Note 250: *livedata* - software notes

Mark Calabretta, ATNF

2002/06/14

1 Overview

livedata is the realtime data-processing pipeline used by the Multibeam system at Parkes. It is also used for the 4-beam system at Jodrell Bank, and is intended to be used for the single beam system at Mopra.

A coordinator at the heart of *livedata* directs and regulates the flow of data between the following six independent data reduction clients:

Reader →			
	→ Bandpass calibrator →		
		→ Data monitor	
		→ Statistics	
		→ Writer	→ Gridder

Data flow is from left to right, and all data reduction clients other than the reader may be disabled or short-circuited, i.e. any line(s) of the above may be deleted other than the first. The gridder is dependent on the writer since it processes the output file; it is fed via a special purpose queue and once started may remain active even after the writer has been disabled.

Reader: reads data from an `MBFITS` format file (the single-dish variant of `RPFITS`), an `SDFITS` file, or an `aips++` measurementset (`MS2`). The input data, which may be selected on a beam-by-beam basis, is packaged up and passed to the next client in the chain.

Bandpass calibrator: does most of the work in calibrating Multibeam data. It has several modes of operation which it selects automatically based on the method of observation:

a) Scanning modes as used in `HIPASS` and `ZOA`.
b) Frequency switching modes.
c) Position switching modes where each element of the Multibeam system is pointed at the source in turn.
d) Extended source modes such as used for High Velocity Clouds.

Each of these modes uses a separate bandpass calibration strategy. These are based on robust statistical estimators, particularly median estimators, that allow rejection of RFI without human intervention.

Data monitor: interfaces to *MultibeamView*, a specially modified version of the *kview* karma application. In fact, it invokes *MultibeamView* twice, once for each polarization, to provide two panels displaying frequency versus time with various image enhancement options. This provides visual inspection for each pair of polarizations of the 13 beams, one pair at a time.

Statistics: Computes and displays basic statistical measures for the spectra from each beam and polarization as a function of time as an observation progresses, and computes global statistics once the observation has finished. Typically it would be used to monitor T_{sys} for each IF.

Writer: data may be written either in `SDFITS` format or as an `aips++` measurementset by the writer client.

Gridder: Each Multibeam integration consists of a spectrum taken at a particular point on the sky. The gridder takes the spectra from a collection of bandpass-calibrated observations and interpolates them onto a regular map grid, writing the result to a `FITS` data cube as it goes. Typically each point in the sky is sampled many times in separate observations and the gridder combines these using a choice of statistics, including robust (median) statistics that suppress radio-frequency interference (RFI).

Being the most computationally intensive client, the gridder is usually used separately for offline data reduction. However, it can be linked into the *livedata* reduction chain, though, unlike the other clients, it may run remotely on another machine in a quasi-batch mode.

The gridder is only loosely coupled to the *livedata* pipeline. In fact, it is most often used in stand-alone mode since it is usually necessary to combine the output of many observations to produce the final data cube.

livedata operates in near-realtime at the Parkes telescope to allow immediate assessment of data quality. However, it may also be used for offline data processing.

2 Implementation

In the following, the term *agent* refers to a Glish "agent" implemented as a Glish subsequence; these are defined via a Glish script (in a `.g` file). These processing entities exist within Glish, they do not have a separate unix `pid` (process identification number).

The term *client* refers to a `C++` program (in a `.cc` file) written using the Glish interface classes and invoked via Glish's `client()` function by the Glish agent. These clients are separate autonomous unix processes with their own `pid`.

Each of the six *data reduction clients* described above is implemented as a `C++` client with a separate controlling Glish agent. They are referred to simply as *clients* where there is no ambiguity or need to distinguish.

2.1 Agents

Glish agents in *livedata* have the following main functions:

Control: they are primarily responsible for invoking and controlling the `C++` clients.

Validation: they validate all parameter values and enforce certain restrictions on them in particular processing modes; the `C++` clients themselves do minimal parameter checking.

