

The WIYN ODI Instrument Software Architecture

Andrey K. Yeatts^{*a}, Daniel Harbeck^{a,b}, John Cavin^b, Eugene McDougall^a

^aWIYN Observatory, 950 N. Cherry Ave, Tucson, AZ 85719 USA,

^bUniversity of Wisconsin, Madison, 475 N Charter St, Madison, WI 53706, USA

ABSTRACT

As camera focal planes become larger, with higher resolutions and increasingly higher data throughputs, the more they resemble the enterprise data systems found in commercial data centers. The WIYN One Degree Imager (ODI) is such a system. ODI is a mosaic imager with 64 independent CCD detectors with a total resolution of approximately a gigapixel, covering 1 square degree of the sky at the WIYN 3.5 m telescope at Kitt Peak. The ODI camera will bring improved seeing, widefield imaging, new modes of operation and automated integration with the NOAO Science Archive. It will also become the workhorse instrument of the observatory, with high availability and reliability. The new flexibility of the camera will allow (and require) constant refinement of imaging techniques, and calibration and maintenance processes.

Large scale, parallel data processing, management and control will be a constant in the operation of the instrument. We are developing an enterprise level data system using typical Java J2EE constructs. With the advent of relatively inexpensive clustered hardware, scaling of image operations and management to the large volumes of data in ODI should be simplified. We describe an architecture in construction for ODI's 2010 deployment.

Keywords: Wide field imaging, image data handling, cluster computing, OTA guiding.

1. INTRODUCTION

As mentioned above, the ODI instrument provides a gigapixel resolution image of 1 square degree of sky at the WIYN 3.5 m observatory at Kitt Peak. The instrument will be the main system imaging at WIYN, and will provide a unique facility for wide field imaging. Other papers^{1,2} in these proceedings describe ODI's mechanical and optical systems in detail. This paper discusses the software architecture and data handling of ODI, primarily focusing on exposure control and data flow.

ODI presents unique challenges for its software architecture. ODI is required to deliver its image to a high resolution quick look display within 20 seconds after image readout, saving the image within the first 10 seconds of that time. It has real-time requirements for video analysis and guiding. Unlike many wide field imagers, ODI is to be used in classical observing modes as well as automated operation. This implies a level of interactivity and fast turnaround not present in more automated queued systems. ODI is a general purpose instrument that serves a wide range of scientific interests. Many of ODI's uses are not presently known, but will evolve over its lifetime. Like most complex systems, it is desired that most common uses will be conveniently accessible, with a minimum of technical involvement, while still providing a manageable interface for new and unforeseen creative uses. Complicating the issues are a limited software budget, and commissioning scheduled for Spring 2010.

ODI images will become a standard for the WIYN observatory. All the current processes of calibration, configuration, image acquisition, image reduction, display, archiving and retrieval will now be on a new, much larger scale. Current processes cannot be expected to slow down to accommodate the larger data flows with gigapixel images; to the contrary, ODI processes are expected to improve productivity. Calibration, image retrieval, image review, etc. must scale by a factor of hundreds from the current megapixel sizes without degradation in productivity. Furthermore, configuration for the ODI focal plane is an order or magnitude more complex than current instruments. Parallel handling and management of ODI image data will be required to maintain and improve existing cadences.

In this paper, we discuss the platform on which ODI is built and some background on the WIYN science requirements. The controller hardware and cluster integration are very important, and influence configuration and object design. The dataflow processes determine how clustering should be used. We describe a system architecture to manage this dataflow, including control and telemetry. From this basis, we present a software architecture based on the underlying hardware and system fabric that appears capable of meeting the requirements of ODI. We conclude with a description of next steps in development and the vision for ODI's future.

1.1 ODI Hardware Platform

The CCD imaging device used for ODI is a $4k^2$ pixel Orthogonal Transfer Array (OTA)³. The device is subdivided into 64 sub-regions or cells of 480×494 pixels each. The sub division of the device allows a form of random access by cell address to sub areas of pixels. Each OTA device has 8 analog outputs, each servicing a column of 8 cells. Any area of pixels within a cell may be rapidly shifted to the output, enabling sub-cell readouts at rapid rates (~20 milliseconds).

The same shifting process of the OTA design provides charge transfer to adjacent pixel wells along the horizontal and vertical device axes. This facility provides image tip-tilt correction to compensate for telescope vibration and atmospheric effects and should improve the seeing at the WIYN site by 10-20%. The correction process reads out a rectangular pixel area around a star, analyzes it to ascertain its centroid and its deviation from previous positions and uses the resulting errors over several guide stars in a feedback loop to provide tip-tilt correction of the science areas of the focal plane⁴. Guide stars may be chosen around the focal plane to support the varying correction models, from coherent motion artifacts (like telescope wind shake) to air cells above the mirror that exhibit fluid behavior. The sampling and correction loop is to operate at a rate of 20-50Hz; with an integration time of 10-50 msec, this leaves 10-40 msec for video analysis and correction. The capabilities outlined above cover both moderately high data rate transfers (16Gbit over 10 seconds to disk) and low latency (up to 256 guide star video streams analyzed at a rate of 20-50 Hz).

