JUC Memo #16

The UniBoard correlator system - an overview

H. Verkouter
June 2014

1. Introduction

At the heart of the Joint Institute for VLBI in Europe's (JIVE) UniBoard based VLBI Correlator (JUC) lies, not surprisingly, the UniBoard hardware. The firmware on its FPGAs perform the complex signal processing that make a collection of data streams from individual telescopes into a synthesized telescope. Without peripheral hard- and software, the UniBoard by itself can not produce scientific output.

This document provides an overview of the whole JUC ecosystem: what components there are and what their role is. For many – not all – components, separate memo's exist to explain in great detail their behaviour and where appropriate, reference to them will be made.

2. The entities

Figure 1 shows the highest level overview of the JUC system. It shows the central role of the UniBoard in the JUC and the flow of data and/or control signals.

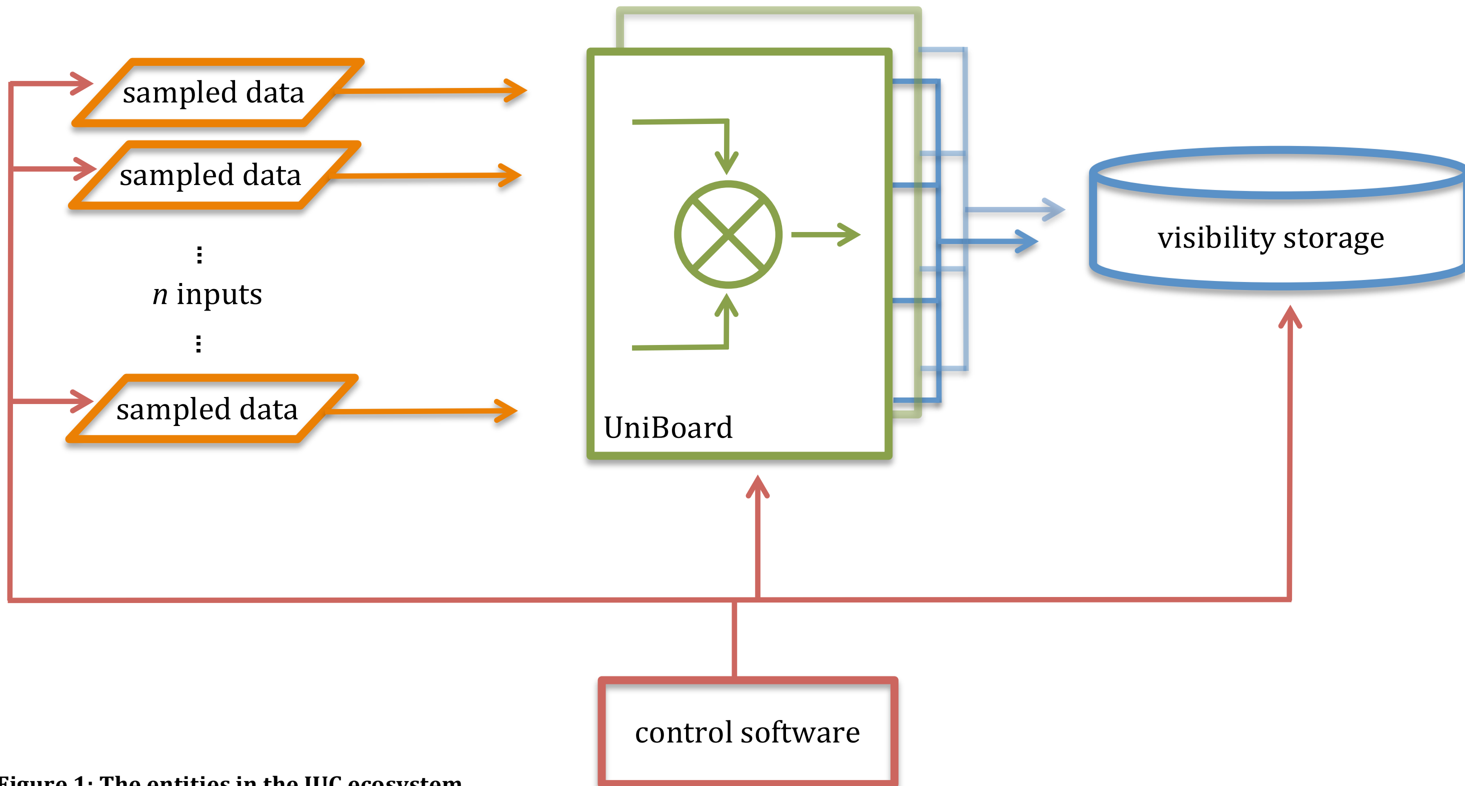

Figure 1: The entities in the JUC ecosystem

As with any correlator, it takes sampled data from a number of inputs, computes the correlation functions of all possible combinations, after which the resulting visibilities are output to persistent storage.

