Medv4D

project documentation

Medv4D: project documentation

Table of Contents

List of Figures

Preface

aaaa

Chapter 1. Library Common

aaaaaaa

Chapter 2. Library Imaging

Main goal of this library is effective implementation of pipeline computation on input datasets. Whole computation should be as parallel as possible in order to utilize resources available in modern processors, etc.

Design of interfaces and class hierarchies is aimed to extensibility and code reusability. Centra

2.1. Compilation

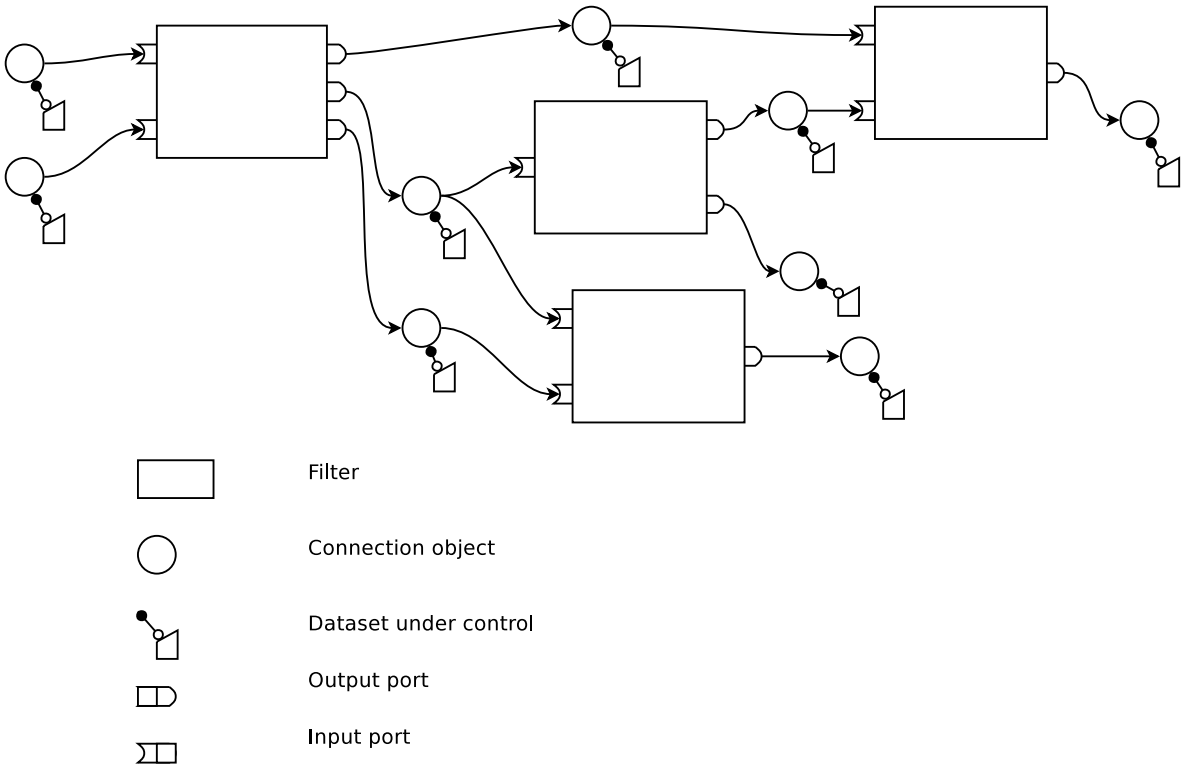
2.1.1. Dependencies

This library should be linked together with libCommon. So all its dependencies come with it.

2.2. Architecture

All declarations are in namespace M4D::Imaging. Whole design of class hierarchies is Figure 2-1

Figure 2-1. Pipeline scheme



Pipeline should be built from objects of certain types, which we will now discuss.

2.2.1. Datasets

Actual data are stored in proper descendants of class `AbstractDataSet`. Its hierarchy is shown in . Purpose of these classes is to provide access methods for reading and writing informations of certain type and optional synchronization.

By optional synchronization is meant set of synchronization methods, which are not called by access methods in class. And user should use these methods only in situations requiring synchronization. This less comfortable though, but more effective.

