Developing an Operational Real-Time Space Weather System: A Case Study

S. Dave Bouwer, SET
www.spacewx.com
dbouwer@spacenvironment.net
What is an “operational system”?

- Automated & real-time
- Addresses the coupling of real-time data with models
- Increasingly, largely decoupled from the User Interface
  - Web-based UI getting replaced with computer-to-computer coupling
  - Web, smart phone apps, tweets, etc. diversifying
- Encompasses or resides in the critical data path
The Evolution of Space Weather Systems

- Legacy Practice: Automated & coupled space weather data ingest, model execution, and data distribution as a *single threaded process*
- Evolving from reading web-based data to direct access from databases between servers and clients as *multiple asynchronous processes*
  - Data distribution to end-users extending to smartphone apps, social media (tweets), etc.
  - Common data formats increasingly ill-suited because data content rapidly changes, programming languages and applications evolve, and stakeholders needs frequently change.
LAPS overview

- The mission: Develop an antonymous space weather system coupling multiple real-time data streams with models to produce thermospheric densities needed for satellite orbit predictions.
- LEO Alert and Prediction System (LAPS)
- History
  - Proof of Concept phase (2008)
  - Beta ops version phase (2010)
  - LAPS (Low-earth orbit Alert and Prediction System) : Transitioning into operations for final testing (2012)
  - Operations Evaluation 2012
  - TRL (Technology Readiness Level) Operations 2013
  - Approximately one million lines of code developed over 10 years (1,000 modules)
The Challenge

• Defining and managing critical path paths
  • For LAPS, approximately 36 I/O unique data streams (8 Primary)
  • The 99% “rule of thumb”:
    • Each data source has a 1% failure rate per product delivery
      • Server/network outages, non-monotonic time steps, inaccurate data, missing fields, etc.
      • Most outages are temporary or self-correcting
  • LAPS Design Goal: Greater than 99.9% uptime
  • Time-line: approximately 4,000 hours (in one year)
Proposed design

- **System engineering priorities**
  - Accuracy, traceability
    - Performance evaluation (scientific accuracy)
    - Mean time between failures (percent uptime)
  - **Costs**
    - Risks
      - Under-estimating scope
      - Vague/changing customer V&V requirements
      - Research vs. operations discontinuities (historical vs. real-time data)
      - Excessive system maintenance (operating system, libraries, apps, etc)
  - **Redundancy**
    - One development server, one operational server, two backup servers
      - All servers running independently from each other
Proposed design (2)

• Reducing software development costs
  • Object-oriented software (e.g., C++, Java)
    • Robust – primary focus
    • Extensible – dealing with changing requirements
    • Code reusability
    • Portable
  • Encapsulating models/legacy processes
    • But separating data from models
  • One integrated application, one computer
  • Test data epochs for scientific evaluation
- 3-tier data I/O framework
- **Data Producers:**
  - Bridge to outside
  - One data stream per producer
  - Asynchronous
- **Data Broker:**
  - Monitors producer performance
  - Verifies data
  - Delivers primary/secondary data streams
- **Data Consumer**
  - Model, web site, etc.
  - May in turn be a producer
Design Pattern (2)

- The Consumer (model) drives the Producer/Broker design specifics
  - No two data set formats are the same
  - Time series need to be monotonically increasing in time
- Object-oriented software (e.g., C++, Java)
  - Robust – primary focus
  - Extensible – dealing with changing requirements
    - Code reusability
    - Encapsulation for models (supporting Fortran, C, etc.)
- One integrated application, one computer
- Iterative & incremental development process
  - For each module (sic. class): write a little, test, then repeat
Available Resources

- Three existing SET servers (providing real-time data streams)
  - Current “base” ops system running for years
  - Most components already in place, and which are consistent with current operations design pattern
  - Four Linux systems to be installed (1 development, 1 operations, 2 redundant).
  - Java SE 1.6
  - MySQL
  - IDL development and runtime licenses
  - gfortran
  - A handful of Perl, PHP, and specialized libraries
    - In the process of deprecating Perl/PHP
Distributed Networks

- An outage in one component compromises many interconnected users/subsystems
  - A “Data Daisy-chain”, but if one link fails early on, multiple systems are impacted
- With multiple systems each at 99% accuracy, anomalies are frequent
- A software architecture that is network-centric with redundant data-streams is needed
LAPS Software Architecture

- Contains multiple Producers & Consumers (Brokers not shown)
- Data are separated from models, reduced to manageable sizes
- Data Producers *pull* data from remote sources
- While all data-streams run asynchronously, a well-crafted crontab avoids latch-ups, out-of-phase latencies, and enhances traceability.
Abstract Data Objects