GUI: parameter values may be set in a number of ways as described below. One method is via an optional graphical user interface provided by the Glish agent using `glish/tk`. The statistics client also provides a graphical display window using the `glish/PGPLOT` interface.

Being a loosely typed language, Glish is capable of some very powerful constructs but at the same time allows great scope for disastrous programming errors. A highly formalized style of Glish programming helps to alleviate this in *livedata*:

- The agents are implemented as Glish subsequences. A subsequence is essentially a Glish function which returns an agent variable created via Glish's `create_agent()` function; this variable may be referred to within the subsequence itself by the name *self*.

  Locally scoped data and function definitions within a subsequence provide a natural form of encapsulation, and the agent variable returned from a subsequence invokation may be treated much like an object instantiation. Communication with these objects is via Glish events tagged to this agent variable. Since agent variables are implemented as Glish records it is also possible to store public information in them in separate fields.

- Each agent contains a Glish record called `parms` which contains operating parameters. These are often data processing options intended for the `C++` client and usually correspond to GUI widgets.

  Parameter values are *always* set by a function called `setparm()` whose single argument is a record variable. Each field name in this record identifies the parameter and the field value is the new parameter setting.

  `setparm()` first validates the parameter value using a global function called `validate()`. Validation rules are defined in a record variable conventionally called `valid` which is passed to this function.

  `setparm()` implements a simple form of polymorphism via a set of functions contained within the `set` record variable, the fields of which match parameter names. When called upon to set a parameter value, `setparm()` first checks to see if `set` contains a specialized function and if so invokes it. These functions are *never* invoked by anything other than `setparm()`.

  Each agent recognizes a record-valued `setparm` event which invokes the `setparm()` function on the record. Likewise, the `init` event also accepts a record argument to set parameters before it initiates processing. They also recognize `printparms` which simply prints the value of the `parms` record.

  Parameter values may be set by any of the following methods

  1. Via the subsequence arguments.
  2. Via the `setparm` event.
  3. Via the `init` event.
  4. Via the graphical user interface.

  Each of these methods invokes `setparm()` to change the parameter value.

- An important part of the design of the Glish agents is that the GUI they provide is optional, and is only one of several methods of setting parameter values.

  Furthermore, although the GUIs for the data reduction clients are normally collected together into a single *livedata* GUI, each is completely autonomous. This design simplifies the development and debugging of a complex GUI.

  The GUI is always constructed by a function called `showgui()` which takes the parent frame as an optional argument; this mechanism allows the separate GUIs to be combined as separate panels within in a single window. If the parent frame is not provided a new top-level frame (i.e. window) is created. Agent variables for the GUI widgets are always contained in a record called `gui`.

  Whenever `setparm()` changes a parameter value it invokes `showparm()` which reflects the changed value in the GUI (if it is active). `showparm()` is a global function with two arguments; the first is `gui`, and the second is a record of parameter values, each field, `item`, of which identifies a widget as

  `gui[item].bn` ... button

  `gui[item].en` ... entry box

  `gui[item].sv` ... label (status value)

  `gui[item].lb` ... listbox

If `gui[item].show` is defined it is taken to be a specialized function for displaying the parameter. Otherwise `showparm()` does a generic widget update.

Aside from the widget updates performed by `setparm()`, widget initialization is the *only* other valid place where `showparm()` may be invoked; this is almost invariably done by `showgui()` via `showparm(gui,parms)`.

- The following global functions are used by most *livedata* agents:

  `showparm(gui,parms)` – Updates widget(s) associated with parameter values if the GUI is active.

  `validate(valid,parms,value)` – Parameter validity checking.

- The following local functions are common to most *livedata* agents:

  `helpmsg(msg='')` – Write a widget help message.

  `sethelp(ref widget, msg='')` – Set up the help message for a widget.

  `setparm(value), set(value)` – Update parameter values, also updating any associated widget(s) using `showparm()` if the GUI is active.