`AbstractPipeFilter`. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

`AbstractImageFilter`. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

`AbstractImageSliceFilter`. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

2.2.2. Filters

aaaaaaa

2.2.3. Connections

aaaaaaa

2.2.4. Ports

aaaaaaa

2.3. Usage

2.3.1. How to build a pipeline

First thing we should know, is how to create filtering pipeline with supposed behaviour using prepared filters. This section will be brief tutorial to pipeline construction. We show possible usage on example class that will have all desired abilities normaly distributed across application.

Before we start construction, we must decide if we want handle storing and deallocation of all objects. There is prepared container for all pipeline objects, which can handle deallocation of stored objects and semiautomatically connects filters, and other objects with ports used as communication channel. For our example we use this `PipelineContainer`. For manual control and construction see its source code.

```
#include "Imaging/Image.h"
#include "Imaging/PipelineContainer.h"

const unsigned Dim = 3
typedef uint16 ElementType;

typedef M4D::Imaging::Image< ElementType, Dim > ImageType;

class ExamplePipelineHandler
{
public:
    //Default constructor - pipeline construction
    ExamplePipelineHandler();

    //Method which pass image to pipeline and start computation.
    void
```

```
    FilterImage( ImageType::Ptr image );

    //This method will be called, when filters finish.
    //Parameter tells, if computation passed withou problems.
    //Implementation of this method depends on its purpose (notify user, ...).
    void
    FinishNotification( bool succesfully )
        { /*Put your code here.*/ }
protected:
    M4D::Imaging::PipelineContainer   _pipeline;
    M4D::Imaging::AbstractPipeFilter  *_firstFilter;
    M4D::Imaging::AbstractImageConnectionInterface  *_inConnection;
    M4D::Imaging::AbstractImageConnectionInterface  *_outConnection;
};
```

Actual pipeline construction should proceed in three basic steps:

- Allocation of filters
- Establishing connections
- Setting message hooks

Now we discuss these steps in detail. With example implementation.

Allocation of filters. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

Establishing connections. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

Passing input datasets and execution. This is a test. This is only a test. Had this been a real example, it would have made more sense. Or less.

```
#include "ExamplePipelineHandler.h"

ExamplePipelineHandler
::ExamplePipelineHandler()
{
    _firstFilter = new
    M4D::Imaging::AbstractPipeFilter *secondFilter = new

    _pipeline.AddFilter( _firstFilter );
    _pipeline.AddFilter( secondFilter );

    _pipeline.MakeConnection( _firstFilter, 0, secondFilter, 0, true );

    _inConnection =
        dynamic_cast< M4D::Imaging::AbstractImageConnectionInterface * >(
             &(_pipeline.MakeInputConnection( _firstFilter, 0, false )) );

    _outConnection =
        dynamic_cast< M4D::Imaging::AbstractImageConnectionInterface * >(
```

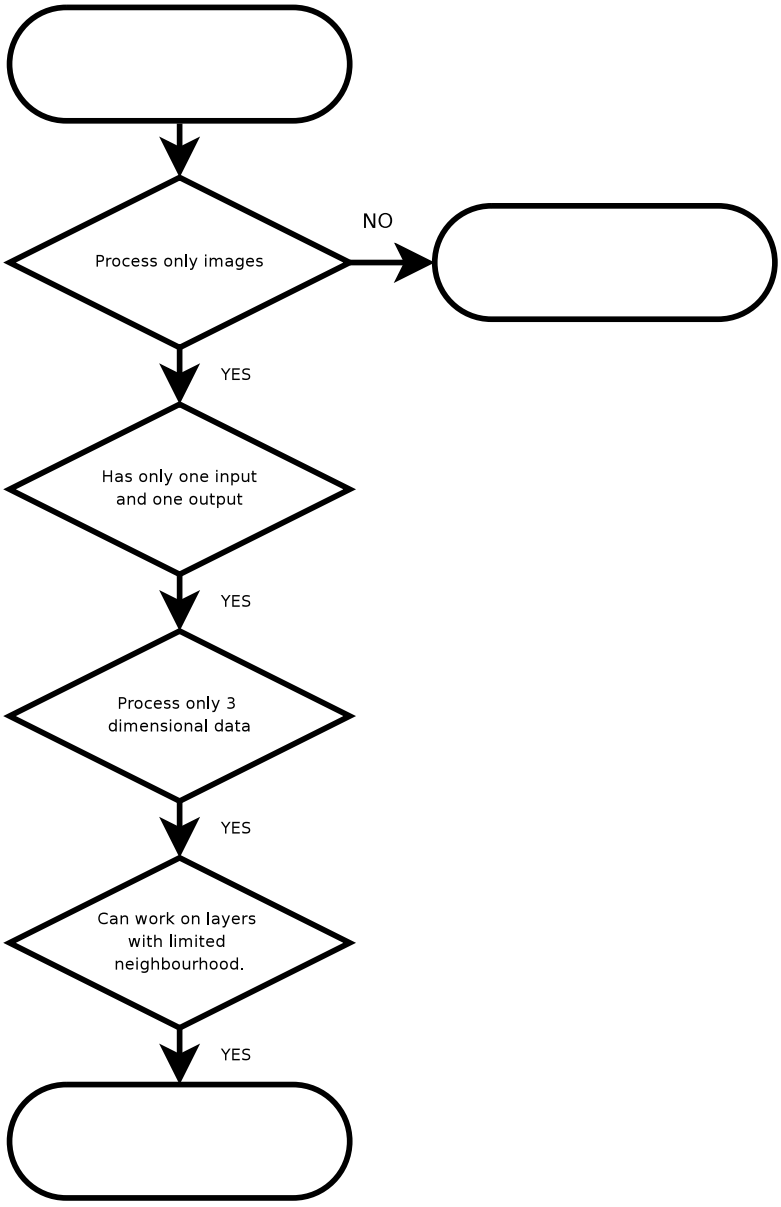
```
            &(_pipeline.MakeOutputConnection( secondFilter, 0, true )) );
}


void
ExamplePipelineHandler
::FilterImage( ImageType::Ptr image )
{
    _inConnection->PutImage( image );

    _firstFilter->Execute();
}
```

