

The RTSJ and Project Mackinac

Greg Bollella
Distinguished Engineer



**2004
Sun Labs
Open House**

Outline

- The Real-time Specification for Java
- Project Mackinac
- The Future of the RTSJ

Why Java as a Real-Time Platform

- More software and more complex apps
 - ➔ low-level programming no longer provides the level of abstraction required by complex apps
 - ➔ need for an easy-to-use real-time programming environment
- Support
 - ➔ vendor-neutral platform and programming language to ensure support over the long haul
 - ➔ community-based specification model
- Real-Time Specification for Java
 - ➔ all advantages of Java + meet needs of RT systems

RTSJ History

- Idea in 1998
- JSR-01 accepted by the JCP in 1999
- 40 companies involved initially
- 0.9 release June 2000
- 1.0 release Nov. 2001
- Reference implementation Jan. 2002
- First product March 2003
- 1.0.1 released this year
 - maintenance release, now available for review at <http://jcp.org>

RTSJ Guiding Principles

- Predictable execution
- Support current real-time software practice
- Backward compatibility
 - no syntactic extension
- Any Java edition
- Allow for implementation trade-offs
- WOCRAC
 - write-once carefully, execute anywhere conditionally

RTSJ Goals

- Create an API set and semantic enhancements which allow Java developers to correctly reason about and control the temporal behavior of applications
 - A high-level, abstract programming model for building real-time systems
- Conformant implementations pass JSR-01 TCK **and** a Java TCK (J2EE, J2SE or J2ME)

RTSJ System Engineering Advantages

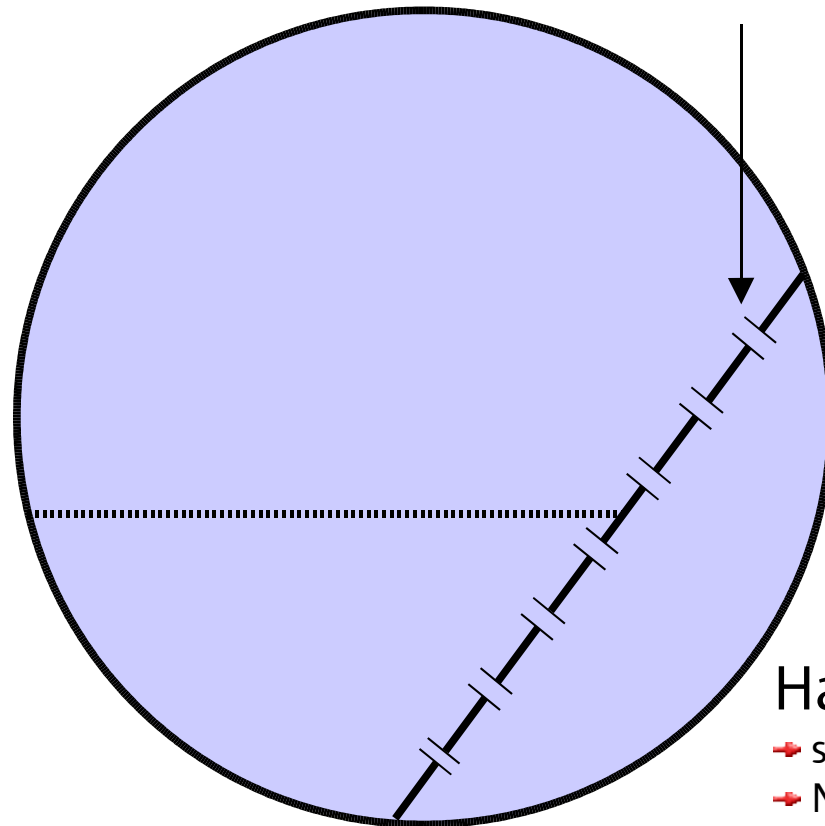
- Encourages use of real-time scheduling
- Encourages temporal decomposition
 - ➔ reduce logic induced cost variation
- Exposes memory management issues explicitly
 - ➔ In real-time systems all resource usage, including memory consumption, must be directly managed by the application or design
- Promotes an all-Java design
- Encourages encapsulation of real-world mode changes via thread sets and ATC
- Encourages application decomposition into 3 categories of temporal correctness
 - ➔ non real-time, soft real-time, and hard real-time

