WIDAR Correlator
Backend Software Design

Martin Pokorny

Revision history

Revision	Date	Description	Name
0.4	19 Dec 2007	Placed pipelinefs layout description into its own section.	mpp
0.3	19 Dec 2007	Updated to include new message types handled by compute node application control socket and console app; added description of pipelinefs to head node software section.	mpp
0.2	19 Sep 2007	Changed names to match those in current WCBE project.	mpp
0.1.2	9 Aug 2007	Fixed up FIXME reference, updated lag set definition.	mpp
0.1.1	8 Aug 2007	Simple corrections and reformatting.	mpp
0.1	8 Aug 2007	Initial version	mpp

Contents

WIDAR Correlator Backend Software Design

1 Introduction

This document describes the design of the WIDAR correlator backend software, from general principles to design details. This is an evolving document, and should remain the most up-to-date source of information about the design of the backend software.

It is expected that the backend software will be used by both the EVLA and eMERLIN WIDAR correlators from the early development stage to the production stage (as well as WIDAR development and testing at DRAO), and therefore the design is based on a generic framework, allowing customized extensions as needed at every stage of development and deployment of any WIDAR correlator installation. The diversity of environments in which the backend software will be used has significantly affected the design of the software, in an attempt to meet the following design goals.

Design goals

flexibility

This is perhaps the goal with the greatest effect on the design of the backend software. Each correlator development or installation site may have unique requirements for the processing or output required of the backend, and development of backend functionality will be an evolutionary process keeping pace with requirements imposed by the correlator and other subsystem development and deployment processes. Flexibility in both software development and system configuration is a necessary quality of the backend software design.

robustness

Required to resist the effects of flexibility's evil twin, instability. While parts of the backend software require great flexibility to be useful, other parts, which may be described as *common functionality*, do not. The design must guard against failures in the common functionality that may be triggered by faults in the flexible parts.

The backend software, generally, is deployed on a computing cluster, consisting of a head node and one or more compute nodes. Backend cluster processes, and the software they run, are classified according to whether they execute on the head node or a compute node in a fully deployed cluster. A configuration in which the cluster consists of a single node will be common during development and early correlator deployment, but this configuration does not change the logical distinction of head and compute node processes and software.

2 Head node software

Design of the backend cluster head node software is incomplete. The only part of this software already designed and in existence is a monitor and control interface to the compute node pipeline processes.

2.1 Compute node pipeline monitor and control interface

The monitor and control interface to the compute node pipelines is based on the control interface described in **Section 3.4.1**. The interface on the head node has been enhanced somewhat from the interface described

in **Section 3.4.1** to provide greater simplicity as well as a single control path from the head node to all compute node pipelines. This interface effectively takes the form of a sort of "device browser", which is, unlike the usual device browser, integrated with the operating system on the head node through a special filesystem. The advantages of this design are that the message passing interface to the compute node pipelines is hidden by an interface that instead uses reading and writing files and directories on the head node, and that the interface is readily available to all users (human and otherwise) on the head node by virtue of its integration with the operating system. By this design, the API used by monitor and control programs on the head node is a subset of the standard `libc` library functions, such as `read`, `write`, `creat`, `rmdir`, *etc.*

The program that implements the interface is named **pipelinefs**, which is also the name of the filesystem that the program creates. **pipelinefs** is based on the **FUSE** library, which is used to implement filesystems in userspace under Linux (and other operating systems).[1] **pipelinefs** itself is written in the Python programming language, which was chosen mainly for reasons of expediency; the program could be reimplemented in another language, if that became desirable, without requiring *any* changes in programs that use the `pipelinefs` filesystem interface.

2.1.1 Layout of `pipelinefs` filesystem

An outline of the `pipelinefs` filesystem structure follows.

- mountpoint
  - compute node pipeline instance 1
    - ⋆ `flow`
    - ⋆ `name`
    - ⋆ `src`
    - ⋆ `live_ts`
    - ⋆ `rate`
    - ⋆ `active`
    - ⋆ `configurations`
      - ▷ `pending`
      - ▷ configuration 1
        - ○ `status`
        - ○ `pipelines`
          - ○ pipeline 1
            - ○ element 1
              - ○ `pads`
                - ○ pad 1
                - ○ `caps`
                - ○ `peer`
                - ○ pad 2

[1] FUSE has been adopted by the official Linux kernel starting at version 2.6.14. If you are using an earlier kernel version, FUSE likely will work (see the FUSE web site for details about usable kernel versions), but you will need to download, build and install it yourself. Note that FUSE depends on a loadable kernel module, which probably requires superuser access to install on your system.

```
                    ○ ...
                ○ ...
            ○ properties
                ○ property 1
                ○ property 2
                ○ ...
        ○ element 2
            ○ ...
        ○ ...
    ○ pipeline 2
        ○ ...
    ○ ...
▷ configuration 2
    ○ ...
▷ ...
– compute node pipeline instance 2
  ⋆ ...
– ...
```

The following list provides descriptions of the entries appearing in the outline above.

mountpoint

The `pipelinefs` filesystem mount point.

compute node pipeline instance 1

Local name of compute node pipeline instance.

`flow`

Pipeline flow control; equivalent to **WCBE_FLOW_MSG** message. The values that can read from or written to this file, together with their WCBE_FLOW_MSG value equivalents are

- "on": WCBE_FLOW_ON
- "off": WCBE_FLOW_OFF

`name`

Pipeline instance (remote) name; equivalent to **WCBE_NAME_MSG** message. This is a read-only file because the (remote) instance name may not be changed.

`src`

Pipeline lag frame source; equivalent to **WCBE_SRC_MSG** message. This file's content is the URI of the pipeline's lag frame source.

`live_ts`

"Live" timestamp flag; equivalent to **WCBE_LIVE_TS_MSG** message. This file only exists if the pipeline's lag frame source is a BLF file. The strings "true" and "false" are the only valid strings for this file's content.

`rate`

Frame playback rate; equivalent to **WCBE_RATE_MSG** message. This file only exists if the pipeline's lag frame source is a BLF file. The file's content is a string representation of the frame playback rate in frames per second.

`active`

Symbolic link to directory for the active configuration (under `configurations` directory). This symbolic link only exists when one of the pipeline's configurations is in the "active" state.

`configurations`

The directory containing the pipeline's configurations.

`pending`

The pending configurations actions for the pipeline. This is a read-only file, which exists only when there are pending configuration actions. Each line of the file's contents is a string comprising a configuration action and a configuration name (or action target). The pending actions are listed in the order in which the actions were created. The valid configuration actions are in the following list: "create", "activate", and "destroy".

configuration 1

Local configuration name.

`status`

Pipeline status. This file's content is either a list of pending configuration actions to be applied to the configuration, or the current configuration status. The valid pending configuration actions are in the following list: "pending-create", "pending-activate", and "pending-destroy". The valid configuration status values are "active", "inactive", "active-retired", and "inactive-retired". The only valid string that may be written to this file is "activate" (but one should not expect to read back the same value).

`pipelines`

The directory containing the configuration's GStreamer pipelines.

The pipeline names under this directory are given by the names specified in the configuration. However, one pipeline, INPUT_STAGE, is never given explicitly by a configuration, and will only appear under one configuration of each compute node pipeline instance at any given time. The INPUT_STAGE pipeline corresponds, unsurprisingly, to the GStreamer pipeline in the input stage of each compute node pipeline.

pipeline 1

GStreamer pipeline name.

element 1

GStreamer element name.

`pads`

Directory containing the element's pads.

pad 1

GStreamer pad name.

`caps`

Value of pad's negotiated capabilities (read-only).

`peer`

Pad's peer (read-only). The name of the peer is provided in the format [element name]/[pad name] where `element name` is the name of an element in the same pipeline as the current pad, and `pad name` is the name of a pad in the named element.

`properties`

Directory containing the elements GStreamer properties.

Note that the values of all the element properties appearing in the `pipelinefs` filesystem are given in string representations. The actual property value types are provided through an extended attribute of each file in this directory.[2] The name of the extended attribute used for the property type metadata is "user.edu.nrao.widar.gtype"; the values of this attribute are GType names.

property 1

Element property name

2.1.2 Usage

2.1.2.1 Mounting `pipelinefs`

To mount a `pipelinefs` filesystem, the **pipelinefs** program may be used as follows:

`pipelinefs` [*`mountpoint`*]

Standard FUSE filesystem options are also available in the **pipelinefs** command, but default values for some of these options are automatically set. Be aware that the filesystem depends on a few of the FUSE options to work properly, but you may of course experiment with alternative values.

A mounted `pipelinefs` filesystem will appear in the `/proc/mounts` file, but will not appear in the output of the **df** command.

2.1.2.2 Unmounting `pipelinefs`

A `pipelinefs` filesystem may be unmounted by using the **fusermount** system command as

`fusermount -u` [*`mountpoint`*]

[2] See the getfattr(1), setfattr(1), attr(5) man pages for information on extended attributes on filesystem objects under Linux.

2.1.2.3 Starting up and/or connecting to a running compute node pipeline instance

To create a compute node pipeline instance under a `pipelinefs` filesystem , or to "connect" a running compute node pipeline instance to the filesystem, simply write a "host:port" address value to a new file in the top-level directory of the `pipelinefs` filesystem. Upon creating such a "file" in the filesystem, a trial WCBE_NAME_MSG message is first sent to the given address in an attempt to contact a running compute node pipeline instance on the given node listening on the given port. If no response to the trial message is received, a new compute node pipeline is spawned on the given host by logging in to that host, and running the **wcbe** application. After spawning a new compute node pipeline instance, a sequence of WCBE_NAME_MSG messages are sent to verify that the instance started up correctly. The following shell command example creates and/or connects to a compute node pipeline instance on the host `compute0` with a control socket on port 9000, and creates the instance's filesystem entries under the name `node0` in the `pipelinefs` filesystem mounted at `pipelines`.

```
echo compute0:9000 > pipelines/node0
```

2.1.2.4 Shutting down a compute node pipeline instance

To shut down a compute node pipeline instance that is under a `pipelinefs` filesystem, simply remove the instance's directory from the filesystem. Note that the directory need not be empty for the remove operation to succeed. The following shell command example would shut down the compute node pipeline instance started in the previous example.

```
rmdir pipelines/node0
```

2.1.2.5 Restarting `pipelinefs`

A list of connected compute node pipeline instances is created in the `pipelinefs` mount point directory when the filesystem is unmounted in order to facilitate the compute node connection process upon remounting the `pipelinefs` filesystem. Every compute node instance with an entry in the `pipelinefs` filesystem just before it is unmounted is written to the `.state` file in the mount point directory. When the `pipelinefs` filesystem is remounted on the same directory, the `.state` file in that directory is read, and an attempt is made to reconnect to each of the compute node pipeline instances listed in the file. For example, after the following sequence of commands

```
pipelinefs pipelines
echo compute0:9000 > pipelines/node0
fusermount -u pipelines
pipelinefs pipelines
```

the directory `node0` should be present under the `pipelines` directory. If the `pipelinefs` filesystem is once more unmounted, one will find the file `.state` under the `pipelines` directory with the content

```
compute0:9000  node0
```

The `.state` file is merely a convenience, and is not essential to the proper functioning of the `pipelinefs` filesystem.

