Project Golden Gate
Addressing the Challenges of Space Mission Software

Daniel Dvorak, Ph.D.
Information Technologies and Software Systems Division
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California
Daniel.Dvorak@jpl.nasa.gov

Outline

• Introduction to Jet Propulsion Laboratory
• Challenges of space mission flight software
  • Timekeeping, timing, coupling, relativity, ...
• Mission Data System
• Project Golden Gate
  • Rocky 7 rover testbed
  • Programming model
  • Performance benchmark suite
• Shaping the Future
  • Better tools
Introduction to JPL

JPL is one of NASA's project implementation

[Map showing locations of NASA centers]
JPL business summary

- $1.4 billion business base
- 5400 employees and contractors
- 177 acres
- 134 buildings and 57 trailers
- 670,000 net square feet of office space
- 860,000 net square feet of non-office space (e.g., labs)

Sixteen JPL spacecraft, and four major instruments, now operating across the solar system

- Two Voyagers on an interstellar mission
- Ulysses, Genesis, and ACRIMSAT studying the sun
- Galileo and Cassini studying Jupiter and Saturn
- GALEX studying UV universe
- Stardust returning comet dust
- Tgope/Fuselion, Quick-Scat, Jason 1, and GRACE (plus SeaWinds, ASTER, MISR, and AHS instruments) monitoring Earth
Mission Data System

Studying neighboring solar systems

- Space Infrared Telescope Facility (SIRTF), to launch in August 2004, seeks stellar planetary disks.

- Kepler, to launch in 2007, observes transits of planets across stars.

- Space Interferometry Mission (SIM), to launch in 2009, performs astrometry of extra-solar planets.

- Terrestrial Planet Finder (TPF), to launch in 2015, performs spectroscopy of extra-solar planets.
Significant recent and future events

2001 Mars Odyssey began mapping February 2002

GRACE Earth gravity measuring mission launched March 17, 2002

GALEX ultraviolet observatory launched April 28, 2003

NASA infrared great observatory SIRTF launch in August 2003

Mars Exploration Rovers launch summer 2003, arrive January 2004

Stardust captures material from Comet Wild 2 in January 2004

Cassini/Huygens arrives at Saturn July 2004

Genesis solar wind sample return September 2004

Cloudsat launch November 12, 2004

Technology and Engineering

JPL is a world leader in key areas critical to deep space exploration

- End-to-end system engineering and project management
- Autonomous mobility
- Deep space communications
- Deep space navigation and highly stable clocks
- Extreme precision formation flying for science and rendezvous
- High precision spaceborne systems in optical to sub-millimeter, including interferometry
- Active sensors for mapping and positioning (SAR, altimeters, GPS)
A renaissance in space exploration in the 21st Century; Analogous to Quattrocento and Cinquecento Florence

- Searching for water and former or extant life:
  - Mars
  - Europa
  - Small bodies (comets, asteroids)

- Characterizing extra-solar planets

- Understanding stellar and galactic evolution

- Understanding dark matter and "dark energy"

- Searching for gravity waves

Some Challenges of Space Mission Flight Software
Hydrobot in Europa Ocean

Mars Exploration Rover
## Characteristics of Deep Space Missions

- Explore in uncertain environments
- Conduct *in situ* science investigations
- Operate far from Earth
- Survive for decades (in some cases)
- Operate semi-autonomously

## Problem Domain

### Common Characteristics (1)

- **Infrequent, scheduled communication**
  - The ‘network’ is not continuously available due to DSN constraints and spacecraft activities that compete for power and pointing.

- **Distance and Time Delay**
  - With a ~10 hour round-trip light-time delay to Pluto, it's impossible for operators on Earth to react to events in a timely manner.

- **Distance and Communication Rate**
  - With a data rate of ~300 bits/sec from Pluto, it isn't feasible to send all of the raw science data; prioritization/summarization is needed.