2.3.2. Creating new filter

One of the main advantage of this library is its extensibility. The biggest potential is easy way to add new filters with all features promised by pipeline design (synchronization, parallel execution, etc.). This is achieved through set of filter abstract classes, each designed for special purpose. Author of new filter have to consider few aspects (dimensionality, type of input/output data, way the filter computes, etc.) and choose right ancestor class for his filter.

To make decision easier you can use prepared flowchart Figure 2-2 and find most suitable ancestor class.

Figure 2-2. Filter decisive flowchart



Now if you have chosen right ancestor class, you can start with actual implementation of your filter. You should keep few rules and concepts, which can not only help you with writing, but even some other parts of library will just work without handling extra issues. This is achieved by generic design of whole library, templates are used almost everywhere.

We now introduce these rules and concepts, and show practical examples from library sources. First of all try design your filters as generic as possible. So try maximally use dataset traits (now only available `ImageTraits`), template specializations, etc. All ancestor classes are templated (with exception of `AbstractPipeFilter`), so it makes that easier.

Try to keep these rules, when designing filter interface:

- Filter class has public typedef to predecessor class with name `PredecessorType`.
- All properties of filter are in one public nested `struct Properties` deriving from `PredecessorType::Properties`. In case, that set of properties is empty, make at least public typedef to `PredecessorType::Properties` with name Properties.
- Prepare default constructor, and constructor with pointer to `Properties`. Default constructor creates default set of properties. In both constructors pointer to instance of `Properties` is passed to predecessor constructor in list of initializers. Using constructor with parameter will completely initialize filter with passed properties, no other method is needed to call.

