P1.2 TURBULENCE REMOTE SENSING OPERATIONAL DEMONSTRATION SYSTEM

Jaimi Yee*, John K. Williams, Gary Blackburn, Steven G. Carson and Jason A. Craig
National Center for Atmospheric Research, Boulder, Colorado

1. INTRODUCTION

In the spring and summer of 2005, under direction and funding from the Federal Aviation Administration (FAA), an operational demonstration was created to test the NCAR Turbulence Detection Algorithm in a real-time setting. This task involved collecting and processing NEXRAD Level II data from sixteen radar sites and integrating that data into a final mosaic product. In addition, turbulence messages were uplinked via ARINC to ACARS printers in selected United Airlines aircraft cockpits. To this end, several new applications were developed and installed in a distributed system consisting of four servers. Operations were maintained on a continual basis, so software tools had to be in place to maintain process control and data flow. In addition, system monitoring tools were employed to make sure any problems were found quickly.

2. ALGORITHMS

The Turbulence Remote Sensing Operational Demonstration system is defined by the algorithms it uses. Indeed the purpose of implementing such a system was to demonstrate the capabilities of the NCAR Turbulence Detection Algorithm (NTDA). This is a fuzzy logic algorithm that uses radar reflectivity, radial velocity and spectrum width to perform data quality control and compute estimates of eddy dissipation rate (EDR) and associated confidences (Williams, 2004). See Figure 1 for a depiction of this algorithm. Note that the NTDA uses an elevation of NEXRAD Level II data. This means that the Level II data must be retrieved and preprocessed before being used in the NTDA.

Another process, called *nexrad2netcdf*, reads NEXRAD Level II data written to disk via Unidata's Local Data Manager (LDM), performs various preprocessing steps on the data and writes it out to disk in netCDF format. It must buffer the data into elevations and compute signal to noise ratio (SNR) and power ratio (PR) data fields. (The power ratio is defined as the ratio of the signal power at each gate to the total overlaid echo power, and is used for data quality control.)

Both *nexrad2netcdf* and the NTDA operate on one radar elevation at a time and produce data on a polar grid. A 3-D mosaic is created by combining the output of the NTDA from several radars into a single Cartesian grid. The process responsible for this is called *confWgtdMosaic,* since it produces a confidence weighted mosaic using the confidence values from the NTDA. This process is triggered by the clock every 5 minutes and reads in all the NTDA data from sixteen radars that was produced in the previous 10 minutes, using only the most recent elevation tilts from each radar. (Note that the time periods and the number of radars are configurable parameters. The values listed above are the current settings for those parameters.)

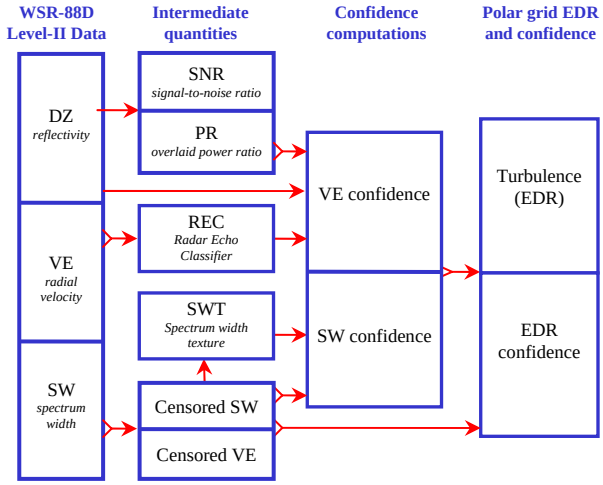

Figure 1: Diagram of the NTDA, as implemented for the WSR-88D (NEXRAD) radar. The Level II reflectivity, radial velocity and spectrum width data are used to censor bad data and compute EDR and an associated confidence for each radar measurement point via a fuzzy-logic framework