- **Distance and Pointing**
  - When transmitting, need to point antenna at where Earth will be *when the signal arrives*, not at where it is now. Vice versa for receiving.
Distance, Data Rate, Time Delay

Effect of distance on data rate for X-band RF communication with 5 watts transmitted power from a 2-meter spacecraft antenna into a 70-meter ground antenna

Data Rate

Mars 1.5 AU 16.7 kbps
Jupiter 5.2 AU 10.0 kbps
Saturn 9.5 AU 16.7 kbps
Uranus 19.2 AU 7.5 kbps
Neptune 30.0 AU 30.0 kbps
Pluto 39.5 AU

Round Trip Light-Time Delay

12 hrs 10 hrs 8 hrs 6 hrs 4 hrs 2 hr

AU = Astronomical Unit = mean Earth-Sun distance

Computing: Flight vs. Ground

Gulf between flight processors and ground processors due to radiation-hardening, long cruise times, and Moore's Law

Mips

10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
100,000


When spacecraft reaches Pluto in 2012, running a 242 Mip processor, desktop computers will be running at 100,000 Mips!
• Special Relativity and Time Dilation
  • Though spacecraft velocity is a tiny fraction of lightspeed, navigation must take relativistic effects into account.

• Limited Flight Processor and Memory
  • Radiation-hardened flight processors are years behind mainstream commercial processors. Flight software must be frugal with CPU cycles and memory.

• Cruise Time and Moore’s Law
  • The disparity between flight and ground processing abilities grows with every year of cruise time.

• Limited Resources and Tight Coupling
  • In a resource-limited system, 'everything affects everything'.

---

The World of Side Effects

*Turning on a disk drive has the following side effects:*

• It reduces available power
• It causes heating
• It causes vibration
• It causes electromagnetic radiation
• It imparts rotational torque
• It stabilizes orientation around axis of rotation

In a server room on Earth, these side effects are negligible.

In a spacecraft, every one of these side effects is significant and must be managed!
The Problem:
- Physics has no respect for our mental simplifications
- "Side effects" (couplings) are everywhere
- And we can't ignore them in some control systems

Mission Data System
**Mission Data System**

**High-risk systems**

<table>
<thead>
<tr>
<th>Linear</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Nuclear plant</td>
</tr>
<tr>
<td></td>
<td>Aircraft</td>
</tr>
<tr>
<td></td>
<td>Marine transport</td>
</tr>
<tr>
<td></td>
<td>Dams</td>
</tr>
<tr>
<td></td>
<td>Power grids</td>
</tr>
<tr>
<td>Low</td>
<td>Chemical plants</td>
</tr>
<tr>
<td></td>
<td>Space missions</td>
</tr>
<tr>
<td></td>
<td>Military early-warning</td>
</tr>
<tr>
<td></td>
<td>Junior college</td>
</tr>
<tr>
<td></td>
<td>Trade schools</td>
</tr>
<tr>
<td></td>
<td>Most manufacturing</td>
</tr>
<tr>
<td></td>
<td>Rail transport</td>
</tr>
<tr>
<td></td>
<td>Airways</td>
</tr>
<tr>
<td></td>
<td>Military actions</td>
</tr>
<tr>
<td></td>
<td>Mining</td>
</tr>
<tr>
<td></td>
<td>R&amp;D firms</td>
</tr>
<tr>
<td></td>
<td>Universities</td>
</tr>
<tr>
<td></td>
<td>Post Office</td>
</tr>
</tbody>
</table>

*Adapted from Charles Perrow, "Normal Accidents: Living with High-Risk Technologies", 1984.*

**Coupling in Space Systems**

*Complex couplings arise from physics and design*

- Amount of mass launched determines a **big** mission cost
- Therefore, minimize size of batteries, size of solar panels, amount of memory, articulation mechanisms, shielding, smaller antenna, low-power transmitter, etc
- That means:
  - Slower CPU and busses and less memory
  - Can't drive and transmit concurrently
  - Can't run heaters while firing thrusters
  - Can't independently point camera and antenna
  - Lower signal-to-noise ratio, so lower data communication rates, so science downlink is limited
  - Must hold reserve power for surviving the night
Coupling in Space Systems

