : "Scanner is GE",
    "rules" : [
      {
         "tag" : [ "0x08", "0x70"],
         "value": "^GE MEDICAL SYSTEMS"
      }
    ]
  },{ "type" : "axial",
    "description": "An axial scan",
    "rules" : [
      {
         "tag" : [ "0x20","0x37" ],
         "value" : [1,0,0,0,1,0],
         "operator": "approx",
         "approxLevel": "0.0004"
      }
    ]
  },{ "type" : "axial",
    "description": "An axial scan",
    "rules" : [
       { "tag" : [ "0x20","0x37" ],
         "value" : [1,0,0,0,1,0],
         "operator": "approx",
         "approxLevel": "0.0004"
       }
    ]
  },{ "type" : "coronal",
    "description": "A coronal scan",
    "rules" : [
       { "tag" : [ "0x20","0x37" ],
         "value" : [1,0,0,0,0,-1],
         "operator": "approx"
       }
    ]
  },{ "type" : "T1",
    "description" : "A T1 weighted image is classified if its from GE and is EFGRE3D",
    "rules" : [
      {
        "rule" : "GEBYMANUFACTURER"
      },{
        "tag": [ "0x19", "0x109e"],
                  "value": "EFGRE3D"
      },{
        "tag": [ "NumFiles" ],
        "operator": ">",
                  "value": "100"
      }
    ]
  },
  { "type" : "T2",
    "description" : "A T2 weighted image",
    "rules" : [
     {
        "rule" : "GEBYMANUFACTURER"
      },{
        "tag": [ "0x19", "0x109c" ],
        "operator": "regexp",
        "value": "Cube"
      }
    ]
  },
  { "type" : "fMRI",
    "description" : "fMRI detected by used b-value",
    "rules" : [
      {
        "rule": "GEBYMANUFACTURER"
      },{
        "tag": [ "0x43", "0x1039", "0" ],
        "operator": "==",
        "value": "4000"
      }
    ]
  }
]
```

Each series type has a name "type" and a short description which is usually ignored and only used as a means to document what the classification tries to implement. The rules for each type are a collection of statements that all have to be true for a scan to be classified as "type". The order of the rules is not important. Every successful classification will add its type to the returned array.

Each rule can contain a reference to another rule (key "rule" with value "id"). This allows for an hierarchical classification of rules. In the example above the rule for detecting if a scan was done on a scanner from GE is referenced in types "T2", "T1", and "fMRI". For debugging a call to "processSingleFile.py test" will list the resolved rules on the command line.

Non-referencing rules contain at least the tags "tag" and "value". If only these two tags are supplied the operation that compares each incoming DICOM files tag value to the one supplied in the "value" field of the rule is assumed to be a regular expression match (python search). The "tag" value can have the following structure:

"tag" [[<key from series level json>]] The tag can describe the number of DICOM slices in this series as "tag": ["NumFiles"].

"tag" [[<dicom group hex code>, <dicom tag hex code>]] The Manufacturer tag can be addressed as "tag" : [

“0x08”, “0x70”]

“tag” [[<dicom group hex code>, <dicom tag hex code>, <vector index>]] If a third argument is supplied the returned tag is assumed to have a vector value and the specific index from that array is used. The b-value for GE diffusion weighted images can be addressed this way as “tag” : [“0x43”, “0x1039”, “0”].

Instead of just using regular expressions tag values can also be interpreted as floating point values. This is forced by the optional tag “operator”. The following operators are available:

“operator” [“==”] Tests for equal value of the tag of the current DICOM file in the series and the value in the rule.

“operator” [”!=”] True of the values are not the same (convertes values to floating point first).

“operator” [“<”] True if value in the DICOM file is smaller.

“operator” [“>”] True if value in the DICOM file is greater.

“operator” [“exist”] True if the tag exists (can be empty).

“operator” [“notexist”] True if the tag does not exist.

“operator” [“contains”] True if the value is in the list of values defined by the tag.

“operator” [“approx”] True if the numerical values of the tag are sufficiently close to the target values. How close can be controlled by an “approxLevel” variable in the rule. The above example uses this to test if the Image Orientation Patient tag that contains the direction cosines for the positive row axis are close enough to be called either axial, sagittal or coronal. A series might contain more than one orientation (like a localizer scan). In this case all three rules might apply as images for that series are classified.

“operator” [“regexp”] Default (non-numeric) regular expression match.

A single rule can be negated by adding the key “negate” with a value of “yes”.

A type can depend on more than one image in the series. For example oblique series are defined as series that are neither coronal, sagittal nor axial. More than one image has to be viewed to be able to make this decission. In order to force a constant update of a type add the key “check” with a value of “SeriesLevel” to the type. Here an example:

```
{ “type” : “oblique”, “description”: “Neither coronal, sagittal nor axial”, “check”: “SeriesLevel”, “rules”:
    [
        { “tag”: [ “ClassifyType” ], “value”: “axial”, “operator”: “contains”, “negate”: “yes”
        }, {
            “tag”: [ “ClassifyType” ], “value”: “coronal”, “operator”: “contains”, “negate”:
            “yes”
        }, {
            “tag”: [ “ClassifyType” ], “value”: “sagittal”, “operator”: “contains”, “negate”:
            “yes”
        }
    ]
}
```

Note: These tests are executed for each file that arrives for a series. If the tags addressed are not series level tags (the same for all files in the series) the outcome of the classification will depend on the order in which files are received.