RTSJ System Model

Non real-time

- ➔ heap memory
- ➔ Java threads
- ➔ maximized throughput

Data transfer queues



Soft real-time

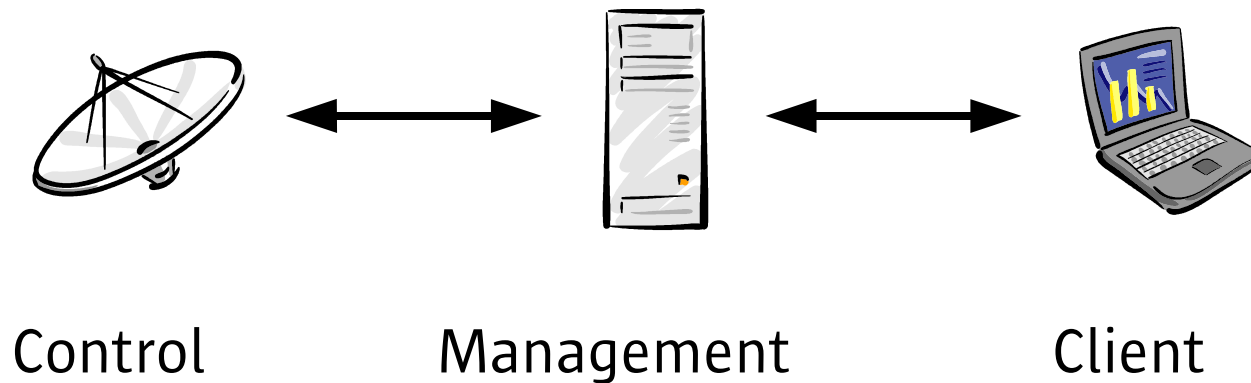
- ➔ scoped memory
- ➔ heap memory
- ➔ Realtime threads
- ➔ jitter subject to GC

Hard real-time

- ➔ scoped memory
- ➔ NoHeapRealtime threads
- ➔ bounded jitter

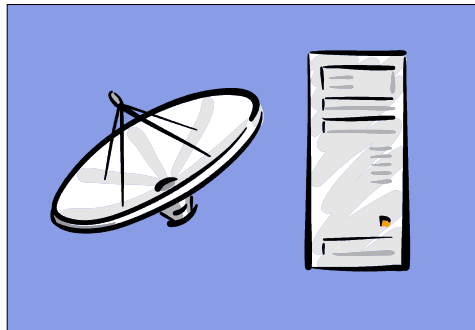
RTSJ System Architecture

- Typical embedded/real-time system architecture now:



RTSJ System Architecture

- Architecture encouraged by the RTSJ:



Control & Management



Client

RTSJ Enhanced Areas

- Scheduling
- Memory management
 - Including framework for hard and soft real-time garbage collection algorithms
- Synchronization
- Asynchronous events
 - First class support for asynchronous activities
 - Scheduled handlers
- Asynchronous transfer of control
- Physical memory access

Scheduling

- Generalize the scheduling/dispatching definition and mechanism
- Base scheduler, fixed-priority, preemptive
 - at least 28 unique priority levels
- Native support for periodic, aperiodic, and sporadic threads
- Temporal reqs expressed in application terms
 - deadlines, periods, interarrival times, costs
- Rich framework for additional schedulers

Feasibility

- As systems grow more complex the need for assuring temporal correctness increases
 - ➔ 'guess, test, increase processor capacity' will no longer suffice as a design methodology
- The RTSJ provides a rich framework that standardizes use of theoretical temporal correctness algorithms
 - ➔ Implementations of the RTSJ may provide additional scheduler and dispatch algorithms