* Some domains of concurrent design in JPL's Project Design Center

Control System Domain

- **Characteristics:**
  - Interacts with world via imperfect sensors & actuators
  - Designed for continuous operation
  - Real-time closed-loop control
  - Embedded systems, often

- **Examples:**
  - Petroleum refining
  - Pharmaceutical manufacturing
  - Nuclear power plant
  - Spacecraft control
addressing the couplings

A State/Model Architecture
Managing Interactions

- Interactions often cross subsystem boundaries
- Managing interactions is key to good design
- Need to elevate interactions to architectural level
- Need to describe how one thing affects another
  - The variables are *states*
  - The equations are *models*

Example Spacecraft States

- Dynamics
  - Vehicle position & attitude, gimbal angles, wheel rotation, ...
- Environment
  - Ephemeris, light level, atmospheric profiles, terrain, ...
- Device status
  - Configuration, temperature, operating modes, failure modes, ...
- Parameters
  - Mass properties, scale factors, biases, alignments, noise levels, ...
- Resources
  - Power & energy, propellant, data storage, bandwidth, ...
- Data product collections
  - Science data, measurement sets, ...
- Data management policies
  - Compression/deletion, transport priority, ...
- Externally controlled factors
  - Space link schedule & configuration, ...
**Example Spacecraft Models**

- **Relationships among states**
  - Power varies with solar incidence angle, temperature, & occultation

- **Relationships between measurement values and states**
  - Temperature data depends on temperature, but also on calibration parameters and transducer health

- **Relationships between command values and states**
  - It can take up to half a second from commanding a switch to full on

- **Sequential state machines**
  - Some sequences of valve operations are okay; others are not

- **Dynamical state models**
  - Accelerating to a turn rate takes time

- **Inference rules**
  - If there has been no communication from the ground in a week, assume something in the uplink has failed

- **Conditional behaviors**
  - Pointing performance can’t be maintained until rates are low

- **Compatibility rules**
  - Reaction wheel momentum cannot be dumped while being used for control

---

**State/Model Architecture**

- Estimators interpret measurement and command evidence to estimate state
- State variables hold state values, including degree of uncertainty
- Model: expresses mission-specific relations among states, commands, and measurements
- A goal is a constraint on the value of a state variable over a time interval
- Controllers issue commands, striving to achieve goals
- Hardware provides access to hardware buses, devices, instruments
- Hardware proxies provide access to hardware buses, devices, instruments
Architectural Relationships

The color coding conveys similarities, e.g., estimators and controllers are goal achievers, sensors and actuators are devices, measurements and commands are time-tagged items.

Project Golden Gate

Moving toward real-time Java in flight software
**Project Golden Gate**

- **What:**
  - A collaboration among NASA (JPL), academia (CMU), and industry (Sun), with HDCP participation
  - An evaluation of real-time Java (RTSJ) and real-time Linux for flight systems

- **Why:**
  - RTSJ is a significant step for real-time community
  - RTSJ is relevant to NASA's spacecraft, rovers, etc
  - RTSJ/Linux may help improve dependability over C++/VxWorks

- **How:**
  - Do thorough technical assessment w.r.t. flight mission criteria
  - Actively work to retire the risks
    - Influence evolution of the specification
    - Feedback to vendors
    - Adjustments to programming model
JPL Overview

Approach
- Evaluate real-time Java against FSW demands
  - Target challenges cited in Nov 1999 JPL report "Using Java for Flight Implementation"
- Use MDS framework/code as a testbed
  - Same hardware functionality as MDS/C++VxWorks
  - Side-by-side performance comparison
- Leverage Distinguished Visiting Scientists
  - Dr. James Gosling, creator of Java, Sun Microsystems
  - Dr. Greg Bollea, lead of RTSJ, Sun Microsystems