If you follow

- Pointer to `Properties` structure is stored as protected member *`_properties`*, but its type is `AbstractFilter::Properties`. So if you want access members of your properties structure you must either cast to right type every time, or put preprocessor macro `GET_PROPERTIES_DEFINITION_MACRO` to private section of your class declaration. Now method `GetProperties()` returning reference to `Properties` is available for usage.
- To make declaration of Get/Set methods easier three macros are prepared: `GET_PROPERTY_METHOD_MACRO( TYPE, NAME, MEMBER_NAME )`, `SET_PROPERTY_METHOD_MACRO( TYPE, NAME, MEMBER_NAME )` and `GET_SET_PROPERTY_METHOD_MACRO( TYPE, NAME, MEMBER_NAME )`. These macros will be unwinded into inline declarations of `TYPE Get'NAME'()const`, `void Set'NAME'( TYPE value )` and both. Parameter *`NAME`* is used in name of Get/Set method and parameter *`MEMBER_NAME`* is name of `Properties` member accessed by these two methods.

```
template< typename ElementType >
class ThresholdingFunctor
{
public:
    void
    operator()( const ElementType&    input, ElementType& output )
    {
        if( input < bottom || input > top ) {
            output = outValue;
        } else {
            output = input;
```

```
        }
    }

    ElementType    bottom;
    ElementType    top;

    ElementType    outValue;
};

template< typename ImageType >
class ThresholdingFilter
: public AbstractImageElementFilter<
    ImageType,
    ImageType,
    ThresholdingFunctor< typename ImageTraits< ImageType >::ElementType >
    >
{
public:
    typedef ThresholdingFunctor
        < typename ImageTraits< ImageType >::ElementType >     Functor;

    typedef Imaging::AbstractImageElementFilter
        < ImageType, ImageType, Functor >                      PredecessorType;

    typedef typename ImageTraits< ImageType >::ElementType     InputElementType;

    struct Properties : public PredecessorType::Properties
    {
        Properties(): bottom( 0 ), top( 0 ), outValue( 0 ) {}

        InputElementType    bottom;
        InputElementType    top;

        InputElementType    outValue;

        void
        CheckProperties() {
            _functor->bottom = bottom;
            _functor->top = top;
            _functor->outValue = outValue;
        }

        Functor    *_functor;
    };

    ThresholdingFilter( Properties  * prop );
    ThresholdingFilter();

    GET_SET_PROPERTY_METHOD_MACRO( InputElementType, Bottom, bottom );
    GET_SET_PROPERTY_METHOD_MACRO( InputElementType, Top, top );
    GET_SET_PROPERTY_METHOD_MACRO( InputElementType, OutValue, outValue );
protected:
```

```
private:
    GET_PROPERTIES_DEFINITION_MACRO;

};

template< typename ImageType >
ThresholdingFilter< ImageType >
::ThresholdingFilter()
    : PredecessorType( new Properties() )
{
    GetProperties()._functor = &(this->_elementFilter);
}

template< typename ImageType >
ThresholdingFilter< ImageType >
::ThresholdingFilter( typename ThresholdingFilter< ImageType >::Properties *prop )
    : PredecessorType( prop )
{
    GetProperties()._functor = &(this->_elementFilter);
}
```

2.3.3. Defining new dataset type

- Designing dataset interfaces and synchronization system
- Implementing connection objects and input/output ports
- Creating filter base classes working on this dataset type

Chapter 3. DICOM Client library

This class is gate to DICOM world. DICOM is a standard describing creating, storing, manipulating and many more actions taken on medical data. For full description informations see (http://www.dclunie.com/dicom-status/status.html). Actual standard is 'little' bigger because it decribes all possible scenarios of all possible types. So when you want to get inside the mather you can read only some parts of it. There are those parts reffered in the code of particular parts.

Data are stored in files (*.dcm). The files are divided into pairs of key and value. There is dictionary of keys. The keys changes with every new specification of the stadard (yearly). There are set of commonly used types of files like Images, Waves of sound, CardioGrams, ... Each type has its own set of mandatory or optionaly tags to be filled when created on a device. Most important tag is the one that contains actual data of the image. Although every DICOM file has set of common tags. Theese are used to identify the file. There are 4 levels of entities that identifies the files. These are: patien, study, serie, image. So to fully identify one image 4 IDs must be provided.

On these tags DICOM server quering is performed, where data are normally stored. There are many implementations of that server. Most common is PACS server. Since there is no universal language SQL like system, actual quering is performed through dicom files. Filled tags means 'SQL WHERE clause', tags provided with no value defines 'wanted columns'. Although DICOM provides variety filetypes, we have focused on filetypes that provides images. Most used are: CT (Computer Tomography), MR (Magnetic Resonator), RT (rentgen). DICOM server implements searching (C-FIND: query file is send and resulting files are recieved), moving (C-MOVE: query file is send like in searching but with already known ID params that identify specific image or set of image. Then particular image files are recieved), storing (C-STORE: new image with unique generated IDs is send, not yet implemeted) functionalities.

Library also provides local filesystem searching funcionality. This is used when actual data are stored on filesystem.

The library uses 3rd party toolkit DCMTK by DICOM@Office (dicom.offis.de) for manipulation with DICOM files (reading, saving, server quiering).