The CCD device controller selected for the ODI camera is the PanSTARRS StarGrasp controller. Each StarGrasp controller provides clocking and bias voltages for two OTA devices, and communicates TCP/IP over optical fiber Ethernet to host computers. Internally, the Star Grasp controller utilizes a PowerPC microprocessor running a custom C command processor on a microkernel with a simple TCP/UDP implementation. Commands are short string sequences, with string replies. Data transfer can run at up to 600 Mb/s over 1000Mb fiber to the control cluster. Aggregated across 32 fiber lines, this provides a possible bandwidth of nearly 20 gigabits per second. Control response is very good, and with the deterministic command engine and wide data path, provide high command throughput on the network side. OTA control is provide by an FPGA clocking engine that can run the devices over their full regime of operation. FPGA processing is fed by a FIFO queue fed by the PPC. These features provide very fine grained and accurate real-time data acquisition without requiring real time behavior from the controlling system. As important as its performance is, a crucial feature of the StarGrasp controller is its buffering and error recovery system. As image data is transferred, a record is kept of success pixel transfers. Retry occurs at each cell row transfer, so packet loss only degrades transmission rate and does not cause image loss.

The cluster used for the WIYN ODI development environment is a Beowulf cluster of 6 compute nodes and a head node running Centos 5.0 and Java 1.6. Each node provides at least 4 2.0 GHz AMD 2200 cores and 4 gigabytes of memory. One of the compute cores provides 16 gigabytes of memory plus 4 2.8 GHz AMD 2212 cores. The head node and one compute node have Intel PRO/1000GT four port Ethernet controller cards. All nodes are connected to a gigabit Ethernet, and all compute nodes are connected with 10Gb/s Infiniband. Currently Infiniband is not used in the system, but it or 10Gb Ethernet will likely play a part in the production system.

Another useful feature of the StarGrasp system in conjunction with a cluster environment is a robust simulator. As the command system is written in C over a minimal kernel, porting to a Linux environment requires only stubbing of the clocking engine. Since communication occurs over IP, it is possible to deploy as many virtual Linux-based StarGrasp simulations as desired. For example, we can deploy multiple simulators across a group of cluster nodes with a single `beo_exec` command.

1.2 ODI Dataflow

Dataflow in the ODI data system breaks down into the phases pictured in Figure 1.

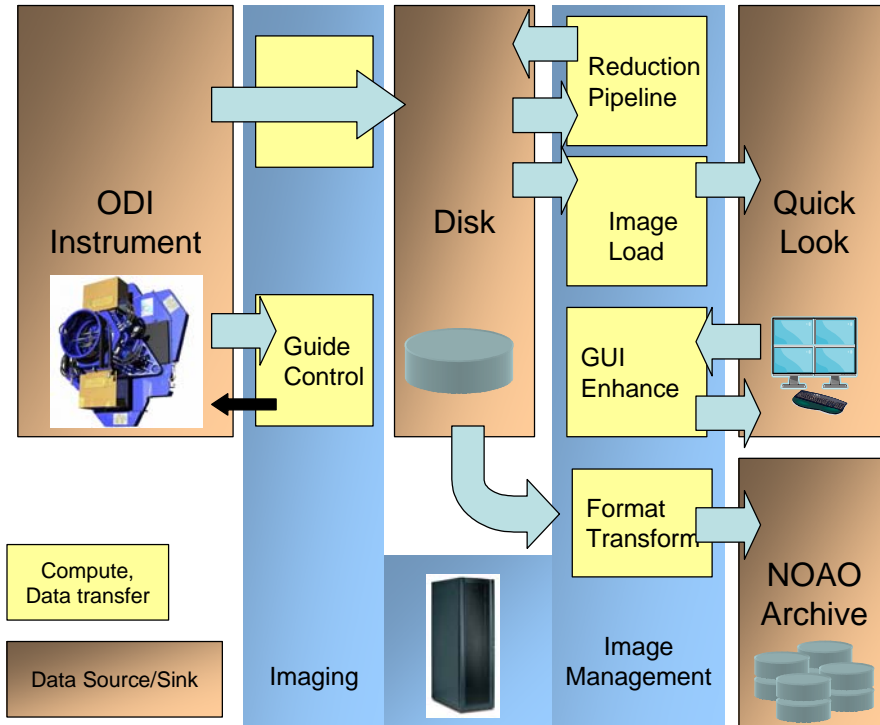


Fig. 1. Dataflow stages in the ODI architecture. The cluster mediates parallel transfers between data sources and sinks, performing pipeline operations in between.

There are two major phases for the ODI image handling: acquiring an image from the instrument and processing the image. The first phase, while it may overlap the first, has real time constraints, and may take advantage of cluster partitioning (separating real time acquisition tasks from image handling). We have measured video data receipt, centroiding and flux computation, data exchange and focal plane shift command computation in Java, and found it to be comparable to a typical C or C++ implementation, with most computations requiring only a few microseconds per star, and data receipt and transmission in the sub-millisecond range. Long term processing will require careful configuration of Java's garbage collection process and thread priorities. With sufficient care and enough resources on the guiding nodes, it may be possible to share guide cluster nodes with low-priority pipeline tasks. Even with 256 32^2 pixel guide stars, this is an incoming data rate of around 200 Mbits/second, or only about 26 Mbits/second per node. The output command stream requires a single command packet or so at 50 Hz, or about 500K bits/sec for each outgoing command stream per node. The computation resource has similar headroom, with a typical 64 core cluster providing several gigaflops per processor.

1.3 Focal Plane Image Acquisition Configuration

The ODI camera control system is shown below. There are 32 independent, hardware synchronizable StarGrasp controllers, each providing clocking and bias voltages to two OTA devices. The controllers each provide a gigabit Ethernet link to the compute cluster. Under commands from the compute cluster, the StarGrasp controllers set device bias and clocking values and initiate operations to read out sub-cell video for guiding or photometry capture, or the entire focal plane for image readout.