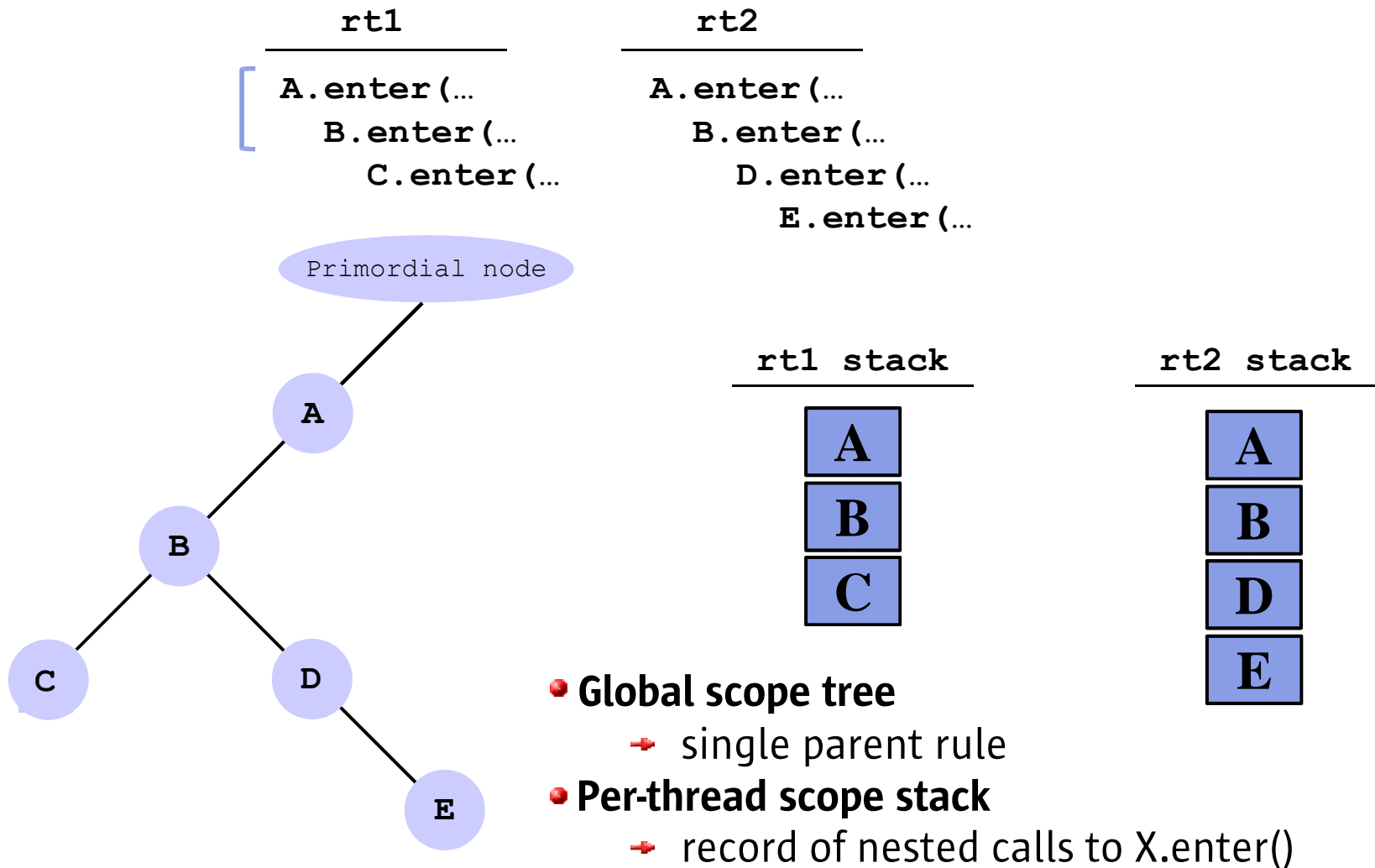
Memory Management

- RTSJ changes the notion of object lifetime
 - i.e., when an object is a candidate for collection
- Manual: lifetime controlled by program logic
 - malloc()/free()
- Automatic: lifetime controlled by visibility
 - garbage collection
- Scoped memory: lifetime controlled by syntactic scope
 - all objects live until control flows out of scope