Maybe not immediately obvious is the crucial role of the control software. The reason for this is the UniBoard being a hardware correlator, with all its valuable resources dedicated to signal processing. No resources have been allocated for autonomous control; the board relies on an external control system.

After the visibilities have been written to disk in the JUC native format, the data can be translated to JIVE's internal intermediate format, the CASA Measurement Set (AIPS++ Measurement Set v. 2.0). From this point on UniBoard data will follow the same data path downstream to the end user as data from other correlators that have been or are in use at JIVE (The European VLBI Network (EVN) MkIV data processor and the EVN software correlator at JIVE, SFXC).

The role and more detailed description of the various entities in the system will be explained by following the steps necessary to perform the correlation of a user experiment.

3. Preparation

Before correlation of data can start the most important information which must be available is a description of the observation. The VLBI experiment definition file format (VEX)[1] is used for this purpose. JIVE uses the same VEX file as was used by the stations to control the observation.

At the start of the JUC control software project is was decided to use a relational database system for storing both the observational parameters and the correlation parameters. The fact that that what can be correlated must be a subset of what was observed played an important role in this decision.

It allows the correlation of an experiment to be divided into correlator jobs, where each correlator job refers to (a subset of) the observation. JIVE software implementation note #25 “The correlator control system database”[2] describes the database scheme in greater detail. After an experiment’s VEX file has been downloaded from INAF’s ‘vlbeer’ archive, it will be checked for validity and inserted into this database.

A VLBI correlator must, during correlation, account for the rotation of the earth and the resulting differential velocities of the antennae used in the experiment. The NASA-developed `CALC` software program[3] is used to compute the full geometrical model including relativistic effects due to gravity of solar system objects, ocean loading and other known perturbations of the purely geometric delay. Refer to JUC Memo #2 “On scaling delays”[4] for details.

For each station in the experiment, the “model” delays are pre-computed with one second resolution and stored in a file on disk for use during correlation, see JUC Memo #6 “File format for delay model output”[5].

The user can now use a graphical user interface to select a part of the experiment to be correlated and set the correlation parameters to use. Depending on the correlator some restrictions may apply. For the JUC the spectral resolution is fixed at 1024 channels per subband at a maximum integration time of 1 second.

---

[1] http://vlbi.org/vex/

[2] http://www.jive.nl/~jive_cc/sin/control%20database.pdf

[3] http://gemini.gsfc.nasa.gov/solve/

[4] http://www.jive.nl/wiki/lib/exe/fetch.php?media=uniboard:delay_2012-08-15.pdf

[5] http://www.jive.nl/wiki/lib/exe/fetch.php?media=uniboard:model_file_format.pdf

Using a relational database allowed the implementation of versioned tables. When a user edits the observational parameters of the experiment, e.g. applying a different clock offset and/or clock rate, an updated table will be generated with a different version number and the new values. A record of who changed the table and at what date/time is kept separately. This creates a complete, searchable, record of the settings used for each correlation job. In the past this information was usually lost because the VEX file was edited, copied or renamed, typically without leaving any comments.

The validity and usability of this database approach has by now been well established given that the database was initially developed for the JUC but has been in (production) use with SFXC for the past two years.

4. Correlation

After the user presses 'start' on the aforementioned GUI, a correlator specific program gets invoked. In JUC's case it is a Python script which boots the distributed JUC control system. After that, control is transferred to a program written in the Erlang[6] programming language which drives the actual correlation. The rationale for choosing this language and an introductory explanation of how an Erlang system works can be found in Appendix A.

