Programming Models for Concurrency and Real-time

PLACES'08, Oslo, Norway

Jan Vitek

based on joint work with

David Bacon and Josh Auerbach (IBM Research)

Jesper Spring and Rachid Guerraoui (EPFL)

COLE POLYTECHNIQUE ÉDÉRALE DE LAUSANNE

PURDUE UNIVERSITY

IBM

Why care about Real-time?

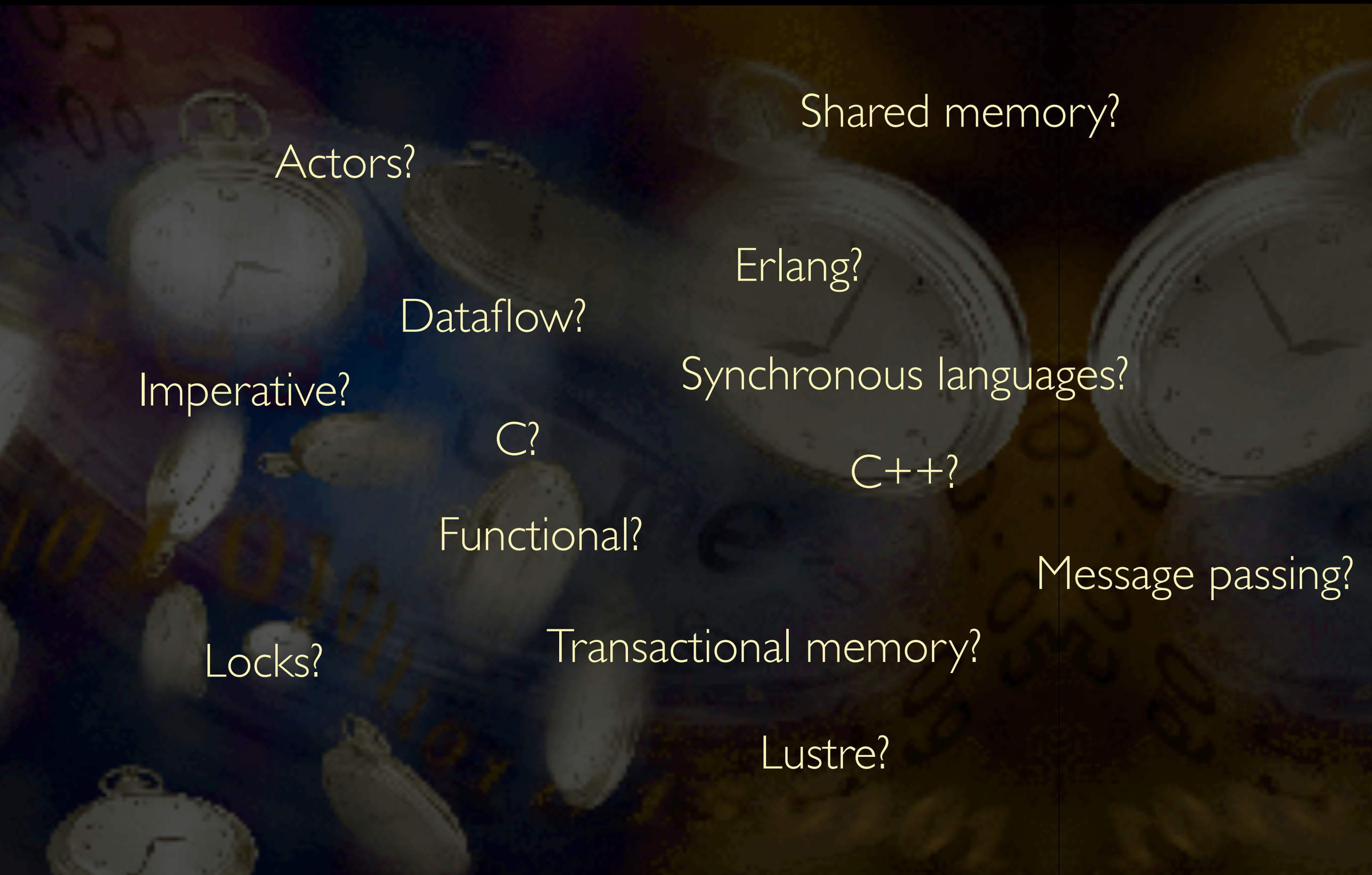
- Real-time systems are inherently concurrent with multiple tasks of different priorities running on the same machine
- Most concurrent systems have implicit timeliness and responsiveness expectations
- Yet, there is very little interest in Real-time from the mainstream programming language community

Why care about Real-time?

- The **programming model** for most real-time systems is 'defined' as a function of the hardware, operating system, and libraries.
- As a consequence real-time applications are not portable across platforms.
- Good news:

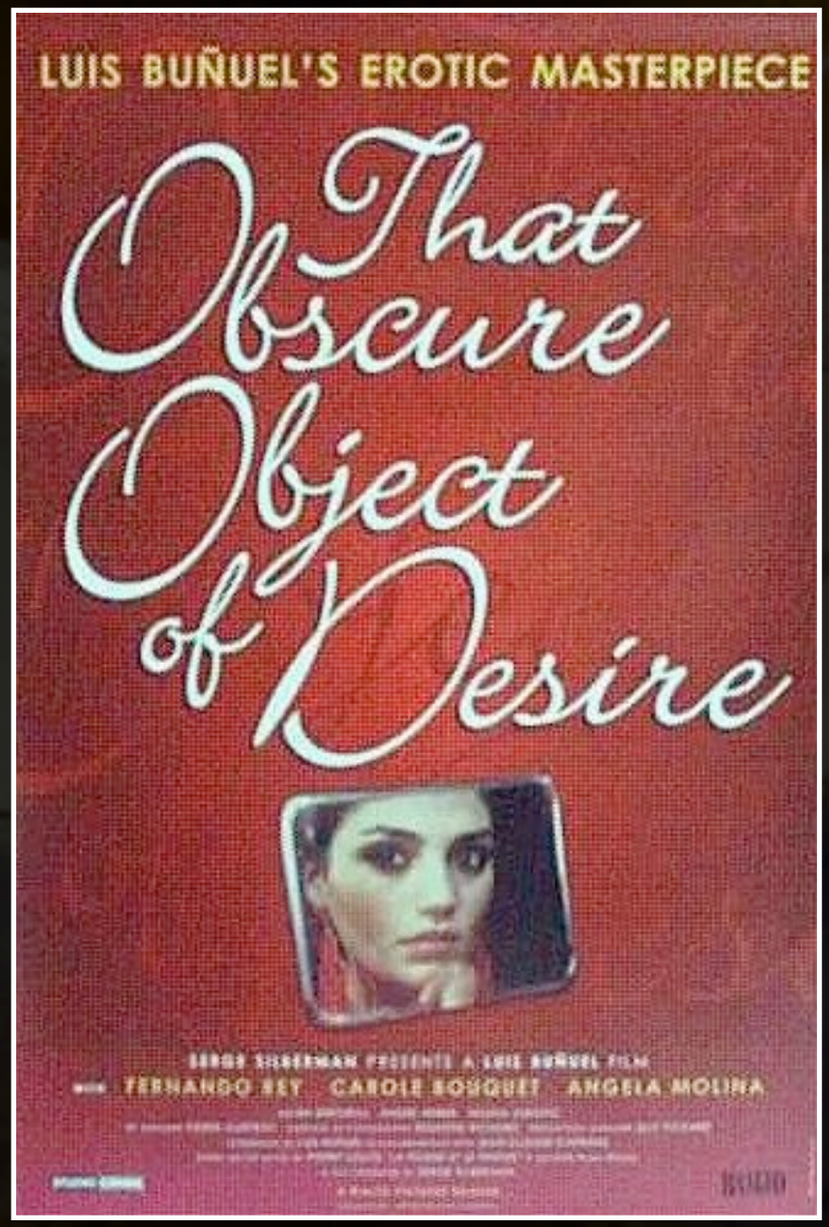
  programming languages have started to wrestle control from the lower layers of the stack and impose semantics (e.g. memory models).

What Programming model?

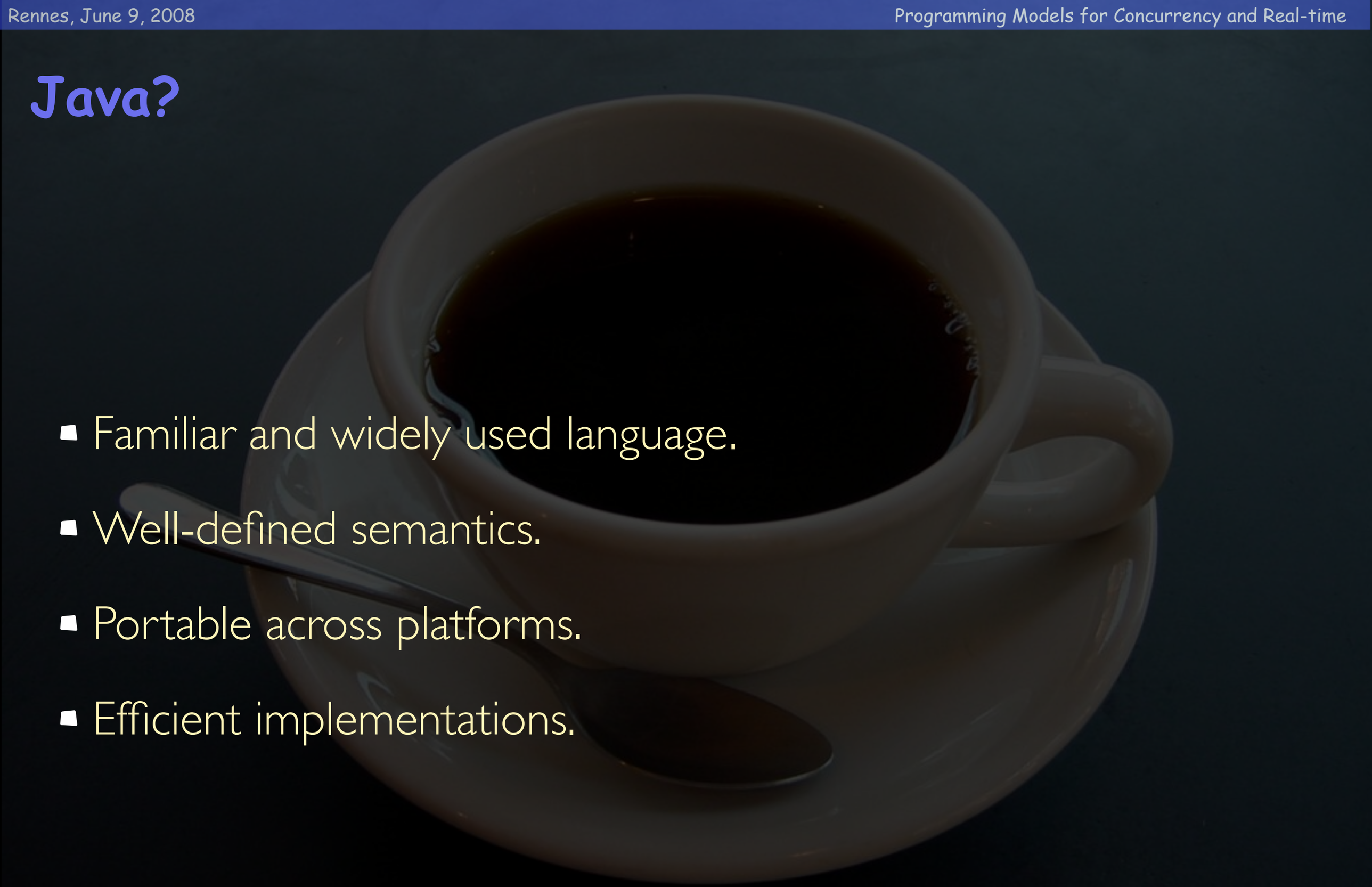


What Programming model?

- For the sake of acceptance, pick something familiar as a starting point…
- … an object-oriented language



Java?