2.1.2.6 Creating a compute node pipeline instance configuration

Configurations are created by copying the contents of a local configuration file to a new file name under the `configurations` directory of a compute node pipeline instance. In the following example, we create a new configuration named `config0` using the file `testconf.xml`.

```
cat testconf.xml > pipelines/node0/configurations/config0
```

2.1.2.7 Destroying a compute node pipeline instance configuration

A configuration is destroyed by removing the configuration's directory from the `pipelinefs` filesystem. Note that the configuration directory need not be empty for the "destroy" action to succeed. For example, to destroy the configuration created in the previous example, the following command is sufficient.

```
rmdir pipelines/node0/configurations/config0
```

Be aware that a destroyed configuration may not immediately disappear from the `pipelinefs` filesystem, which will occur if the targeted configuration is active when it is destroyed. This behavior is a consequence of the manner in which configuration destruction is handled by the compute node pipelines. A sign that an active configuration has received the "destroy" command and will be removed from the filesystem after its deactivation is that the contents of the configuration's `status` file is "active-retired".

3 Compute node software

Design of the backend cluster compute node software is well advanced, although incomplete. Most of the high level design is complete, with the exception of details regarding the handling of auxiliary data that is not contained in the lag frames sent by the WIDAR correlator baseline boards. Some design work on data representations used internally by the backend are also incomplete. Implementation is complete for the high-level backend system compute node executable, the input stage pipeline, and the lag processing stage pipeline container processes.

3.1 Design elements

Compute node software is based on a number of design elements that may be described separately. The following sections present a relatively high-level overview of these design elements, and how they are used by the backend software.

3.1.1 GStreamer framework

The design of all pipeline processing components is based on the GStreamer framework. GStreamer is an open source multimedia application framework, which "allows the construction of graphs of media-handling components." A high-level overview of GStreamer features may be found on the GStreamer website at http://gstreamer.freedesktop.org/features/.[3] The following sections provide simple descriptions of the GStreamer framework in the context of the backend software.

[3] All GStreamer documentation may be found on the website at http://gstreamer.freedesktop.org.

3.1.1.1 Pipeline elements

GStreamer supports the construction of pipelines as arbitrary, directed graphs of processing elements. Such pipeline elements are the basic building blocks of the pipelines in the backend. Elements are connected *via* the "pads" that may exist on an element. Terminal elements in a pipeline are classified as source or sink elements, depending on whether pipeline data is put into or taken out of the pipeline by that element. Non-terminal pipeline elements may be classified in a variety of ways, such as filter, demuxer, *etc.* Elements are available to application programs by dynamically loaded plugins, providing the backend software with a clear mechanism for extensibility.

3.1.1.1.1 Pads

Pads are the component structures of elements over which data flows in a pipeline. Every pad may be classified as either a sink or source pad (as seen from a point of view outside of the element; in this way, a source element has only source pads, and a sink element, only sink pads). Elements are connected in a pipeline through their pads. Since any given element may be able to process data in a limited number of formats, each pad has associated with it a set of allowed data types (media types, in the GStreamer terminology). Whether two pipeline elements can be connected depends on the type of the data that would pass between the elements, and the data types allowed by elements' respective sets of source and sink pads.

Data types

Data (or media) types in the GStreamer sense are based on MIME types with added properties. Pad *capabilities* (or *caps* in GStreamer terminology) are effectively sets of MIME types together with arbitrary properties associated with each type. Whereas a pad's set of allowed caps is typically specified at compile time, *caps negotiation* occurs at run time before data can pass between connected pads. Caps negotiation ensures that connected elements pass data in a format acceptable to both sender and receiver.

The set of data types used by the backend software is intentionally somewhat limited, but the set may be extended using the plugin mechanism.

3.1.1.1.2 Properties

GStreamer pipeline elements are based on GObject objects, and, consequently, have object properties available to them. GObject properties function as simple attributes in an object-oriented model. Various pipeline element properties are available to provide monitor and control functionality for the backend pipelines. For example, an element may provide a boolean-valued property to be used to switch on or off the data flow through a pipeline.

3.1.1.1.3 Bins

Individual elements may be linked together into higher order elements termed *bins*. In the context of the backend software, it is envisioned that bins may be a useful mechanism by which to extend the available pipeline elements. Because certain elements of backend pipelines are likely to be used universally, and those elements will have fixed pad capabilities, new elements will be required to use a common backend data type

as input, and produce a common backend data type as output. Thus, it may be useful to implement a new element as a bin whenever intermediate data types are desired.

Another potential application of bins in the backend software would be to package the most commonly used pipelines. Bundling elements in this way would allow a simplified configuration specification whenever the backend were to be configured in some widely applicable, common mode. Whether this approach might be useful or not is uncertain.

3.1.1.2 Signals

Signals are a notification mechanism used in the GStreamer framework[4] to connect arbitrary application-specific events with any number of listeners. Signals are user-definable, and are used to trigger callbacks conforming to an interface specified by the type of the signal. In a multi-threaded framework, such as GStreamer, it is important to understand that signals are synchronous, and the callbacks triggered by a signal run in the same thread in which the signal was emitted. In the backend software, signals may be used as a low-level notification mechanism between components of a pipeline (such as between an element and the application).

3.1.1.3 Message bus

A message bus is a system to forward messages from the pipeline threads to the application thread. This allows an application to remain thread-unaware, although GStreamer pipelines are heavily threaded. In the backend software, a commonly used pattern is for callbacks connected to particular signals to simply create application messages and post them to the message bus, which has the effect of transferring signals from the pipeline threads to the application thread.

One use of the message bus in the backend software is to read from and write to pipeline element properties, which provides the pipeline application with important monitor and control capabilities. Another use of the message bus is described in **Section 3.3.2.2**.

3.1.1.4 Events

Events in GStreamer are "control particles that are sent both up- and downstream in a pipeline". Downstream events are commonly used in GStreamer to notify elements of stream states,[5] and these types of events are the only ones used by the backend software. Events are also classified according to whether they are serialized in the data stream or not, allowing both data-synchronous and asynchronous event types.

One use of application-specific events in the backend is to signal changes in the backend configuration, as described in **Section 3.4.2**. Because events may carry data, they provide a means to bring auxiliary data (for example, the zero lags, or state counts) to the pipeline elements. While it is planned to use events as carriers of auxiliary information, this feature is not yet implemented.

[4] These signals have nothing to do with standard Unix system signals.
[5] Upstream events are commonly used in GStreamer to allow an application to control data streaming, for example, seeking the position of an audio stream by a playback application.

3.1.2 Process separation

Process separation is the most important principle employed in the backend design to provide robustness to the system. The lag frame processing functionality of the backend is clearly divided into two stages: the input stage, which sorts and orders the WIDAR lag frames arriving from the WIDAR baseline boards *via* an Ethernet network; and the lag processing stage, which applies a sequence of various transformations and side-effect producing operations to the complete lag sets[6] produced by the input stage. These two stages are separated through the mechanism of process isolation. As described below, the input stage is likely to be relatively stable, while the lag processing stage is likely to be less stable; therefore, process isolation is used to prevent faults in the lag processing stage from causing faults in the input stage. Because the lag processing stage comprises multiple pipeline processes (see **Section 3.2**), this design also ensures that a fault in any lag processing pipeline process is isolated to that single process.

3.1.2.1 Input stage

By design, the function of the input stage is invariant across all instances of the WIDAR correlator backend. If the backend uses the software under discussion in this document, it is, by definition, using the input stage of this software.[7] The central role of the input stage in all backend instances requires that it be implemented efficiently and operate robustly. For these reasons, the input stage cannot be (dynamically) extended with new functionality.

3.1.2.2 Lag processing stage

In contrast to the input stage, the lag processing stage functionality is quite broad, and subject to different requirements in different backend instances. This stage is where most of the ongoing development of backend functionality will occur. The GStreamer framework, through its mechanisms for pipeline extensibility, provides the greatest benefit to the backend software in this stage.

Note that both transformations, such as lag normalization, Fourier transformations, and backend integration, and side-effect producing operations, such as output creation and data duplication, may occur in this stage. This may be a point of confusion insofar as the general conception prior to the present design separated the "data processing" from the "output" functionality. The present design is superior in that it reinforces the distributed nature of output production that is foreseen as the standard output mode of the backend. In this design, pipeline elements that produce output may co-exist simultaneously in multiple pipelines.

3.1.2.3 Inter-stage communication

Communication between the two backend stages requires some form of inter-process communication (IPC) because the stages run in distinct processes. The current design uses datagram-oriented Unix domain sockets as the IPC mechanism for this purpose. The benefits of this method are primarily in its ease of use when compared with stream-oriented sockets or shared memory. Another important benefit of using this method is that the backend stages are loosely coupled as a consequence (as long as the sockets operate in non-blocking

[6] I use the term *lag set* to describe the Cartesian product of a lag spectra, a set of phase bins, and a set of polarization products *i.e.* lag spectrum $\otimes$ phase bins $\otimes$ polarization products

[7] Of course, it is conceivable that another, improved input stage could be developed; but, in that case, the new input stage would likely be adopted as the standard input stage.

mode). Loose coupling of the two stages is promoted by the independence of communication endpoints resulting from the use of this IPC method.

3.1.2.3.1 Lag data passing

Lag data are passed in binary form as **lag sets**. Each lag set is passed in one message over a socket. Of course, lag data are passed from the input stage to the lag processing stage only.

3.1.2.3.2 Message passing

Message passing in the backend system is built on the message bus facility of GStreamer. However, because the two backend stages do not run in a single process, the backend software must include an additional mechanism to pass messages between processes. This is accomplished by serializing as text application messages received on the message bus of the transmitting pipeline, sending the text over a socket in a single datagram (*i.e.*, socket message), and then deserializing the text and posting the message to the message bus of the receiving pipeline.

As described in **Section 3.4**, the input stage process is effectively the master process of the backend system, and application messages should be posted on the message bus of the input stage pipeline only. All messages posted in the lag processing stage pipelines will be sent to the input stage, but nothing about this design precludes also using the message bus internally in any pipeline of the backend system.

3.1.2.3.3 Event passing

Events also cross the interface between backend stages *via* datagram-oriented Unix domain sockets. Events are passed in a binary form for efficiency (and the flexibility of serializing arbitrary data types is not needed). GStreamer events received by the sink element of the input stage pipeline are converted to a simple binary form, sent *via* socket messages to the lag processing stage pipelines[8], and are then converted by the source elements in the lag processing stage pipelines into GStreamer events.