Figure 1 shows the highest level view and thus leaves out most of the details. In order to understand what happens during correlation a more detailed view is necessary:

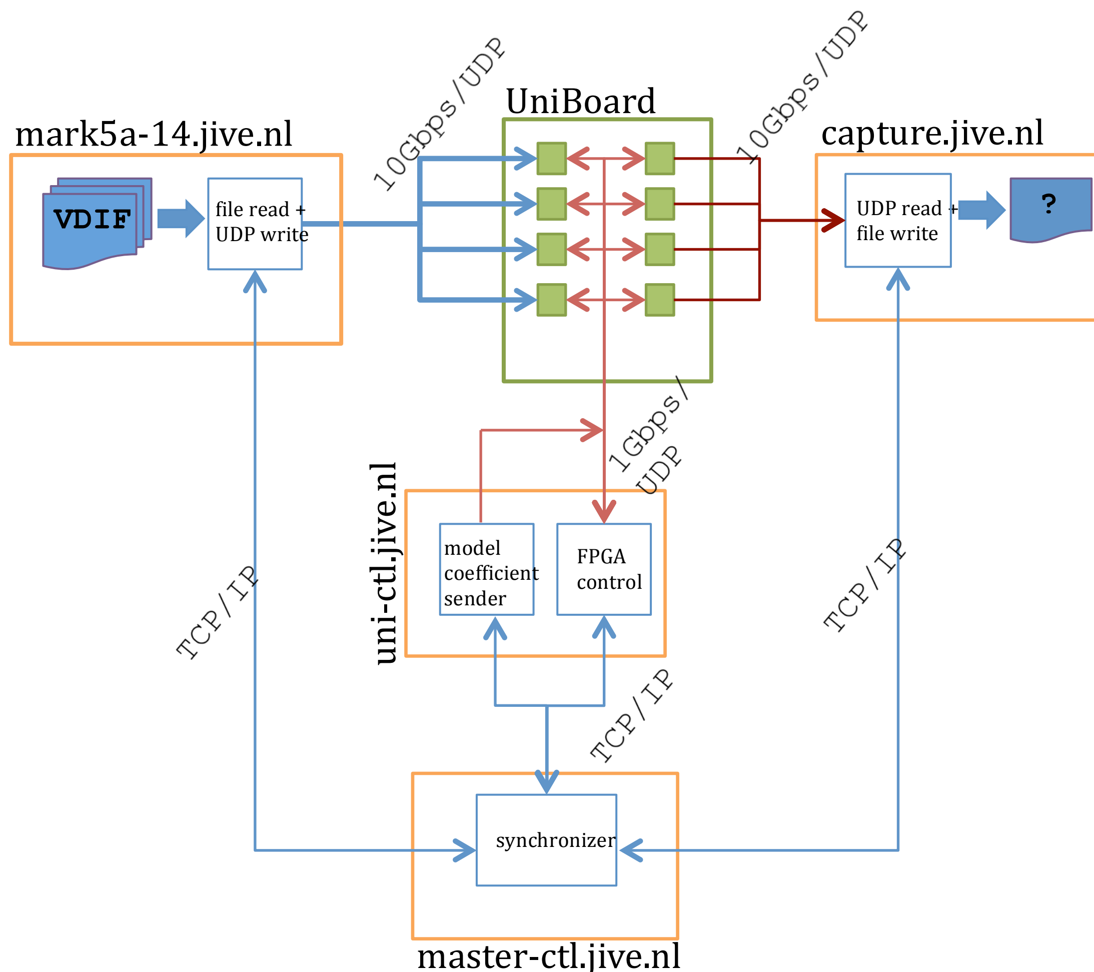

Figure 2: A more complete view of the JUC ecosystem

Figure 2 shows the distributed system surrounding the UniBoard correlator, although, for reasons of clarity, only one data input node out of the availlable 22 has been pictured, "mark5a-14.jive.nl". On the involved hosts (orange boxes with example host names), the Erlang process(es) (blue boxes) and their functionality is shown. Arrows depict the data- and command/control flow and direction as well as the mechanism over which the indicated flow happens.

[6] http://www.erlang.org

4.1 Synchronization

The JUC correlator is passive. It does not read data, does not read the model files nor does it decide by itself to start correlating.

As can be readily seen in Figure 2, data and model coefficients follow different routes into the UniBoard. Given that the correct model coefficients should be applied to the data, some form of synchronization mechanism is necessary.