3.1. DCMTK

DCMTK (DICOM ToolKit) is public domain library for handling DICOM files. It is written in C++. That is the main reason we chosed it to use. Another reason was that it implement some example programs that you can compose working DICOM server. We use them to have a DICOM server and be able to test with it. Library also contains some programs that serves as DICOM client. From that programs our DICOM client code was derived.

That was definitely pros but there are some cons. For instance there are networking absolutely hidden to library user. So we had to use in manner the DCMKT authors wanted. But we want some scather-gather

like behaviour to be able to make the incoming data be written to directly to one big memory array that is essential for image-processing algorithms operating upon the data. It was impossible to do that due to the networking hiding and impossibility knowing DICOM file sizes befor actual data are recieved that would be needed to scater-gather ops. So we agreed to following scenario: Whole file is recieved. Than type of the elemens as well as image sizes are read from dataSet (see next) to set up the places in memory where the data should go. Then actual data is copied to the place from dataSet. And then are deleted from it. So whole dataSet with all atributes about the image but actual data remains (’empty bottle’).

DataSet is map-like container for atrubutes (key-value pairs) that the DICOM file is composed from. When a file is being retrieved from server or from filesystem, the dataSet is build automaticaly by the DCMTK library. The building is atomical so no posibillity of knowing some attributes befere others (discussed above).

3.2. Compilation

3.2.1. Dependencies

Common library

3.3. Architecture

Public declarations are in namespace M4D::Dicom while private (not visible from outer) are in M4D::DicomInternal. Whole design of class hierarchies is Figure 2-1

3.3.1. DcmProvider

`DcmProvider` is the class all DICOM functionality is provide from. It has methodes for comunication with DICOM server and files or fileSets manipulation (C-FIND, C-MOVE, ...) as well as searching local fileSystem folders. It also contains basic class that represents one DICOM file, `DicomObj` and some structures representing found results such as TableRow for view found information in table.

3.3.2. DicomObj

Represents one DICOM file. When retrieving whole serie, then the DicomObjs are stored in vector (DicomObjSet). It has methodes for Saving and Loading to filesystem. As well as method for copying data to defined place when ’overspilling the bottle’. It also contains methodes to retrieve basic information from dataSet like width height, element’s data type, ...

3.3.3. DicomAssociation

This is base class for DICOM assotiation. Assotiation is something like connection. It is defined by IP address, port and 'Application Entity' (AE). AE is like name. Both sides (client and server) has its own AE. This class contains pointers to DCMTK library objects that contains actual assotiation and his properties. As well as some action members that establish (request), aborts and terminate the assotiation. Next item contained in this class is address container that holds neccessary properties for different assotiation. Assotiation has different properties when different services are called. So the container is a map indexed by string each called service. The container is filled from config file. There are some supporting methodes taking care of it. The container is shared between all instances (static).

3.3.4. AbstractService

This is base class for all services that is requested to the side of DICOM server. There is poiner to DCMTK Network object which need network subsystem on windows system initialized at the beginning of usage and unloaded when is no more needed. So there is reference counting.

Each service is devided into 2 parts. SCP (Service Class Producer = server) and SCU (Sevice Class User = Client). Both sides of an service has to agree while establishing assotiation what role to play. Normally DICOM server plays server role (SCP) for C-FIND, C-MOVE, C-STORE but for C-MOVE subassotiations when image are transfered to client plays SCU role (it requests sending of data). Each scenario is described later in doc of successors. Another class member is query dataSet that is used as a query to server, similary like SQL query string. Each query can be done on one of 4 levels: Patient, Study, Series, Images. For example: for Study level are all matched studies returned, for Series Level all matched series, ... On each level are relevant only sepecified set of matchable attributes so its quite hard to send robust query. Some other filtering has sometimes to be done on returned records.

Common scenario of all services is to prepare query dataSet that selects wanted data files. Then proceed the query to main TCMTK performing function and then retrieve resulting data through callbacks to final data structures. Ancesting classes implementing the services contain supporting callback definitions that cooperated with DCMTK functions and definitions of structures that are then passed to appropriate callbacks.

3.3.5. FindService

Implements C-FIND service to DICOM server. Process decription in a nutshell: client (SCU) establish assotiation to server (SCP) and sends query dataSet. Server process query dataSet and sends back matched results. For more details see DICOM doc ([ver]_08.pdf chapter 9.1.2) and coresponding annexes).