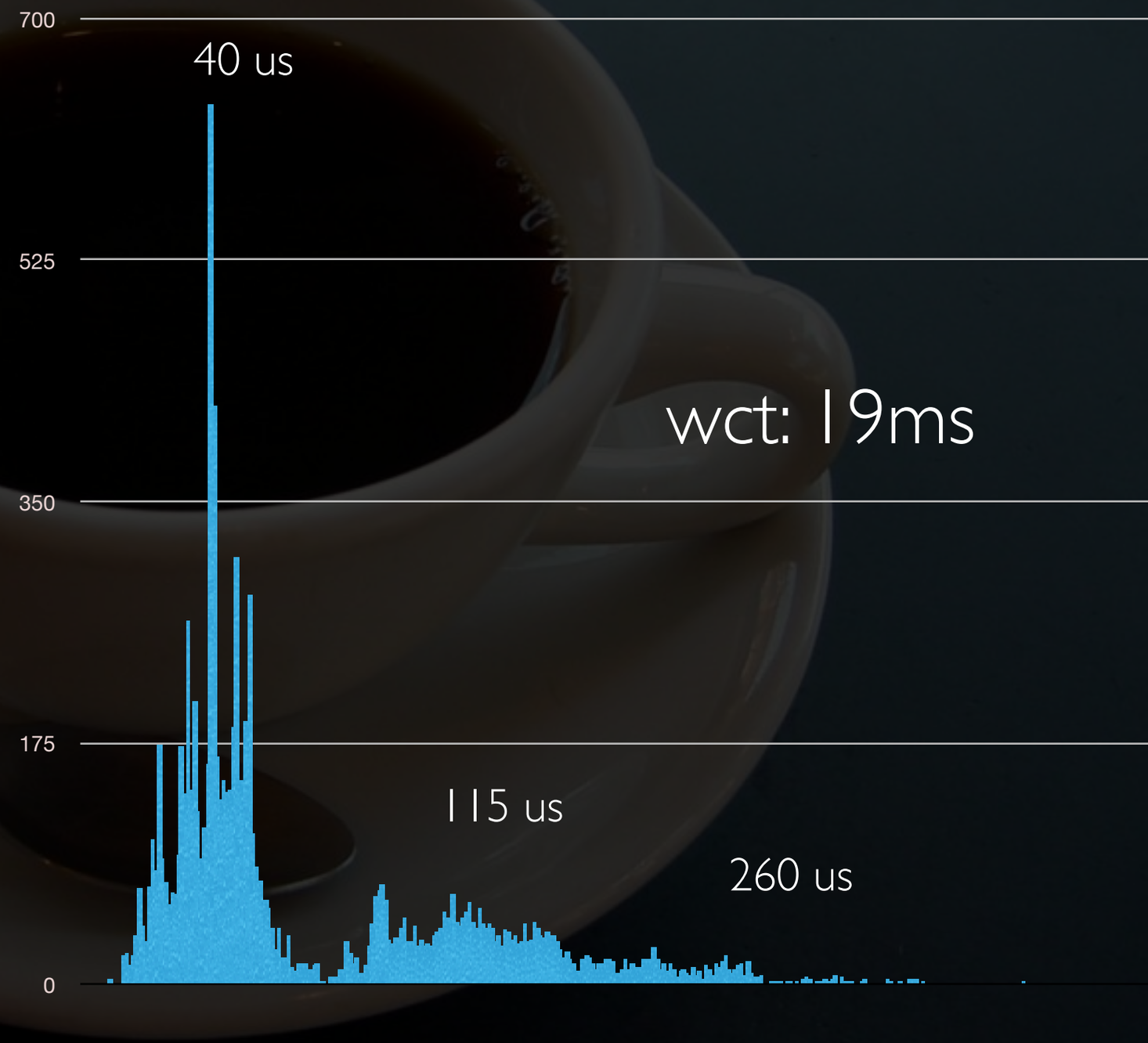
- Familiar and widely used language.
- Well-defined semantics.
- Portable across platforms.
- Efficient implementations.

Java?

- Predictable?
- *Not really.*

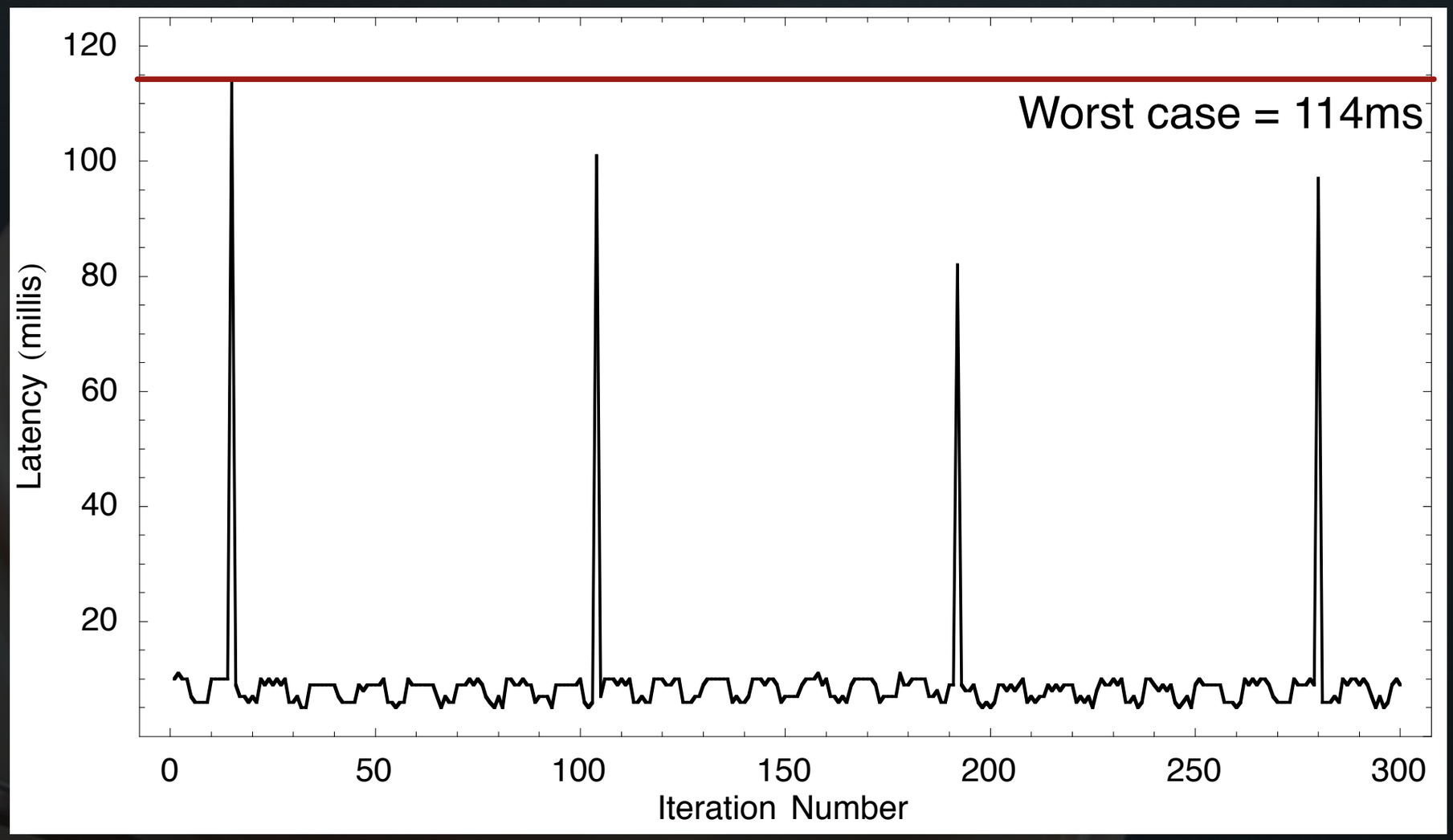
Call sleep(10ms) and get up 20 milis.sec. variability.

Hard real-time often requires microsecond accuracy.



Java?

- Predictable?



- **Java Collision Detector** running at 20Hz.
  - *Bartlett's Mostly Copying Collector. Ovm. Pentium IV 1600 MHz, 512 MB RAM, Linux 2.6.14, GCC 3.4.4*
  - GC pauses cause the collision detector to miss up to three deadlines… *and this is not a particularly hard problem should support KHz periods*

Java?

- Java for **hard** real-time?

RTSJ

The Real-time Specification for Java

Personal history

- Started on Real-time Java in 2001, in a DARPA funded project. At the time, no real RTSJ implementation
- Developed the Ovm virtual machine framework, a clean-room, open source RT Java virtual machine
- Fall 2005, first flight test with Java on a plane

Duke's Choice Award

RTSJ Programming Model

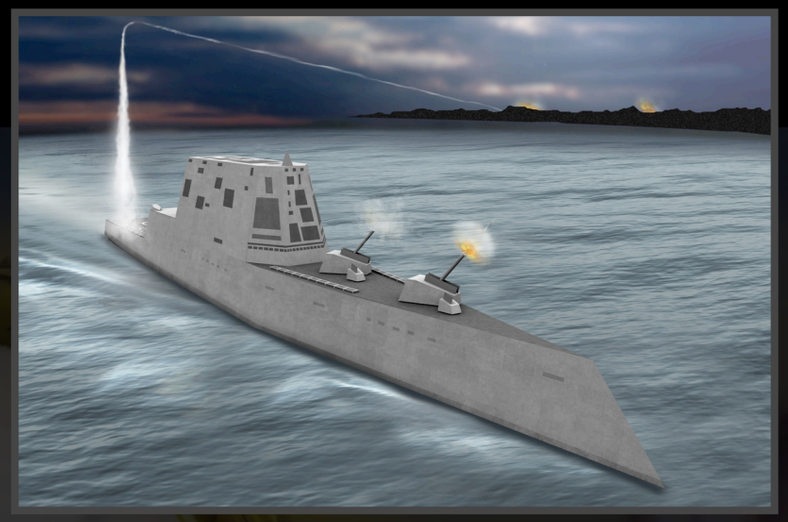
- Java-like:
  - Shared-memory,
  - lock-based synchronization,
  - first class threads.
- Main real-time additions:
  - Real-time threads + Real-time schedulers
  - Priority avoidance protocols: PIP or PCE
  - Region-based memory allocation (to avoid GC pauses)

RTSJ

- Commercial implementations:
  - PERCS (AONIX), JamaicaVM (AICAS) ahead-of-time compiler
  - McKinack (SUN) based on Hotspot, JIT, SMP, RTGC
  - Websphere (IBM) based on J9, ahead/JIT, SMP, RTGC

Real-time applications

- Shipboard computing
  - US navy Zumwalt-class Destroyer,
    Raytheon / IBM
    5 mio lines of Java code
    Real-time GC key part of system.
- Avionics
  - Zedasoft's Java flight simulator
  - Boeing Scaneagle UAV
- Financial information systems

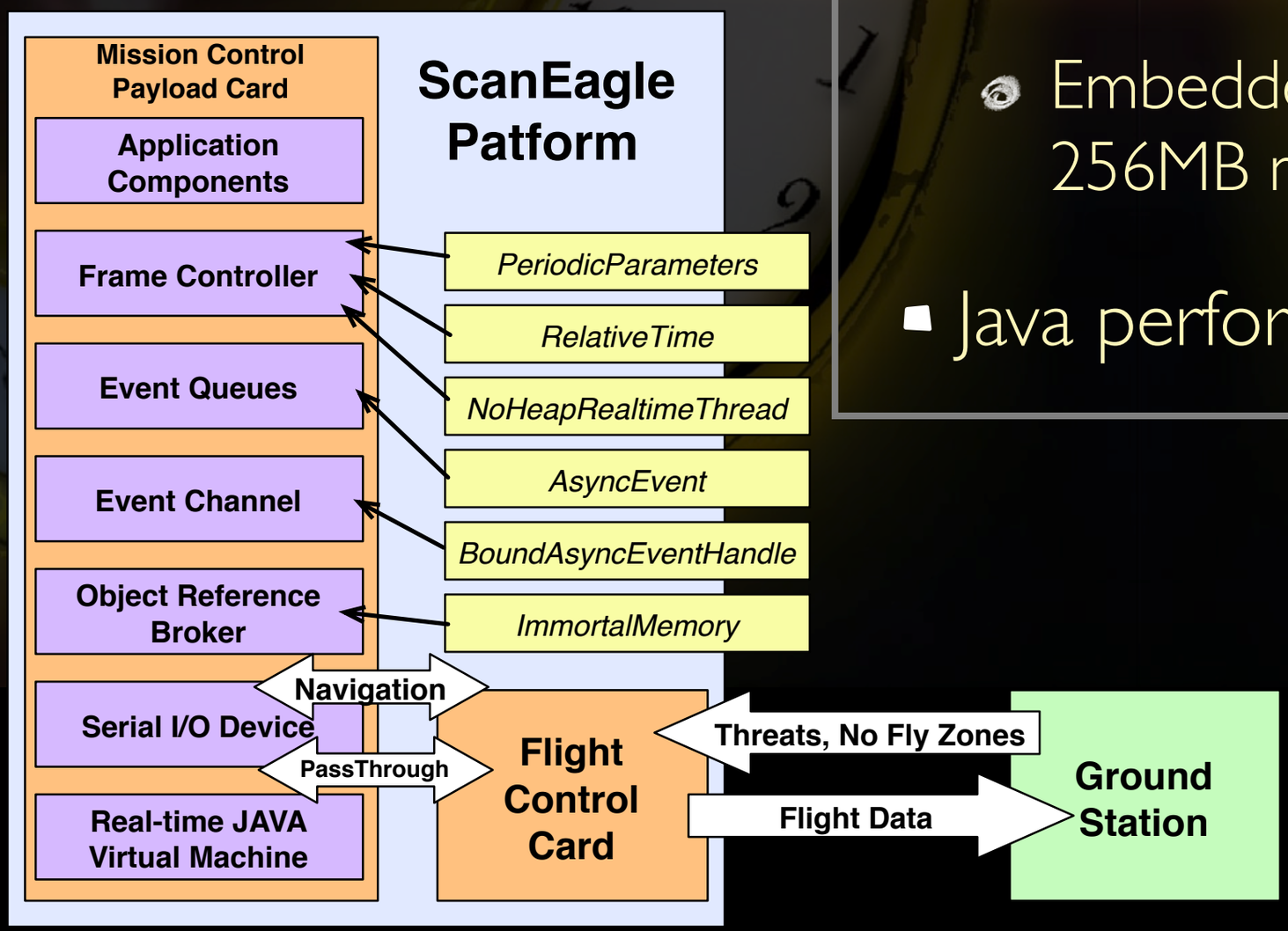


Case Study: ScanEagle

ScanEagle

ScanEagle

- Flight Software:
  - 953 Java classes, 6616 methods.
    Multiple Priority Processing:
    - High (20Hz) - Communicate with Flight Controls
    - Medium (5 Hz) - Computation of navigation data
    - Low (1 Hz) - Performance Computation
  - Embedded Planet 300 Mhz PPC, 256MB memory, Embedded Linux
- Java performed better than C++