To this effect correlation is done one UT second at a time. The central Erlang program "synchronizer" instructs the data- and model senders to prepare a new UT second to correlate. The data senders load all data they have for the requested UT second into buffers in the front node FPGAs. The model senders read entries from the model files on disk, interpolate and produce polynomial coefficients for every integration in the requested UT second.

As soon as all senders acknowledge the synchronizer program that they're done, the synchronizer instructs the UniBoard to process the integrations in this UT second. The synchronizer repeats these steps for all UT seconds in the correlation job. JUC Memo #11 "Timing and synchronization" describes this process in detail.

Note that it is not just the different routes necessitating this external synchronization mechanism. The fact that the UDP over IPv4 protocol is used to load data as well as coefficients into the UniBoard severely hampers 'automatic' synchronization, as is explained in Appendix B. It should be noted that for real-time operation the reliable TCP protocol cannot be used for sending the data.

4.2 Sending of data

Figure 2, whilst being far more detailed, still does not represent all the detail the JUC must be able to deal with. One such area is the data input section. One specific data input mechanism is shown in Figure 2 on the host "mark5a-14.jive.nl"; data is read from file on disk. At the moment there are - or will be in the near future - in total three types of recorders present in contemporary EVN observations.

The EVN VLBI correlators must be able to correlate data from all of these systems. To this effect the JUC can process:

- data that was recorded with the MIT Haystack developed Mark5 family of recorders[7] on disk packs
- data that is streamed into the correlator in real-time when the EVN is in electronic VLBI observing mode (e-VLBI)
- data that was written to files on disk, as might have been produced by extraction from Mark5 disk packs or recordings made on the MIT

[7] http://www.haystack.mit.edu/tech/vlbi/mark5/index.html

Haystack Mark6 recording system[8] or the EVN developed FlexBuff recorder

An extra complication is the fact that the JUC *only* accepts data formatted in the relatively recently ratified VLBI Data Interchange Format (VDIF)[9]. More specifically, it only allows a subset of valid VDIF as input – each "channel" (VEX channels in this context) must be sent to a specific IPv4/UDP port number address. More specifically even, the JUC only accepts single-channel VDIF; exactly one channel of data may be carried in a VDIF thread.

Some of the recording systems can never record in VDIF format. Some digital backends (DBEs) allow the output format to be programmable to, amongst others, non-VDIF or incompatible VDIF. In all these situations the correlator environment/control system is responsible to reformat those (recorded) data streams into UniBoard input buffer compatible form.

In the European Commission sponsored project "Novel Explorations Pushing Robust e-VLBI Services" (NEXPReS) the JIVE developed software package `jive5ab` was modified to support this data format translation-and-distribution, also known as "corner turning" or "dechannelizing". The report "D5.10: A cornerturning platform for radio astronomical data"[10], a deliverable of the NEXPReS project, describes this operation in great detail.

The data input section can be run in one of the following modes:

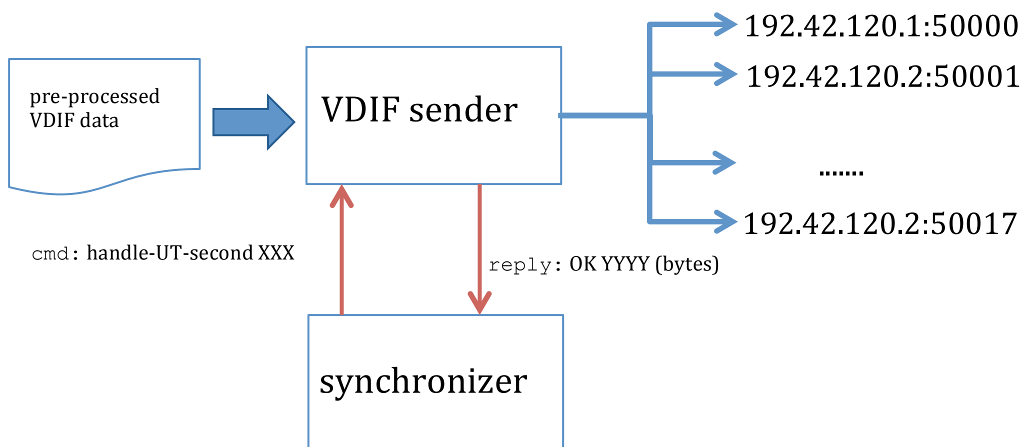

Figure 3: Dealing with pre- dechannelized data

In this mode the JUC software assumes that someone else has performed the task of splitting the recorded data channels into individual VDIF threads. VDIF frames are read and distributed to IPv4/UDP port destinations based on the VDIF thread ID and time stamp found in the header.