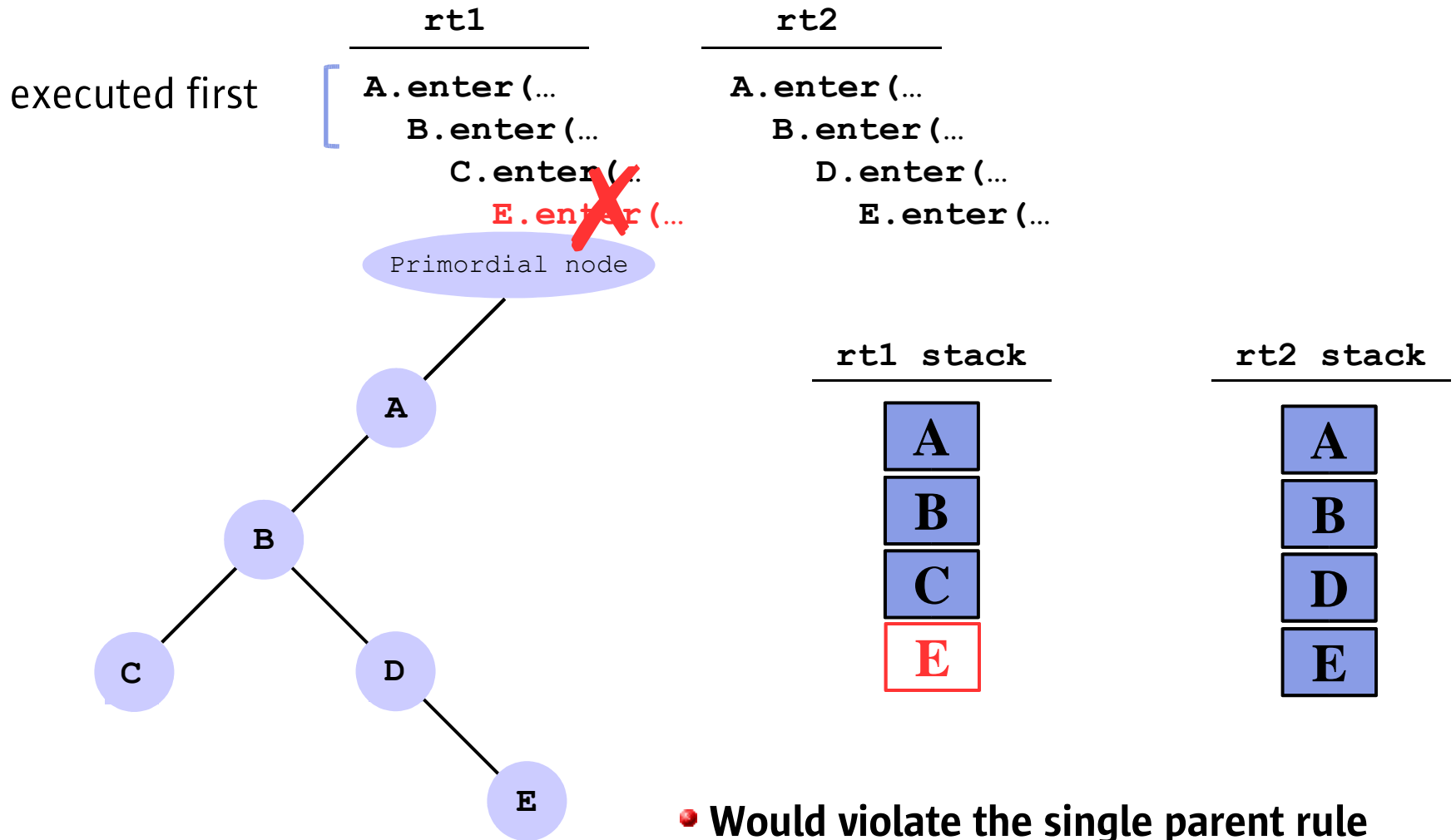
Scoped Memory

- Application defined and managed heaps without automatic memory reclamation
 - eliminate all latency of garbage collector
 - temporary objects
 - NoHeapRealtimeThreads can preempt GC indefinitely
 - 'immortal memory' for permanent objects

Scope tree and stack



Scope tree and stack (cont.)



Scoped Memory

- Why controversial
 - requires the application to consider memory management
 - requires access checks
 - aren't real-time GC good enough?

ScopedMemory and, (not vs), RT-GC

- The original intent of ScopedMemory and the NoHeapRealtimeThread class was for only those pieces of logic for which absolute minimum jitter, overhead, and latency would suffice
- The RTSJ expects that real-time GC algorithms will be implemented in various offerings

Project Mackinac



Project Overview

- Initially a 'micro' business unit funded by directly by Sun's CTO and based in Sun Labs
- Currently transitioning to a product group
 - announcement made at JavaOne 2004
- Implement the RTSJ to product-level quality
 - first commercial implem. of the RTSJ by Sun
- Work closely with industrial partners via an early access program

Mackinac Goal

- Demonstrate Sun and Java can meet the challenge of industrial computing
 - ➔ “Industrial Strength Java”

Application Domains

- Industrial control systems
- Military command and control systems
- Telecommunication switches
- Commercial shipboard control systems
- Power grid control systems
- Spacecraft control systems
- Large industrial-vehicle control systems

- “This is not a small scale. This is real-time in the large sense of the word.” *J. Gosling*

Why Mackinac?