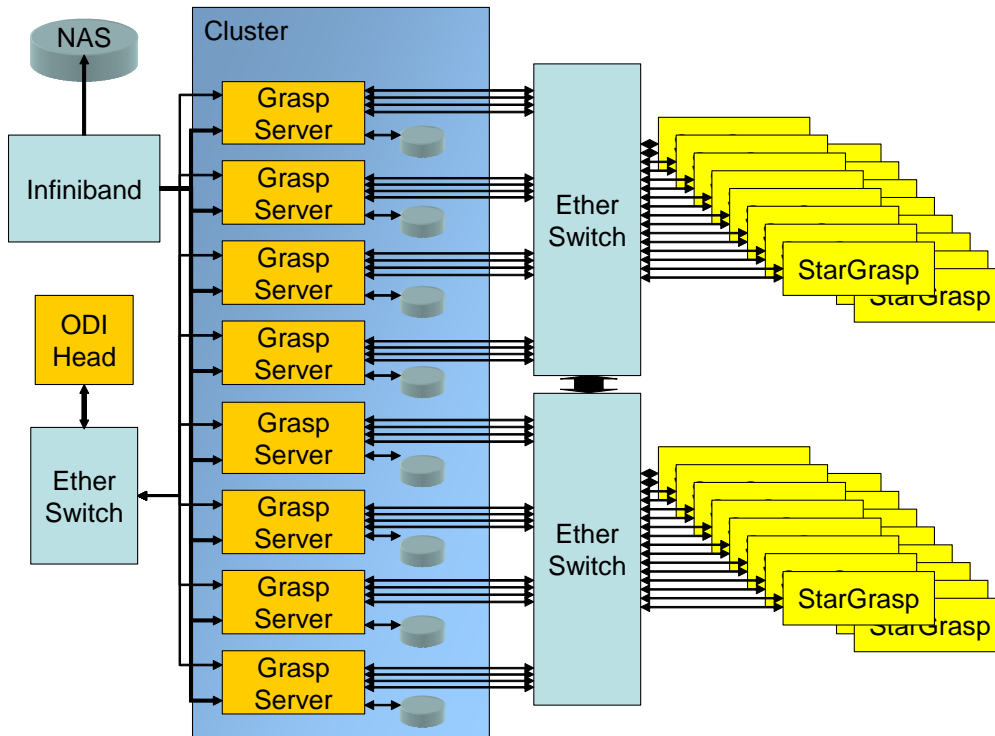


Fig. 2. Physical organization of the cluster, head node, and StarGrasp complement for ODI.

1.4 ODI Control and management

In addition to the complexity of the camera control hardware, an understanding of the imaging artifacts and instrument signatures will accrue over years of experience and practice. It is not feasible to rewrite the image management system repeatedly, so a method of rapid prototyping, followed by refinement and incorporation into an imaging system framework is desired. This is at odds with the complexity of the system, as configuration processes can take place at many levels, from bias voltage tuning of the OTA devices to process optimizations, such as service queues or automation of analysis.

ODI has several demands for immediate analysis of pixel data on site:

- Guide star information must be analyzed in real time as part of the feedback loop. Up to 256 guide stars (from 256 to 4096 pixels each) must be centroided, a correction model obtained, and a guide correction applied to the focal plane controllers in 20 to 50 milliseconds. Data transfer requires a few dozen to less than 2000 microseconds per video transfer, depending on the quantity and size of the guide star extraction boxes. The computation should be limited a few milliseconds to avoid introducing control issues and feedback issues in the guide process, as device clocking and readout latency limit control loop speed.
- Classical observation – each image (with basic overscan correction) is to be viewable within 20 seconds of image readout. The raw image will be saved within the first 10 seconds. The large size of the images precludes direct display on current hardware, so they must be resampled to a lower resolution, and pan and zoom techniques (even if just at a few fixed levels) must be used to confirm target image quality. Raw image storage will run at a rate of about 27 Mbits/sec per OTA. Quick look throughput must aggregate the 32 streams and resample into the parallel graphics level streams. While the final configuration of the graphics system is unknown, the incoming image data rate from a read to the Quick Look system is 17 Mb/s per OTA for a raw image, 35 Mb/s for a 4 byte precision image.
- Basic, level 1 instrument de-trending – bias, dark and flat field correction are to be performed on-site to provide observers with baseline data by the next morning. Assuming 200-250 images produced per night, and an 18 hour duty cycle for computation at the facility, this requires that corrections be performed at a rate of about 4

minutes per image. The dataflow per OTA is raw image, bias image, dark and flat images, or 14 bytes/pixel * 4k² pixels per 4 minutes, about 8 Mbits per second per OTA. The most expensive operation in the sequence is convolution. A convolution must be performed on flat images to eliminate OTA shifting artifacts (different pixels may have different responses) before correction can be done. With the configuration described previously, we can process a 9x9 kernel convolution of a 512x512 pixel cell in about 100 msec, or 6 seconds per serially processed OTA. With one CPU dedicated to each OTA, basic detrending could be accomplished in as little as 30 seconds across an entire image.

- Queue quality control monitoring – while queued mode observing is not in the near term plan for ODI software development, the support of queue mode requires features similar to immediate, classical observation procedures.
- Data export - for the observer, insertion into the NOAO Science Archive, and for engineering quality assessment. Observer and science archive images will likely have metadata requirements differing from the engineering assessments, and so image format transformation will be necessary.

A very useful property of clusters is the pooling of computing resources. By designing the architecture in such a way as to be configurable by resource allocation, scaling is made much simpler, and bottlenecks can be eased by the addition of inexpensive hardware. For example, if we wish to provide image processing during the exposure process without affecting the guide system, simply adding extra compute nodes to service image handling should be sufficient.