A Modern Software Platform for Real-Time Embedded Systems

Work Plan
FY 2003:
- Install Linux/RT and RTSJ JVM on Rocky7
- Prototype, run, and measure major MDS components
- Run AFRL/Boeing test suite for RTSJ
FY 2004:
- Extend to run full MSL test scenario on Rocky7
- Compare performance to C++VxWorks version
- Complete evaluations on latest product release

Evaluation Criteria
- Performance measurements
  - CPU usage
  - Throughput
  - Real-time response & timing jitter
  - Cache hit ratio
- Maturity of RTSJ and RT Linux technology
- Multi-language development
- Application development effort

A Modern Software Platform for Real-Time Embedded Systems

Overview

Rocky 7 Hardware

- 6 driving motors
- 2 steering motors
- 3 stereo camera pairs
- 3-axis accelerometer
- 1-axis gyroscope
- 2 DOF arm w/ 2 DOF scoops
- 3 DOF mast
- PPC 750, 256 MB RAM
- Camera frame-grabber
- Digital I/O
- Compact PCI backplane
Software Overview

Flight Software

Rocky7 Control
Mission Data System

What we have running:
- Driving & steering
- Picture taking
- Telemetry downlink
- Telemetry display

Ground Software

Ground Operations
Mission Data System

Application Software

Driving & Steering
- 9 periodic control loops in NoHeapRealtimeThread
- 6 driving wheels
- 2 steering angles
- rover position & heading

Cameras
- Periodic snapshots in RealtimeThread

Data Transport
- Queuer & Sender as periodic RealtimeThread
The Software Inside

- We have converted a substantial body of MDS/MSL software from C++ to RTSJ and run it on Rocky7
  - and learned a lot in the process

Amount of software in repository:
- 31 packages
- 343 classes
- 2003 functions
- 13,000 non-comment source statements

Rocky7 adaptation packages:
- LM 629 device
- PCI device
- S72Q device
- Motor
- Motor simulator
- Position & heading controller
- Angular position

MDS Framework packages:
- Components & connectors
- Data catalog
- Value history
- Data transport
- Math
- Physics
- Resources
- State knowledge
- Goal achievers
- Goal network
- Component scheduler
- Hardware adapter
- State types
- Utilities

Performance Benchmark Suite

**Performance metrics**

- Support comparisons among 3 platforms:

<table>
<thead>
<tr>
<th>RTSJ</th>
<th>C++</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>Linux</td>
<td>Linux</td>
</tr>
</tbody>
</table>

- **Focused measures**:
  - interrupt response latency, jitter, scheduling and dispatching overhead, garbage collection execution, and processor throughput, including numerically-intensive computation

- **System-level measures**:
  - CPU utilization, cache hit ratio, memory footprint, and language-induced overheads, such as the necessity of moving data among RTSJ's different memory areas.

[^1]: RTSJ = Real-Time Specification for Java
Conclusions

- JPL is interested in architectures, languages, and tools that help us build complex and reliable space mission software.
- Golden Gate is paving the way for a new flight software platform for future missions.
- The Suramadu performance benchmark is offered to the real-time community as a way to help all of us to compare and influence commercial tools to meet our needs.

Questions?

*Artist’s conception: A Mars sample-return mission blasting off from Mars*
End

The remaining slides provide background on RTSJ

RTSJ and its Effects on Program Design

(RTSJ = Real Time Specification for Java)

Dan Dvorak
May 6, 2003
Daniel.Dvorak@jpl.nasa.gov

(Some slides from Greg Bollella, of Sun Microsystems Laboratory)
Outline

- RTSJ Overview
- Main features of RTSJ
- Effects on Program Design
- Other approaches to real-time Java

Java for Real-Time Systems ???

Java Technology was not intended for real-time systems because:

- Requires run-time garbage collection
- Supports threads, but no scheduling control
- Synchronization delays unpredictable
- Very coarse timer support
- No event processing
- No safe asynchronous transfer of control
### RTSJ Guiding Principles (partial)

- **Predictable Execution**
  - "hold predictable execution as first priority in all tradeoffs"

- **Backward Compatibility**
  - Existing Java programs run on RTSJ implementations

- **No Syntactic Extensions**
  - No new keywords or other extensions to Java language

- **Support Leading-Edge Scheduling**
  - Priorities considered harmful

- **Appropriate for any Java platform**
  - J2ME, J2SE, J2EE, ...