  `showgui(parent=F)` – Build a graphical user interface for the client.

  `readgui()` – Read values from GUI entry boxes.

- The following received events are common to most *livedata* agents:

  `setparm(record)` – Set parameter values.

  `setconfig(string)` – Set configuration (a particular set of parameter values).

  `printparms()` – Print parameters values.

  `lock()` – Disable parameter entry.

  `unlock()` – Enable parameter entry.

  `init(record)` – Initialize the client; parameter values may optionally be specified. The client will respond with an `initialized()` event (see below).

  `showgui(agent)` – Create the GUI or make it visible if it already exists. The parent frame may be specified.

  `hidegui()` – Make the GUI invisible.

  `terminate()` – Close down.

- The following sent events are common to most *livedata* agents:

  `done()` – Agent has terminated.

  `log(record)` – Log a message.

  `initialized(record)` – Sent in response to an `init` event once the client has been initialized.

  `fail()` – Client has terminated unexpectedly.

The following annotated example illustrates these concepts:

```
grus% glish -l pksbandpass.g
Glish version 2.7.
- bandpass := pksbandpass(smoothing='Hanning', prescale_mode='median')
- print bandpass
[*agent*=<agent>, name=pksbandpass]
-
```

`bandpass` is the agent variable for this instantiation of the bandpass calibration data reduction client. The agent name, `pksbandpass`, was included in the agent record by the subsequence and this illustrates the mechanism for transferring public information out of the subsequence. Note that the values of the `smoothing` and `prescale_mode` parameters were set in the subsequence invokation.

```
- bandpass->printparms()
- [config=general, client_dir=, smoothing=Hanning, prescale_mode=median,
   fit_order=0, velocity_frame=BARY, rescale_axis=T, src_size=compact,
   estimator=median, bp_recalc=4, nprecycles=24, npostcycles=24,
   boxsize=20, nboxes=5, maxcycles=250, fast=T, check_field=T,
   check_time=T, tmin=0, tmax=300, tjump=20, check_position=T,
   dmin=15, dmax=300, djump=10]
-
```

This illustrates the general method of communicating with a Glish agent, in this case, the `printparms` event instructs it to print the value of the `parms` variable containing parameter values.

```
- bandpass->showgui()
- bandpass->setparm([smoothing='Tukey'])
-
```

Instructs the agent to construct a GUI and then resets the `smoothing` parameter to `'Tukey'`; the new value is reflected in the GUI. Alternatively, the GUI itself could have been used to reset the value and then `printparms` would reflect the changed value.

```
- bandpass->setparm([smoothing='Nonsense'])
- Invalid parameter assignment:
   ignored [smoothing = Nonsense] (string type)
   remains [smoothing = Tukey] (string type)
```

An attempt to set `smoothing` to an invalid value results in a warning from `validate()` that the assignment was ignored.

2.1.1 Coordinating agents

The data reduction clients are coordinated by a Glish agent, the *reducer*, that directs and regulates the flow of data between them by means of Glish events. Data is transferred in the form of a Glish record via the event value. When the clients run on the same host, as is usual, the data transfer is virtual, it occurs by exchanging a pointer to a section of memory, and does not involve real I/O.

The basic unit of work for each data reduction client consists of processing one integration (typically 1024 spectral channels, 13 beams, 2 polarizations, and of 5 s duration). The operation of the *reducer* is completely asynchronous, once a client's output has been accepted by the next client(s) in the pipeline the *reducer* instructs it to process the next integration. The clients therefore run in parallel keeping the CPU fully occupied. In addition, the *reducer* manages the data reduction pathway as clients are enabled or disabled, and also maintains the input queue for the gridder. It is certainly the most complex piece of Glish programming in the *livedata* suite.