Aircraft route and position information are ingested via the Aircraft Situation Display to Industry (ASDI) data stream by a process called *asdi2spdb*. (SPDB stands for symbolic product database and is an internal data format at NCAR's Research Applications Laboratory.) The asdi2spdb application was developed for another project, but it was significantly modified to read and process aircraft route information for this project. It reads the ASDI data stream on a continual basis. Once the turbulence mosaic product is produced, another process called *DrawUplink* uses the route and position information along with the mosaic to produce text messages depicting turbulence over a 80 nm wide by 114 nm region ahead of selected aircraft. (See Figure 2) These messages are then parsed by a process called *asciiParse* to separate out the text messages that will actually be uplinked via ARINC to ACARS printers in specified United Airlines aircraft cockpits. A website was created to allow pilots to review the series of uplinked

* *Corresponding author address:* Jaimi Yee, National Center for Atmospheric Research, P.O. Box 3000, Boulder, CO 80307; email: jaimi@ucar.edu.

messages and provide feedback via a questionnaire and comment form.

```
EXP TURB FI UAL••••••••••••••••
-- 20 Oct 2005 22:43:59Z
FL 360 orient. 270 deg
'X'=aircraft, '+'=waypoint, '*'=route
' '=no_data, 'o'=smooth, 'l'=light
'M'=mod, 'S'=severe
-------------(33nm to BAYLI)----------
      ll          ll *    o
112nm ll           *
        llll       **
108nm lMMllllll     *
      lMMMlll       *
104nm MMMMl l       *
      MMMMl l  lo   *
100nm MMMMM   lll   *     l
      lMMMMl lll    l*llllllll
096nm llMlll ll     l*llllllll
      ll  ll  ll    l*llllll      oo
092nm llllll  l     l*llllll
      llllll ll     ll*llllll
088nm llll  ll      l*llllll
      ll            l*lllllllll l
084nm lll           l*lllllllllllll
+CAP   ll           l+lllllll llllll
080nm               l*llllll   llllll
                     *llll     llllll
076nm                * ll      ll lll
                     *            lll
072nm M              *           llll
      M  ll        llll*          lll
068nm Mllll        llll*ll
      Mllll         lll*ll    MM  lll
064nm llllll       lll*lll   MMMllllll
      lllllll       ll*lll   MMllllllll
060nm llllllll       l*llll    llllll
      llllllll       l*llll    llll l
056nm MMllllll       M*Mll
      MMllllll       M*Mll l
052nm Mllllllll      M*lll ll
      l  llllllo      *    l
048nm l   oooo       *
      l    ool       l*l l    ll
044nm       oo       l*lllllllll
                      l*llllllolll
040nm                  *llllll
                       * l
036nm         ll       *
              llll     *
032nm         llll     *
                l lll  *
028nm             llllll *      ool
                  llloll *  ll   o
024nm              oool  *  ll
                       l *
020nm           oooo   l *
                 ooool  *
016nm            ooolll *
                   oolll *
012nm               ollll*
                     ll  *
008nm                    *
                         *
004nm                    *
                         *
valid ----------------X-*--------------
2240Z -40nm     (39.9N, 87.7W)    +40nm
```

Figure 2: Sample text-based turbulence map generated for a flight from Washington Dulles to San Diego on 20 October 2005. The initial aircraft location is indicated by the X near the bottom, asterisks denote the filed route, and waypoints are indicated by a "+" along the route and labeled in the left margin, which also shows distance in nm along the expected path. Turbulence intensities are denoted by "o" (smooth), "l" (light), "M" (moderate) and "S" (severe).

These processes make up the core of the Turbulence Remote Sensing Operational Demonstration system. Maintaining these processes and the associated inter-process communication is the driving force for the rest of the system design. Figure 3 gives a simplified version of the operational system in order to demonstrate how the processes work together.

Note that the display is not considered part of the core of the system. This is because the display is a Java application that the user downloads to his or her own host. It runs there and communicates via the internet to retrieve data from the operational demonstration system. See Figure 4 for an example of this display and see http://www.rap.ucar.edu/projects/jade for documentation of this software.