Lessons from the RTSJ

1/9/90 Rule

- The main reason of RTSJ's success is that it allows mixing freely real-time and timing-oblivious code in the same platform
- Rule of thumb: 1% hard real-time, 9% soft real-time, 90% non-RT
- RT Programming Model should:
  - Allow non-intrusive injection of real-time in a timing-oblivious system
  - Avoid paradigm switches---single semantic framework for RT/non-RT

Small Footprint

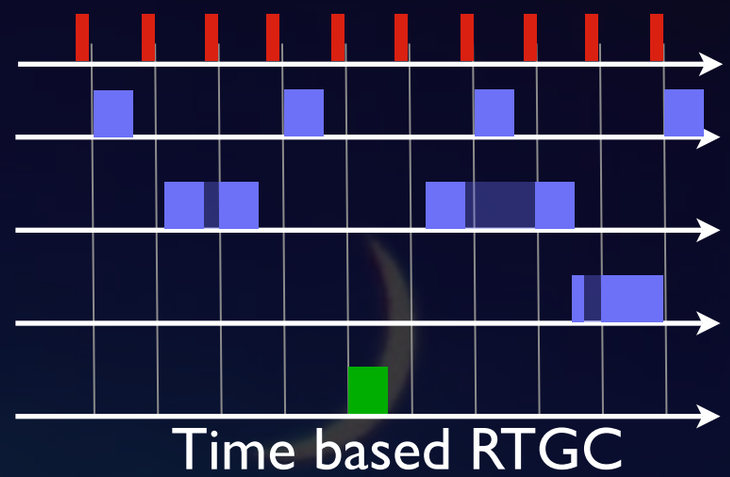
- The RTSJ requires sweeping changes in the Virtual Machine and has complex interaction with Java features.
  - The RTSJ changes the semantics of "normal" Java code:
    - any access to a reference variable may throw an exception,
    - meaning of "throws" changed to support asynchronous exception,
    - finalization interacts with memory management.
- Avoid complex APIs with ill-defined feature interactions
- Don't change the semantics of non-RT code

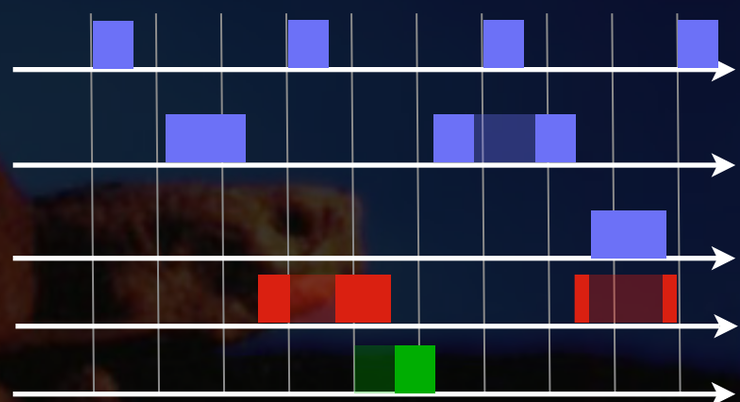
Schedule Flexibly

- Support for real-time scheduling is crucial
  with a clean integration in the language concurrency model
  - Priority Preemptive Scheduling for periodic tasks
  - Support PIP and PCE locks for priority inversion avoidance
- Provide facilities for writing non-blocking algorithms
- Allow for user-defined schedulers.

Don't Fear Garbage

- Real-time Garbage Collectors good enough for most RT apps
  - Time-based collectors (Metronome) can easily bound pauses to one millisecond.
  - Automatic defragmentation for long lived apps.
  - Slack-based collectors (SUN's) ensure RT code never preempted by GC.
  - Ensuring that the GC can keep up requires some understanding of whole-program allocation behavior.

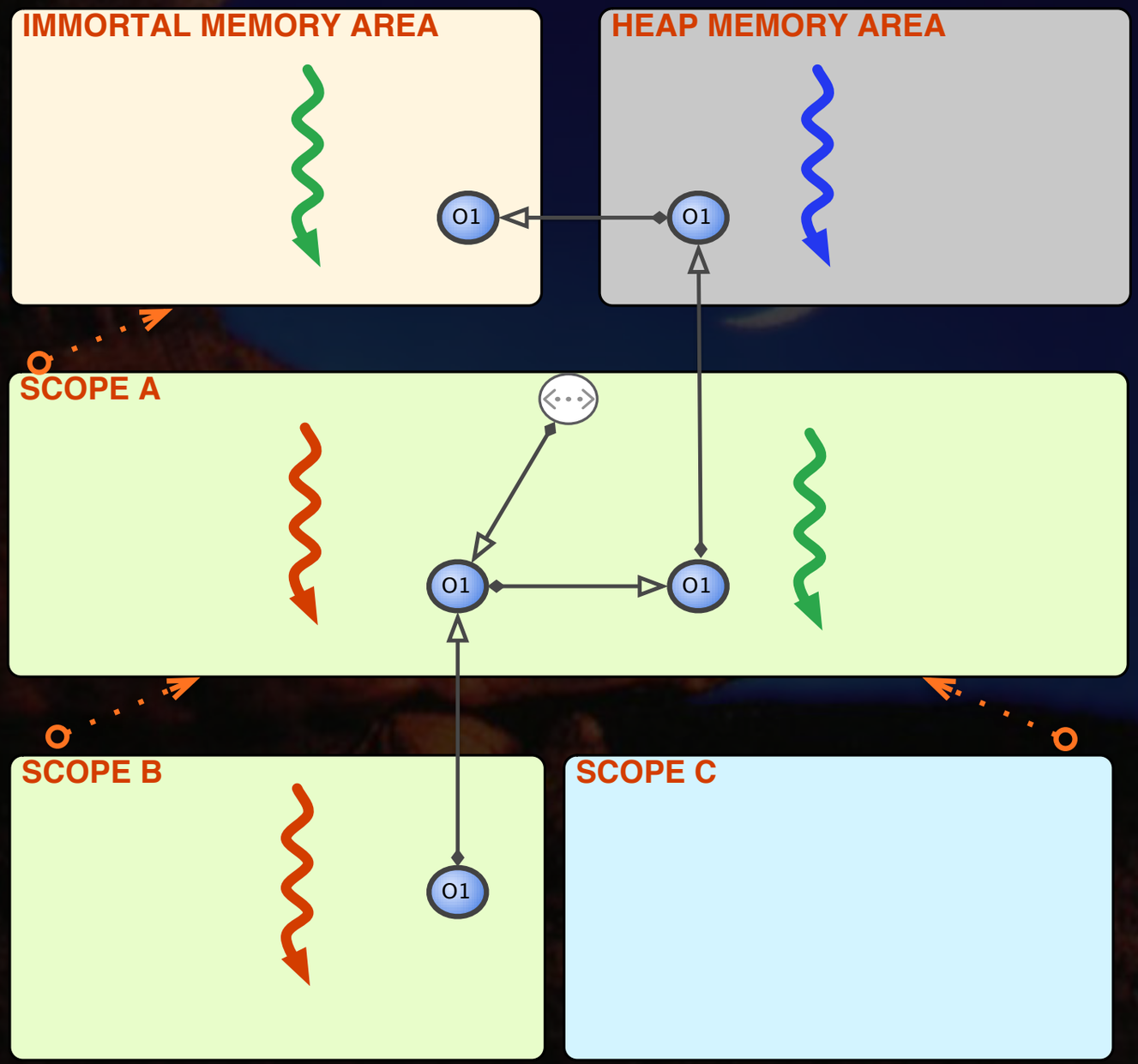




- Avoid manual memory management unless absolutely necessary.

Type-check Region-based Allocation

- Regions are the worst mistake of the RTSJ
  - Dynamic safety checks slow down programs and cause runtime errors that are hard to prevent
  - Distinction between heap-using and non-heap thread is tricky and error prone.
- Use static type system for assurance and efficiency



Shared-memory Concurrency is Hard

- Real-time presents a number of concurrency related challenges.
  - Critical sections must be short --- avoid over-synchronization
  - Avoid priority inversion between real-time and non-RT code
  - Avoid blocking operations in RT code
- Limit/localize communication between RT and non-RT.
- Support flexible isolation to encourage determinism.

Flexible Task Graphs

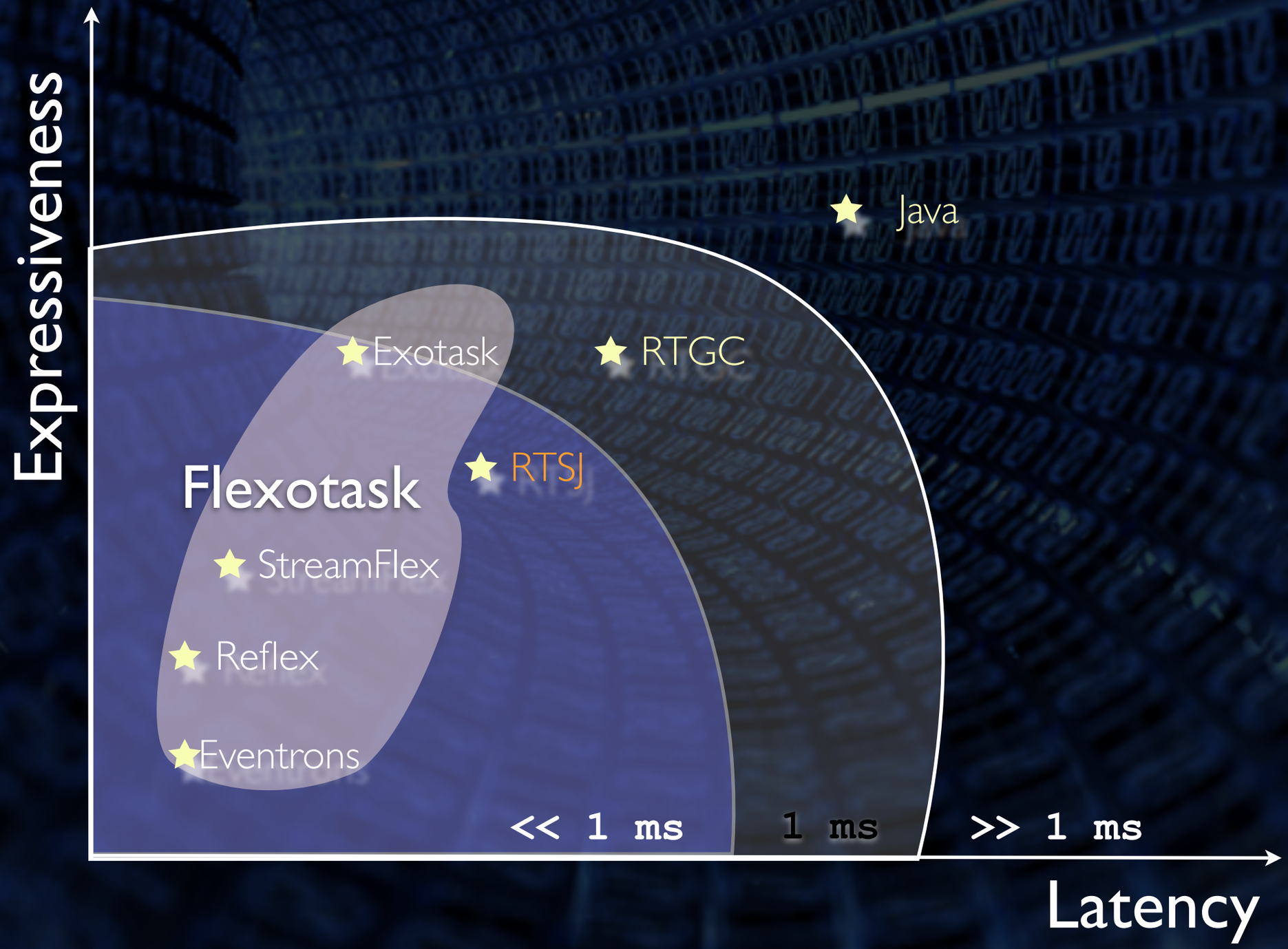
Goals

- Design a new real-time programming model that allows embedding hard real-time computations in timing-oblivious Java applications
- Principle of Least Surprise
  - Semantics of non-real-time code unchanged
  - Semantics of real-time code unsurprising
- Limited set of new abstractions that compose flexibly
- No cheating
  - Run efficiently in a production environment