---

[8] http://www.haystack.mit.edu/tech/vlbi/mark6/index.html
[9] http://vlbi.org/vdif/docs/VDIF_specification_Release_1.1.1.pdf
[10] http://www.jive.nl/nexpres/lib/exe/fetch.php?media=nexpres:cornerturning-nexpres-1.0.pdf

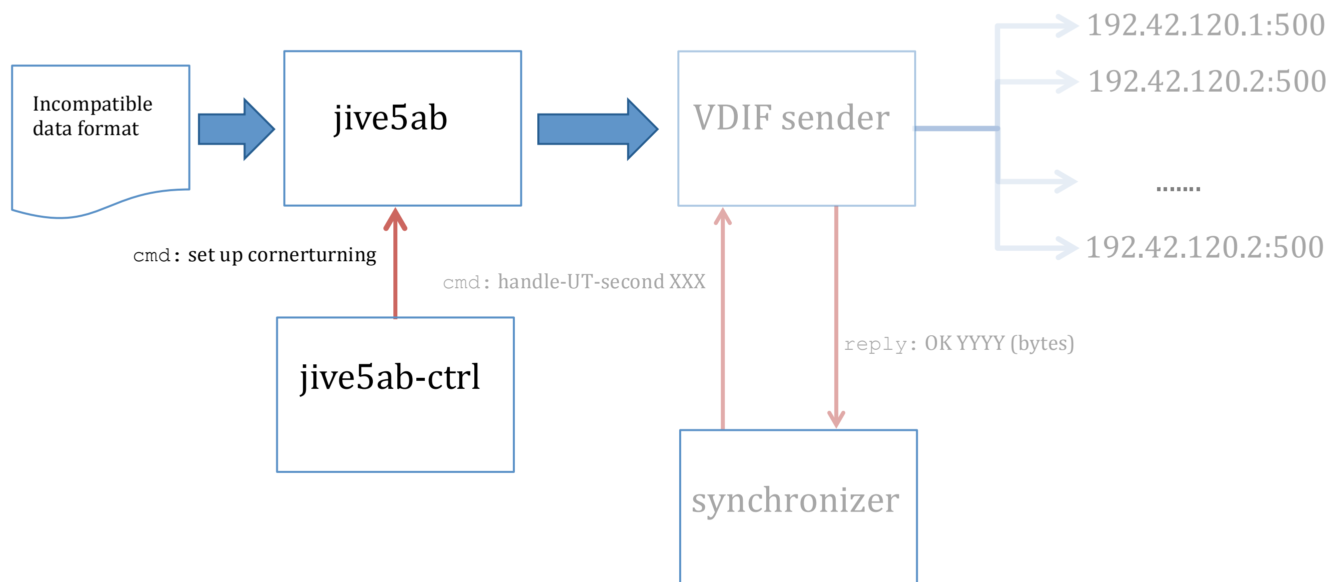

Figure 4: Dealing with incompatible data formats

In this mode an instance of `jive5ab` is started for the data stream(s) which need(s) to be processed. `jive5ab` reads the original data from file, Mark5 disk pack or the network, does the corner turning and conversion to (appropriate) VDIF format. The data are forwarded via a local socket, where the VDIF sender from the previous mode can pick it up. From here on the data flow is identical for all data formats.

Note that in this case another program, the jive5ab controller, must also be started.

The JUC control software inspects the data source for a station when the correlation is configured. It recognizes the need to do on-the-fly data conversion and arranges for the extra software component to be started and configured.

4.3 Communicating with the UniBoard FPGAs

What may not be immediately obvious from Figure 2 is that communication to the UniBoard FPGAs is not direct.

As the “EVN Correlator design” document mentions, each FPGA is controlled using a tiny microprocessor which is responsible for processing command packets received from the network. On a normal ethernet network a lot of packets travel, many of them being broadcasts, e.g. where one particular host on the network is looking for another particular one. None of these packets are of any actual interest to the UniBoard and unnecessarily occupy the tiny microprocessor’s sparse resources.