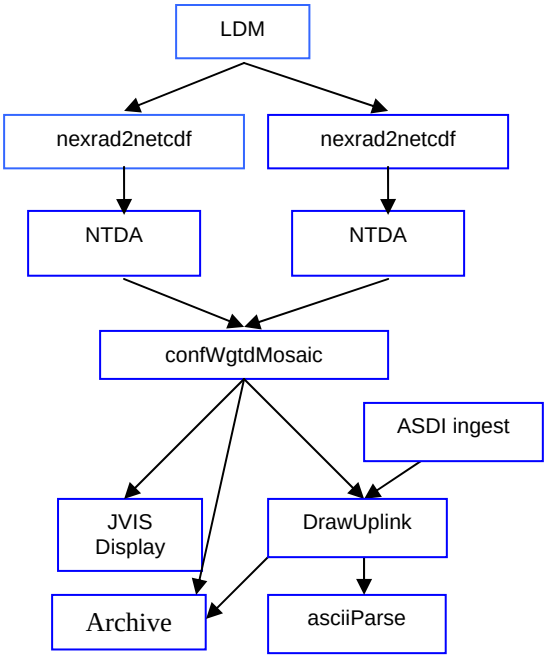

Figure 3: Simplified depiction of algorithm/data flow in operational demonstration system

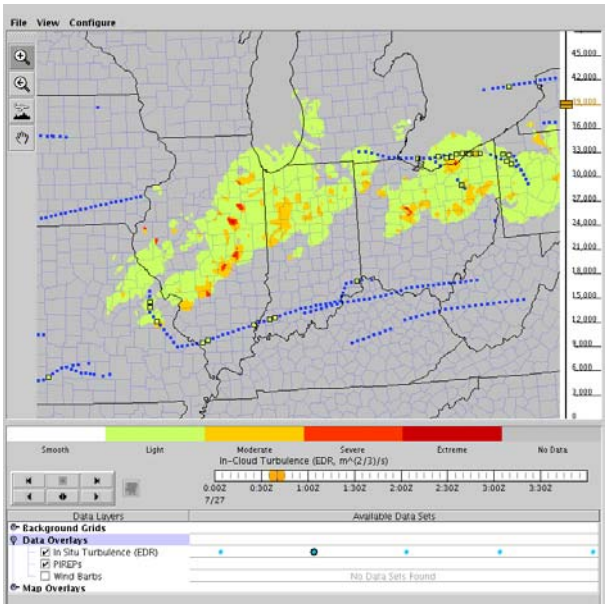

Figure 4: Interactive Java display for disseminating the NTDA operational demonstration data, shown for 0:40 UTC on 27 July 2005. Overlaid are *in situ* turbulence values reported by United Airlines B-757 aircraft.

3. SYSTEM ARCHITECTURE

As mentioned above, *nexrad2netcdf* and the NTDA are designed to work on a single elevation of radar data at a time. Since this system uses sixteen radars, it is necessary to run sixteen instances of each of these processes. In addition two mosaic products are produced. One product is the eddy dissipation rate (EDR) data and its associated confidence on a Cartesian grid having horizontal spacing of approximately two km and vertical levels of multiples of 3,000 ft. The other product consists of the Cartesianized reflectivity field. Thus there are also two instances of the mosaic as well as many other processes that run in this system. It is for this reason the operational demonstration is a distributed system.

Role	Host
Control	delphi1
Ingest1	delphi1
Ingest2	delphi2
Ingest3	delphi3
Ingest4	delphi4
EDR Mosaic	delphi3
Reflectivity Mosaic	delphi4
Uplink	delphi3
Archive	delphi2

Table 1: List of roles in the operational demonstration system with their associated servers

The current system is made up of four 3.4GHz dual Xeon processor servers running Debian Linux. Each server or host takes on one or more “roles”. For example, the uplink host is responsible for maintaining the processes associated with the turbulence uplink product. (See Table 1) Thus processes with like goals are grouped on a given host. But it also means that processes can be moved to a new host with relative ease.

The role definitions are defined in a central file. When a host starts up, it determines what its roles are and then concatenates lists of processes and crontabs, etc. to start and maintain the given processes and any support processes.

The architecture is a centrally controlled one, however. One host is defined as the control host. When the system startup command is evoked on this host, it starts itself and tells the other hosts to start themselves. Likewise, it shuts itself down and sends word to the other hosts to shutdown as well when the system shutdown command is evoked.