As described in **Section 3.1.1.4**, the backend system uses only downstream GStreamer events, and, therefore, events are only passed between the two backend stages in the downstream direction.

3.1.3 Configuration handling

A backend instance must be configured, in both stages, before it can process any lag frames from the WIDAR correlator baseline boards. The backend software allows the creation of multiple backend configurations, only one of which is active at any moment. Backend configurations may also be destroyed after they are no longer needed. There are only three "commands" to a backend system that are needed to control the backend configuration: *create*, which creates a named configuration; *activate*, which causes a named configuration to become the active backend configuration; and *destroy*, which destroys a named configuration (except when the named configuration is the active configuration, in which case the configuration's destruction is deferred until another configuration is activated).

[8] Sending an event from the input stage to the lag processing stage requires a fanout of the event from the single input stage process to all of the lag processing stage processes in the active configuration.

Configuration commands are implemented as downstream, serialized events in the backend system. This ensures that configuration commands arrive at the pipeline elements in the correct order and can be inserted between lag frames precisely (and will maintain their position in the data stream). Configuration deactivation, which occurs implicitly just before the active configuration changes, causes internal buffers in the pipeline elements to be flushed, ensuring that a new active configuration will not apply to any lag data in the pipeline ahead of the activation event.

Backend configurations in general are specified as XML documents. To limit the number of pipeline elements that need to parse XML documents, only elements in the input stage pipeline will have access to XML configuration documents. This guideline is intended to simplify the development of elements that are designed for pipelines in the lag processing stage. Although the current design follows this guideline, it is somewhat uncertain at this time whether the guideline can be maintained in the future. The uncertainty stems from doubts about being able to supply configuration parameters to all pipeline elements efficiently. The use of GStreamer bins may slightly mitigate against some measure of inefficiency, but whether this is actually the case remains to be seen.

Because the input stage pipeline of the backend is always running, while the lag processing stage pipelines are configuration specific, configuration changes are handled somewhat differently by the two stages. Some of these differences are described below.

3.1.3.1 Input stage configuration

As alluded above, the input stage contains a single pipeline. The primary tasks of the input stage are to sort the WIDAR lag frames into lag sets, and to direct a time-ordered sequence of complete lag sets to the proper lag processing stage pipeline. Both of these tasks may be considered as a kind of sorting, where the sorting "instructions" are (partly) defined by the backend configuration. The relatively static nature of the input stage functionality requires only a single pipeline in the input stage. When a new configuration is activated, all that is required of the pipeline elements is to flush (and possibly free) some internal pipeline buffers under the old configuration, initialize buffers for the new configuration, and change the sorting instructions.

3.1.3.2 Lag processing stage configuration

The processing specification for the lag processing stage of the backend is equivalent to the specification of a set of processing pipelines (and their inputs), so each processing pipeline in this stage exists as part of only one configuration. Therefore creating (or destroying) a configuration corresponds to creating (or destroying) a set of pipelines. Activating (or deactivating) a configuration requires no action on the part of the lag processing stage because the input stage sends a flush event into each of the pipelines belonging to the deactivated configuration after the final lag set under the deactivated configuration has been sent to that pipeline.

3.1.4 Cross pipeline communication

MPI!

Remark

Use of MPI is planned, but no real design work has been done, partly for the reason that the requirements in this area are far from clear.

3.2 Lag processing stage design

The lag processing stage comprises many GStreamer-based pipeline processes. As mentioned above, these pipelines are grouped into sets by configurations. The number of lag processing pipelines is specified by a configuration, and is a function of the number of different processing paths required by the lag sets being produced by the input stage under that configuration. In general, if two lag sets share the same processing steps using identical values of all the configuration parameters required to specify those steps, then those lag sets may flow into a common pipeline in the lag processing stage. As an example, this might mean that all lag sets in one spectral window can use a single lag processing stage pipeline.

The current design maintains all of the lag processing stage pipelines in (virtual) memory as long as the configuration owning those pipelines is in existence. This design could conceivably create pressures on the system address space. If that proved to be the case, pipeline processes would have to be created and destroyed as their configurations were activated and deactivated, respectively; the only significant effect on performance in that case being a longer time required to change backend configurations. Other system resources used by pipeline processes are minimal, one important reason being that no shared memory is currently employed by the GStreamer-based pipelines in the backend. If performance improvements were to require the use of shared memory, inactive pipeline processes might have to be shut down rather than being allowed to remain in memory.

3.2.1 Pipeline elements

Pipelines in the lag processing stage can be constructed from any of a number of elements. The only restriction is that the pipeline work in the backend system framework as described previously. What this means is that the pipeline must receive lag sets and application-specific events on its input socket. The bare minimum requirements can be met with a pipeline constructed with one specific element as its source element, which is therefore regarded as a mandatory element in all lag processing stage pipelines.

Note

There is no pipeline element that sends messages to the input stage; that task is handled by the lag processing pipeline **application**. This scheme ensures that the use of the pipeline message bus by the application is not determined by any element in the pipeline, but remains an application-specific feature. Nevertheless, as described in **Section 3.1.2.3.2**, other uses of the message bus within the pipeline are also possible.

3.2.1.1 Mandatory elements

At present, there is only one mandatory element in all lag processing stage pipelines. There is a possibility that other elements may be introduced in the future that will also be mandatory (most likely a standard sink element).

3.2.1.1.1 `wcbe_lagset_src`

This is the standard source element in a lag processing stage pipeline. It is able to handle all incoming messages on a socket from the input stage (*i.e.*, receiving lag sets, and events) according to the protocol used by the input stage.

Lag sets are associated with a particular correlator product as specified by the current configuration. As described in **Section 3.5.1**, correlator products are identified by a unique string given by the configuration. However, as elements likely will often refer to the particular product to which a lag set belongs, lag sets carry an equivalent integer identifier that exists solely within a pipeline for faster indexing operations on the lag set product identification. The `wcbe_lagset_src` element generates this integer for use throughout the rest of the pipeline.

Note

The generation of integer identifiers is the only modification made by the element to the lag sets as received from the input stage. The input stage also uses similar identifiers, which get transmitted with the lag set; however, because the identifiers as received by the lag processing stage were generated within another process, they must be regenerated within the receiving process to ensure uniqueness. These identifiers are *only* to be used internally by a pipeline process, and should never be used in any way by other processes.

Events received by this element are simply recreated as GStreamer events, and passed downstream. Currently, only the GST_EVENT_EOS, GST_EVENT_FLUSH_START, GST_EVENT_FLUSH_STOP, and GST_EVENT_NEW_SEGMENT events are passed downstream by this element.

The following figure is derived from the output of the **gst-inspect wcbe_lagset_src** command, which provides information on this GStreamer element.

```
Factory Details:
  Long name:   WIDAR lagset source
  Class:       Source/WIDAR
  Description: WIDAR lagset source
  Author(s):   Martin Pokorny <mpokorny@nrao.edu>
  Rank:                none (0)

Plugin Details:
  Name:                        wcbecoreelements
  Description:         WIDAR backend lag processing stage core elements
  Filename:            /users/mpokorny/projects/wcbe/_mpp/plugins/.libs/libwcbecoreelements.so
  Version:             0.3
  License:             GPL
  Source module:       wcbe
  Binary package:      wcbe
  Origin URL:          http://www.aoc.nrao.edu/evla

GObject
 +----GstObject
       +----GstElement
             +----GstBaseSrc
                   +----GstPushSrc
                         +----WcbeLagsetSrc

Pad Templates:
  SRC template: 'src'
```

```
    Availability: Always
    Capabilities:
      application/x-widar-lagset

Element Flags:
  no flags set

Element Implementation:
  Has change_state() function: gst_base_src_change_state
  Has custom save_thyself() function: gst_element_save_thyself
  Has custom restore_thyself() function: gst_element_restore_thyself

Element has no clocking capabilities.
Element has no indexing capabilities.

Pads:
  SRC: 'src'
    Implementation:
      Has getrangefunc(): gst_base_src_pad_get_range
      Has custom eventfunc(): gst_base_src_event_handler
      Has custom queryfunc(): gst_base_src_query
    Pad Template: 'src'

Element Properties:
  name                : The name of the object
                        flags: readable, writable
                        String. Default: null Current: "wcbelagsetsrc0"
  blocksize           : Size in bytes to read per buffer (0 = default)
                        flags: readable, writable
                        Unsigned Long. Range: 0 - 4294967295 Default: 4096 Current: 4096
  num-buffers         : Number of buffers to output before sending EOS
                        flags: readable, writable
                        Integer. Range: -1 - 2147483647 Default: -1 Current: -1
  typefind            : Run typefind before negotiating
                        flags: readable, writable
                        Boolean. Default: false Current: false
  input               : Lag set source socket
                        flags: readable, writable
                        String. Default: null Current: null
  num-lagsets         : Get number of lag sets sent
                        flags: readable
                        Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 0 Current: 0
```

3.2.1.2 Optional elements

There are no other backend-specific elements yet available or designed for the lag processing stage. A short list of the functions to be provided by individual pipeline elements to be developed follows.

DV normalization
Fourier transform
Van Vleck correction
Integration
BDF output

3.2.2 Application

As noted in the foregoing, each GStreamer pipeline in the lag processing stage cannot, by itself, be a part of the backend system. The integration of a pipeline into the lag processing stage is accomplished by a GStreamer application that creates and manages the pipeline, and takes care of passing on messages posted to the pipeline message bus to the input stage. This application can create any GStreamer pipeline whose first element includes a sink pad that will accept lag sets. The pipeline created by the application automatically includes a `wcbe_lagset_src` source element at its head; that element should never be explicitly included in the pipeline specification passed to the application.

The application command line synopsis is

`lagset_pipeline` [-i | --input]*`NAME`* *`pipeline`* [--help]

`input`

The name of the socket from which the lag sets processed by the pipeline are read.

pipeline

A pipeline description in the syntax used by GStreamer's **gst-launch** utility.

Note

It is additionally planned to support the use of a file of recorded lag sets as input (much like **wcbe** allows), but this feature has not yet been implemented. Being able to use lag sets from a file would be useful for the development and testing of new pipeline elements.

Note

The syntax of the pipeline description can be obtained from the **gst-launch** man page. It is uncertain whether this syntax for lag processing stage pipeline descriptions will satisfy all requirements or be efficient to use; although no real instances have yet been identified that could be used as an argument in support of changing the syntax, the possibility of a future change should not be discounted.

The application listens for application messages on the pipeline's message bus, and will forward upstream all such messages to the input stage. Messages are forwarded with a slightly changed `src` field value (which normally identifies the message's source element name) to include the name of the pipeline; this is done so that message handlers in the input stage can determine both the pipeline and the element that originated the message.

A GST_MESSAGE_EOS message posted on the pipeline's message bus will be received by the application, which will subsequently initiate its own termination.