In practice this can easily be fixed by isolating the UniBoard control interfaces from the normal network. A generic computer can be outfitted with two network cards. One network card directly connects to the UniBoard, the other to the “outside world” network. As long as the computer is configured to the effect that

it does not (automatically) forward traffic received on the outside network's interface to the interface connected to UniBoard, the UniBoard is reachable via this dedicated host but not burdened with unsolicited traffic.

A consequence is that program(s) wishing to communicate with the UniBoard should either (1) be run on this specific UniBoard control host ("uni-ctl.jive.nl" in Figure 2) or (2) arrange for software running on "uni-ctl.jive.nl" which communicates with the UniBoard on behalf of the program(s).

The Erlang programming language facilitates approach 2 trivially: communication to the UniBoard is transparently available from *any* of the hosts in the distributed system depicted in Figure 2. This feature is exploited in another aspect of the system, explained further below.

Communication from the UniBoard control host to and from the UniBoard's FPGAs is implemented using a simple binary client/server protocol over IPv4/UDP. The details of it are explained in the "UniBoard FPGA protocol-version 1.2" document.

4.4 The actual correlation

The JUC is an FX correlator. This means that the data streams are first fourier transformed from the time domain (time sampled data) to the frequency domain and only then correlated.

The Fourier transformation is done on the front nodes of the UniBoard; correlation is performed on the back nodes, after which the data is output in IPv4/UDP packets. Each back node correlates data from one of the four subbands of input data.

Please refer to the "EVN Correlator design" document for details of the whole signal processing chain – explaining it here is beyond the scope of this document.

4.4 Configuring and capturing the output

The JUC takes data from up to two polarizations of four subbands of sampled base band data of up to 32 stations. The firmware always computes all possible correlation products of these inputs. This amounts to 2112 products in total:

> 32 stations = (32 x 33) / 2 baselines = 528 baselines
> 528 baselines x 4 polarization combinations = 2112 products

Each back node of the UniBoard features a programmable "product table". This table of 2112 boolean valued registers specifies which specific products are to be output over the network for the subband correlated by that back node.

Depending on which inputs have actually been filled with data by the data senders (see section 4.2), the correlator control software can select which of the 2112 products on each back node carry meaningful results and programs those products to be output over the network.

On the data capturing host "capture.jive.nl" in Figure 2 a packet receiving program is run. Its only function is to capture IPv4/UDP packets sent at it and write them to a file on the hard-disk for future processing. The packet format is extensively documented in the "EVN Correlator design" document.

The big question is: how does a back node know where to send its output data to? Each back node features a programmable IPv4 address and UDP port number where to send the data to. Thus, in principle, each back node's output could be captured on a different machine. Given the potential output data rate the UniBoard can generate this may well be a necessity rather than a hypothetical possibility.

The capturing software has been written such that it only gets passed the back node(s) it is supposed to capture data from as one of its main input parameters.

The code will program its own IPv4 address and UDP port number into the back nodes' output configuration registers. Even though the host "capture.jive.nl" is not directly connected to the UniBoard, the Erlang distributed system allows code running on "capture.jive.nl" to remotely execute code on "uni-ctl.jive.nl" to allow it to access the back node's configuration registers. The use of Erlang makes this work as simple as it sounds because it has solved most of the technical difficulties that are usually encountered when writing applications that run in a distributed, heterogenous, system such as the JUC ecosystem.

4.4 Post processing and export/delivery of the data product

After the data have been succesfully stored to disk by the capturing program, the data can be moved to JIVE's off-line processing machine.

This dedicated machine is equipped with JIVE's internally developed data translation programs `j2ms2` and `tConvert`.

The `j2ms2` program reads, in JUC's case, the file(s) containing the data payload of the packets received from the UniBoard's back nodes and converts them into so-called Measurement Sets (CASA Measurement Set, AIPS++ Measurement Set version 2.0)[11], the JIVE internal intermediate data format. `j2ms2` combines information from the original VEX file, a ".json"[12] file generated by the control software that specifies which station's data was sent to which input together with the data from the capture file into a consistently labelled data set.

---

[11] http://casa.nrao.edu/Memos/229.html

[12] http://json.org/

Data inspection and flagging occurs using utilities written in various programming languages (C++, glish, Python) operating on Measurement Sets. Finally, the `tConvert` program is used to convert the data from Measurement Set format to FITS-IDI format for distribution to the observer.