4. DATA ARCHITECTURE

The data hierarchy is defined in a similar way on all the hosts in the system. All the data resides under a single directory on a single disk. If the data does not all fit on one disk, links may be set from the location of the data into the main data directory. Each dataset should be grouped under a data directory that identifies its format and type. For example, the NTDA writes out netCDF files, so the data for a given instance are found in a path named in the following way:

netcdf/ntda/<radar name>/<date>

Other data in the system are stored similarly.

5. PROCESS CONTROL

All of the above mentioned processes are designed to be running all the time. They either wait until data arrives, or, as the case of the mosaic, begin processing according to the clock time. This has several advantages. First, it means that applications do not waste time doing startup initialization tasks every time they process data. It also means that it is very easy to tell if there is a problem with a given process. If it is not running, there is some kind of error. In addition, it can make inter-process communication simpler, since the processes that must communicate are always in existence.

However, this type of architecture requires that there is a mechanism for restarting processes that die for various reasons. Fortunately, the Turbulence Remote Sensing Operational Demonstration was able to leverage work that has already been done in NCAR’s Research Applications Laboratory (RAL). Years of trial and error and redesign has led to some very stable and useful tools for process control.

The process control scheme begins with an application called *procmap*, or the process mapper. All the processes in the system that are to be restarted when they die or are hung must register with the process mapper on a regular basis (at least once a minute). (See Figure 5) When they exit, they must unregister with *procmap*.

Name	Instance	Host	User	Pid	Heartbeat
DataMapper	primary	delphi3	trs	2569	0:0:9
DrawUplink	primary	delphi3	trs	2595	0:0:8
DsFCopyServe	manager	delphi3	trs	2851	0:0:50
DsFileDist	primary	delphi3	trs	2668	0:0:6
DsMdvServer	5440	delphi3	trs	2638	0:0:7
DsServerMgr	primary	delphi3	trs	2542	0:0:9
Janitor	primary	delphi3	trs	2554	0:0:9
NCARTurbDete	KDTX	delphi3	trs	2747	0:0:2
NCARTurbDete	KILN	delphi3	trs	2762	0:0:1
NCARTurbDete	KLOT	delphi3	trs	2778	0:0:59
NCARTurbDete	KLSX	delphi3	trs	2793	0:1:0
Nexrad2Netcd	KDTX	delphi3	trs	2683	0:0:5
Nexrad2Netcd	KILN	delphi3	trs	2698	0:0:5
Nexrad2Netcd	KLOT	delphi3	trs	2714	0:0:4
Nexrad2Netcd	KLSX	delphi3	trs	2729	0:0:4
Scout	primary	delphi3	trs	2581	0:0:9
asciiParse	uplink	delphi3	trs	2609	0:0:8
asciiParse	web	delphi3	trs	2624	0:0:7
confWgtdMosa	edr	delphi3	trs	2654	0:0:6

Figure 5: Listing of processes that are registered with the process mapper. Note the heartbeat column. This

gives the time in seconds since the last registration of that process with *procmap*.

A list of processes that are to be maintained in this way is created and stored on disk. This list contains the process name, instance name, start script and kill script for each process. The start script must first check to see if the given process, with the given instance, is running. If it is already running, it does not start the process. This is a preventative measure to avoid starting too many instances of a given process.

An application called *auto_restart*, reads the process list and determines how long it has been since a given process, with the given instance name, in that list has registered with *procmap*. If the time between registrations has been too long or if the process has unregistered, the process is killed with the kill script and restarted with the start script.

Now it is possible that *procmap* or *auto_restart* could die, so these processes are maintained via cron. Other processes in the system that do not need to be running all the time, such as data archival scripts, can also be added to the cron.

6. DATA FLOW

One of the key parts of system like this is the inter-process communication or the data flow. The Turbulence Remote Sensing Operational Demonstration system makes use of latest data information files. The files of this type that are used by this system are actually message queues. When an application writes out a data file, it writes a message to the queue. Downstream processes that make use of that data watch the appropriate queue. When a new message is written, those processes watching the queue “wake up” and grab the data file that matches the file name and time written to the message queue.