We are currently in the process of developing an ODI resource profile for the commissioning (“production”) cluster. By modeling bandwidth requirements as above, we hope to meet performance expectations and allow planning for future capabilities.

1.5 Control and Pipeline System Layout

The ODI camera control and pipeline system is shown below. Under commands from the compute cluster, the StarGrasp controllers set device bias and clocking values and initiate operations to read out sub-cell video for guiding or photometry capture, or the entire focal plane for image readout.

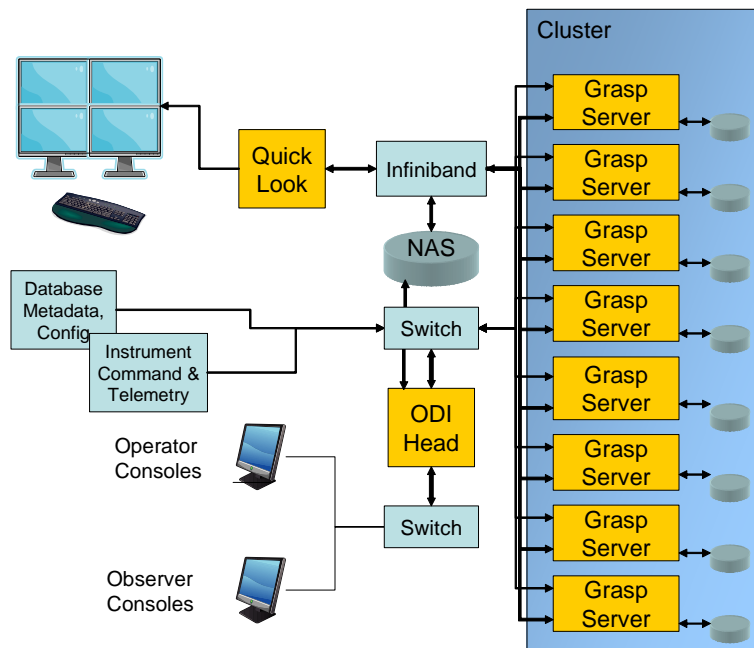


Fig. 3. System configuration for ODI. The ODI head node orchestrates operations for the cluster, mediates telemetry and database recording, and presents an HTTP console for operators and observers.

2. SOFTWARE ARCHITECTURE

2.1 Focal plane control and image management

Parallel programming is complex, and a helpful abstraction will provide a single threaded programming model to the user. The physical resources for focal plane control are controllers and clocking engines; however, the ODI FocalPlane model treats the focal plane as a resource like telescope guiding or the filter change mechanism. The resource is configured, and a remote command is issued. In the case of the FocalPlane, a script is provided describing an atomic operation, such as video readout or an exposure guide process.

While clocking engines may run independently, a simple model that serially schedules work for the clocking engines is sufficient, as currently data may be sent by one clocking engine at a time. For shifting, we will make an exception with shifting command to shorten the command cycle as much as possible. Shifting to two OTAs on the same controller will be a simultaneous operation, expressible across both clocking engines with a single command.

2.2 The StarGrasp Model

The OTA controllers are commanded through a TCP/IP string interface. For instance, the “readout” command takes several parameters, including the designation of the OTA on the controller - 0, 1. Readout also takes parameters to specify binning, etc. A parameterized command might look like:

```
readout dev 0 rowbin=2 colbin=2 ...
```

Each StarGrasp controller is paired with a control thread on a cluster node with its own point to point TCP/IP connection. Groups of control threads are bundled on each Grasp Server node, which presents a network interface to control multiple StarGrasps. The services deployed to a server node are illustrated in the following figure.

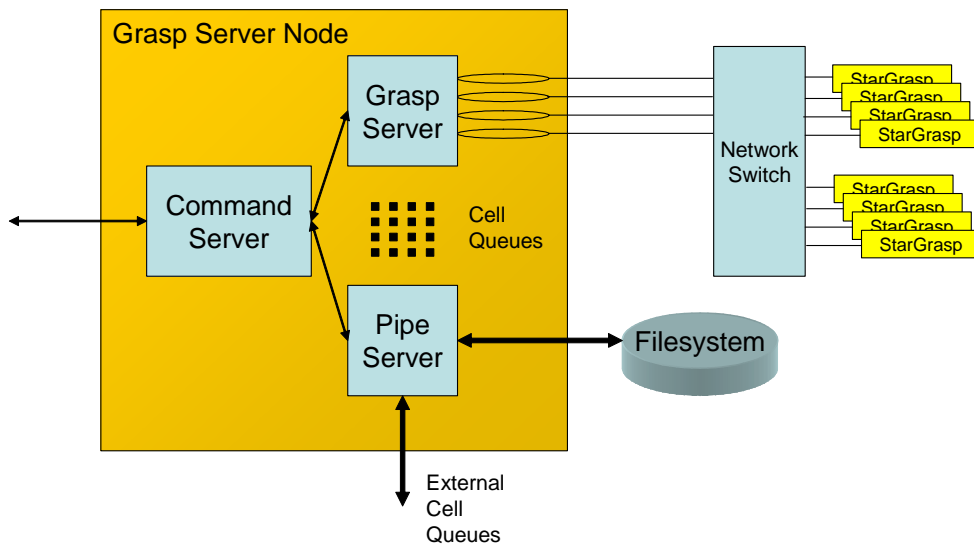


Fig. 4. The Grasp Server node organization. Each Grasp Server node runs a command server that manages the StarGrasp command controller, and a pipeline sever that manages data transformation. Communications between StarGrasp data and pipeline operations is via queues of OTA cell buffers.