By default, when instantiated, the *reducer* instantiates a reader client but none of the others; these must be enabled specifically. The *reducer* includes the agent variables for these clients in its own agent variable thus allowing direct communication with them. For example

```
grus% glish -l livedatareducer.g
```

```
Glish version 2.7.
- ldr := reducer(bandpass=T, stats=T)
- print ldr
[*agent*=<agent>, name=reducer, busy=F,
 reader=[*agent*=<agent>, name=pksreader, open=F, busy=F, message=IDLE],
 bandpass=[*agent*=<agent>, name=pksbandpass, busy=F, buffering=F,
         message=IDLE],
 monitor=[busy=F, message=inactive],
 stats=[*agent*=<agent>, name=pksstats, busy=F, message=IDLE],
 writer=[busy=F, message=inactive],
 gridder=[busy=F, message=inactive]]
-
```

The output (which has been reformatted for clarity) shows that the reader, bandpass, and stats clients are active but idle, while the other clients are inactive. Now it would be possible to communicate with the bandpass client, say, via

```
- ldr.bandpass->setconfig('HVC')
- ldr.bandpass->printparms()
- [config=HVC, client_dir=, smoothing=Hanning, prescale_mode=none,
   fit_order=6, velocity_frame=LSRK, rescale_axis=F, src_size=extended,
   estimator=medmed, bp_recalc=4, nprecycles=24, npostcycles=24,
   boxsize=20, nboxes=5, maxcycles=250, fast=T, check_field=F,
   check_time=F, tmin=0, tmax=300, tjump=20, check_position=F, dmin=15,
   dmax=300, djump=10]
```

This feature may be used in batch scripts to set parameter values for each client.

An additional *scheduler* agent is added for interactive use. It provides a queueing mechanism that supplies input to the data reduction pipeline. It provides *livedata*'s near-realtime capability by automatically discovering `MBFITS` files as they are written by the Multibeam correlator. The *scheduler* includes the *reducer*'s agent variable as part of its own.

livedata may be run in batch mode by invoking the *reducer* directly (usually from a Glish script) thus bypassing the *scheduler*.

2.2 C++ clients

The `C++` clients perform the bulk of the computationally intensive parts of the data reduction. The reader and writer also utilize special-purpose object libraries such as `cfitsio` and `RPFITS` which are not directly accessible from Glish.

The source code for the `C++` clients has a less formal structure than the Glish agents. However, they do have a number of common features:

a) Global function pksmbSetup() performs generic Glish setup operations.

b) The `C++` clients are initialized via an `init` event which contains operating parameters in a Glish record. These are extracted from the event's value via the global getParm() overloaded functions.

c) Message logging is done via the global logMessage(), logWarning(), and logError() functions.

d) The reader, writer, and gridder clients use a common set of general-purpose IO classes, the object diagram for which is shown in Fig. 1.