Unification of previous work

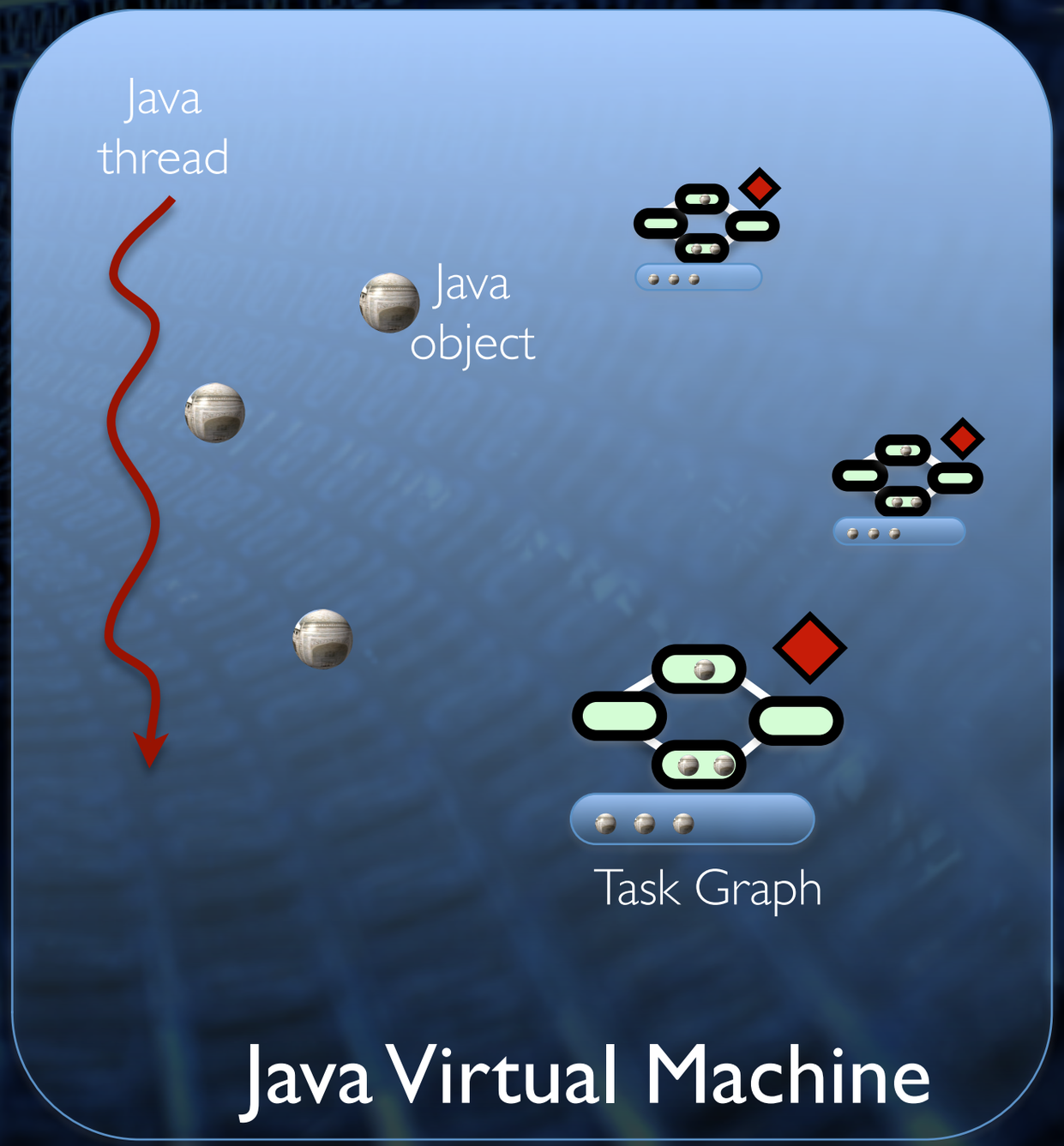
- **Eventrons** [PLDI'06] (IBM)
- **Reflexes** [VEE'07] (Purdue/EPFL)
  - Inspired by RTSJ and Eventrons
- **Exotasks** [LCTES'07] (IBM)
  - Inspired by Giotto, and E-machine
- **StreamFlex** [OOPSLA'07] (Purdue/EPFL)
  - Inspired by Reflexes, StreamIt and dataflow languages

Design space



Flexible Task Graphs

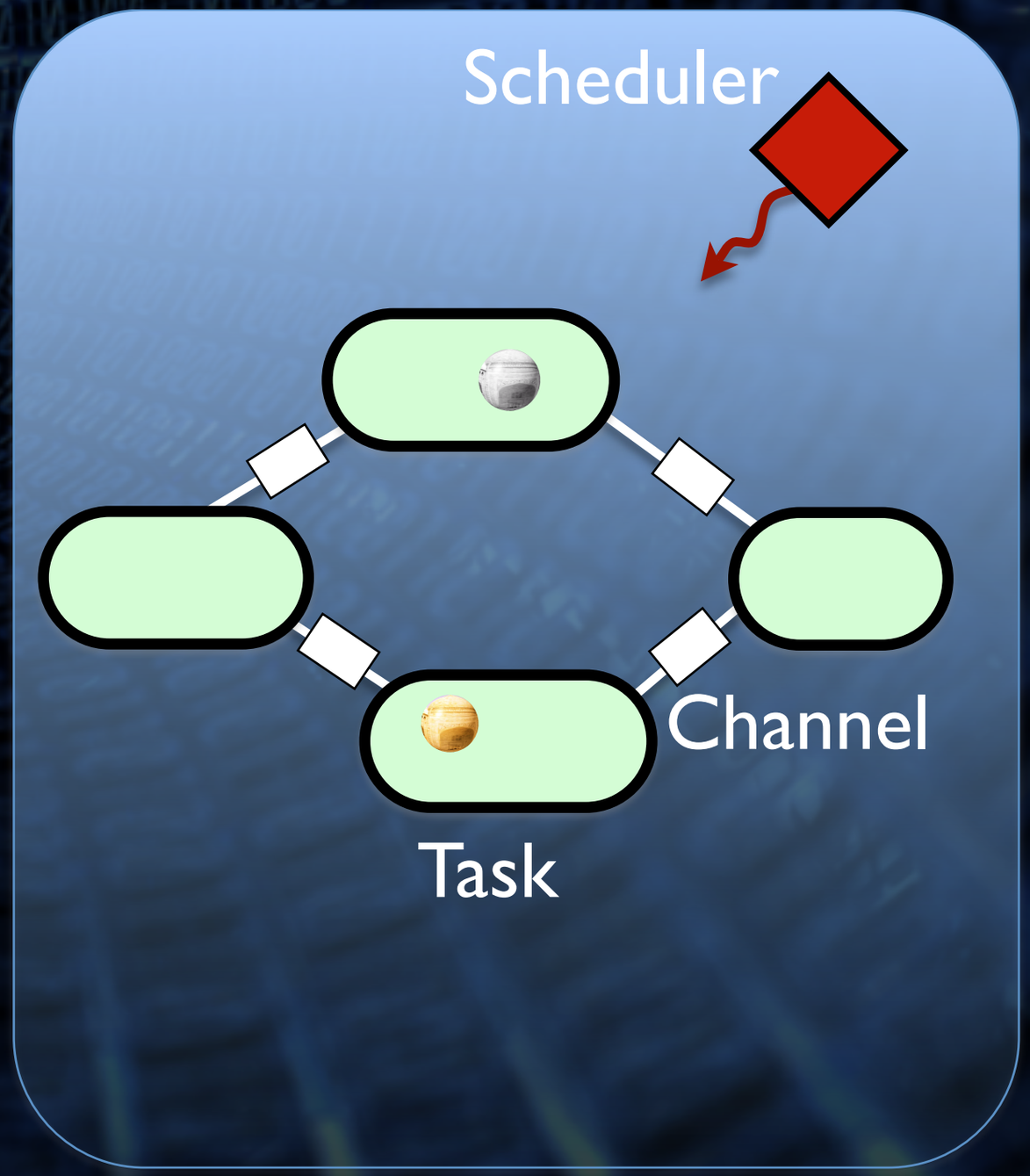
- A *FlexoTask Graph* is a set of *concurrently* executing, *isolated*, tasks communicating through non-blocking *channels*
- Semantics of legacy code is unaffected
- Real-time code has restricted semantics, enforced by compile and start-up time static checks



Task Graph

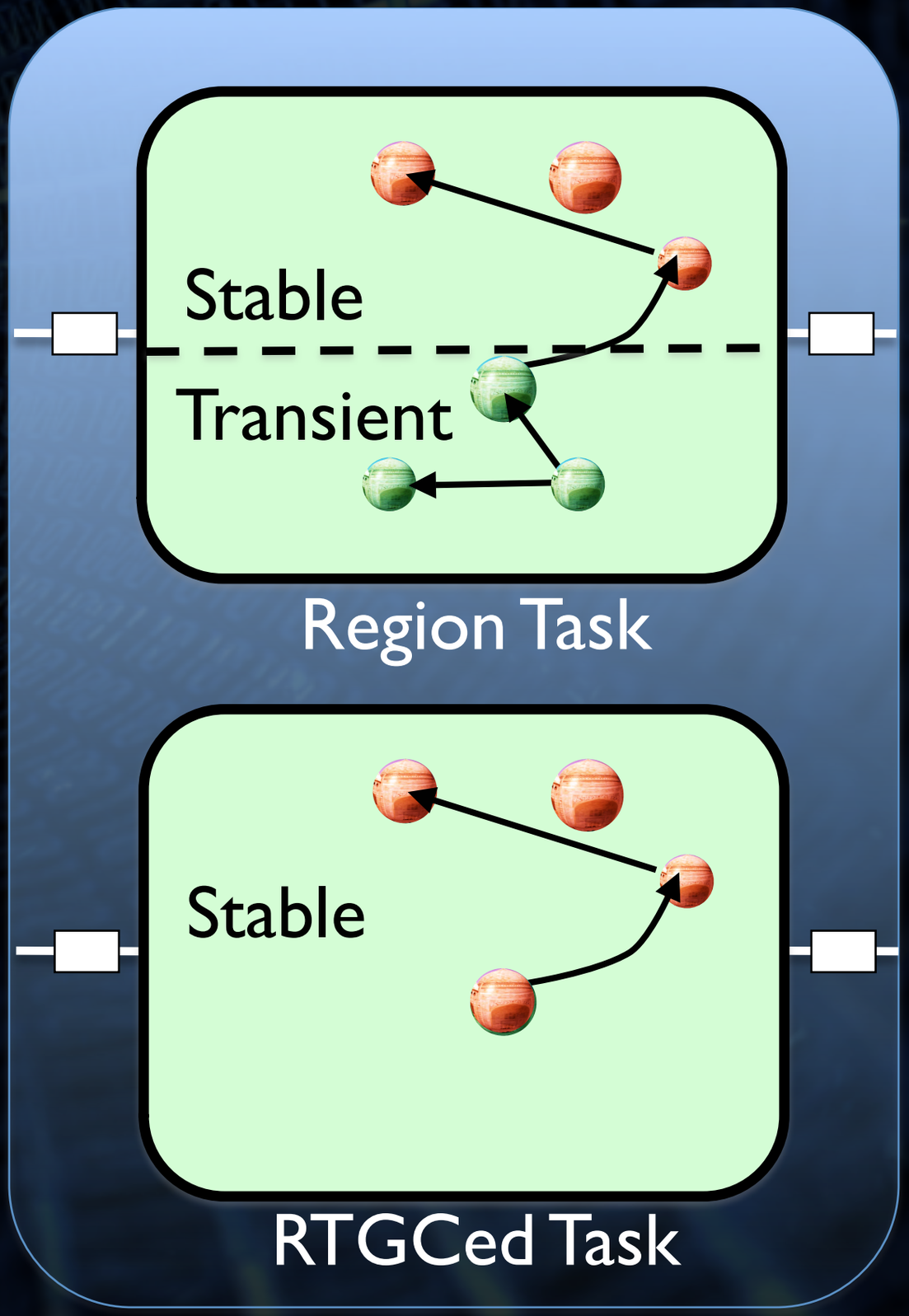
- A *FlexoTask Graph* is a set of *concurrently* executing, *isolated*, tasks communicating through *channels*