The interface to the Grasp Server command server is a set of command lists that designate the StarGrasp controller and the list of commands to be run on the StarGrasp. In Java, this is managed with an

```
HashMap<String, List<Pair<String, String>>> object.
```

The first String is the host name of the StarGrasp, and Pair object pairs a StarGrasp command and the expected response. The response is a regular expression that matches the expected StarGrasp reply to determine success.

While the commands can be sent to any number of controllers independently, most commands are essentially broadcast or multicast to a subset of the focal plane. However, parameters may vary depending on OTA characteristics, and

subsetting groups of controllers or devices requires information about wiring and connectivity. The network configuration is also an input to the `FocalPlane` so various assignment strategies may be used by different commands as required.

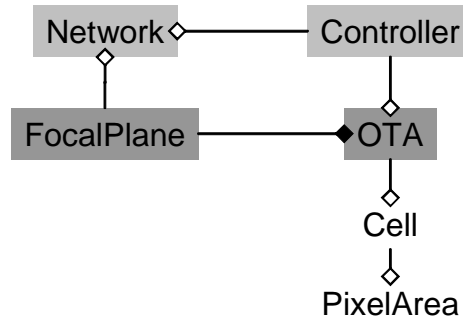


Fig. 5. ODI Focal Plane Component Hierarchy (“geometry”)

In the Focal Plane hierarchy (previous figure), there are three structural sets, the Focal Plane and OTAs (dark shaded), Controller and Network configuration (light shaded), and the Cell and PixelArea set (unshaded). The first two sets are separately configurable, so that the network and controller configurations can be supplied from sources different from the device hardware (the OTAs). Cell and PixelArea are further separated so that they may be created on the fly from scripts, external tools or other sources to describe runtime exposure processes. The FocalPlane is composed of OTAs in rectangular orientation, which have various characteristics like pixel size in microns, pixel width and height of cells, cell configuration, device serial numbers, etc. If the FocalPlane is provisioned with a network and controller configuration, it is capable of transmitting commands to the GraspServer cluster. Cells and PixelArea objects can be created with reference to one FocalPlane (from say, the serialized output of a catalog tool) and retargeted to a different FocalPlane (an actual runtime FocalPlane operating on the telescope). Allowing separable lifetimes eases configuration tasks and simplifies tool integration.

A `FocalPlane` object models the focal plane configuration and accepts lists of StarGrasp commands for execution on the StarGrasps. Rather than requiring each command to be crafted with string operations for each StarGrasp, we abstract them as JavaBeans, setting parameters and the geometry as a group. For example, to perform the readout command on the OTAs in the upper left hand quadrant of the focal plane with 2x2 binning, one could write:

```

FocalPlane focalPlane = FocalPlane.getFocalPlane("ODI1.0");
ReadCmd rc = new ReadCmd();
rc.setRowBin(2); rc.setColBin(2);
for (int row = 7; row >= 4; row--)
    for (int col = 0; col < 4; col++)
        rc.add(focalPlane.getOTA(row, col));
focalPlane.execute(rc);

```

The `FocalPlane` object examines the command given, translates it to the StarGrasp string format, and dispatches it to a set of Grasp Server nodes. Note that the `FocalPlane` is free to allocate Grasp Server nodes as needed using the network to make point to point connections.

JavaBean Command objects translate themselves into StarGrasp string equivalents at the time of the `FocalPlane` execute call. They use a template that extracts StarGrasp node names, device ids, and command parameters to form the `List` objects noted previously. The `HashMap` of commands is assembled and a remote `GraspServer` command interface is allocated and called.

2.3 Focal Plane Scripting

Rather than require the user to assemble sequences of commands for every operation, we group common sequences of commands (charge dumping, or “cleaning”, guiding, readout and image saving) into short scripts called GraspScripts. The GraspScript provides a lookup service, providing copies of named scripts, and allows short, text-based attribute configuration of script instances: for example `gs.config("command[ReadCmd].rowBin", 2)` sets the first `ReadCmd` object’s row binning parameter to 2.

The `FocalPlane` object translates the given GraspScript to a list of destinations and string commands. It then dispatches the lists using JBoss Remoting⁶ to `GraspServer` nodes that handle the mechanics of dispatch and collection of replies. In the next figure, a `save` GraspScript has its geometry set to three OTAs at (0,1), (1,2) and (1,3) on the focal plane. Each of the individual commands is configured with its specific parameters. Upon execution by the `FocalPlane`, the script is translated and remotely executed by a pair of `GraspServer` nodes.