3.3.6. MoveService

Implements C-MOVE service to DICOM server. Its purpose is to move data files (records) from DICOM server. There are two main functions that retrive data files. One retrive one SINGLE image. The image is specified by unique IDs on all levels (patient, study, serie, image). The other retrieve all images from specified serie (SET). That means specification of IDs on all levels but image. Process decription in a nutshell: Client (SCU) establish assotiation to sever (SCP), send query dataSet, server find matching image files, then establish another subassotiation (as SCU) with calling client (that plays SCP role) and transmit data files over the subassotiation. For more details see DICOM doc ([ver]_08.pdf chapter 9.1.4) and coresponding annexes).

3.3.7. StoreService

Implements service that performs C-STORE operatoin to DICOM server. It's purpose is to generate unique IDs and send new data to server. Behavior in a nutshell: Client (SCU) generates unique ID for sent (new) data, establish assotiatoin with a server (SCP) and sends the data to server. For more informations see DICOM doc ([ver]_08.pdf chapter 9.1.1) and coresponding annexes). Generation of unique IDs is based on prefix and the rest is delegated to DCMTK functions. More informations about unique UDs generation see DICOM doc.

3.3.8. LocalService

Implements searching and gettting functions to local FS dicom files. It sequentialy loads data files in specified folder (and subfolders through queue), read ID info, based on that info and given filter inserts or not inserts (if matching found) record into result. Each search run writes the folder that is performed on, build structure of information that is used when aditional informations concerning data from the same run are required. One run is quite expensive while loading each file is needed (there is no other way how to read required IDs). So it is sensitive how wide and deep is the subtree of the given folder. Maybe some timeouts will be required. All functions are private beacause are all called from friend class DcmProvider.

3.4. Usage

Usage is quite simple. First we have to construct DcmProvider instance. Then we only create objects needed for holding data (ResultSet) and issue them along with some other parametrs to member functions of DcmProvider class instance created at the beginning. Example follows:

```
#include "dicomConn/DICOMServiceProvider.h"


M4D::Dicom::DcmProvider dcmProvider;  // DICOMProvider instance
// ....
```

```
// ....
// somewhere when finding record based on some filter form values
M4D::Dicom::DcmProvider::ResultSet resultSet;   // create resultSet container
dcmProvider.Find(     // issue filter values and resultSet to provider method
  &resultSet,
  firstName,
  lastName,
  patientID,
  fromDate,
  toDate,
  referringMD,
  description
  );

// now we have in resultSet (vector) all found items ...
```

Chapter 4. Remote computing (CellBE library)

Library used to send some parts of pipeline to remote mashines to be executed and result sent back. The name is from Cell Broadband Engine architecture name comming from IBM. This architecture contains also supercomputers like Blade servers as well as not so huge system like the one in PlayStation3 console. CellBE should originally be the one that have be target of remote computing because there is some mashines available for testing on faculty. But This library is not bound to specific architecture. It was one of primary request to be platform indemendet. It has to eliminate such scenarios that is usual in hospitals: 'Here you have the CT mashine with software that can work and communicate ONLY with our supercalculating server that is budled with. All for such small price of few milions'.

4.1. Compilation

4.1.1. Dependencies

Imaging lib

4.2. Architecture

All declarations are in namespace M4D::CellBE. System is classic Client-Server architecture. The client parts is in the library while the server part in another project. Purpouse of the library is to be linked with main program that has to have ability of remote computing. The both parts have some parts in common that reflects class hierarchy. Figure 2-1

4.2.1. Job

Job is entity that represents remote computation. Job has 2 main parts: Definig Contatiner and input&output dataSets. Definig Contatiner contains information about filters that the remote pipeline represented by this job will consist of and their settings. Each filter has Properties inner class. Instance of that class give us all neccessary information of that filter because it is teplated as well (it's inner class of templated class) and has all properties of filter that represents. So we don't need to instatiate any filter when defining remote pipeline. Filter::Properties instances are satisfied. According these information is actual pipeline created on ther server side. Input&output dataSets are actual dataContainers that are dataRead from and send to server (input) and recieved and written to (output). When you look at the class hiearchy, you can see there are commons parts for job on client and server side. But both 'types' of job has different behaviour when on client side and server side. So 2 branch in hierachy reflects that. Each hierarchy member will be described now in more details:

4.2.1.1. BasicSocket

Base class containing network functionality. Used BOOST::Asio for asynchronous networking and scatter gather. It contains pointer to socket that all the communication is performed through.