3.3 Input stage design

The input stage consists of only one process. This single process combines the input stage pipeline functionality and the overall control of the compute node backend system, creating the superficial appearance of a single backend processing pipeline. This section of the document will describe how the input stage pipeline is designed, and how the input stage manages the processes in the lag processing stage. The control interface of the compute node backend system is described later in **Section 3.4**.

The input stage pipeline is fixed with regard to the pipeline elements and their connections. Of course, the pipeline responds to the creation, activation and destruction of backend configurations dynamically, on demand. One other feature of the input stage pipeline that may be controlled dynamically is the generation of lag frame dump files (see **Section 3.3.1.2**). Other input stage parameters are provided at start-up and cannot be changed in a running system.

3.3.1 Pipeline elements

The standard input stage pipeline consists of the following elements, in the order presented here.

3.3.1.1 `wcbe_src`

The source element for WIDAR lag frames. As such, its function is to put lag frames into the input stage pipeline. The lag frames may originate from a UDP socket, in which case they are live frames coming from the WIDAR baseline boards, or from a file in the so-called BLF (for "binary lag frame") format.

When using a BLF file as the source of lag frames, the frames may be pushed into the pipeline at a user-determined rate, and, optionally, lag frame timestamps (and checksum values) can be rewritten by the element to reflect the wall clock time just before each frame is pushed into the pipeline.

Using a live frame source is somewhat simpler in that there are no element properties to set other than the IP address and port number of the socket.

Element properties other than `uri`, `fps`, and `live-timestamps` should not generally be set. The `num-frames` property can be read at any time for monitoring purposes.

Note

The `widar-max-framesize` property may be needed only if the maximum size of WIDAR lag frames is ever changed.

The following figure is derived from the output of the **gst-inspect wcbe_src** command, which provides information on this GStreamer element.

```
Factory Details:
  Long name:   WIDAR frame source
  Class:       Source
  Description: WIDAR frame source
  Author(s):   Martin Pokorny <mpokorny@nrao.edu>
  Rank:                none (0)

Plugin Details:
  Name:                        wcbeinputstage
  Description:         WIDAR backend input stage elements
  Filename:            /users/mpokorny/projects/wcbe/_mpp/plugins/.libs/libwcbeinputstage.so
  Version:             0.3
  License:             GPL
  Source module:       wcbe
  Binary package:      wcbe
  Origin URL:          http://www.aoc.nrao.edu/evla
```

```
GObject
 +----GstObject
       +----GstElement
             +----GstBaseSrc
                   +----GstPushSrc
                         +----WcbeSrc

Implemented Interfaces:
  GstURIHandler

Pad Templates:
  SRC template: 'src'
    Availability: Always
    Capabilities:
      application/x-widar-raw

Element Flags:
  no flags set

Element Implementation:
  Has change_state() function: gst_base_src_change_state
  Has custom save_thyself() function: gst_element_save_thyself
  Has custom restore_thyself() function: gst_element_restore_thyself

Element has no clocking capabilities.
Element has no indexing capabilities.

Pads:
  SRC: 'src'
    Implementation:
      Has getrangefunc(): gst_base_src_pad_get_range
      Has custom eventfunc(): gst_base_src_event_handler
      Has custom queryfunc(): gst_base_src_query
    Pad Template: 'src'

Element Properties:
  name                : The name of the object
                        flags: readable, writable
                        String. Default: null Current: "wcbesrc0"
  blocksize           : Size in bytes to read per buffer (0 = default)
                        flags: readable, writable
                        Unsigned Long. Range: 0 - 4294967295 Default: 4096 Current: 1068
  num-buffers         : Number of buffers to output before sending EOS
                        flags: readable, writable
                        Integer. Range: -1 - 2147483647 Default: -1 Current: -1
  typefind            : Run typefind before negotiating
                        flags: readable, writable
                        Boolean. Default: false Current: false
  uri                 : Set frame source
                        flags: readable, writable
                        String. Default: null Current: null
  widar-max-framesize : Set maximum frame size
                        flags: readable, writable
                        Unsigned Long. Range: 0 - 4294967295 Default: 1068 Current: 1068
  num-frames          : Get number of frames sent
                        flags: readable
                        Unsigned Integer64. Range: 0 - 18446744073709551615 Default: 0 Current: 0
```

```
  fps                 : Set playback rate of file input
                        flags: readable, writable
                        Unsigned Integer. Range: 0 - 4294967295 Default: 100 Current: 100
  live-timestamps     : Use live timestamps in data frames?
                        flags: readable, writable
                        Boolean. Default: false Current: false
```

3.3.1.2 wcbe_framedump

This element is a passthrough element, meaning it does not change the data going through it in any way. Its use is to dump WIDAR lag frames to files. The files can be written in either an ASCII or binary format, the latter being the BLF file format that may be used as input by the **wcbe_src** element. The generation of frame dump files is controllable at run-time. Output files are closed (as a set) and a new set is opened when the largest of the set reaches or exceeds a specified size.

The following figure is derived from the output of the **gst-inspect wcbe_framedump** command, which provides information on this GStreamer element.

```
Factory Details:
  Long name:   WIDAR frame dump
  Class:       Transform/WIDAR
  Description: Dump WIDAR binary frames to files
  Author(s):   Martin Pokorny <mpokorny@nrao.edu>
  Rank:                none (0)

Plugin Details:
  Name:                        wcbeinputstage
  Description:         WIDAR backend input stage elements
  Filename:            /users/mpokorny/projects/wcbe/_mpp/plugins/.libs/libwcbeinputstage.so
  Version:             0.3
  License:             GPL
  Source module:       wcbe
  Binary package:      wcbe
  Origin URL:          http://www.aoc.nrao.edu/evla

GObject
 +----GstObject
       +----GstElement
             +----GstBaseTransform
                   +----WcbeFramedump

Pad Templates:
  SINK template: 'sink'
    Availability: Always
    Capabilities:
      application/x-widar-raw

  SRC template: 'src'
    Availability: Always
    Capabilities:
      application/x-widar-raw

Element Flags:
  no flags set
```

```
Element Implementation:
  Has change_state() function: gst_element_change_state_func
  Has custom save_thyself() function: gst_element_save_thyself
  Has custom restore_thyself() function: gst_element_restore_thyself

Element has no clocking capabilities.
Element has no indexing capabilities.

Pads:
  SRC: 'src'
    Implementation:
      Has getrangefunc(): gst_base_transform_getrange
      Has custom eventfunc(): gst_base_transform_src_event
    Pad Template: 'src'
  SINK: 'sink'
    Implementation:
      Has chainfunc(): gst_base_transform_chain
      Has custom eventfunc(): gst_base_transform_sink_event
      Has bufferallocfunc(): gst_base_transform_buffer_alloc
    Pad Template: 'sink'

Element Properties:
  name                : The name of the object
                        flags: readable, writable
                        String. Default: null Current: "wcbeframedump0"
  qos                 : handle QoS messages
                        flags: readable, writable
                        Boolean. Default: false Current: false
  format              : Output format switch
                        flags: readable, writable
                        Enum "WcbeFramedumpFormat" Current: 0, "ASCII"
                           (0): ASCII            - WCBE_FRAMEDUMP_ASCII
                           (1): BINARY           - WCBE_FRAMEDUMP_BINARY
  do-output           : Produce frame dump files?
                        flags: readable, writable
                        Boolean. Default: false Current: false
  max-file-size       : Maximum dump file size
                        flags: readable, writable
                        Integer64. Range: 102400 - 1073741824 Default: 10485760 Current: 10485760
```

3.3.1.3 `wcbe_decode`

Decoder element that converts binary lag frames into C language structures for later processing by the input stage pipeline. Frames are only partially decoded for efficiency in later pipeline elements that need to do some kinds of lag frame sorting. Lag frame checksums are validated by this element, and invalid lag frames are dropped from the pipeline.

Note

The implementation of this element is missing a few features. The most important missing feature is a counter for invalid lag frames, and possibly a signal that is triggered by invalid lag frames.

The following figure is derived from the output of the **gst-inspect wcbe_decode** command, which provides information on this GStreamer element.

```
Factory Details:
  Long name:   WIDAR decoder
  Class:       Decoder/WIDAR
  Description: Decode WIDAR binary lag frames
  Author(s):   Martin Pokorny <mpokorny@nrao.edu>
  Rank:                none (0)

Plugin Details:
  Name:                        wcbeinputstage
  Description:         WIDAR backend input stage elements
  Filename:            /users/mpokorny/projects/wcbe/_mpp/plugins/.libs/libwcbeinputstage.so
  Version:             0.3
  License:             GPL
  Source module:       wcbe
  Binary package:      wcbe
  Origin URL:          http://www.aoc.nrao.edu/evla

GObject
 +----GstObject
       +----GstElement
             +----GstBaseTransform
                   +----WcbeDecode

Pad Templates:
  SRC template: 'src'
    Availability: Always
    Capabilities:
      application/x-widar

  SINK template: 'sink'
    Availability: Always
    Capabilities:
      application/x-widar-raw

Element Flags:
  no flags set

Element Implementation:
  Has change_state() function: gst_element_change_state_func
  Has custom save_thyself() function: gst_element_save_thyself
  Has custom restore_thyself() function: gst_element_restore_thyself

Element has no clocking capabilities.
Element has no indexing capabilities.

Pads:
  SRC: 'src'
    Implementation:
      Has getrangefunc(): gst_base_transform_getrange
      Has custom eventfunc(): gst_base_transform_src_event
    Pad Template: 'src'
  SINK: 'sink'
    Implementation:
      Has chainfunc(): gst_base_transform_chain
```

```
      Has custom eventfunc(): gst_base_transform_sink_event
      Has bufferallocfunc(): gst_base_transform_buffer_alloc
    Pad Template: 'sink'

Element Properties:
  name                : The name of the object
                        flags: readable, writable
                        String. Default: null Current: "wcbedecode0"
  qos                 : handle QoS messages
                        flags: readable, writable
                        Boolean. Default: false Current: false
```

3.3.1.4 wcbe_sort

Element that sorts WIDAR lag frames into lag sets, orders lag sets by time, and handles missing or late lag frames. This is the first element in the input stage pipeline that needs to be passed an XML configuration document before doing any useful work. Lag frames that do not appear in a configuration are dropped from the pipeline. When the active configuration is changed, the element ensures that buffers containing lag sets that arrived before the configuration change event are flushed before the new configuration takes effect.

The implementation of this element is somewhat complex due to the sorting and timeout functionality, as well as the handling of configurations. Although the element is functional, there are a few features that still need to be implemented, mainly having to do with dropped lag frames. This element implements the heart of the backend input stage, and will require much testing to ensure that it is functioning correctly; more properties and signals for monitoring would be very helpful.

Note

The `configs` property value configuration names are tagged in an unusual style to indicate the current status of each existing configuration. These tags should be documented here, but, in the meantime, interactive user access to the backend system through the **wcbe_console** application hides these tags and reports configuration status using meaningful text strings.