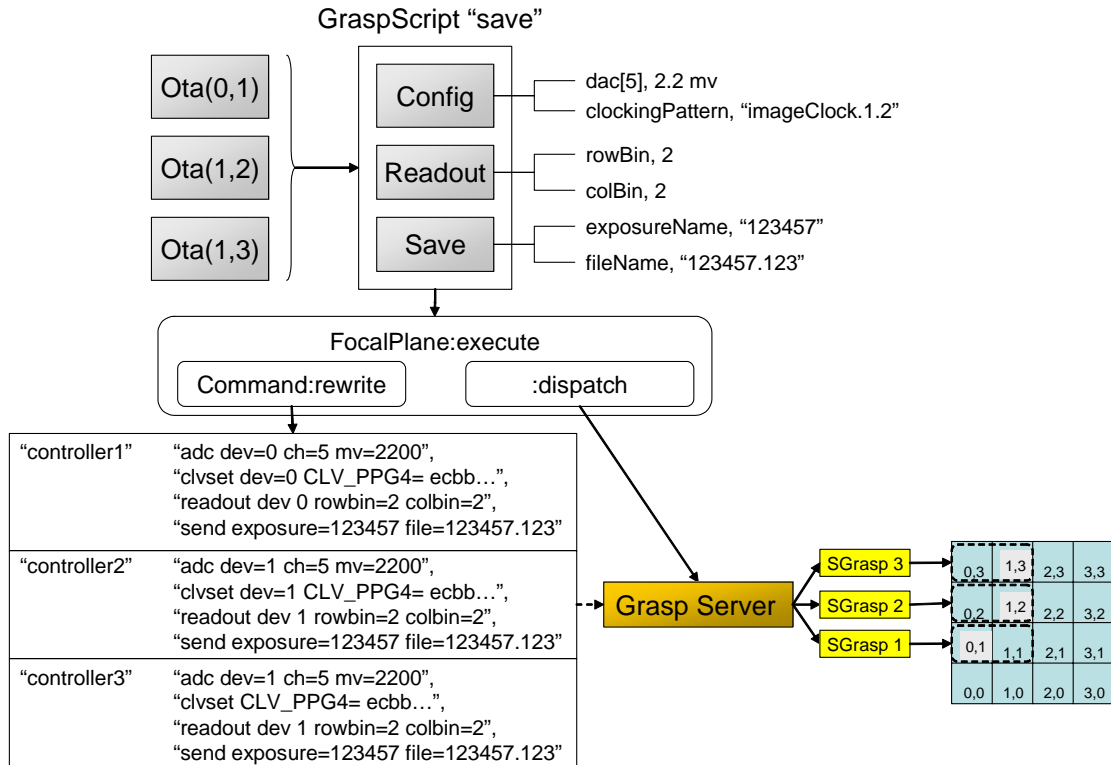


Fig. 6. Processing a script of commands to the Grasp Servers. A sequences of StarGrasp command beans are configured and sent to the FocalPlane to execute. After the commands have been translated to the StarGrasp string format, they are dispatched to the hosts controlling the specified OTAs.

The `FocalPlane` object returns a `HashMap<String, List<String>>` where the key string (the first string in the type declaration) in the `HashMap` is the StarGrasp controller host, and the `List<String>` it indexes are the sequence of replies to each of the commands sent. In the above case, the reply list length would be four, corresponding to replies from the `adc`, `clvset`, `readout` and `save` commands.

Another use of GraspScripts is to provide higher speed and more deterministic response to a sequence of commands than the higher level scripting can provide. While the JBoss Remoting service is efficient the following steps are required:

- marshalling a set of commands `HashMap<String, List<Pair<String, String>>>`
- dispatching them via the network to a `GraspServer`, which
- executes each individual list on StarGrasp command thread and network connection, and

- receives each reply and inserts it into a `HashMap<String, List<String>>`, which is
- transferred to the `FocalPlane` object at the head.

This requires a 10-20 msec overhead roundtrip currently. Commands handled by the GraspServers by contrast, execute in sequence in tens of microseconds of overhead per StarGrasp command, and is thus quite responsive.

2.4 Pipelining

Another resource scheduled on the cluster's Grasp Server Nodes are image processing pipelines. Pipelines are modeled as named event queues, with `Pipeline` operators performing event handling. Each `pipeline` operator reads from an input queue and may write to an output queue. The `PipeServer` instantiates its `Pipeline` operator, configures its input and output queues and starts it on a thread execution service.

Pipelines are configured with a special `Command` bean, `PipeCmd`. `PipeCmd` encapsulates a `Pipeline` and its configuration.

`ImageQueues` are created to interconnect data sources and pipelines. The output from a GraspServer save operation is routed to an `ImageQueue` named for the OTA. The save operation is parameterized with an exposure name (ID), which names the `ImageQueue` set it will produce. On deployment of the pipeline, the `PipeServer` builds an image queue per OTA with the exposure name plus the OTA row and column id. For example, a `SaveCmd` on two OTA (0, 0) and (1,0) with `exposureName` "12345" results in two queues named "12345.0.0" and "12345.1.0" being built by the `PipeServer`.

The diagram below illustrates the configuration after the GraspScript save is deployed on a Grasp Server Node. Two pipeline tasks are created to save files to disk corresponding to the data output from the two OTAs (0,0) and (1,0) serviced by the Grasp Server Node. Two `ImageQueues` are constructed naming the exposure and OTA belonging to it. The save command initiates an image data transfer from the StarGrasp controller, and puts the resulting data on the configured `ImageQueue`.

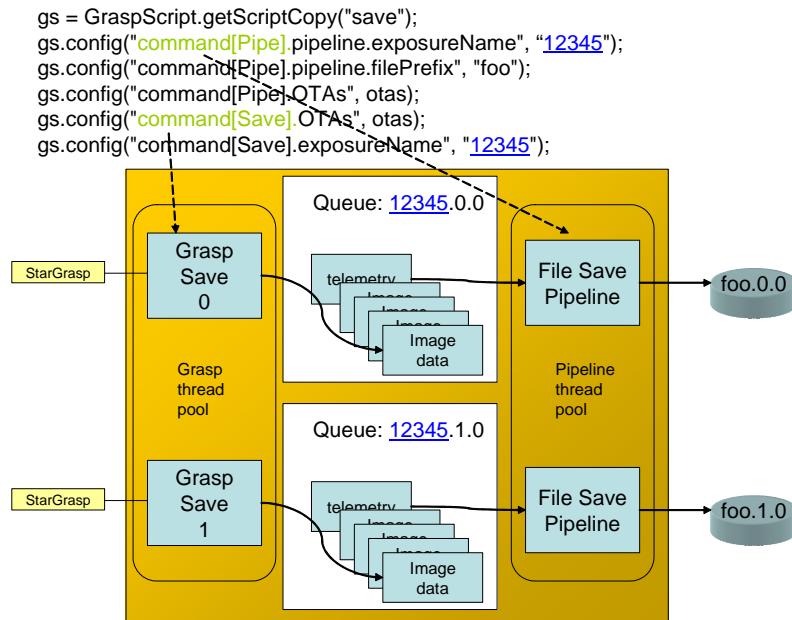


Fig. 7. Pipeline setup as a result of execution of a save script.