4.2.1.2. ClientSocket

Client has to be able to establish connection to server. This class can do it.

4.2.1.3. ClientJob

Now we fucus on particular branches of the hierarchy. Lets start with client one.

This class advances functionality for special for client job behaviour. That is ID generation, sending definition container and his content. It has also supporting members special for client async operations as well as members that do them.

4.2.1.4. ServerJob

The last class in Job hierarchy remains. This class form server side brach.

Is opposite to ClientJob. It’s purpouse is to handle operation that the client job performs (definition container unpacking and building pipeline and creating some supporting structures that are used in re-recieving definition container content. Will be dicused later on)

4.2.2. Serializers

Because this system (or library) is closely related to Imaging library that we wanted to let independent to any other library we have to solve how to perform serialization on the objects from the Imaging library that are mostly templated. Serializers object was developed.

Serializer is object that can perform serialization of its content. We have 2 types of serializers. These are FilterSerializers perform Serialization of Filter::Properties classes. Used in sending definitioin container. And DataSetSerializers. These performs dataSet serialization. Each Serializer performs serialization of 2 main types. First is class info serialization (CIS). In this stage is template parameters along with class type serialized. These information should be enough for other side for creating appropriate templated class instance. The second serialization tier is for actual content serialization (ACS) of entities (properties for filterProperties, dataSet atributes for dataSet).

The whole library is closed system that should not be chaged. Only these Serializers are subjects to edit. Serializers reflects hieararchy of objects in Imaging library. So when some new item in Imaging library appears and its author wants it to be able to use it remotely so new serializer has to written and registered. Registering is done in according type of GeneralSerializer, that serves as recognizer of particular serialiers. Details of creating new Serializer will follow later on. Now we will discuss each type of serializer in more details.

4.2.2.1. FilterSerializer

Each filterSerializer (FS) should be derived from AbstractFilterSerializer class which is the base class for every FS and defines interface of its behaviour. CIS is performed by SerializeClassInfo & DeSerializeClassInfo pair of member functions. ACS by SerializeProperties & DeSerializeProperties pair.

4.2.2.2. DataSetSerializer

DataSetSerializer has more work to do than FilterSerializer. It performs CIS (within GeneralDataSetSerializer and appropriate switch cases, desribed later) and ACS in SerializeProperties & DeSerializeProperties pair of functions that are in interface defined by AbstractDataSetSerializer that each new DataSet should inherit.

New is Actual Data Serialization (ADS). This is performed through another part of AbstractDataSetSerializer iterface parts: Serialize, OnDataPieceReadRequest, OnDataSetEndRead functions. For more detail of these function purpouse see appropriate headers.

Main idea of dataSetSerialization is to divide the whole dataSet into smaller parts, that can be transported throug network separately (dataPieces, DP) and alternatively the calculation can be started on independently from other DP that have not yet arrived. Each DP has its header that says how long it is. So on reciving side when such header is recieved its passed to DataSetSerializer's OnDataPieceReadRequest method that will decide where the data that the DP reffers to are going to be recieved. This can tell through DataBuffs vector by putting DataBuf sctruct into it. DataBuf is struct of void * pointer pointing somewhere into memory and size. So the DataBuffs vector defines place where data of DP should be transmitted (in geather manner).

Another mandatory function is OnDataSetEndRead. This is called when whole dataSet is recieved. No more DP are going to be recieved. This is recognized by recieving of special DBHeader. Function can perform some clean up or something.

Serialize is another mandatory function. It has to perform actual whole ADS. It gets pointer to iPublicJob interface and through its function PutDataPiece should perform serialization. PutDataPiece is the same as in deserialization but it writes data into network in scatter manner. In this function the programer should define meaning of DP and reflect the meaning in deserialization functions. Perfect example is serialization of 3d image. DP is one slice. So whole dataSet is sent with slice granularity. On the other

side DP (slice) is recieved directly into the right place in big array for 3d image and calculation can be started if another slices not needed. This is ideal case. Current implementation do not allow it due to dataSet locking and lack of universal dataSet handling. So dataSet is currently transferred whole and whole recieved.

Because this serialization is statefull the inner state has to be in serializer instance. Reset mandatory method should reset the state for seralizer reusage.