- Contains attributes and methods
- Abstract – applies to all data
- Is passed between classes as an element of a vector*
- Very computationally expensive
  - Usually irrelevant, but can be optimized
  - Can revert to encapsulating faster C or Fortran programs

```java
public class Generic_Data_Record {
    // Class Attributes
    String Time_Key;  // YYYYMMDDhhmm of meas.
    double JulDay;    // 2452719.541655
    float measVal;    // Single measurement
    boolean IsValid;  // true if no problems
    String inpRec;    // Data line from inp. file
    String outStr;    // Formatted line for model

    public Generic_Data_Record( String inpRec ) {
        // Methods
        run() {
            parse_inpRec( inpRec );    // sets attributes
            IsValid = validate();     // validates
            outStr = make_lineRec();   // for out file }
        // report on self (YYYYMMDDhhmm JD data isOK)
        print();
    }

    // self-test, inpLine = "2003 080 01:00 1.234"
    public static void main( inpLine ) {
        genRec = new Generic_Lya_Record( inpLine );
        genRec.run();
        if( !genRec.isValid() ) print, "error ..."
        genRec.print();
    }
}
```

* dataRec =
  (Generic_Data_Record) dataVec.elementAt( i )
float a_datum = dataRec.measVal;

Space Weather Community Operations Workshop, March 2013
Mitigating Vulnerabilities: The “Deadman” File

- When cron jobs fail, it is important to rapidly be alerted to the problem, and to diagnose the point of failure for quick resolution
  - Each data stream, each procedure or class, reports its status at key points to a Deadman file, e.g., “Input Data verified”, “Exception detected in parsing input”, etc.
  - The Deadman file uses a consistent format, so it can be read by an independent performance monitor or Data Broker to decide on what procedure or data stream to implement next
  - A Green/Yellow/Red message/alert available to operators

<table>
<thead>
<tr>
<th>System Time (UT)</th>
<th>Status Flag (0-5)</th>
<th>Color Code (Gre, Yel, Red)</th>
<th>Operator Action</th>
<th>Descriptive Text (class and method)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011/01/07 21:10:00</td>
<td>2</td>
<td>Gre</td>
<td>Waiting for some event; no action required</td>
<td>DST_QL_Kyoto_main started, run() executing get_DST_RT_dataFile()</td>
</tr>
</tbody>
</table>
Research to Ops: In Practice

- Unit Design Phase
  - Port from current operations
    - *Hidden dependencies exposed*

- Integration, Beta ops version
  - Test/debug cycle
  - Simulation testing
    - *Ops to research*
    - *Ops framework ill-suited for historical simulation*

- Final ops evaluation (scientific and system)
  - LAPS: One end-product “datagram” file fed to satellite drag models
  - Data Matrix
System Implementation

- Code refactoring: Port of SET ops to LAPS
  - Legacy data file I/O problematic
  - Hidden dependencies
  - Additional components needed

- Real-time system integration
  - Code, unit tests, integrated test, repeat
    - All components (classes) containing a self-test method highly advantageous

- Integrated real-time test: Performance
  - Despite three satellite failures, a major network outage, and a power supply failure, LAPS maintained operations 99% of the time over two months
    - Required writing new code: fast

- In the past two months, no catastrophic failures, 1% run-time exceptions (self-correcting); greater than 99.9% ops uptime
Postmortem: What did not work (well)

- Under-estimated scope (a lot)
- Changing performance criteria
- Different flavors of Unix (esp. IDL/Fedora)
- Lack of a source control system
  - Accustomed to 2-3 people collaborating in software development
    - Steep learning curve to implement
  - Complex source control/versioning applications are available, but too elaborate for LAPS needs
  - Nevertheless, evolving software on multiple systems requires a source versioning solution
- Reproducing historical conditions (ops-to-research)
- Legacy/Procedural programs prone to single-point-of-failure exceptions
- Models tend to be developer-specific
  - Difficult to debug/maintain
Postmortem (2): What did work

- Asynchronous/Redundant data streams paradigm validated:
  - Mitigated critical-path failures
  - Object-oriented design made development (debugging) easier
  - Model encapsulation was effective
  - Data Producers/Brokers/Consumers Design Pattern well-received
  - Separating data from models design made development (debugging) easier
- Exception Handlers extremely important in achieving resiliency
  - Deadman File, run-time logs
    - `grep -i error *.log ... the most useful ops unix command!
- MySQL database
  - Highly efficient
  - Prevents data I/O collisions
- Limiting reliance on 3rd-party applications and open-source libraries
- System administration support
Conclusions

• Transitioning models into operations if typically far more difficult than expectations
  • Experience indicates for a single model it can take 6-18 months

• Time for development, then maintenance, are the greatest long-term costs
  • “Organizational memory” (or lack thereof) is important
  • The more modular and tested a system is, the lower the maintenance costs
  • Documentation is often ignored, but meaningful in-line comments in code and a final users guide (for both system admin. and end-user) are needed

• There is still an un-addressed risk is in the distributed network “daisy-chain”: If the first link fails, who are you going to call?
  • Develop a “POC” call list?
Conclusions

- *All processes will fail — get used to it.*