The version of *nexrad2netcdf* that ran operationally this summer did not make use of the latest data information queue to read data files from the LDM. Instead it polled the input directories looking for a new file. It made use of the fact that the LDM files for a given volume are named with the start time of the volume and a sequence number. The end of the volume is marked by a file with an E instead of the sequence number. This allowed *nexrad2netcdf* to look for the next file in the sequence and wait until it appeared. However, after running this operationally, it was apparent that this was not the most efficient way to find the next file. Therefore, the LDM was setup to write messages to latest data information queues using an application called *LdataWriter* in preparation for the next version of *nexrad2netcdf*. This new version will make use of the latest data information queues.

When it writes a file, *nexrad2netcdf*, writes a message to the latest data information message queue. This allows the NTDA to determine when to read in the appropriate netCDF file. The NTDA also writes a message to the queue on output.

As was mentioned before, there are sixteen instances of *nexrad2netcdf* and the NTDA running as part of this system. There are four servers in the operational demonstration system, so there are 4 instances of each of these applications running on each server. The two instances of the mosaic are running on separate servers in the system. Each instance of the mosaic requires output from all sixteen instances of the NTDA. Thus any NTDA data that does not reside on the server where the mosaic is running must be pushed there from the other servers. This is accomplished by using an application called *DsFileDist*. This process uses the latest data information messages to determine when to copy a given file to another server. It also keeps the latest data information message queue up to date on the remote server.

The mosaic algorithm then makes use of the latest data information message queues for all sixteen radars by finding the time of the latest file in each case. It then reads in all the data for the previous ten minutes from each radar, using the most recent data from each elevation tilt to create the mosaic.

As mentioned above, the *DrawUplink* application uses the EDR mosaic data as well as the ASDI data processed by asdi2spdb. *DrawUplink* uses the latest data information queue to trigger data processing. It then reads in the mosaic data and contacts a process called *DsSpdbServer* to retrieve data from the data bases created by asdi2spdb (aircraft position information and aircraft route information). *DrawUplink* creates data in two formats: netCDF and ASCII. There are two instances of the process called *asciiParse* running in this system. In both cases, they wait until they detect that a new file has appeared in the appropriate data directory to read the ASCII formatted output from *DrawUplink*. The first instance creates text files that are then pushed using *DsFileDist* to another server outside of the operational demonstration. These messages are then sent via ARINC to the ACARS printer in the appropriate United Airlines cockpit. The second instance of *DrawUplink* creates files that can then be used in a web display.

The mosaic data is also used in the web based display. The data is retrieved by the display through a process called *DsMdvServer*, which processes requests from a client and sends the appropriate data back.

The above mentioned processes such as *DsSpdbServer* and *DsMdvServer* are part of a general suite of client/server based applications developed in RAL for data service. Any of these processes can be started by themselves, but they can also be controlled by another process called *DsServerMgr*, or the server manager. This process waits until a request for a specific type of data is made. It then starts the appropriate data server. It also kills data servers that have been inactive for a lengthy period of time.

DsFileDist works in cooperation with another server application called *DsFCopyServer*. This server runs on the receiving host and is also maintained by the server manager.