- A new market for Sun
- Where all the excitement will next be
- Computer generations
 - Mainframes, Mini, Workstation, PC, ?
- Necessary but not sufficient condition
- High margins
- Currently a fractured market
- Long term engagements (20+ years)

Mackinac stack

- Sun's J2SE HotSpot JVM 1.4
 - ➔ modified to conform to the RTSJ, while retaining HotSpot's native performances
 - ➔ latency target... 10 microseconds
 - ➔ jitter target... 2 microseconds
- Solaris/SPARC platform
 - ➔ Solaris 9+ operating environment
 - ➔ enhanced real-time capabilities via extra kernel modules
 - ➔ support for high-availability

Mackinac Early Release Evaluations

- Power generation plant control system
 - implement turbine control system
 - 16 ms period control loop
 - 3000+ methods call each period
 - additional non real-time activity
 - 80 ms fault detection and failover to hot standby
- U.S. Navy next generation destroyer program
 - all shipboard command and control loops
- Digital acquisition card vendor
 - MATLAB/Simulink to RTSJ backend

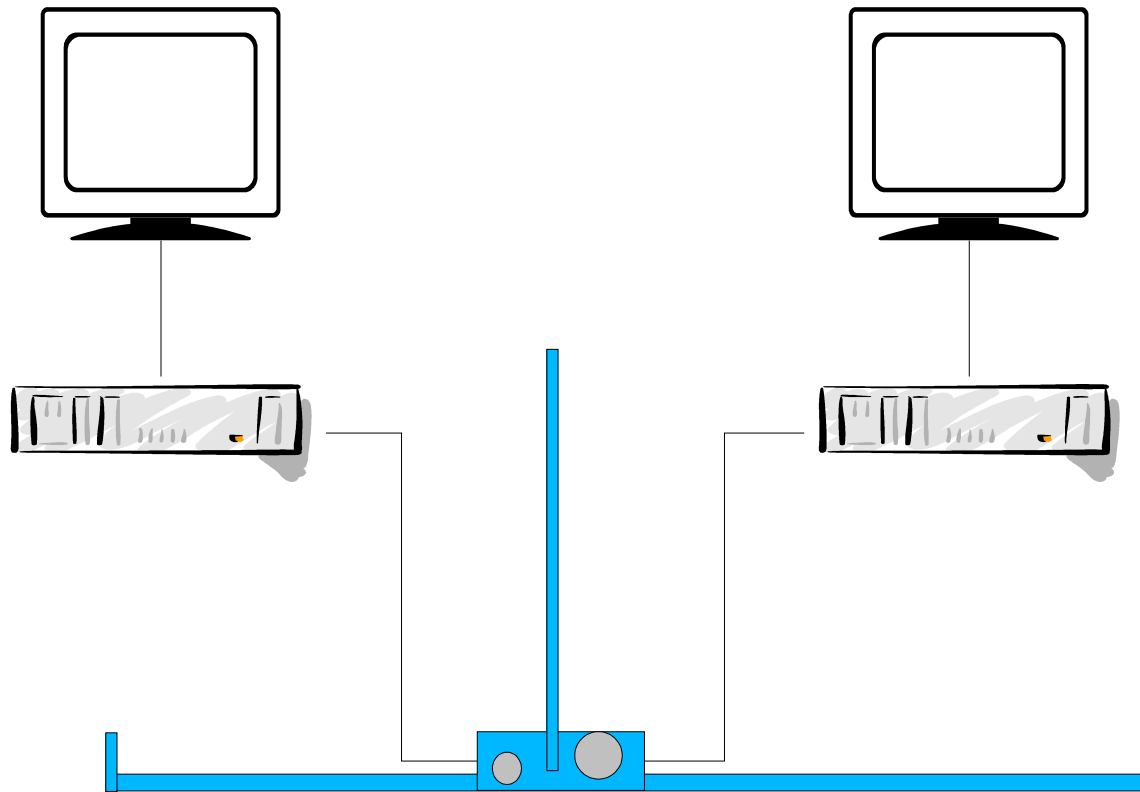
Mackinac Schedule

- Currently accepting more early release evaluations
 - ➔ Contact Gail Yamanaka gail.yamanaka@sun.com
- Code complete (Alpha) Q3 2004
- Mostly done (Beta) Q4 2004
- Tested to product-level quality Q1 2005

Demo

- Inverted pendulum
 - control law implemented fully in Java
 - 5 ms period control loop
 - 1 ms fault detection and failover to hot standby
 - no missed deadlines
 - 2-3 microseconds jitter
 - significant non real-time load in JVM
- Public performance
 - Sun Network Shanghai 2004 and JavaOne 2004

Demo System



RTSJ/Mackinac Futures?

- Distributed
- RT-GC
- Product Plans
- Safety-Critical
- Mission-Critical

Questions?

greg.bollella@sun.com



**2004
Sun Labs
Open House**