2.5 Instrument Level Scripting

The next level of scripting primarily provides a means of handling exposure level processes, such as telescope, shutter, and filter operations, and the metadata and configuration process required there. The following BeanShell script

configures 2x2 binning on a readout of the 4x2 OTA “sim” focal plane instance. The FocalPlaneControl object calls the given script on the FocalPlane when it is run:

```
FocalPlaneControl fpc = Controls.getFocalPlaneControl();
fpc.setFocalPlane("sim"); // 4x2 OTA model

GraspScript gs = GraspScript.getScriptCopy("readout");
gs.config("command[ReadCmd].rowBin",2);
gs.config("command[ReadCmd].colBin",2);
fp = fpc.getFocalPlane();
otas = new HashSet();
otas.add(fp.getOTAs());
gs.config("command[ReadCmd].OTAs",otas);

fpc.setScript(gs);
fpc.run();
```

Using this method of scripting and FocalPlane operation dispatch makes it relatively easy to build compositions of exposure operations. For example, preimages could be used to automatically acquire a set of guide stars. This short, unguided analysis exposure would feed a star extraction process, and populate the guide stage for a guided image acquisition. This process in turn, might be driven by a dither over a set of RA/DEC pointings. A possible script hierarchy is shown in the next figure.

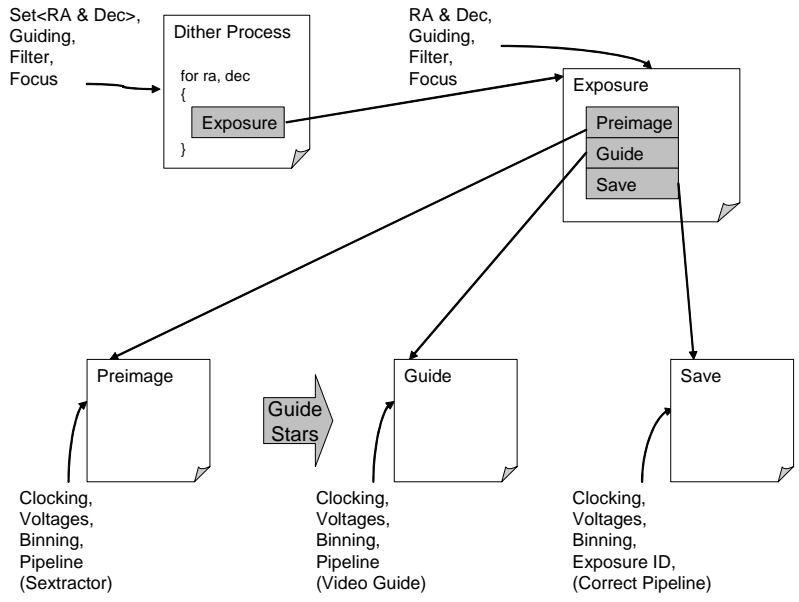


Fig. 8. Pipeline setup as a result of execution of a save script.

For each FocalPlane process, we will create corresponding pipeline processes: FitsFileSave, VideoGuide, BiasDarkFlat, Resample, Sextractor are some of the pipelines to be built to serve ODI operations.

3. CONCLUSIONS

To date, test results with the architecture have been positive. Reading out 4 OTAs and saving to disk in a simulated read of a parallel portion of an ODI focal plane completes in around 10 seconds, with the first cells being written to disk within 200 milliseconds. This would comport well with plans for image pipelining.

Guide processing design and development are next tasks. The StarGrasp command details for sub-cell readout are in discussion, and we hope to begin preliminary tests in the fall. We have tested a rudimentary, Java based guide loop telemetry simulation. using the `java.util.TimerTask`⁷. One server sends pre-captured video to an analysis client that computes centroids and returns a command to the video server. At 50 msec intervals, 90% of cycles were within a few microseconds of the absolute selected time, with the other 10% within 1 millisecond of 50 milliseconds. At 20 msecs, about 80% of the samples were within a few microseconds, with the other samples falling within 2 microseconds. Absolute drift appeared to be within a few milliseconds over several minutes.

We have used several standard utilities of the Java 5 environment to create our pipeline and Grasp Server services, such as the `ExecutorService`, `ThreadPool` system, and `BlockingQueue` objects for multithreading⁸. These high level utilities provide good performance and high quality, tested and flexible interfaces, and rich functionality with no additional libraries outside of the standard Java system. This is a considerable development savings over using hand built thread pooling or synchronization services, and is far more maintainable than custom solutions.