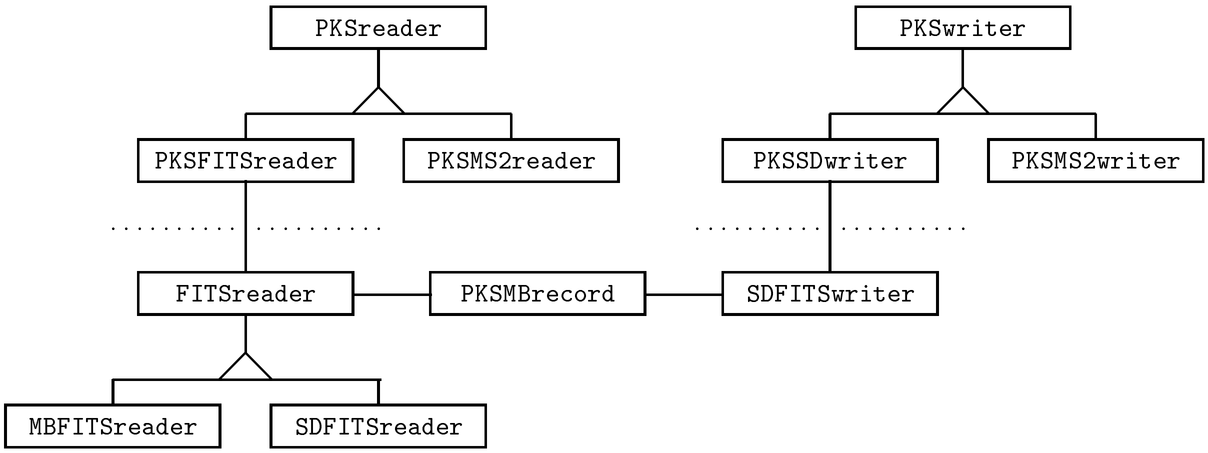

Figure 1: Class hierarchy of the `PKSIO` reading and writing classes. Classes below the level of the dotted line are independent of `aips++`; a stand-alone utility, `rp2sdfits`, uses `MBFITSreader` and `SDFITSwriter` to translate `MBFITS` data files to `SDFITS` format. `PKSMBrecord` is basically a data structure that stores an `MBFITS` single-dish data record.

Prototypes for the general-purpose global functions are declared in `pksmb_support.h` which resides in the `atnf/implement/pks` module, and the IO classes are defined in various files in `atnf/implement/PKSIO` module.

2.3 Files

livedata's main component files are as follows:

`livedata` Bourne shell script that initiates *livedata*. It checks that certain essential environment variables are correctly defined and starts Glish with `livedata.g` as input.

`livedata.g` Startup Glish script for *livedata*. It copies certain environment variables into Glish global variables and then creates a *scheduler* agent with GUI.

`livedatascheduler.g` Defines the *scheduler* agent (subsequence) with optional GUI for realtime (live) and offline data reduction. Realtime reduction differs from offline reduction in that newly created files may be discovered automatically (auto-queued) and that several attempts are made to read files which may be incomplete. The *scheduler*'s main job is to create a *reducer* agent and supply it with work.

`livedatareducer.g` Defines the *reducer* agent (subsequence) with optional GUI. This agent is *livedata*'s pipeline controller, it provides the glue that ties *livedata*'s data reduction clients together and regulates their activities.

`pksreader.g` Defines the *pksreader* agent (subsequence) with optional GUI.

`pksreader.cc` The *pksreader* client reads a Parkes Multibeam dataset and emits Multibeam Glish records. It will determine for itself whether the input is an `MBFITS` or `SDFITS` format file or an `aips++` measurementset (v2).

`pksbandpass.g` Defines the *pksbandpass* agent (subsequence) with optional GUI.

`pksbandpass.cc` The *pksbandpass* client applies bandpass calibration to Multibeam data. It has several modes of operation as described in Sect. 1.

`pksmonitor.g` Defines the *pksmonitor* agent (subsequence) with optional GUI. This agent also creates and manages two instances of *MultibeamView*, a version of the karma *kview* utility adapted for *livedata*, one for each polarization.

`pksmonitor.cc` The *pksmonitor* client prepares Multibeam data for display. It applies data selection, time averaging, and frequency smoothing options.

`pksstats.g` Defines the *pksstats* agent (subsequence) with optional graphical (`PGPLOT`) display window.

`pksstats.cc` The *pksstats* client accumulates data and computes running statistics of the spectra for each beam and polarization, for example, the mean, median, and T_{sys}. It passes these back to the *pksstats* agent for display. Finally it computes statistics for the observation as a whole from accumulated data.

`pkswriter.g` Defines the *pkswriter* agent (subsequence). This agent has no GUI of its own, the few parameters it requires are set in the *scheduler* GUI.

`pkswriter.cc` The *pkswriter* client writes data to file in either `SDFITS` or `MS2` format using the `PKSIO` writer classes.

`gridzilla.g` Defines the *gridzilla* agent (subsequence) with optional GUI. The gridder is most often used independently of *livedata* and a separate Bourne shell script, `gridzilla`, and startup Glish script, `gridzillarc.g`, are provided for this purpose.