========== Data on host 'localhost' at time 2005/11/03 23:22:27 ==========

DataType	Dir	HostName	Latest time
ascii	TurbRemoteSensing/ascii/all	delphi3	2005/11/03-23:20:19
javascript	TurbRemoteSensing/js/web	delphi3	2005/11/03-23:19:30
mdv	TurbRemoteSensing/mdv/mosaic	delphi3	2005/11/03-23:20:00
nc	TurbRemoteSensing/netcdf/messages	delphi3	2005/11/03-23:20:02
nc	TurbRemoteSensing/netcdf/nrda/KCLE	delphi3	2005/11/03-23:20:59
nc	TurbRemoteSensing/netcdf/nrda/KDTX	delphi3	2005/11/03-23:18:20
nc	TurbRemoteSensing/netcdf/nrda/KDVN	delphi3	2005/11/03-23:19:30
nc	TurbRemoteSensing/netcdf/nrda/KGRR	delphi3	2005/11/03-23:17:39
nc	TurbRemoteSensing/netcdf/nrda/KILN	delphi3	2005/11/03-23:19:33
nc	TurbRemoteSensing/netcdf/nrda/KILX	delphi3	2005/11/03-23:17:51
nc	TurbRemoteSensing/netcdf/nrda/KIND	delphi3	2005/11/03-23:19:53
nc	TurbRemoteSensing/netcdf/nrda/KIWX	delphi3	2005/11/03-23:19:19
nc	TurbRemoteSensing/netcdf/nrda/KJKL	delphi3	2005/11/03-23:19:10
nc	TurbRemoteSensing/netcdf/nrda/KLOT	delphi3	2005/11/03-23:20:03
nc	TurbRemoteSensing/netcdf/nrda/KLSX	delphi3	2005/11/03-23:20:09
nc	TurbRemoteSensing/netcdf/nrda/KLVX	delphi3	2005/11/03-23:20:17
nc	TurbRemoteSensing/netcdf/nrda/KMKX	delphi3	2005/11/03-23:20:07
nc	TurbRemoteSensing/netcdf/nrda/KPAH	delphi3	2005/11/03-23:19:05
nc	TurbRemoteSensing/netcdf/nrda/KPBZ	delphi3	2005/11/03-23:19:30
nc	TurbRemoteSensing/netcdf/nrda/KRLX	delphi3	2005/11/03-23:19:20
nc	TurbRemoteSensing/netcdf/radarRaw/KDTX	delphi3	2005/11/03-23:18:20
nc	TurbRemoteSensing/netcdf/radarRaw/KILN	delphi3	2005/11/03-23:19:33
nc	TurbRemoteSensing/netcdf/radarRaw/KLOT	delphi3	2005/11/03-23:20:03
nc	TurbRemoteSensing/netcdf/radarRaw/KLSX	delphi3	2005/11/03-23:20:09

Figure 6: Listing of datasets that are registered with the data mapper. Note that this is not the full listing. There are other fields available, such as start time, end time and number of files. However, the primary file format for this system is netCDF, which is not a supported data format for those fields.

Just as the process mapper keeps track of which processes are running, the data mapper keeps track of which data sets are up to date. Data sets that have associated latest data information queues are usually registered with the data mapper. This means that each time a data file is written and the message queue is updated, the data mapper is notified. (See Figure 6) Thus it is possible to see when there is a delay in the arrival of a data file. Data sets that do not have associated latest data information queues can still be updated in the data mapper via an application called the *Scout*. This process looks through the data tree to see when data files have been written to the various directories and reports that information to the data mapper.

7. DATA MANAGEMENT

One of the issues with a system like the Turbulence Remote Sensing Operational Demonstration is the vast amount of data that is produced. This requires some specific data management to ensure that appropriate data is saved, while also making sure that the disks do not fill.

The operational demonstration system made use of a two TB Apple Xserve RAID that was connected to one of the servers to archive well chosen data sets. This was done by creating several simple scripts, which were run daily, to copy data to that RAID. Not all the datasets were archived, but those key to later validation efforts were maintained. As this was the first year of operations, considerable trial and error took place to determine which datasets could feasibly be archived and how to maintain the datasets in such a way that they did not take up too much room.

The other goal of data management is to ensure that the data disks do not fill. To that end, the operational demonstration system made use of an existing process called *Janitor*. This process traverses the data directory tree and first zips older files and then deletes them at the appropriate file age. (All of those ages are configurable.) In the Operational Demonstration System, five days of data were maintained on disk, and hence available to users of the Java display.

8. SYSTEM MONITORING

For a distributed system as complicated as this one, system monitoring tools are a great advantage. They allow the developers to see problems at a glance and to locate and solve those problems quickly. System monitoring for the operational demonstration consists of three parts. First, there are the reliability statistics. These give an indication of the average reliability and health of the system over a span of a day. Second, there is a current system view. This gives a snapshot of the current condition of the processes in the system. Third there is hardware status monitoring, which gives warnings when the hardware is not behaving as expected.