Appendix A

The Erlang programming language is particularly well suited to build distributed monitoring and control systems. The decision to go forward with it was made deliberately, after doing a few pilot projects that demonstrated the language could live up to its claims.

It is a so-called compiled byte code language and can, in that respect, be likened to Java and Python. Program text is compiled into byte code which in turn can be executed by a runtime environment – the abstract machine.

Like Java, Erlang's byte code is hardware independent. Quite unlike Java and Python, Erlang is a functional programming language, with all the good and bad that comes with such a language, with a definite surplus of good[13].

Erlang's strong points are (1) parallel processing is for free, (2) parallel processing is *completely* transparent across hardware and operating systems, as long as the `erl` interpreter runs on it, (3) syntax for en- or decoding binary data is part of the language – all of these points were necessities to build a distributed control system that has to communicate with hardware registers.

A distributed Erlang system consists of a number of Erlang nodes, where each node is typically an instance of an Erlang abtract machine. Such a node is just a process that is started on a machine. Nodes can be connected to each other after which compiled byte code can be remotely loaded and executed.

Erlang is a soft real-time system and it is interpreted byte code. The focus of the language has never been brute force performance but rather transparency, distributivity and reliability. Sometimes, however, it is necessary to get a (significantly) higher performance out of the system.

For these situations Erlang allows for easy replacement of the relevant bits of slow code with compiled code. So-called "ports" are external processes that provide a service on behalf of the Erlang code but cannot bring the system down.

Ports being external processes to the Erlang system means they can be written in any programming language.

[13] the fact that languages like C++ and Java are updated to implement features originally found in functional languages is a definite sign of this

Appendix B

Implementing the transmission control protocol (TCP) is prohibitively expensive in logic resources and complexity on resource constrained- or very low level programmable devices like FPGAs. The IPv4/user datagram protocol (UDP), in direct contrast, is very easy to implement in these systems and as such is used by a lot of embedded devices and systems. The JUC is no exception to this.

At this point a slight digression into explaining some of the basics of TCP is appropriate. It is one of the most used IPv4 based protocols. It is reliable because it implements ACK(nowledgment) packets, re-transmission of un-ACK-knowledged packets and local/remote buffer fill level locking.

What this means in practice is that a TCP transmission, from a user's perspective, is extremely simple as well as 'synchronous'. The writer writes bytes into the write end of the connection and at the read end they can be read at the reader's own speed. The TCP mechanism blocks the sender if the receiver's buffer is full and likewise blocks the reader if the sender has not sent any data (yet). Sender and receiver are perfectly synchronized due to TCP's implementation. In order to be able to support this, a TCP implementation needs to keep a lot of state information (logic resources) about the connection and has a non-trivial state transition diagram (complexity).

The UDP protocol has none of this all. It is completely unreliable and has no state. A sender writes a packet on the network and that is it. There is no confirmation the packet arrived at the destination or an indication that the receive buffer at the remote end is already full or wether there is a receiver at all.

In fact, a careless user of the UDP protocol might find him- or herself overwriting their own send buffer, destroying data before it was even sent onto the network, of course without as much as an error or warning.

These properties make it inherently difficult to synchronize a sender and a receiver if they are using UDP to transfer data. The sender must *assume* that the receiver has ample resources (buffer space, processing speed) to keep up with the rate at which it is sending.

A special case where UDP's properties are actually beneficial is in real-time e-VLBI observations. TCP's reliability and fairness priciple[14] keeps it from being able to sustain the transmission speeds needed for this observing mode whereas it is (VLBI-)scientifically perfectly acceptable to loose a fraction of a percent of network packets.

Of course it would be possible to implement ACKs or alternative TCP-like communication on top of UDP but that would be in direct contrast with the reasons to decide on the use of UDP in the first place.

---

[14] TCP prevents a single data stream dominating (in bandwidth usage) if multiple data streams are present on the network

Within JUC's context the following UDP packet loss counter measures have been taken:

- loading data/model coefficients: none
- output of correlated data: none
- communicating with the FPGAs: each command has a reply, so a simple "try/wait for reply/retry" mechanism is implemented, assuming the FPGA commands are idempotent