- Schedulers control the execution of tasks with user-defined policies (eg. logical execution time, data driven)
  - atomically update task's in ports
  - invoke task's **`execute`**()
  - update the task's output ports



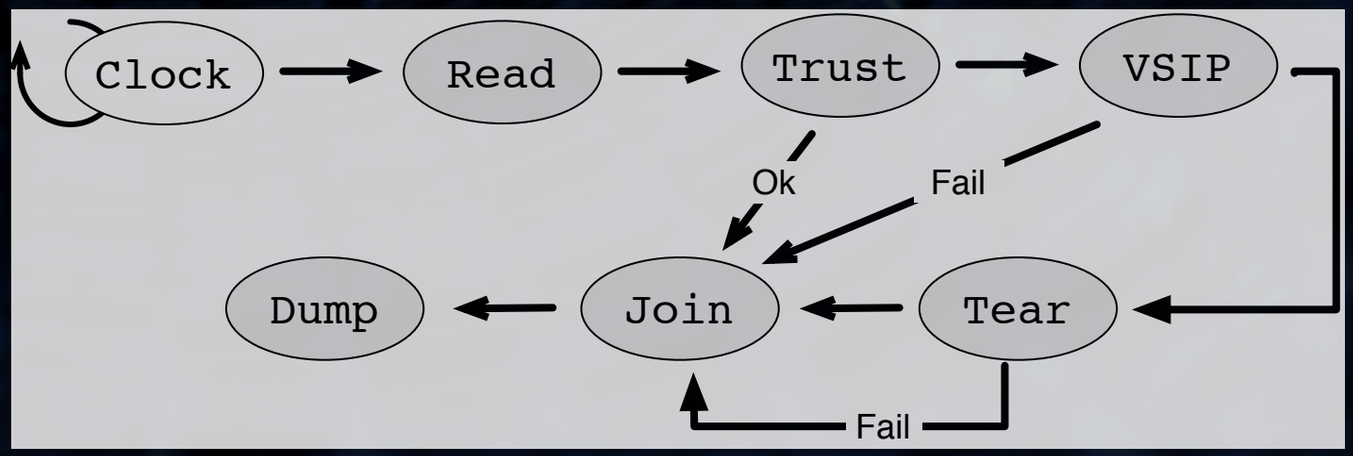
Memory management

- Either garbage collected with a real-time GC, or a region allocator for sub-millisecond response times.
- Region-allocated tasks preempt task RTGC and Java GC
- Region tasks are split between
  - Stable objects
  - Transient (per invocation) objects



Task communication: Channels

- Design goals: scalability and expressiveness
- Rule # 1: isolation to allow local, per task GC (avoid dependencies on global allocation rates)
- Rule #2: allow for decomposition of problems in stages

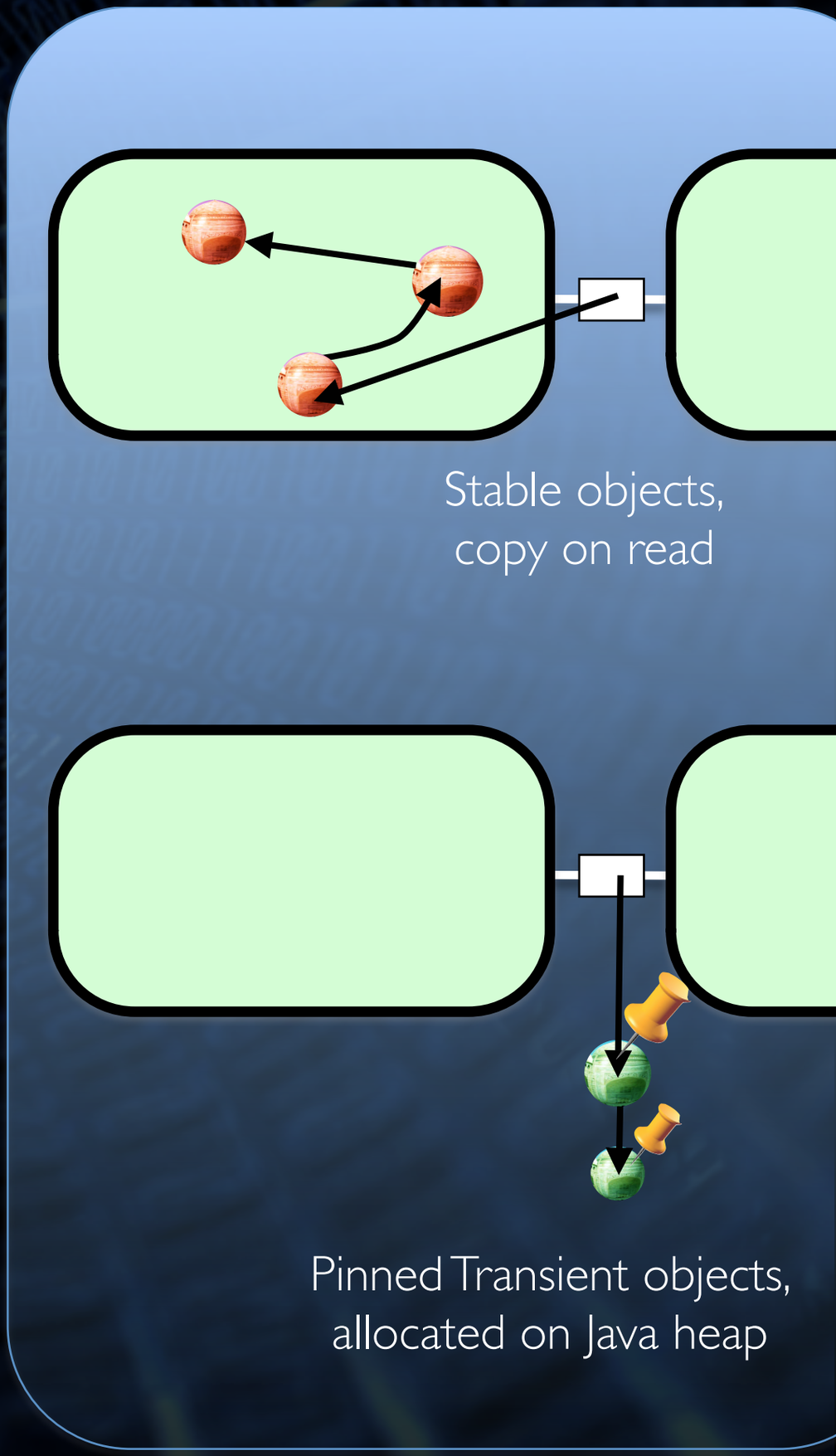


e.g. Intrusion detection task graph running at 750Mb per sec [OOPSLA07]

Channels

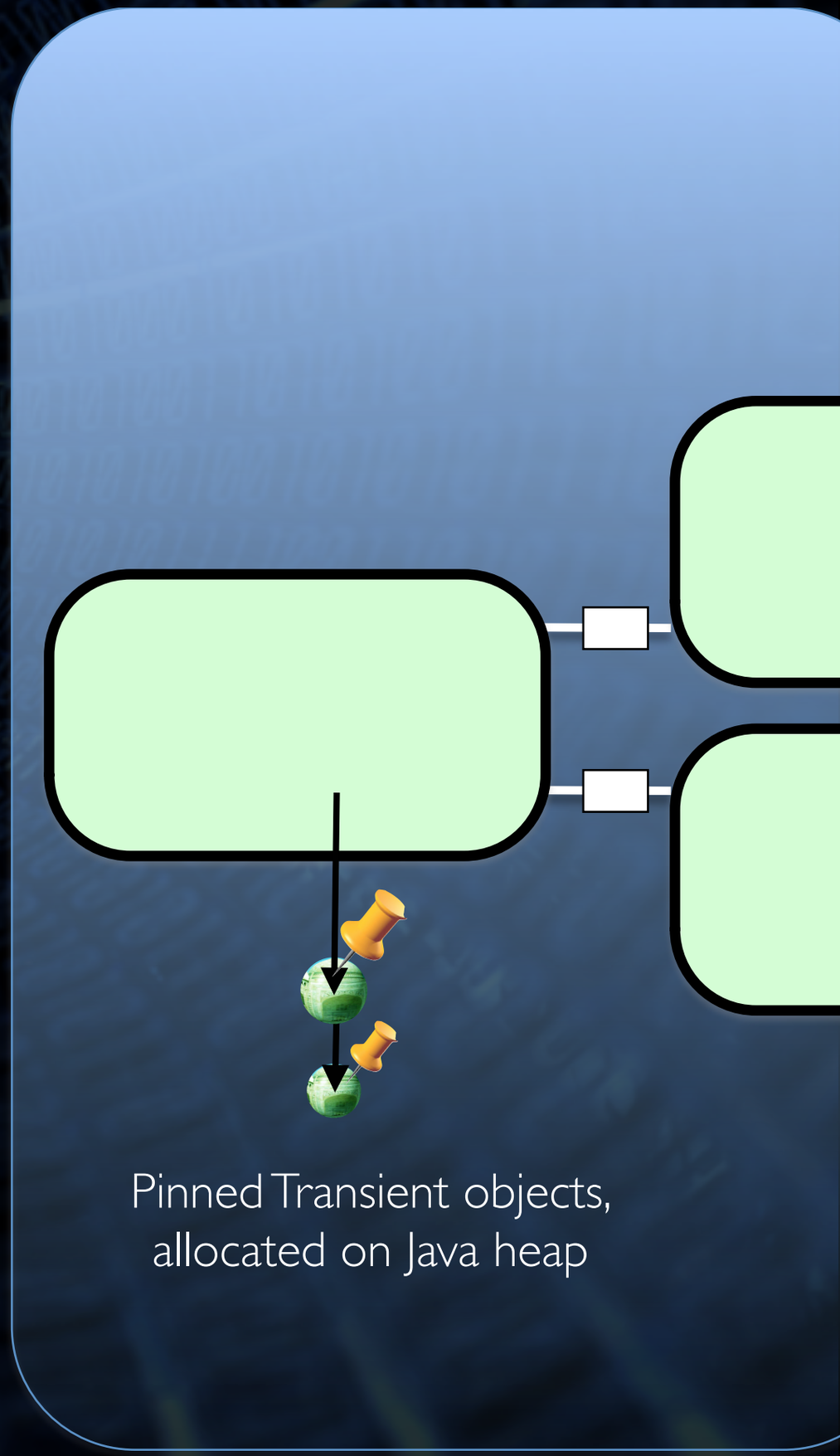
- Stable channels
  - Can refer to any stable object (complex structures)
  - Deep copy on read (atomic)

- Transient channels
  - Can refer to Capsules (transient objects, arrays)
  - Zero-copy (linear reference)