8.1 Reliability Statistics

Every time the *auto-restarter* kills and restarts a process, that information is saved in a log file. The auto-restart reliability statistics parse those log files and create statistics over a period of a day. This data is displayed on an internal web page, which allows the developers to see a summary of which processes restarted, how many times and why the process was restarted, i.e. whether the process was hung or missing.

Statistics are also produced for file distribution. These statistics indicate how many errors occurred during data pushes. The web page again allows the user to “drill down” to see a summary of the error messages generated by *DsFileDist*.

The other part to the reliability statistics internal web page is a count of error messages in the system. Each process in the system writes error messages to a separate log file in the data tree. These log files are parsed to determine how many error messages appeared for each process. A total count appears on the web page, but again the user can view a summary which includes the name of the process producing the error messages and the number of error messages included in the count for that process.

Scripts to compute these statistics, developed previously in RAL, are run nightly as part of cron. Note that creating a web page for this requires that statistics from all four hosts reside on a single host. This means

that summary statistics must be pushed to this central host via *DsFileDist*.

8.2 System View

A current view of the system is very helpful when determining whether it is working correctly. The operational demonstration system makes use of another RAL tool called *SysView*, which is a graphical depiction of the key processes and datasets in the system.

The system developer creates a diagram of the system configuration using *SysView*. This diagram allows the developer to specify the process names and instances, specify the datasets and the amount of time allowed before each dataset is considered late and draw flow lines indicating the data flow. When *SysView* runs it communicates with the process mapper and the data mapper to determine if processes are currently running and if datasets are current. It then assigns the appropriate color to the given process or dataset to indicate its status. (See Figure 7) *SysView* also dumps images on a regular basis that can be displayed on a web page.

8.3 Hardware Status

Many operational systems make use of spong. This is a shareware system for monitoring servers for things like disk usage, CPU usage, NFS, etc. It sends email to the specified user when disk usage or the load on the machine exceeds some limit, or when NFS is down, etc. Obviously this is a significant tool for setting up and monitoring a distributed system. See Figure 8 for an example.

9. PROJECT REVISION CONTROL

One key aspect of a complicated system such as this is to ensure that the current system setup is protected and to keep a clear record of the changes that were made to the various parts of the system over time. Concurrent Versions System or CVS has been an excellent tool to this end. All source code used in this system is checked into CVS. The project setup itself is also maintained since the start/kill scripts, system level scripts, archive scripts, parameter files for each of the applications, process lists used by the *auto-restarter*, crontabs and documentation are checked into CVS.

Using this revision control system for the source code has the added benefit of making it straightforward to do a complete build of all of the software used in the system on a regular basis. Of course, prior to becoming operational, the code should be frozen and no additional builds should be done. But up until this point, the Turbulence Remote Sensing Operational Demonstration system made use of build scripts to systematically upgrade all of the software in the system.