The following figure is derived from the output of the **gst-inspect wcbe_sort** command, which provides information on this GStreamer element.

```
Factory Details:
  Long name:   WIDAR lag set sorter
  Class:       Sorter/WIDAR
  Description: Assemble and temporally sort WIDAR lag sets
  Author(s):   Martin Pokorny <mpokorny@nrao.edu>
  Rank:                none (0)

Plugin Details:
  Name:                        wcbeinputstage
  Description:         WIDAR backend input stage elements
  Filename:            /users/mpokorny/projects/wcbe/_mpp/plugins/.libs/libwcbeinputstage.so
  Version:             0.3
  License:             GPL
  Source module:       wcbe
  Binary package:      wcbe
  Origin URL:          http://www.aoc.nrao.edu/evla
```

```
GObject
 +----GstObject
       +----GstElement
             +----WcbeSort

Pad Templates:
  SRC template: 'src'
    Availability: Always
    Capabilities:
      application/x-widar-lagset

  SINK template: 'sink'
    Availability: Always
    Capabilities:
      application/x-widar

Element Flags:
  no flags set

Element Implementation:
  Has change_state() function: wcbe_sort_change_state
  Has custom save_thyself() function: gst_element_save_thyself
  Has custom restore_thyself() function: gst_element_restore_thyself

Element has no clocking capabilities.
Element has no indexing capabilities.

Pads:
  SRC: 'src'
    Implementation:
    Pad Template: 'src'
    Capabilities:
      application/x-widar-lagset
  SINK: 'sink'
    Implementation:
      Has chainfunc(): 0x6e48cf
      Has custom eventfunc(): 0x6e5494
    Pad Template: 'sink'

Element Properties:
  name                : The name of the object
                        flags: readable, writable
                        String. Default: null Current: "wcbesort0"
  configs             : Get list of all available configurations
                        flags: readable
                        Array of GValues
  active-config       : Get name of active configuration
                        flags: readable
                        String. Default: null Current: ""

Element Signals:
  "config-created" :  void user_function (GstElement* object,
                                          gchararray arg0,
                                          gboolean arg1,
                                          gpointer user_data);
  "config-activated" :  void user_function (GstElement* object,
                                            gchararray arg0,
```

```
                                          gboolean arg1,
                                          gpointer user_data);
  "config-deactivated" :  void user_function (GstElement* object,
                                            gchararray arg0,
                                            gpointer user_data);
  "config-destroyed" :  void user_function (GstElement* object,
                                          gchararray arg0,
                                          gboolean arg1,
                                          gpointer user_data);
```

3.3.1.5 wcbe_lagset_sink

Element that routes lag sets to the appropriate lag processing stage pipelines as determined by the active backend configuration. The element also forwards input stage downstream pipeline events to the lag processing stage pipelines in the currently active configuration as they arrive. Pipeline flush events are created and sent to all lag processing stage pipelines in the active configuration before switching to a different configuration.

See this note regarding the value of the `configs` element property.

The following figure is derived from the output of the **gst-inspect wcbe_lagset_sink** command, which provides information on this GStreamer element.

```
Factory Details:
  Long name:   WIDAR lagset sink
  Class:       Sink/WIDAR
  Description: Write WIDAR lag sets to Unix domain sockets
  Author(s):   Martin Pokorny <mpokorny@nrao.edu>
  Rank:                none (0)

Plugin Details:
  Name:                        wcbeinputstage
  Description:         WIDAR backend input stage elements
  Filename:            /users/mpokorny/projects/wcbe/_mpp/plugins/.libs/libwcbeinputstage.so
  Version:             0.3
  License:             GPL
  Source module:       wcbe
  Binary package:      wcbe
  Origin URL:          http://www.aoc.nrao.edu/evla

GObject
 +----GstObject
       +----GstElement
             +----GstBaseSink
                   +----WcbeLagsetSink

Pad Templates:
  SINK template: 'sink'
    Availability: Always
    Capabilities:
      application/x-widar-lagset

Element Flags:
  no flags set
```

```
Element Implementation:
  Has change_state() function: gst_base_sink_change_state
  Has custom save_thyself() function: gst_element_save_thyself
  Has custom restore_thyself() function: gst_element_restore_thyself

Element has no clocking capabilities.
Element has no indexing capabilities.

Pads:
  SINK: 'sink'
    Implementation:
      Has chainfunc(): gst_base_sink_chain
      Has custom eventfunc(): gst_base_sink_event
      Has bufferallocfunc(): gst_base_sink_pad_buffer_alloc
    Pad Template: 'sink'

Element Properties:
  name                : The name of the object
                        flags: readable, writable
                        String. Default: null Current: "wcbelagsetsink0"
  preroll-queue-len   : Number of buffers to queue during preroll
                        flags: readable, writable
                        Unsigned Integer. Range: 0 - 4294967295 Default: 0 Current: 0
  sync                : Sync on the clock
                        flags: readable, writable
                        Boolean. Default: true Current: false
  max-lateness        : Maximum number of nanoseconds that a buffer can be late before it is dropped (-1
unlimited)
                        flags: readable, writable
                        Integer64. Range: -1 - 9223372036854775807 Default: -1 Current: -1
  qos                 : Generate Quality-of-Service events upstream
                        flags: readable, writable
                        Boolean. Default: false Current: false
  sockdir             : Directory for lagset sockets
                        flags: readable, writable
                        String. Default: null Current: ""
  configs             : Get list of all available configurations
                        flags: readable
                        Array of GValues
  active-config       : Get name of active configuration
                        flags: readable
                        String. Default: null Current: ""

Element Signals:
  "config-created" :  void user_function (GstElement* object,
                                          gchararray arg0,
                                          gboolean arg1,
                                          gpointer user_data);
  "config-activated" :  void user_function (GstElement* object,
                                            gchararray arg0,
                                            gboolean arg1,
                                            gpointer user_data);
  "config-deactivated" :  void user_function (GstElement* object,
                                              gchararray arg0,
                                              gpointer user_data);
  "config-destroyed" :  void user_function (GstElement* object,
                                            gchararray arg0,
```

```
                                        gboolean arg1,
                                        gpointer user_data);
```

3.3.2 Application

wcbe is the application that implements the backend input stage. This application seamlessly integrates the input and lag processing stages to give the appearance of a single processing pipeline.

The application command line synopsis is

`wcbe` [--name | -n]*`NAME`* [[--src | -s]*`URI`*] [--live-timestamps | -l] [[--rate | -r]*`RATE`*] [[--sockdir | -k]*`DIR`*] [[--framedump | -d] [none | ascii | binary]] [[--flow | -f] [on | off]] [[--conf | -c]*`FILE`*]

`name`

The only required option; the name of the backend input stage instance. Of course, running the application with just this option doesn't do much, but the application does open a socket over which the application may be controlled and monitored, and `name` is used to provide the name of that socket.

`src`

A URI naming either a socket and port number to which lag frames are sent (by the WIDAR baseline boards), or a file of recorded lag frames in the BLF format. The `live-timestamps` and `rate` options are only used when the `src` option names a file.

`live-timestamps`

Determines whether the lag frame timestamps (and checksums) are rewritten by the `wcbe_src` pipeline element.

`rate`

Determines the rate at which the lag frames in the file are pushed into the pipeline.

`sockdir`

Name of a directory, with a default value determined by inspecting the environment variables `TMPDIR`, `TMP`, and `TEMP` in that order, and finally the `/tmp` directory if none of the environment variables are defined. The value is the name of the top-level directory in which a working directory for the input stage instance will be created.

Remark

Yes, the name of the option should be changed.

It is best normally not to use the `sockdir` option.

`framedump`

Determines whether lag frames are dumped to a file, as well as the dump format.

`flow`

A switch to control the flow of lag frames through the input stage pipeline (and consequently, the lag processing stage, as well).

`conf`

The initial configuration of the input stage instance. The option is normally not used, since the input stage can handle a multitude of configurations, but the option has an application in testing and development.

3.3.2.1 Directory structure

The `name` and `sockdir` option values together determine the name of a directory created by the input stage for its general use. This directory is simply named `[sockdir]/[name]`; it is removed by the application on exit, but could be orphaned by the application under unusual circumstances. In this directory are created the following named Unix domain sockets: `ctlsock`, the application control socket (may not be present, see **Section 3.4.1**); `issink`, used as the destination socket by the lag processing stage applications to send messages to the input stage; and `lagsets`, the lag set source socket name.

Remark

I might be able to do away with this last one, but it does no harm, so I'm in no rush.

Under the directory created for the input stage instance, a directory is created for each existing configuration, named according to the configuration name. This directory only contains the named Unix domain sockets from which the lag set processing stage pipelines receive their lag sets. Each process in the lag set processing stage will be associated with exactly one socket in the sub-directory associated with the configuration to which it belongs. These "lag set sockets" are named according to their "id" value in the configuration.

3.3.2.2 Lag processing stage management

Configurations are generally handled by the input stage as described in **Section 3.1.3**, but to do so requires the input stage to manage the processes in the lag processing stage. This section provides more detailed information on the design of this aspect of the input stage.

When a configuration is created, the input stage determines the number of lag processing stage pipelines in the configuration, and spawns one **lagset_pipeline** process for each pipeline. Each of these processes is monitored at the operating system level by the input stage application to determine whether the process is running or not.

When a configuration is destroyed, a GST_MESSAGE_EOS message is first sent by the input stage application to each lag processing stage pipeline in the configuration to request the pipelines to shut down normally. Process monitoring by the input stage allows the input stage to determine whether the pipeline processes have exited as requested; if any process has not exited after some period (approximately five seconds, currently), it is sent a kill signal to force it to exit.

In the event that a lag processing stage pipeline process in an existing configuration exits prematurely, the input stage will be notified of that event and will attempt to restart the failed process. An attempt to start a pipeline process may be repeated up to a small number of times (three, currently) before no further attempts are made. A failed pipeline process should have no impact on any part of the system aside from the loss of the lag sets that are destined for the faulty pipeline.

In addition to the process monitoring by the input stage at the operating system level, the input stage expects to receive certain messages from each of the pipelines reporting on its state. Currently, a created

configuration is not ready for use until all its pipelines have reported their readiness to the input stage. If for some pipeline a readiness message is not received by the input stage before a timeout occurs (five seconds, again, currently), the pipeline process is sent a kill signal. Another attempt may be made to start the pipeline depending on the number of attempts that have already been made to start the process, as described previously.

Note

I think this behavior should be changed. Although one configuration may contain several nominally independent lag processing stage pipelines, the current behavior improperly ties together the operation of these various pipelines. There is no problem in waiting for the state message from each pipeline, but the fate of an entire configuration should not be determined by any single pipeline process. My mistake.

3.4 Application interface

The compute node software described in this document functions as the main correlator backend application on the compute nodes. For that reason, this application needs to have an interface that is accessible by the head node processes, while, simultaneously, there must be a human accessible interactive interface for diagnostic, development and testing purposes and to provide control during the period before the head node software is available. To achieve these goals, the input stage process contains an integrated socket based control and monitoring interface, and a command line application has been developed that is able to communicate over this interface with the input stage process in a relatively user-friendly manner.