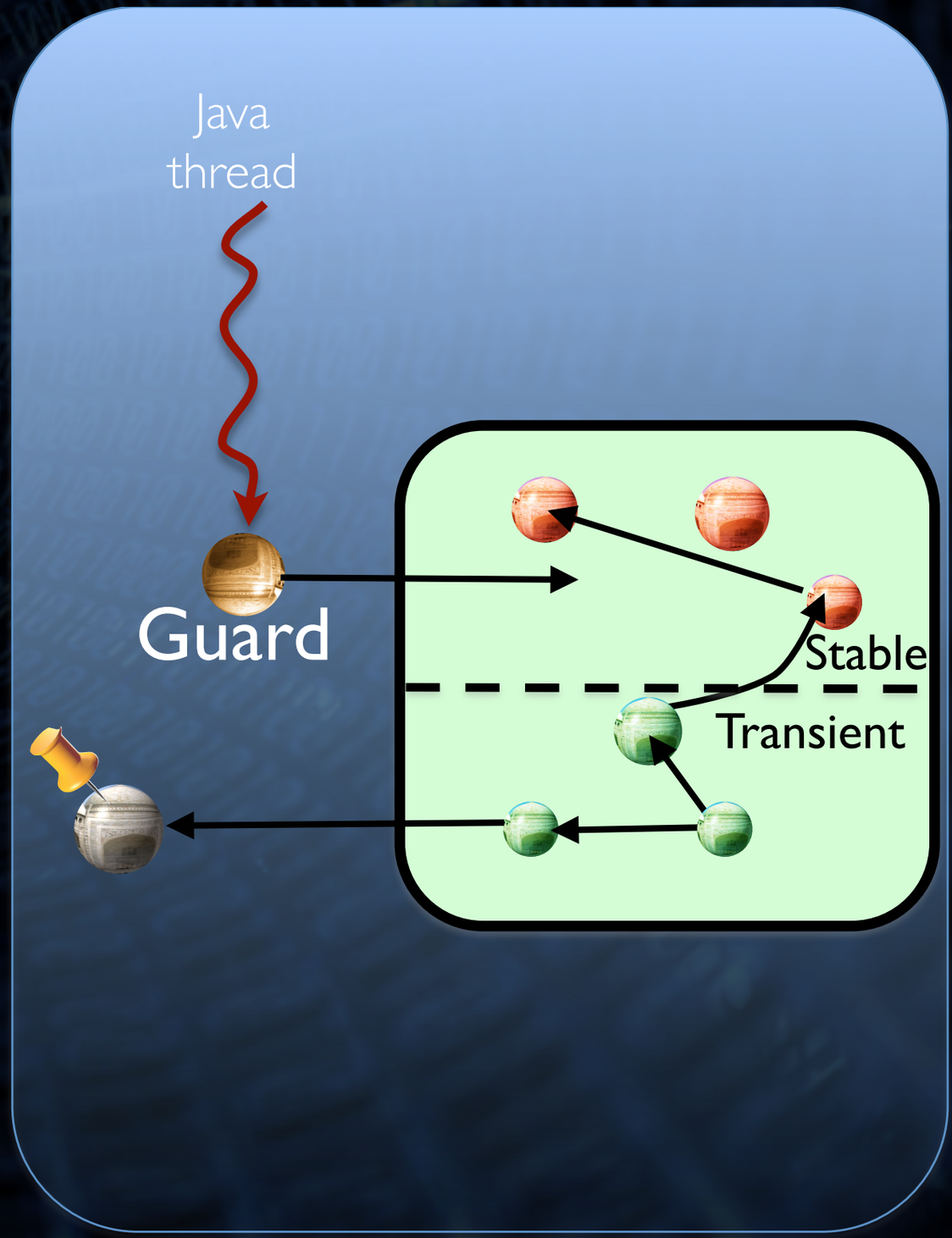
Channels

- Allow double send of a capsule?
- Immutable capsules
  - *will force users to perform copies of the data*
- Mutable capsules
  - Zero-copy
    - *mutable capsules are shared introducing data races*
  - Copy
    - *copies are done atomically when the task execute() method returns*



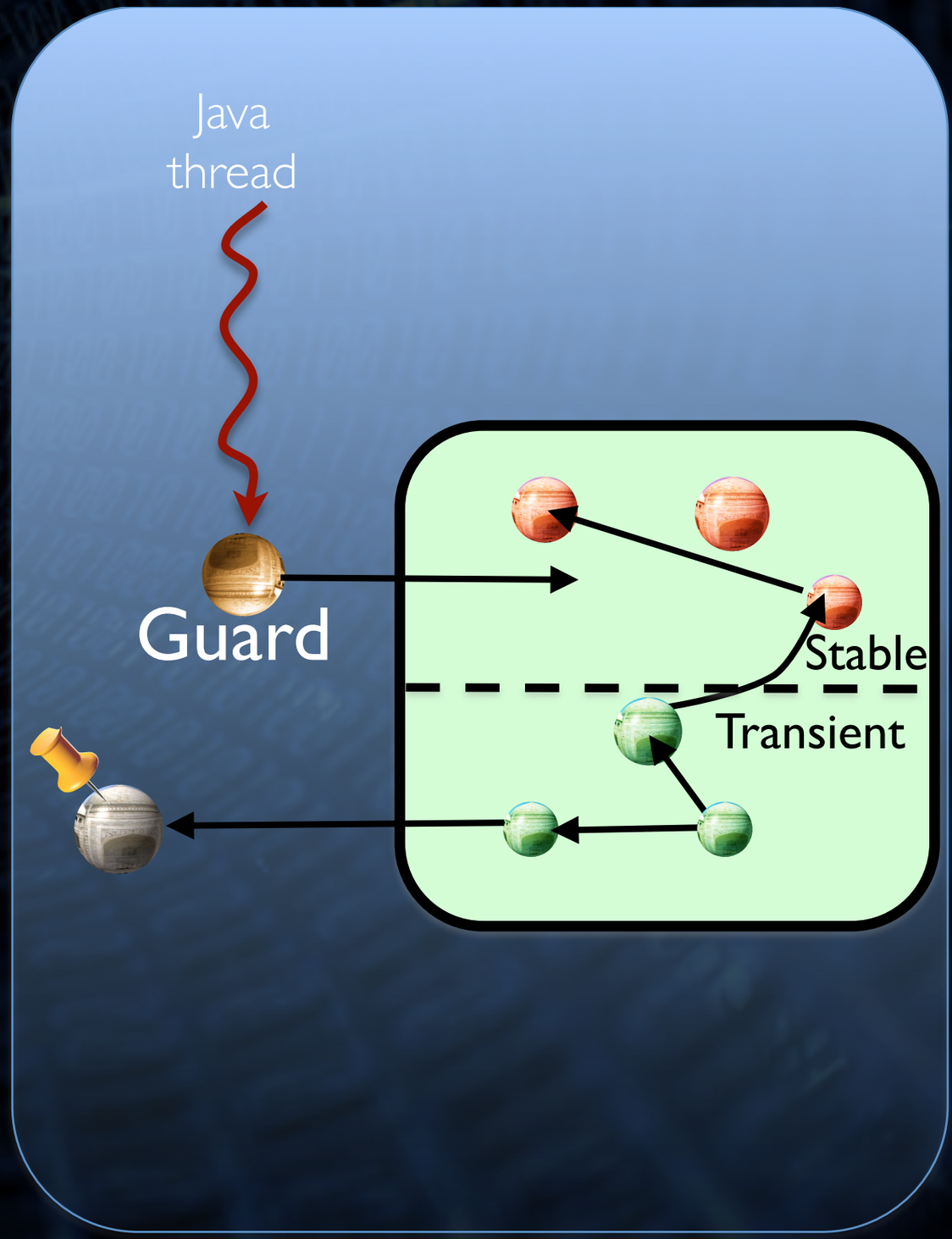
Communication with Java

- Challenges:
  - Avoiding priority inversion
  - Enforcing isolation



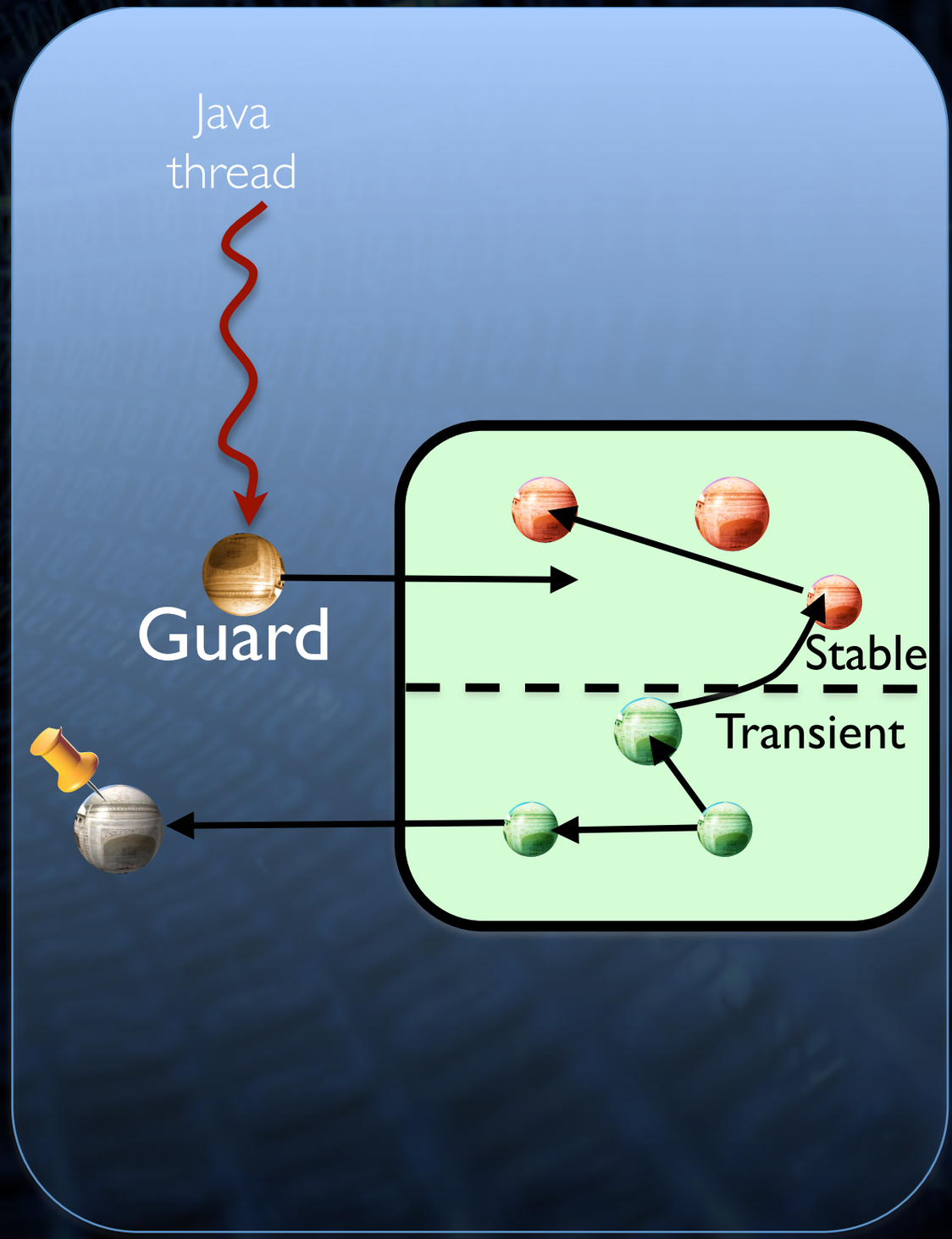
Communication with Java

- Every task has an automatically generated proxy-object
- User-defined atomic methods can be called from Java with transactional semantics
- Arguments are reference-immutable pinned objects



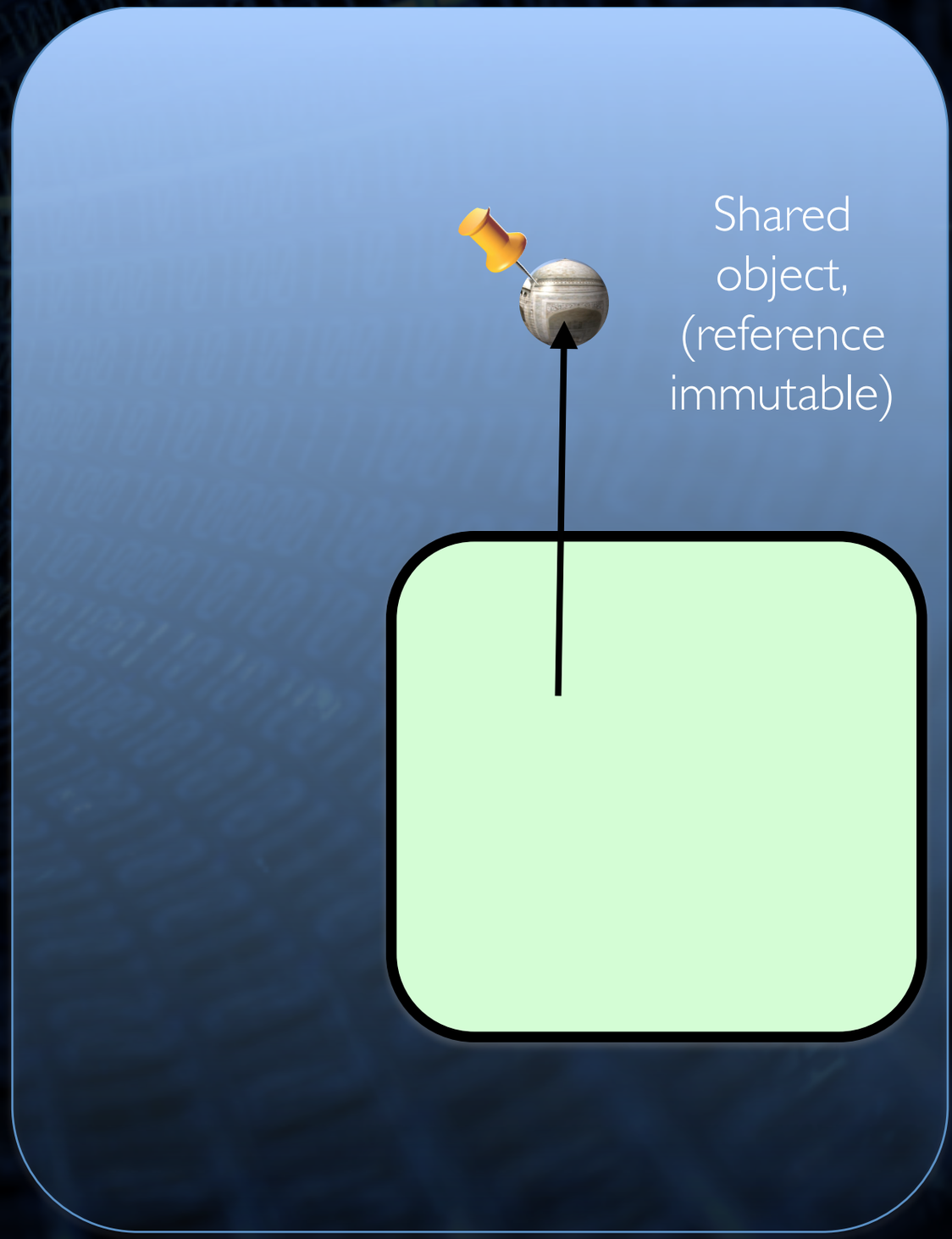
Communication with Java

- Atomic Methods:
  - acquire a lock on guard & pin all reference-immutable arguments
  - start transaction;
  - execute method
  - commit transaction
  - reclaim transient memory
  - unpin all arguments & release lock on guard
- If during execution of the method the Task is scheduled, the transaction is immediately aborted.