4.3. Usage

4.3.1. FilterSerializer

Next is the list of neccessary steps for adding new FilterSerializer:

- Add enum FilterID item in cellBE/filterIDEnums.h header
- Write actual FilterSerializer derived from AbstractFilterSerializer and implementing its interface. Right place where the code to place is cellBE/FilterSerializers folder
- Add the new Serializer instance into array in FilterSerializerArray class constructor and appropriate edit size of the array (all described in code and by next example that is taken from code. For details see ThresholdingSerializer.h & .tcc)

```
// supportig function
template< typename ElementType, unsigned Dim >
void
CreateThresholdingFilter(
     M4D::Imaging::AbstractPipeFilter **resultingFilter
   , AbstractFilterSerializer **serializer
   , const uint16 id
   , M4D::CellBE::NetStream &s )
{
 typedef typename M4D::Imaging::Image< ElementType, Dim > ImageType;
 typedef typename M4D::Imaging::ThresholdingFilter< ImageType > Filter;
  typedef typename FilterSerializer< Filter > FilterSerializer;

 Filter::Properties *prop = new Filter::Properties();

 *resultingFilter = new Filter( prop );  // id
  *serializer = new FilterSerializer( prop, id);  // id
}

//////////////////////////////////////////////////////
template< typename InputImageType >
class FilterSerializer< M4D::Imaging::ThresholdingFilter< InputImageType > >
```

```
 : public AbstractFilterSerializer   // inheritation from AbstractFilterSerializer
{
public:
  // typedef to Properties of filter this serializer is for. Just for simplier usage
 typedef typename M4D::Imaging::ThresholdingFilter< InputImageType >::Properties Prope

 // ctor - subject of customization, must call typed ancestor ctor
 FilterSerializer( Properties * props, uint16 id)
 : AbstractFilterSerializer( FID_Thresholding, id )
 , _properties( props )
  {}

  // member performing CIS
  void SerializeClassInfo( M4D::CellBE::NetStream &s)
  {
    s << (uint8) ImageTraits< InputImageType >::Dimension;
 s << (uint8) GetNumericTypeID< ImageTraits< InputImageType >::ElementType >();
  }

  // member performing CIS deserialization
  void
  DeSerializeClassInfo(
      M4D::Imaging::AbstractPipeFilter **resultingFilter
    , AbstractFilterSerializer **serializer
    , const uint16 id
    , M4D::CellBE::NetStream &s
    )
  {
    uint8 dim;        // read from netStream
 uint8 typeID;
 s >> dim;
 s >> typeID;

 // build apropriate classes based read info with assistance of CreateThresholdingFilt
 // templated function defined above
 NUMERIC_TYPE_TEMPLATE_SWITCH_MACRO( typeID,         // usage of special macro. See Co
 DIMENSION_TEMPLATE_SWITCH_MACRO(
        dim, CreateThresholdingFilter<TTYPE, DIM >(
          resultingFilter, serializer, id, s ) )
 );
  }

  // pair of members performing ACS
 void
 SerializeProperties( M4D::CellBE::NetStream &s)
 {
 s << _properties->bottom << _properties->top << _properties->outValue;
 }
 void
 DeSerializeProperties( M4D::CellBE::NetStream &s )
 {
 s >> _properties->bottom >> _properties->top >> _properties->outValue;
 }
```

```
protected:
 Properties *_properties;   // pointer to properties (actual content)
};

////////////////////////////////// ...

// within FilterSerializerArray constructor .... creation of buddy instance of our new
// serializer class on place within the array defined by our new enum member (step 1)
m_serializerArray[ (uint32) FID_Thresholding] =
  new FilterSerializer< typename ThresholdingFilter< Image<uint8, 3> > >(
    NULL, 0 );

// .....
```

4.3.2. FilterSerializer

Next is the list of neccessary steps for adding new DataSetSerializer:

- Add enum DataSetType item in Imaging/dataSetClassEnum.h header
- Write actual DataSetSerializer derived from AbstractDataSetSerializer and implementing its interface. Right place where the code to place is cellBE/DataSetSerializers folder
- Register new dataSetSerializer by adding new switch case of new enum member (step 1) into GeneralDataSetSerializer::GetDataSetSerializer method and writing the content of switch case that instatiates DataSetSerializer based on inner properties (like TypeID, Element type, dimension. Properties of image). And another switch case in GeneralDataSetSerializer::DeSerializeDataSetProperties method that on the other side can instantate DataSetSeralizer class (CIS).