3.4.1 Control interface

The control (and monitoring) interface is based on a simple protocol using message passing, in which all message transactions are initiated by any process that is not the input stage process. The interface is based on a message-oriented socket, either a Unix domain socket or a UDP/IP socket, the choice of which is made at compile time.[9] Generally, the Unix domain socket is rarely used, low level development and testing being the cases in which it is most likely of use. A message transaction is initiated with a WCBE_REQUESTmessage received on the input stage process' control socket, `ctlsock`, followed by a return WCBE_RESPONSE message to the originating socket. WCBE_REQUEST messages are tagged so that a WCBE_RESPONSE message can be properly associated with the initial WCBE_REQUEST message by the originating process.[10] WCBE_REQUEST messages are classified as either WCBE_COMMAND or WCBE_QUERY messages, whose meanings are hopefully clear to the reader.

3.4.1.1 Message types

There follows a list of the message types that are used by the control interface. The message names are the enumeration symbols used in the source code. In the description of each message type, if a mention is made of the fields in the message, these only refer to the fields specific to the message in question, and do not refer to fields common to all message types. WCBE_COMMAND WCBE_RESPONSE messages generally

[9] I am investigating the use of an SCTP socket as well, but this work in not done.

[10] The system is relatively robust to simultaneous access by multiple processes, assuming that each process has its own socket by which to communicate with the input stage.

return the same message content that was sent in the WCBE_REQUEST message. WCBE_QUERY WCBE_REQUEST messages generally leave the message type-specific field(s) empty.

WCBE_CONFIG_CREATE_MSG

Create configuration. Two message fields: `name`, a configuration identifier; and `desc`, the XML configuration document (that is, the entire document as a string, not a file name). Only exists as a WCBE_COMMAND message.

WCBE_CONFIG_ACTIVATE_MSG

Activate configuration. One message field: `name`, the configuration name. WCBE_COMMAND only.

WCBE_CONFIG_DESTROY_MSG

Destroy configuration. One message field: `name`, configuration name. WCBE_COMMAND only.

WCBE_CONFIG_LIST_MSG

Configuration list. One message field: `list`, a NULL-terminated list of existing configuration names and pending configuration actions (each configuration name being a NULL-terminated string). The name of a configuration is surrounded by square brackets ("[name]") when the configuration is active but marked for destruction, angle brackets ("<name>" when active but not marked for destruction, parentheses ("(name)") when inactive, asterisks ("*name*") when pending creation, hash marks ("#name#") when pending activation, and the at sign ("@name@") when pending destruction. Pending configuration actions occur when the **wcbe** application has been started, but its pipeline is not running due to lack of a lag frame source setting, or the pipeline flow being turned off. WCBE_QUERY only; WCBE_REQUEST `list` field is ignored by input stage.

WCBE_SRC_MSG

Input stage lag frame source. One message field: `name`, lag frame source URI.

WCBE_FRAMEDUMP_MSG

Raw lag frame dump value. One message field: `format`, WCBE_DUMP_NONE to inhibit frame dumping; WCBE_DUMP_ASCII to dump frames in ASCII format; WCBE_DUMP_BINARY to dump frames in binary format.

WCBE_FLOW_MSG

Pipeline flow. One message field: `state`, WCBE_FLOW_OFF to inhibit pipeline data flow; WCBE_FLOW_ON to allow pipeline data flow.

WCBE_LIVE_TS_MSG

"Live timestamps" value. One message field: `state`, TRUE to rewrite file lag frame data timestamps, FALSE otherwise.

WCBE_RATE_MSG

Frame rate (file lag frame source only). One message field: `fps`, frame rate in frames per second.

WCBE_NAME_MSG

Input stage instance name. One message field: `name`, instance name. WCBE_QUERY only.

WCBE_QUIT_MSG

Quit input stage application. One reserved, unused message field. WCBE_COMMAND only.

WCBE_PIPELINE_LIST_MSG

List of GStreamer pipeline names in a given configuration. One message field: `configuration`, the configuration name. WCBE_QUERY only.

WCBE_ELEMENT_LIST_MSG

List of GStreamer element names in a given pipeline. One message field: `pipeline`, the pipeline name in the format [configuration name]/[pipeline name] WCBE_QUERY only.

WCBE_PROPERTY_LIST_MSG

List of properties of a given GStreamer element. One message field: `element`, the element name in the format [configuration name]/[pipeline name]/[element name] WCBE_QUERY only.

WCBE_PROPERTY_MSG

GStreamer element property value. Message fields:

`property`

Property name in the format [configuration name]/[pipeline name]/[element name]/[property name]

`type`

Value type (a GType name). A return value for a WCBE_QUERY message: the property value type. An input parameter for a WCBE_COMMAND message: the type of the value given in the `value` message field.

`value`

Property value. A return value for a WCBE_QUERY message. An input parameter for a WCBE_COMMAND message.

Both WCBE_QUERY and WCBE_COMMAND.

WCBE_PAD_LIST_MSG

List of pads for a given GStreamer element. One message field: `element`, the element name in the format [configuration name]/[pipeline name]/[element name] WCBE_QUERY only.

WCBE_PAD_CAPS_MSG

The negotiated capabilities of a given pad on a GStreamer element. Two message fields:

`pad`

The pad name in the format [configuration name]/[pipeline name]/[element name]/[pad name]

`caps`

The (negotiated) pad caps.

WCBE_QUERY only.

WCBE_PAD_PEER_MSG

The peer of a given pad on a GStreamer element. Two message fields:

`pad`

The pad name in the format [configuration name]/[pipeline name]/[element name]/[pad name]

`peer`

The pad's peer in the format [element name]/[pad name] where the named element is in the same pipeline as the queried pad.

WCBE_QUERY only.

3.4.2 Configuration

Configuration of the input stage application depends on two parameters in important ways. First, the pipeline source parameter is needed before a pipeline can be created, since the `wcbe_src` pipeline element depends on having a lag frame source. A consequence is that the input stage application cannot construct a pipeline until the WCBE_SRC_MSG message has been received. Secondly, since configuration commands are implemented in the input stage pipeline as downstream, serialized events, configuration commands cannot be acted on in the input pipeline until the pipeline is "flowing".

Without a flowing pipeline, the application cannot handle configuration commands immediately, and instead queues configuration commands in a list of pending commands. These pending commands will be acted on, in sequence, as soon as a flowing pipeline is ready. Merely stopping the data flow with a WCBE_FLOW_MSG message will not affect any existing configurations, whereas a change of source will cause all existing configurations to be destroyed. Also, on a source change, the flow parameter will automatically be set to WCBE_FLOW_OFF. Finally, when using a file as the lag frame source, the end of file will trigger a GST_MESSAGE_EOS message on the input pipeline, which will terminate the input pipeline, thus removing the source from the input stage.

3.4.3 Application

wcbe_console is a command line application that can be used to control and monitor the WIDAR backend compute node application. The execution of this application is independent of the backend application itself; the command line application can create and destroy backend application instances, but it can also connect to a running backend application (input stage process).[11] Because the backend compute node application instances are named, it is possible to allow multiple instances of the application to run simultaneously.[12]

[11] The command line application is, in essence, a front end to the backend compute node application. To limit overloading the terms *front end* and *back end* yet further, this document will refer to the front end application as the "command line application", or something similar.

[12] It would be wise to use different lag frame sources for different backend application instances; since the lag frames arrive *via* UDP socket, there is no way to guarantee that a particular lag frame is pulled from the socket by a particular backend application instance. There is, however, no mechanism to prevent such (mis)configuration.

The **wcbe_console** application is a Python application that runs from the command line. The application opens an interactive console that can be used to control and monitor backend compute node application instances. The command synopsis to start the command line application follows.

`wcbe_console` [[--name | -n]*NAME*] [[--sockdir | -s]*SOCKDIR*]

`name`

The name of the backend compute node application instance to select initially. The instance name can be set interactively from this application's console, so the need for this option is minimal.

`sockdir`

The directory in which the compute node application instance directories exist. The default value of this option corresponds with the **default value** used by the **wcbe** application. Not used in most cases.

Once the application is started, the user is presented with a command line console. The console implements a command history saving mechanism, command editing and tab completion for commands. A *help* command has not been implemented, but is planned. The first thing a user ought to do is create and/or select a backend compute node application instance. When an application instance has not been selected, the prompt consists of a simple >; when an instance has been selected, the instance name serves as a prefix to the prompt, for example `cbe0>`. When the socket that would connect with the selected compute node application instance is not found, the command line application will spawn a new compute node application with the given name; this provides a simple way to start up the compute node application. To create and/or select a backend instance, the command is **`set name [name]`**.

To destroy a compute node application instance from the command line application, the instance must first be selected. After the instance is selected (an action that could fail if the compute node application were not responding), the instance can be terminated with the **`kill`** command.

A brief description of other commands available from within the **wcbe_console** application follow. Further details concerning the effects of these commands may be gleaned by referring to **Section 3.4.1.1**.

`quit`

Quit the **wcbe_console** application. Note that this does not change the state or configuration of any compute node backend application instances.

`get name`

Get the name of the selected compute node backend application instance.

`set name [NAME]`

Select a compute node application instance by name. Creates a new instance if an instance with the supplied name cannot be found.

`get src`, **`set src [URI]`**

Get or set the lag frame source URI.

get flow, set flow [on|off]

Get or set the pipeline flow state.

get framedump, set framedump [ascii|binary|none]

Get or set the frame dump option.

get live_ts, set live_ts [true|false]

Get or set the "live timestamps" option (useful for file-based lag frame source only).

get rate, set rate [RATE]

Get or set the lag frame rate (useful for file-base lag frame source only).

get configs

Get the list of all existing configurations and pending configuration actions. The state of each existing configuration is indicated in the resulting output.

get config, set config [NAME]

Get or set the active configuration.

get all

Get all compute node application instance parameters.

create [NAME] [FILE]

Create a configuration named "NAME" using the XML backend configuration document in the file "FILE".

destroy [NAME]

Destroy the named configuration. Note that the named configuration will not be immediately destroyed if it is the active configuration; however, it will be destroyed after another configuration is activated. You cannot *undo* a destroy operation, even when the destruction has been deferred because the configuration is active at the time the command is given.

kill

Terminate the currently selected backend compute node application instance.

get pipelines [config]

Get the list of pipelines in the configuration named **config**. Note that the input stage pipeline, INPUT_STAGE, only appears in the pipeline list for the active configuration.

get elements [pipeline]

Get the list of elements in the pipeline named **pipeline**. The format of the **pipeline** parameter is **[config name]/[pipeline name]**.

`get pads [element]`

Get the list of pads for the element named **element**. The format of the **elment** parameter is **[config name]/[pipeline name]/[element name]**.