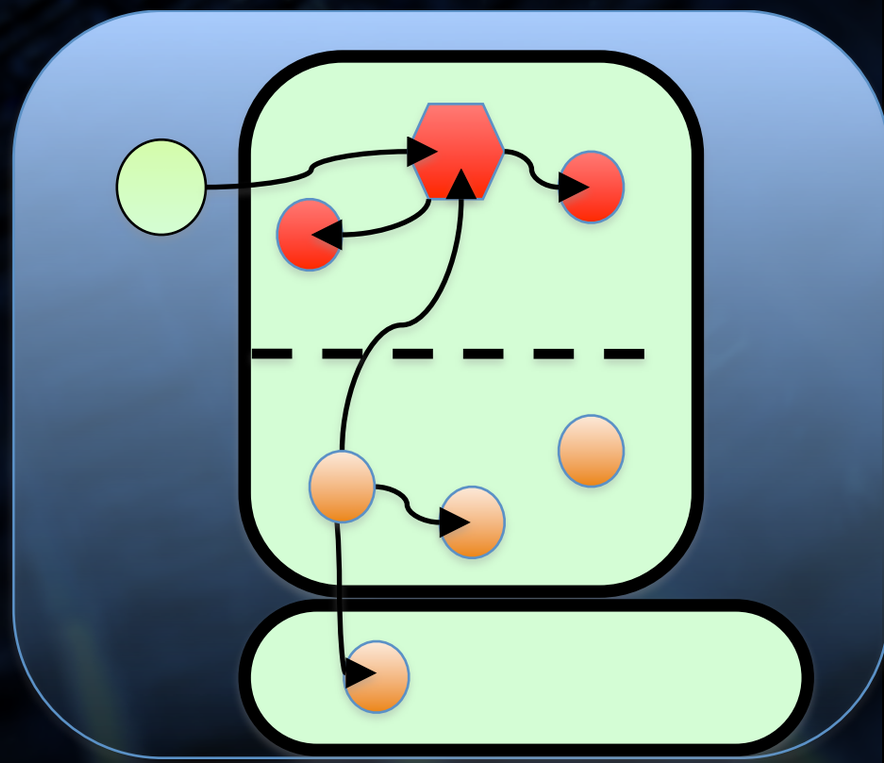
Shared Objects

- Shared objects can be given as startup argument to a task
- They must be pinned
- Restricted to reference-immutable objects to prevent breaking isolation
- *This is a loophole! Breaks determinism guarantees, but sometimes necessary.*



Static safety

- Safety checks prevent references to transient objects after they have been deallocated and to capsules once they have been sent.
- A form of ownership types is used where **`Stable`** is a marker interface for data in the stable heap and **`Capsule`** for messages. Some polymorphism is needed for arrays.
- Checking is done statically, no dynamic tests are need.



Main program

```
// Obtain the graph template:
in =  getResourceAsStream("Detector.ftg");
spec = FlexotaskXMLParser.parseStream(in);


// To instantiate graph with sharing:
sharedArray = new RawFrameArray();
params.put("DetectorTask", sharedArray);
graph = spec.validate("TTScheduler", parameters);


// To obtain reference to Detector
detector = graph.getGuardObject("DetectorTask");
graph.start();


// To communicate a new frame:
int frameIndex = detector.getFirstFree();
frames.get(frameIndex).copy(lengths, callsigns, posns);
detector.setNextToProcess(frameIndex);
```

Detector task

```
interface DetectorGuard implements ExternalMethods {
  void setNextToProcess(int idx) throws AtomicException;
  ...
}

class StateTable implements Stable { … }

class DetectorTask extends AtomicFlexotask
                   implements DetectorGuard {
  private StateTable state;
  private RawFrameArray frames;
  private int nextToProcess;

  void initialize(...,  Object param) {
    frames = (RawFrameArray) param;
    state = new StateTable();
  }
```

Detector task (ctd)

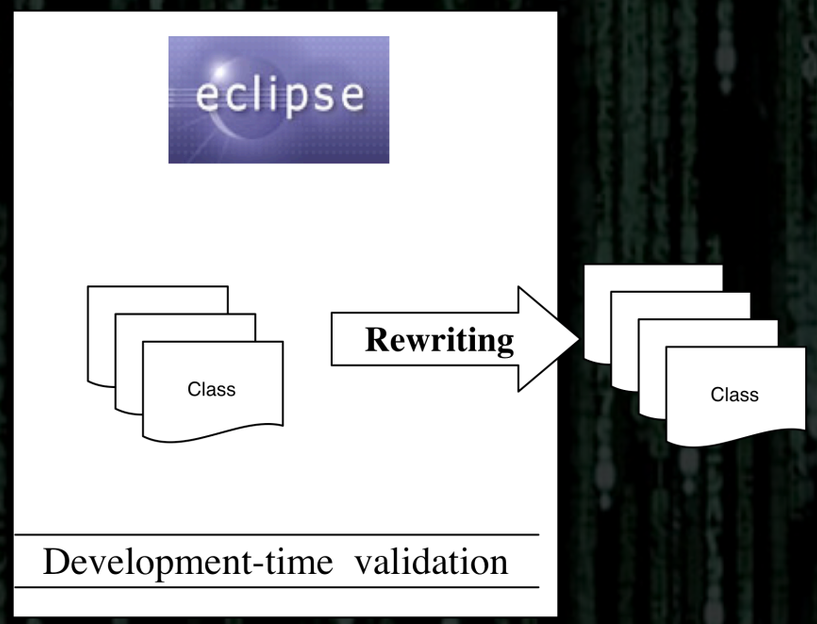
```
void execute() {
    if (nextToProcess != firstFree) {
        cd = new Detector(state, Constants.GOOD_VOXEL_SIZE);
        cd.setFrame(frames.get(nextToProcess));
        cd.run();
        nextToProcess = firstFree = increment(nextToProcess);
    }
 }
```

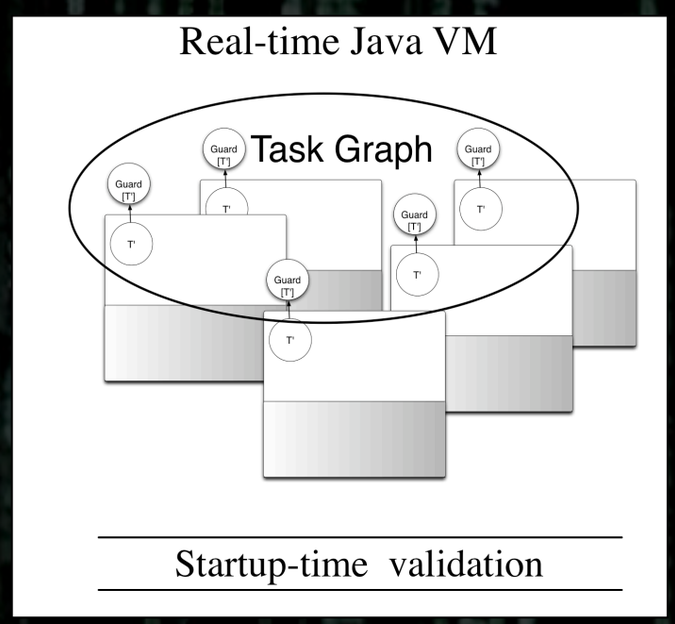
```
int getFirstFree() throws AtomicException {
    int check = increment(firstFree);
    if (check == nextToProcess) return -1;
    int ans = firstFree;
    firstFree = check;
    return ans;
}

void setNextToProcess(int ntP) throws AtomicException {
    this.nextToProcess = ntP;
}
```

Implementation

Implementation Overview

- Eclipse
  - IDE support for task graph construction
  - Static checker for development time warnings
  - Byte-code rewriter for transactions and memory management
- IBM's J9 production real-time VM
  - Start up time verifier
  - Runtime support



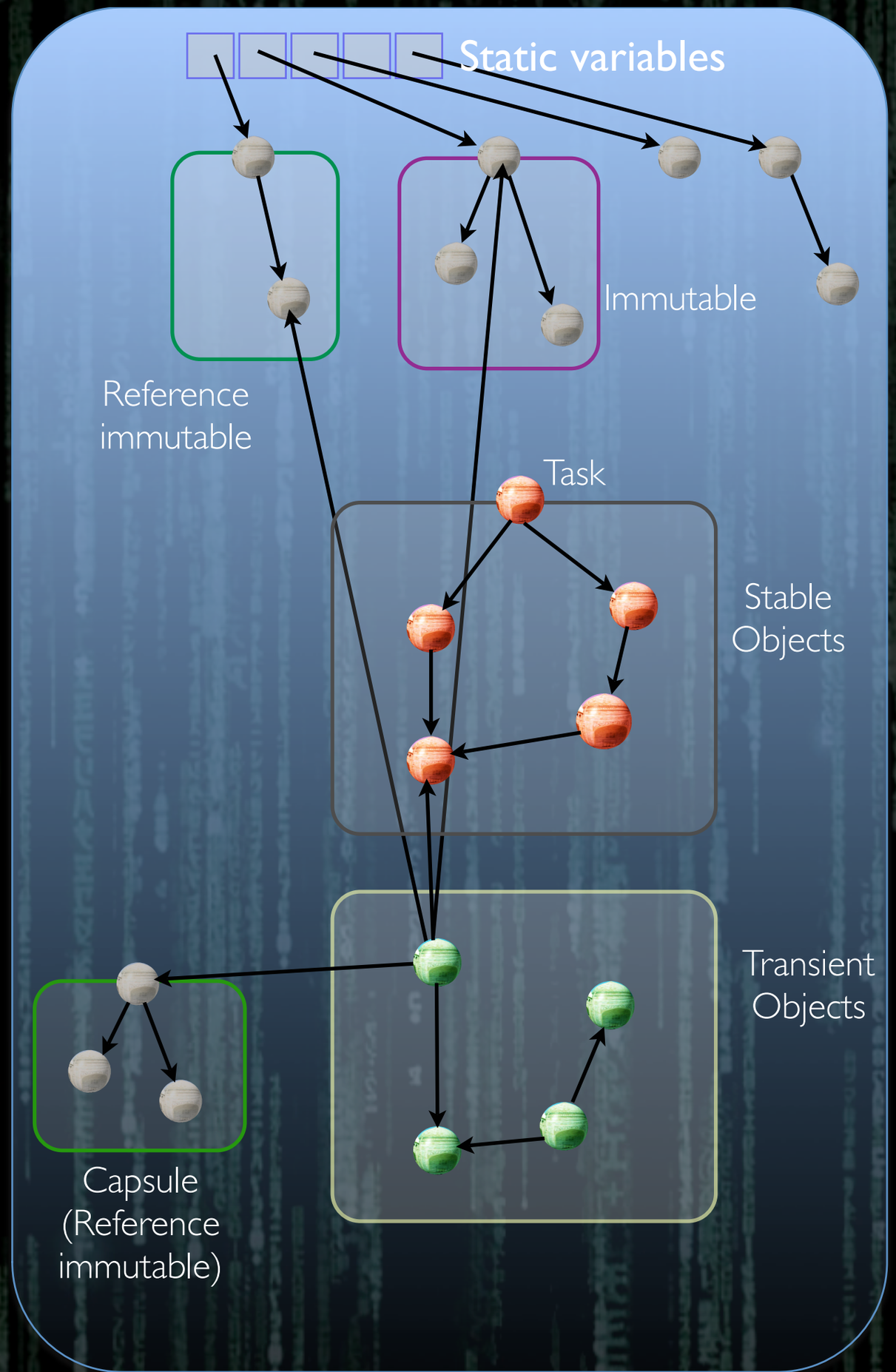


Static checking

- Validation performed at compile-time and at start-up time
- Enforce many restrictions for isolation (e.g. limits on `getstatic`), memory safety (ownership and reference immutability), programming model compliance (e.g. no thread operations, or synchronized statements or finalization).
- Use Rapid Type Analysis with value information at initialization time.