An important feature of enterprise systems is the ease with which they can be integrated with high level services, such as web sites, SOAP, XML/RPC or other remote procedure call services. Embedding ODI's operating software will provide similar benefits. We plan to run the Focal Plane and instrument scripting service inside the JBoss Java Application Server. This will enable us to implement maintenance and management services, such as observation record keeping in a typical web tier application, and allow integration of tools implementing the PLASTIC⁵ interface (for example) to script observations or other camera operations via the PLASTIC XML/RPC interface. Below, Figure 8 depicts the controlling cluster head, and the services to be deployed there.

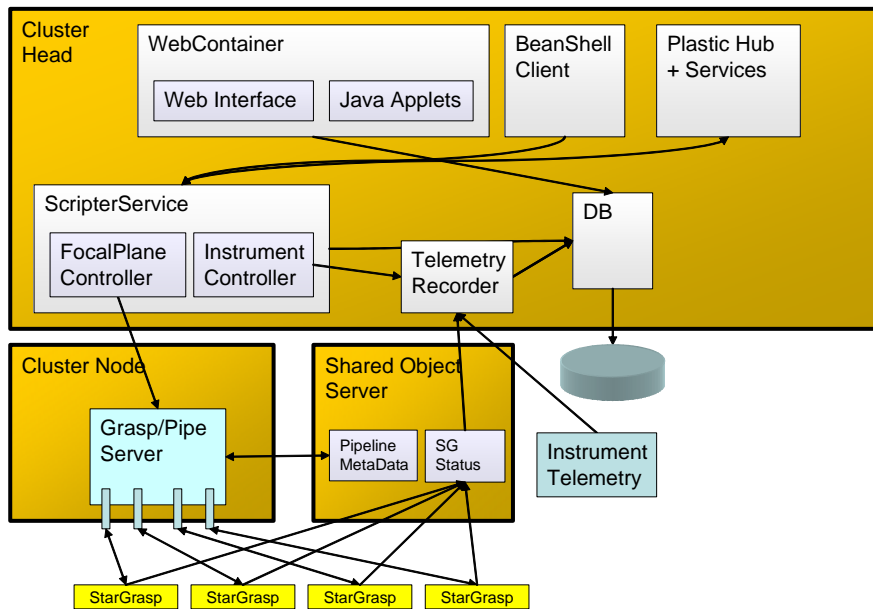


Fig. 9. ODI Management Node (Cluster Head) services.

For example, after installing the PLASTIC hub, it would be relatively easy to add an interface to execute an exposure via the PLASTIC RPC mechanism. The signature might look like:

```

ivo://votech.org/ODI/takeDitherImages(ra2000: double[], dec2000: double[],
    imageName: String,
    exposureDuration: double,
    filters: String[] ) -> success: Boolean
  
```

This enables scripting at the level of tool-integration, allowing ODI to interact with catalogs and high level GUI-based tooling. This further provides an interface for queuing and scheduling in the future. ODI BeanShell scripts can be pulled from a database or operator's HTTP interface, bound with runtime parameters (campaign or account information, or particular system tunings, for example), and dispatched to the ODI system.

Pipelines will similarly be remotely deployed across the cluster via external tools. Enabling pipelines with scripting and managing both pipelines and exposures with the same mechanism could yield useful productivity as well. As a new exposure process is created, its complementary processing pipeline can be developed and maintained. This will aid in the evolution of processes to automate various tasks, from engineering to operations to science.

A key component in such combined interfaces is treatment of image metadata. We are working on methods to distribute metadata across the set of GraspServers that integrate both focal plane information and file-based metadata. This will also provide for metadata transformations, to allow simple re-formatting transformations to remain simple.

Today, great amounts of computing power and communications bandwidth are available in abundance. A number of models have emerged and adapted to take advantage of this "hardware glut." We have begun to apply some of these methods (application servers, Java remote procedures, and Java libraries) and find that they appear to scale well. A useful property of these systems is that workarounds are generally obtainable for performance issues, so we believe that this architecture will scale gracefully and economically as it encompasses new operations and ideas for science.

The authors would like to thank the WIYN Board for supporting the ODI project. QUOTA (QUad OTA), the OTA CCD and filter development projects are supported, in part, by NSF/AST grant 0352979. ODI is supported, in part, by NSF under the TSIP contract C10482T.

REFERENCES

- [1] Jacoby, G. Harmer, C., Muller, G., et. al., "The WIYN One Degree Imager: Optical Design," Proceedings of the SPIE 7014-171, (2008)
- [2] Muller, G., Harbeck, D., Jacoby., G., et. al., "The WIYN One Degree Imager Mechanical Design," Proceedings of the SPIE 7014-170, (2008)
- [3] Tonry, J., Luppino, G., Kaiser, N., Burke, B., and Jacoby, G., "Rubber Focal Plane for Sky Surveys," Proceedings of the SPIE, (2002)
- [4] Burke, B., Tonry, J., et. al., "The Orthogonal-Transfer Array: A New CCD Architecture for Astronomy," Proceedings of the SPIE, (2002)
- [5] Boach, T., Comparato, M., Taylor, J., Taylor, M., Winstanley, N., "PLASTIC – A Protocol for Desktop Application Interoperability Version 1.0", <http://www.ivoa.net/Documents/latest/PlasticDesktopInterop.html>, (2006)
- [6] Elrod, T., Sigal, R., Suconic, C., Haynie, J., Voss, M., Lee, T., Lloyd, D. M., "JBoss Remoting", <http://www.jboss.org/jbossremoting>, (2006)
- [7] Sun Computer Corporation, "JavaDocs, Java 1.5", <http://java.sun.com/j2se/1.5.0/docs/api/java/util/Timer.html>, (2004)
- [8] Sun Computer Corporation, "Concurrency Utilities ", <http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/index.html>, (2004)