`get caps [pad]`

Get the caps for the pad named **pad**. The format of the **pad** parameter is **[config name]/[pipeline name]/[element name]/[pad name]**.

`get peer [pad]`

Get the peer of the pad named **pad**. The format of the **pad** parameter is **[config name]/[pipeline name]/[element name]/[pad name]**. The format of the returned string is **[element name]/[pad name]**, where the returned element name refers to an element in the same pipeline as the pad named in the query.

`get properties [element]`

Get the list of properties of the element named **element**. The format of the **element** parameter is **[config name]/[pipeline name]/[element name]**.

`get property [property]`

Get the value of the property named **property**. The format of the **property** parameter is **[config name]/[pipeline name]/[element name]/[property name]**.

`set property [property] [value]`

Set the value of the property named **property** to **value**. The format of the **property** parameter is the same as described above for the **get property** command. The parameter **value** is a string that should be convertible to the proper type for the property.

3.5 Configuration

This section discusses the XML documents used by the backend for configuration. The information in this section is incomplete and should be seen as a fluid work in progress, although it reflects accurately the current state of design. Current implementation may lag behind what is presented here.

Note

I've used the compact syntax form of the RELAX NG XML schema language[13] rather than the more popular W3C XML schema language as the primary form of the schema in this document. I have found that the RELAX NG schema language is generally easier to work with, and is more expressive than the W3C alternative. However, I have endeavored to design a RELAX NG schema that can be accurately translated to the W3C language, and have provided the W3C forms as well for reference.

[13] See **http://relaxng.org** for information.

3.5.1 Schema

In designing this schema, I have used short, perhaps in cases somewhat cryptic element and attribute names. The reason for doing this is simply to reduce the size of the configuration documents, which may be several megabytes in size. The ultimate reason for the large size of the configuration documents is that each correlator output product (lag spectrum) must be specified in order that the backend correctly assembles the lag frames, and each correlator output product must be assigned to a spectral window.

3.5.1.1 RELAX NG form

```
start = config.elt

# backend configuration
#
# Configuration is split into three parts: input configuration,
# specifying the correlator hardware configuration w.r.t. the products
# it produces; output configuration, specifying the subarrays and
# spectral windows in the backend output; and pipeline configuration,
# mapping correlator products to spectral windows and specifying the
# processing pipeline into which the input products go.
#
config.elt = element config {
   # input configuration
   inputConfig.elt,
   # output configuration
   outputConfig.elt,
   # pipeline configuration
   pipelineConfig.elt
}

# backend input configuration
#
# Configuration is split into two parts: the specification of the
# correlator baseline board input streams, and the products produced
# by the baseline boards from their input streams.
#
inputConfig.elt = element inputConfig {
   # streams
   stream.elt*,
   # products
   (product4.elt
    | product7.elt)*
}

stream.elt = element strm {
   # product id
   attribute id { xsd:ID },
   # station id
   attribute stn { xsd:positiveInteger },
   # baseband id
   attribute bb { xsd:nonNegativeInteger },
   # subband id
   attribute sb { xsd:nonNegativeInteger }
}
```

```
# 'product4' and 'product7' could be combined with the addition of a
# 'mode' attribute to reflect 4 or 7 bit products, but this produces a
# nondeterministic schema that cannot be accurately translated to
# the W3C XML Schema language
#
# product =
#   element product {
#     attribute id { xsd:ID },
#     attribute lags { xsd:positiveInteger },
#     attribute bins { xsd:positiveInteger },
#     attribute streamA { xsd:IDREF },
#     attribute streamB { xsd:IDREF },
#     ((attribute mode { xsd:integer "4" },
#       corrIntegration, lagChainSegment+)
#       | (attribute mode { xsd:integer "7" },
#          corrIntegration,
#          subProduct00,
#          subProduct01,
#          subProduct10,
#          subProduct11))
#   }

product.common.content =
   attribute id { xsd:ID },
   # number of lags
   attribute nLg { xsd:positiveInteger },
   # number of bins
   attribute nBn { xsd:positiveInteger },
   # stream "A" reference
   attribute strmA { xsd:IDREF },
   # stream "B" reference
   attribute strmB { xsd:IDREF },
   corrIntegration.elt

# four bit correlation product
product4.elt = element prd4 {
   product.common.content,
   lagChainSegment.elt+
}

# seven bit correlation product
product7.elt = element prd7 {
   product.common.content,
   # LSN (least significant nibble) - LSN product
   subProduct00.elt,
   # LSN - MSN (most significant nibble) product
   subProduct01.elt,
   # MSN-LSN product
   subProduct10.elt,
   # MSN-MSN product
   subProduct11.elt
}

subProduct.elements = lagChainSegment.elt+
subProduct00.elt = element sPrd00 {
   subProduct.elements
}
```

```
subProduct01.elt = element sPrd01 {
   subProduct.elements
}
subProduct10.elt = element sPrd10 {
   subProduct.elements
}
subProduct11.elt = element sPrd11 {
   subProduct.elements
}

# correlator integration
corrIntegration.elt = element cIntn {
   # minHW integration time (not sure this is needed)
   attribute minHW { xsd:positiveInteger },
   # hardware integration time (TODO: units)
   attribute hw { xsd:positiveInteger },
   # LTA integration time (TODO: units)
   attribute lta { xsd:positiveInteger }
}

# lag chain (spectrum) segment
lagChainSegment.elt = element lcSeg {
   # rack id
   attribute rck { xsd:positiveInteger },
   # crate id
   attribute crt { xsd:nonNegativeInteger },
   # slot id
   attribute slt { xsd:nonNegativeInteger },
   # corr chip x coordinate
   attribute x { xsd:nonNegativeInteger },
   # corr chip y coordinate
   attribute y { xsd:nonNegativeInteger },
   # first CCC
   attribute ccc0 { xsd:nonNegativeInteger },
   # lasg CCC
   attribute cccN { xsd:nonNegativeInteger },
   # recirculation factor
   attribute rec { xsd:positiveInteger },
   # corr chip X input
   xInp.elt,
   # corr chip Y input
   yInp.elt
}

inp.content =
   # stream label
   attribute strm { "A" | "B" },
   # lag block
   attribute inpBlk { xsd:nonNegativeInteger }

xInp.elt = element xInp {
   inp.content
}

yInp.elt = element yInp {
   inp.content
}
```

```
# output configuration
#
# Comprises a number of subarray configurations.
#
outputConfig.elt = element outputConfig {
   subarray.elt*
}

# subarray configuration
#
# A subarray consists of a number of stations, and it has an
# unchanging number of APC bins and integration time for all spectral
# windows. Auto- and cross-correlation configurations are given
# separately to account for different allowed values of the number of
# polarization products (this could be simplified were we to ignore
# W3C schema compatibility).
#
subarray.elt = element subarray {
   # subarray id
   attribute id { xsd:ID },
   # station id list
   attribute stns { list { xsd:nonNegativeInteger+ } },
   # num APC bins
   attribute nAPC { xsd:positiveInteger },
   # integration (TODO: units?)
   attribute intn { xsd:positiveInteger },
   # auto-correlation products
   autoCorrelations.elt?,
   # cross-correlation products
   crossCorrelations.elt?
}

# autocorrelations configuration
#
# Comprises a list of I/Fs and the spectral windows in each I/F.
#
autoCorrelations.elt = element aCorr {
   element if {
      # I/F id
      attribute val { xsd:nonNegativeInteger },
      # spectral window
      element sw {
         # spectral window id
         attribute id { xsd:ID },
         # number of (phase) bins
         attribute nBn { xsd:positiveInteger },
         # number of spectral channels
         attribute nCh { xsd:positiveInteger },
         # number of polarization
         attribute nPn { xsd:integer "1" | xsd:integer "2" }
      }+
   }+
}

# cross-correlations configuration
#
# Comprises a list of I/Fs and the spectral windows in each I/F.
```

```
#
crossCorrelations.elt = element xCorr {
   element if {
      # I/F id
      attribute val { xsd:nonNegativeInteger },
      element sw {
         # spectral window id
         attribute id { xsd:ID },
         # number of (phase) bins
         attribute nBn { xsd:positiveInteger },
         # number of spectral channels
         attribute nCh { xsd:positiveInteger },
         # number of polarization products
         attribute nPn { xsd:integer "1" | xsd:integer "2"
                       | xsd:integer "4" }
      }+
   }+
}

# pipeline configuration
#
# Pipeline configuration has two functions: assigning input products
# to spectral windows, and describing the pipeline used to move and
# convert the input products to the spectral windows. The assignment
# of input products to a baseline and polarization product in a
# spectral window may make this element very large. Perhaps we could
# create rules to describe the assignments in order to reduce the size
# of this element.
#
pipelineConfig.elt = element pipelineConfig {
   pipeline.elt*
}

inputProduct.elt = element prd {
   attribute bbA { xsd:nonNegativeInteger },
   attribute bbB { xsd:nonNegativeInteger }
}

pipeline.elt = element pln {
   # pipeline id
   attribute id { xsd:ID },
   # (output) spectral window ref
   attribute sw { xsd:IDREF },
   # pipeline description
   attribute desc { text },
   # (baseband, subband)->(pol'n product, sw offset) mapping
   element map {
      # subband id
      attribute sb { xsd:nonNegativeInteger },
      # offset of (subband) product in spectral window
      attribute off { xsd:nonNegativeInteger },
      # LL polarization product
      element pLL {
         inputProduct.elt
      }?,
      # LR polarization product
      element pLR {
```

```
         inputProduct.elt
      }?,
      # RL polarization product
      element pRL {
         inputProduct.elt
      }?,
      # RR polarization product
      element pRR {
         inputProduct.elt
      }?
   }+
}
```

3.5.1.2 W3C XML form

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <!--
    backend configuration

    Configuration is split into three parts: input configuration,
    specifying the correlator hardware configuration w.r.t. the products
    it produces; output configuration, specifying the subarrays and
    spectral windows in the backend output; and pipeline configuration,
    mapping correlator products to spectral windows and specifying the
    processing pipeline into which the input products go.

  -->
  <xs:element name="config">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="inputConfig"/>
        <xs:element ref="outputConfig"/>
        <xs:element ref="pipelineConfig"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <!--
    backend input configuration

    Configuration is split into two parts: the specification of the
    correlator baseline board input streams, and the products produced
    by the baseline boards from their input streams.