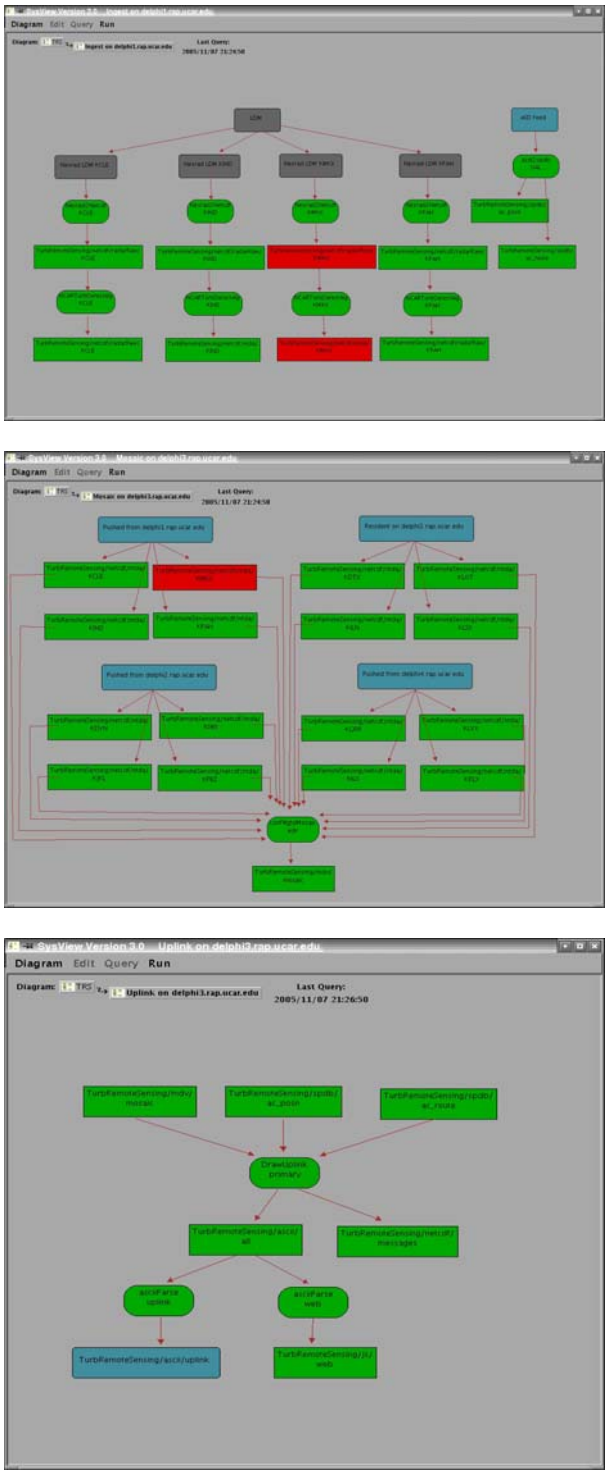

Figure 7: Sample SysView diagrams from the operational demonstration system. The top diagram depicts the processes and datasets for one of the ingest roles. The center diagram depicts processes and datasets for the EDR mosaic and the bottom shows the uplink processes and datasets. The red color indicates that the datasets are late, which, in this case is a result of problems at the given radar.

Figure 8: Example of spong display

10. CONCLUSION

In the summer of 2005, an operational demonstration of the newly-developed NCAR Turbulence Detection Algorithm (NTDA) was performed by NCAR's Research Applications Laboratory (RAL) under direction and funding from the FAA's Aviation Weather Research Program. The system designed for this operational demonstration consisted of a distributed system using four servers. Several central algorithms were developed, including an ingester (*nexrad2netcdf*), the NCAR Turbulence Detection Algorithm (NTDA), a mosaic algorithm (*confWgtdMosaic*) and hazard uplink message generation algorithms. In addition, existing code for reading and interpreting ASDI data was significantly modified to process aircraft route information. Existing tools were employed to keep this system operational throughout the summer and to ensure that the data was flowing through the system properly. Several methods of monitoring the system were used to ensure the health of the system and CVS was used to ensure this system setup was saved for current and future uses.

11. ACKNOWLEDGEMENTS

Many of the tools used in this system were developed by the hard work and experience of many software engineers in Research Applications Laboratory (RAL) at NCAR, including Michael Dixon, Deirdre Garvey, Niles Oien, Nancy Rehak, Rebecca Ruttenburg, and many others. Their work has produced very reliable systems and has made design and installation a more productive and straightforward endeavor. In addition, the Oceanic Weather project at NCAR did much of the early work in uplinking messages to ACARS printers in selected cockpits via ARINC. That work was most helpful in this project.

The JADE software used by this demonstration for the display was developed by a team of software engineers in RAL at NCAR. Their very competent work has been essential to the success of this project. In particular, Paddy McCarthy, Aaron Braeckel and Shelly Knight should be noted for their work on the display for this project.

This work is in response to requirements and funding by the Federal Aviation Administration (FAA). The views expressed are those of the authors and do not necessarily represent the official policy or position of the FAA.

12. REFERENCE

Williams, J. K., L. Cornman, D. Gilbert, S. G. Carson, and J. Yee, 2004: Improved remote detection of turbulence using ground-based Doppler radars. AMS 11th Conference on Aviation, Range, and Aerospace Meteorology, CD-ROM, 4.5.