`pksgridzilla.cc` The *pksgridzilla* client grids Multibeam data into a spectral data cube. Most of the gridding is actually done by the `MBGridder` class.

`pkslib.g` Defines global functions used by *livedata*.

Generally the `C++` client code resides in specific subdirectories of `atnf/apps` together with other files which define helper classes or template definitions.

The Glish scripts reside in `atnf/apps/livedata` as it is more convenient to have them in one place for debugging – Glish's search path finds modified files in the current directory before looking elsewhere. One exception is the gridder code which all resides in `atnf/apps/pksgridzilla`.

3 Batch processing example

The following Glish script uses *livedata* and *gridzilla* in batch mode to bandpass calibrate and grid a number of data files. It does create a GUI for each in order to display the processing parameters but this is not necessary for its operation.

```
include 'livedatareducer.g'
include 'gridzilla.g'

# Check that DISPLAY is defined.
if (!has_field(environ, 'DISPLAY')) {
  print 'DISPLAY is not defined - abort!'
  exit
}

# Check that AIPSPATH is defined.
if (!has_field(environ, 'AIPSPATH')) {
  print 'AIPSPATH is not defined - abort!'
  exit
}

# AIPS++ system area.
aipsarch := paste(split(environ.AIPSPATH)[1],
```

```
                  split(environ.AIPSPATH)[2], sep='/')


# Input directory.
if (has_field(environ, 'HVCDIR')) {
  read_dir := environ.HVCDIR
} else {
  read_dir := '/DATA/MULTI_5/mputman/hvcred'
  read_dir := '.'
}

# Output directory.
write_dir := spaste(read_dir, '/new')
if (!len(stat(write_dir))) shell('mkdir', write_dir)


# Instantiate a livedata reducer.
ldred := reducer(icon_dir   = spaste(aipsarch, '/libexec/icons'),
                 bandpass   = T,
                 monitor    = F,
                 writer     = T,
                 stats      = T,
                 read_dir   = read_dir,
                 write_dir  = write_dir)

# Create a GUI so we can see what's going on.
ldred->showgui()
t := client("timer -oneshot", 3.0)
await t->ready


# Configure the bandpass client.
ldred.bandpass->setconfig('HVC')


# Process a batch of files through livedata.
files := shell('cd', read_dir, '&& ls -d 2000-01-24_*.hpf')

for (read_file in files) {
  print 'Processing', read_file
  write_file := read_file ~ s|\.hpf|.mscal|
  ldred->start([read_file=read_file, write_file=write_file])
  await ldred->finished
}

# Finished with livedata.
print 'Closing down livedata...'
ldred->terminate()
await ldred->done


# Start the gridder.
files := shell('cd', write_dir, '&& ls -d *.mscal')
```

```
lsg := gridzilla(remote = T,
                 autosize = T,
                 pixel_width = 4,
                 pixel_height = 4,
                 rangeSpec = 'velocity',
                 startSpec = -508.0,
                 endSpec   =  680.0,
                 pol_op = 'A&B',
                 spectral_id = '0&1',
                 projection = 'SIN',
                 doEquatorial = T,
                 statistic = 'median',
                 clip_fraction = 0.0,
                 beam_weight = 0,
                 beam_FWHM = 14.4,
                 beam_normal = F,
                 kernel_type = 'top-hat',
                 kernel_FWHM = 6.0,
                 cutoff_radius = 6.0,
                 storage = 25,
                 directories = write_dir,
                 files = files,
                 selection = ind(files),
                 write_dir = write_dir,
                 p_FITSfilename = 'hvcred',
                 short_int = F)

# The optional GUI simply reflects what's happening.
lsg->showgui()
await lsg->guidone

# Start processing with these specific arguments.
lsg->go()
await lsg->finished
print 'Finished processing.'

exit
```