  -->
  <xs:element name="inputConfig">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="strm"/>
        <xs:choice minOccurs="0" maxOccurs="unbounded">
          <xs:element ref="prd4"/>
          <xs:element ref="prd7"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

```
<xs:element name="strm">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:ID"/>
    <xs:attribute name="stn" use="required" type="xs:positiveInteger"/>
    <xs:attribute name="bb" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="sb" use="required" type="xs:nonNegativeInteger"/>
  </xs:complexType>
</xs:element>
<!--
  'product4' and 'product7' could be combined with the addition of a
  'mode' attribute to reflect 4 or 7 bit products, but this produces a
  nondeterministic schema that cannot be accurately translated to
  the W3C XML Schema language

  product =
    element product {
      attribute id { xsd:ID },
      attribute lags { xsd:positiveInteger },
      attribute bins { xsd:positiveInteger },
      attribute streamA { xsd:IDREF },
      attribute streamB { xsd:IDREF },
      ((attribute mode { xsd:integer "4" },
        corrIntegration, lagChainSegment+)
        | (attribute mode { xsd:integer "7" },
           corrIntegration,
           subProduct00,
           subProduct01,
           subProduct10,
           subProduct11))
    }
-->
<xs:element name="product.common.content" abstract="true">
  <xs:complexType>
    <xs:attribute name="minHW" use="required" type="xs:positiveInteger"/>
    <xs:attribute name="hw" use="required" type="xs:positiveInteger"/>
    <xs:attribute name="lta" use="required" type="xs:positiveInteger"/>
  </xs:complexType>
</xs:element>
<xs:complexType name="product.common.content">
  <xs:sequence>
    <xs:element ref="product.common.content"/>
  </xs:sequence>
  <xs:attribute name="id" use="required" type="xs:ID"/>
  <xs:attribute name="nLg" use="required" type="xs:positiveInteger"/>
  <xs:attribute name="nBn" use="required" type="xs:positiveInteger"/>
  <xs:attribute name="strmA" use="required" type="xs:IDREF"/>
  <xs:attribute name="strmB" use="required" type="xs:IDREF"/>
</xs:complexType>
<!-- four bit correlation product -->
<xs:element name="prd4">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="product.common.content">
        <xs:sequence>
          <xs:element maxOccurs="unbounded" ref="lcSeg"/>
        </xs:sequence>
      </xs:extension>
```

```
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<!-- seven bit correlation product -->
<xs:element name="prd7">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="product.common.content">
        <xs:sequence>
          <xs:element ref="sPrd00"/>
          <xs:element ref="sPrd01"/>
          <xs:element ref="sPrd10"/>
          <xs:element ref="sPrd11"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:complexType name="subProduct.elements">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" ref="lcSeg"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="sPrd00" type="subProduct.elements"/>
<xs:element name="sPrd01" type="subProduct.elements"/>
<xs:element name="sPrd10" type="subProduct.elements"/>
<xs:element name="sPrd11" type="subProduct.elements"/>
<!-- correlator integration -->
<xs:complexType name="corrIntegration.elt">
  <xs:sequence>
    <xs:element ref="cIntn"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="cIntn" substitutionGroup="product.common.content"/>
<!-- lag chain (spectrum) segment -->
<xs:element name="lcSeg">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="xInp"/>
      <xs:element ref="yInp"/>
    </xs:sequence>
    <xs:attribute name="rck" use="required" type="xs:positiveInteger"/>
    <xs:attribute name="crt" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="slt" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="x" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="y" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="ccc0" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="cccN" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="rec" use="required" type="xs:positiveInteger"/>
  </xs:complexType>
</xs:element>
<xs:attributeGroup name="inp.content">
  <xs:attribute name="strm" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="A"/>
        <xs:enumeration value="B"/>
```

```
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="inpBlk" use="required" type="xs:nonNegativeInteger"/>
</xs:attributeGroup>
<xs:element name="xInp">
  <xs:complexType>
    <xs:attributeGroup ref="inp.content"/>
  </xs:complexType>
</xs:element>
<xs:element name="yInp">
  <xs:complexType>
    <xs:attributeGroup ref="inp.content"/>
  </xs:complexType>
</xs:element>
<!--
  output configuration

  Comprises a number of subarray configurations.

-->
<xs:element name="outputConfig">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="subarray"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!--
  subarray configuration

  A subarray consists of a number of stations, and it has an
  unchanging number of APC bins and integration time for all spectral
  windows. Auto- and cross-correlation configurations are given
  separately to account for different allowed values of the number of
  polarization products (this could be simplified were we to ignore
  W3C schema compatibility).

-->
<xs:element name="subarray">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" ref="aCorr"/>
      <xs:element minOccurs="0" ref="xCorr"/>
    </xs:sequence>
    <xs:attribute name="id" use="required" type="xs:ID"/>
    <xs:attribute name="stns" use="required">
      <xs:simpleType>
        <xs:restriction>
          <xs:simpleType>
            <xs:list itemType="xs:nonNegativeInteger"/>
          </xs:simpleType>
          <xs:minLength value="1"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="nAPC" use="required" type="xs:positiveInteger"/>
```

```
    <xs:attribute name="intn" use="required" type="xs:positiveInteger"/>
  </xs:complexType>
</xs:element>
<!--
  autocorrelations configuration

  Comprises a list of I/Fs and the spectral windows in each I/F.

-->
<xs:element name="aCorr">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="if">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" name="sw">
              <xs:complexType>
                <xs:attribute name="id" use="required" type="xs:ID"/>
                <xs:attribute name="nBn" use="required" type="xs:positiveInteger"/>
                <xs:attribute name="nCh" use="required" type="xs:positiveInteger"/>
                <xs:attribute name="nPn" use="required">
                  <xs:simpleType>
                    <xs:restriction base="xs:integer">
                      <xs:enumeration value="1"/>
                      <xs:enumeration value="2"/>
                    </xs:restriction>
                  </xs:simpleType>
                </xs:attribute>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
          <xs:attribute name="val" use="required" type="xs:nonNegativeInteger"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<!--
  cross-correlations configuration

  Comprises a list of I/Fs and the spectral windows in each I/F.

-->
<xs:element name="xCorr">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="if">
        <xs:complexType>
          <xs:sequence>
            <xs:element maxOccurs="unbounded" name="sw">
              <xs:complexType>
                <xs:attribute name="id" use="required" type="xs:ID"/>
                <xs:attribute name="nBn" use="required" type="xs:positiveInteger"/>
                <xs:attribute name="nCh" use="required" type="xs:positiveInteger"/>
                <xs:attribute name="nPn" use="required">
                  <xs:simpleType>
                    <xs:restriction base="xs:integer">
```

```
                    <xs:enumeration value="1"/>
                    <xs:enumeration value="2"/>
                    <xs:enumeration value="4"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:attribute>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="val" use="required" type="xs:nonNegativeInteger"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
</xs:element>
<!--
  pipeline configuration

  Pipeline configuration has two functions: assigning input products
  to spectral windows, and describing the pipeline used to move and
  convert the input products to the spectral windows. The assignment
  of input products to a baseline and polarization product in a
  spectral window may make this element very large. Perhaps we could
  create rules to describe the assignments in order to reduce the size
  of this element.

-->
<xs:element name="pipelineConfig">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="pln"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:complexType name="inputProduct.elt">
  <xs:sequence>
    <xs:element ref="prd"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="prd">
  <xs:complexType>
    <xs:attribute name="bbA" use="required" type="xs:nonNegativeInteger"/>
    <xs:attribute name="bbB" use="required" type="xs:nonNegativeInteger"/>
  </xs:complexType>
</xs:element>
<xs:element name="pln">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="map"/>
    </xs:sequence>
    <xs:attribute name="id" use="required" type="xs:ID"/>
    <xs:attribute name="sw" use="required" type="xs:IDREF"/>
    <xs:attribute name="desc" use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="map">
  <xs:complexType>
```

```
      <xs:sequence>
        <xs:element minOccurs="0" ref="pLL"/>
        <xs:element minOccurs="0" ref="pLR"/>
        <xs:element minOccurs="0" ref="pRL"/>
        <xs:element minOccurs="0" ref="pRR"/>
      </xs:sequence>
      <xs:attribute name="sb" use="required" type="xs:nonNegativeInteger"/>
      <xs:attribute name="off" use="required" type="xs:nonNegativeInteger"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="pLL" type="inputProduct.elt"/>
  <xs:element name="pLR" type="inputProduct.elt"/>
  <xs:element name="pRL" type="inputProduct.elt"/>
  <xs:element name="pRR" type="inputProduct.elt"/>
</xs:schema>
```

3.5.2 Examples

Nothing useful here yet.

3.6 Plugin design issues

3.6.1 Plugin development

Development of plugins for the backend compute node lag processing stage should be possible without having access to a complete backend system. It is already possible to run a backend compute node application instance on a workstation, and this section will describe what is needed to develop a plugin in such an environment. The focus will be on development for the lag processing stage in particular, as that is the only part of the backend system intended to be extensible.

To develop a plugin for the lag processing stage, one can run the lag processing stage **application** standalone, or through the backend compute node **application**. It will probably be most useful to run the **lagset_pipeline** application, as running the full compute node application requires a lag frame source. On the other hand, running **lagset_pipeline** requires a lag set source instead, but a file-based lag set source should be easier to generate than a lag frame source, since most of the WIDAR hardware-specific fields are not contained in a lag set. Once in possession of a file of lag sets appropriate to the plugin being developed, one also needs a configuration file for the lag processing stage. Creating a configuration file for this narrow purpose should prove fairly simple, but this process has not been attempted, so there is some uncertainty. Ultimately, a tool to create lag sets together with an appropriate configuration file might be useful, but experience developing plugins is a prerequisite to designing such a tool.

To ensure that the plugin development environment is compatible with the deployment environment, it may be useful to create a development toolkit. A toolkit of this kind would comprise customized makefiles together with the copies of the source code for libraries deployed on the target backend system. This toolkit should be relatively easy to create, will greatly ease plugin development, and will minimize problems caused by differences in the development and deployment environments.

GStreamer elements are based on GLib's GObject object-oriented framework for C. While this framework is well developed and widely used, it will not be familiar to most programmers. There are complexities in

the GObject approach that take some time for a new developer to understand, but, in the limited scope of developing new backend elements, the complexities can be minimized, and the development can be guided by employing the object-oriented nature of GObject. A good example of this approach is GStreamer itself, which provides base classes for the development of certain types of elements, such as filters, sources and sinks. For backend plugin development, even more narrowly defined base classes can be prepared to simplify the development of new elements. What classes, exactly, would be needed in this approach, while now uncertain, will become clearer as development of the core pipeline elements proceeds.

3.6.2 Plugin deployment

Plugins in GStreamer are implemented as dynamically linked libraries, and, as such, deployment of the plugins (*i.e.*, libraries) on the backend compute nodes must be considered. Most of these issues ought to be addressed by the cluster management software that will be used for the backend cluster. While the development of new pipeline functionality is eased by the use of GStreamer plugins, it will not be the case that the backend software will be customized by every user through employing user-customized plugins for the backend pipelines. What is foreseen is a library of available pipeline elements to be employed as needed by the users and developers, although general user access will be mediated through the use of high-level processing specifications. Development or testing of new plugins deployed alongside standard plugins will be supported through standard methods employed by the operating system. Using cluster management software is the preferred method for the deployment of standard element plugins, while the deployment of new element plugins for testing should be left to the system experts.