### Observations about RTSJ

- **Virgil Champlin, CMU West**
  - Application programming model: RTSJ vs. Java

- **Brian Giovannoni, CMU West**
  - Core libraries, JERI, scoped memory, ...
RTSJ Preview

- RTSJ = Real-Time Specification for Java
  - Nov. 2001: JCP approved RTSJ 1.0
  - Jan. 2002: Reference Implementation

- RTSJ allows real-time applications to be coded in Java
  - without interference from garbage collector

- RTSJ allows developers to reason about time
  - Periods, costs, deadlines, overrun handler, ...

RTSJ Features

- Full Java capability
- Threads (RealtimeThread, NoHeapRealtimeThread)
- Asynchronous Event handling
- High resolution time
- Precise Timer support
- Asynchronous Transfer of Control
- Flexible memory management
  - Makes using Java heap optional
  - Avoids or controls garbage collection under application control
- Raw memory access (e.g., memory-mapped I/O)
RTSJ: Why should you care?

- It's interesting technology for real-time systems
- RTSJ may become the default language for embedded systems at JPL
- Some of your code may have to run in soft real time or even hard real time
- "I expect The Real-Time Specification for Java to become the first real-time programming language to be both commercially and technologically successful."
  - E. Douglas Jensen, in Foreword of RTSJ book

Definitions

- **Real-time** (system or code): A system (or code) which requires that computation have temporal correctness criteria in addition to functional correctness criteria
- **Hard real-time**: A real-time system which requires that the temporal correctness criteria are always met (often, incorrectly IMHO, defined as 'less than n time-units latency')
- **Soft real-time**: A real-time system which requires that the temporal correctness criteria are almost always met (often, incorrectly IMHO, defined as 'more than n time-units latency')
Concrete Examples

- Temporal correctness criterion: A deadline
- **Hard real-time:** All computation must be complete at or before its deadline (or the system goes into an abnormal state)
- **Soft real-time:** Computation is allowed to complete after its deadline (the really interesting questions are: by how much and how often)

**Memory Areas**

- **Java**
  - Normal Java heap
  - Accessible by all threads
  - Subject to garbage collecting
  - Prone to execution latencies due to unpredictable garbage collection

- **RTSJ**
  - Limited size memory area that lives until end of application
  - Accessible by all threads
  - Never subject to garbage collecting
  - Unlike heap objects, these exist even when there are no references
  - Limited size, limited lifetime memory areas (think "scratchpad")
  - Scope is emptied when no threads have reference to this area
  - Requires application to consider memory management

- **Physical**
  - Provides access to a range of physical memory addresses
  - Enables memory-mapped I/O
  - Provides accessors to get/set byte, short, int, long
  - Can map a physical address range into virtual memory
Threads

Java
- Normal Java thread
- No scheduling control
- Cannot be asynchronously interrupted

RealtimeThread
- Temporal demands specified (deadline, period)
- Processor demands specified (execution & memory costs)
- Can specify deadline miss handler and cost overrun handler
- Can run as a periodic, aperiodic, or sporadic
- Can allow for asynchronous interrupts

NoHeapRealtimeThread
- Cannot allocate or reference objects in heap memory
- Can always execute in preference to garbage collector

Release Parameters (for scheduling)

ReleaseParameters
- Cost (execution time)
- Deadline (duration following eligible-to-run)
- Cost overrun handler
- Missed deadline handler

Periodic
- Start time
- Period (interval between successive unblocks)

Aperiodic
Can become active at any time

Sporadic
- Minimum interarrival time
Periodic Task in RTSJ vs. RTOS

```java
while(true) {
    work();
    periodNumber++;
    startOfNextPeriod=(periodNumber*period)+start;
    t = getTime();
    if ( t > startOfNextPeriod ) {
        actualPeriodNumber = ceiling((t - start)/p);
        numberOfMisses = actualPeriodNumber - periodNumber;
        takeCareOfMiss(numberOfMisses);
    } else {
        //Non-atomicity of these stmts is really bad
        //on an OS (not so bad on an RTOS)
        t = getTime();
        sleep(startOfNextPeriod - t);
    }
}
```

```
pp = new ReleaseParameters();
pp.setPeriod(new RelativeTime(10,0));
mh = new missHandler();
periodicThread = New PT(null,pp,mh);
periodicThread.start();

Class PT extends RealtimeThread {
    void run() {
        while(true) {
            WORK();
            waitForNextPeriod();
        }
    }
}

Class missHandler extends AsyncEventHandler{
    handleAsyncEvent() {
        takeCareOfMiss();
    }
}
```
Asynchronous Event Handler

- `AsyncEvent`
  - `addHandler`
  - `removeHandler`
  - `fire`

- `AsyncEventHandler`
  - `addIfFeasible`
  - `handleAsyncEvent`

Worst Case Execution Time and Scheduling Utilization

- How to calculate the cost of a task?
  - Even ignoring hardware issues control flow causes wide variability

- For hard real-time, set cost \( \geq \) WCET but WCET is typically much, much greater than average execution time.
  - RTSJ structures help equalize control flow
normalCaseLittleWork = new AsyncEvent();
normalCaseLittleWork.addHandler(new normalH);
errorCaseLotsOfWork = new AsyncEvent();
errorCaseLotsOfWork.addHandler(new errorH);

B = readDevice();
if (notError(B)){
    normalCaseLittleWork();
} else {
    errorCaseLotsOfWork();
}

• What is the cost?
  • WCET includes error case

Class normalH extends AsyncEventHandler {
    normalCaseLittleWork();
}

Class errorH extends AsyncEventHandler {
    errorCaseLotsOfWork();
}

• Normal and error cases separately schedulable (different priorities, deadline, etc.)
Most applications have a small real-time side and a larger non-real-time side.

Data needs to move between these sides.

RTSJ's memory areas complicate this.

**Non-real-time side**
- Heap memory
- Java threads
- Realtime threads
- Maximized throughput

**Real-time side**
- Scoped & immortal memory
- NoHeapRealtime threads
- Very low jitter

Data transfer queues
Memory Area Assignment Rules

**Primordial Scope**

Legend: \[ \rightarrow \] means that \( X \) can be assigned a reference to an object in \( Y \)

Legend: \[ \rightarrow \rightarrow \] means that \( X \) can **NOT** be assigned a reference to an object in \( Y \) because \( Y \) can disappear before \( X \)

Illegal Memory Area Assignments
Development Model Comparison

"Priorities considered harmful"

- Priorities alone
  - Developer assigns
  - Decrease utilization when failures occur
  - Test as much as schedule will allow
  - Processor access relative and global

- Scheduling and Temporal Programming
  - Algorithm assigns
  - Known utilization bounds
  - Temporal requirements in application terms
  - Test for overrun cases
  - Processor access locally determined

Summary

- RTSJ allows real-time applications to be coded in the Java language
- RTSJ allows developers to reason about time
- RTSJ enables better processor utilization
- Memory areas complicate the programming model
- www.rtl.org
**Heading into the Future**

- Embedded
- Desktops
- Mini-computers

... C  C++  Java  RTSJ  ...

**Time and Timing**

- Special relativity
- 1 sec error amounts to x kilometers error
- Interferometers and their control loops
- Different time frames: UCT, TAI, ...
- A comm pass at Pluto: Earth starts transmitting ~4.5 hours before spacecraft starts listening, and vice versa
- Ranging
- LISA mission: formation flying
State-Based Goal-Driven Architecture

Planning & Execution

goals &
goal status

State Variables

estimates

Estimators

estimates, x-goals

Models

measures & commands

Controllers

commands

Hardware

Hardware

Adapters

You are here