Static checking

- A class is reference immutable if all of its reference fields are effectively final and of reference immutable type
- Inference of 'effectively' final fields.
- Can infer Stable classes.



Atomics

- Use Rapid Type Analysis to approximate call graph of atomic methods. Generate a transactional version of all reachable methods.
- Use a single roll-forward transaction log per task. Only log changes to stable objects.

Evaluation

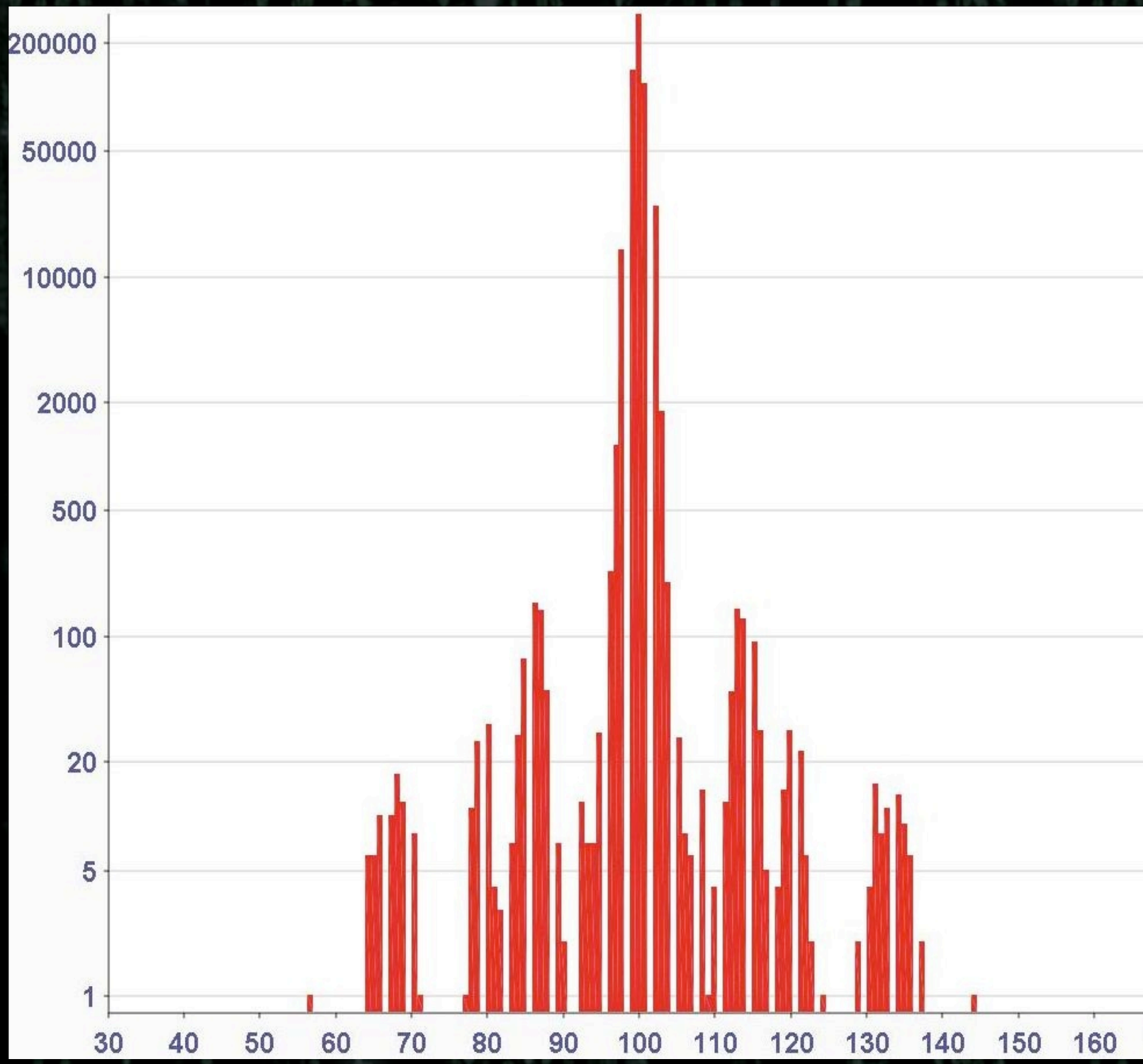
- Experimental setup
  - IBM Websphere Real Time (WRT) VM
  - RHEL 5 Linux, kernel 2.6.21.4 (real-time config)
  - IBM Blade server, 4 dual-core AMD Opteron 64 2.4 GHz, 12GB RAM

Predictability

- High Frequency Reader, 540 LOC.
  - Scheduled for 100us periods, plain Java thread invoking transactional method every 20ms
  - Noise maker thread allocating 2MB per second using 48 byte objects, maintaining live set of 40,000 objects

Predictability

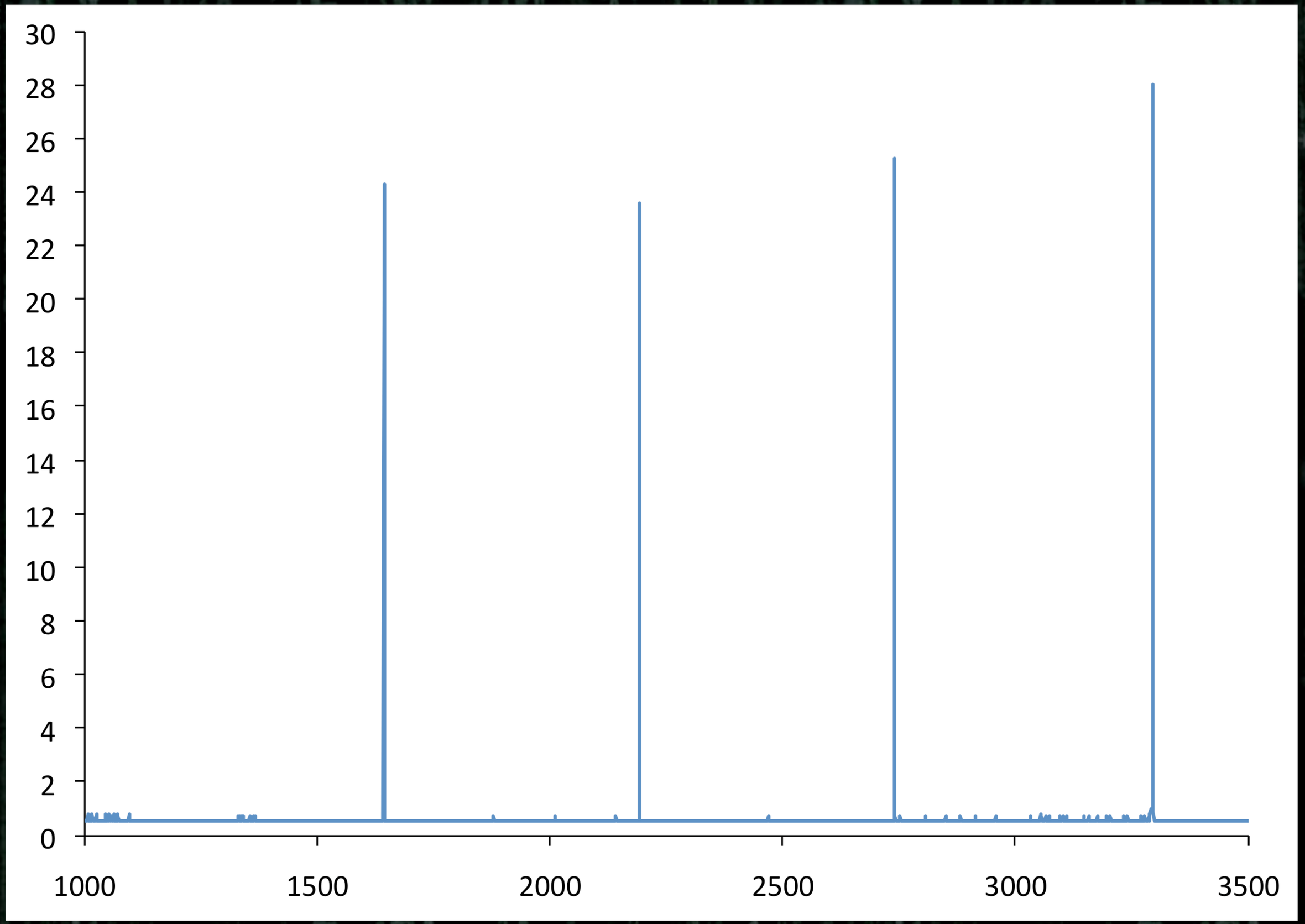
- Summary:
  - 600,000 periodic invocations
  - Inter-arrival time between 57 and 144us
  - 516 aborts of the atomic method



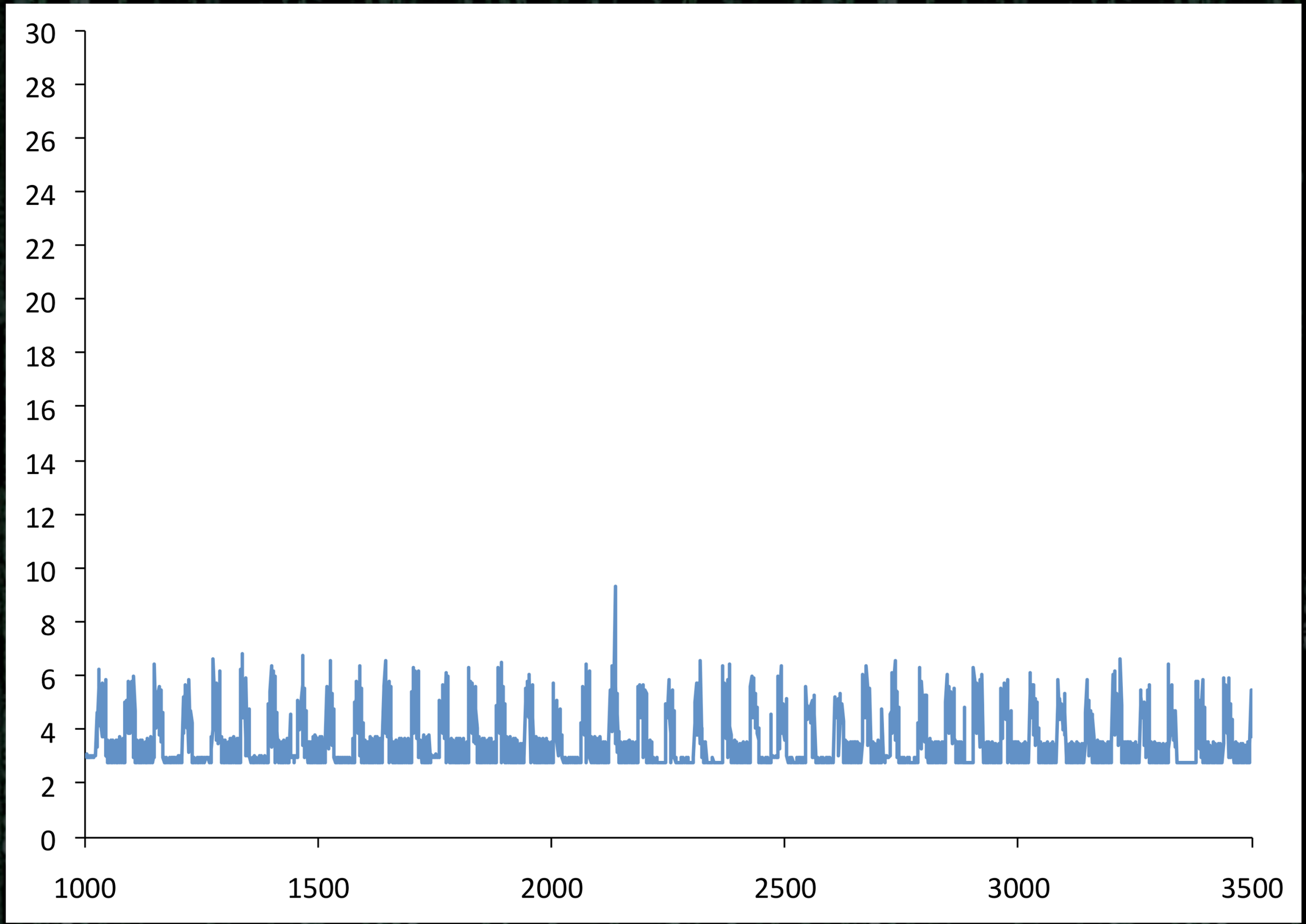
Performance

- Avionics Collision Detector, 30KLOC
  - Flexotask (detector) scheduled for 20ms periods. Plain Java thread (simulator) communicating with Detector every 20ms
  - Noise maker thread allocating 2MB per second using 48 byte objects, maintaining live set of 150,000 objects
  - Four variants: Plain Java, Plain Java/RTGC, (RTSJ/NHRT) and Flexotasks

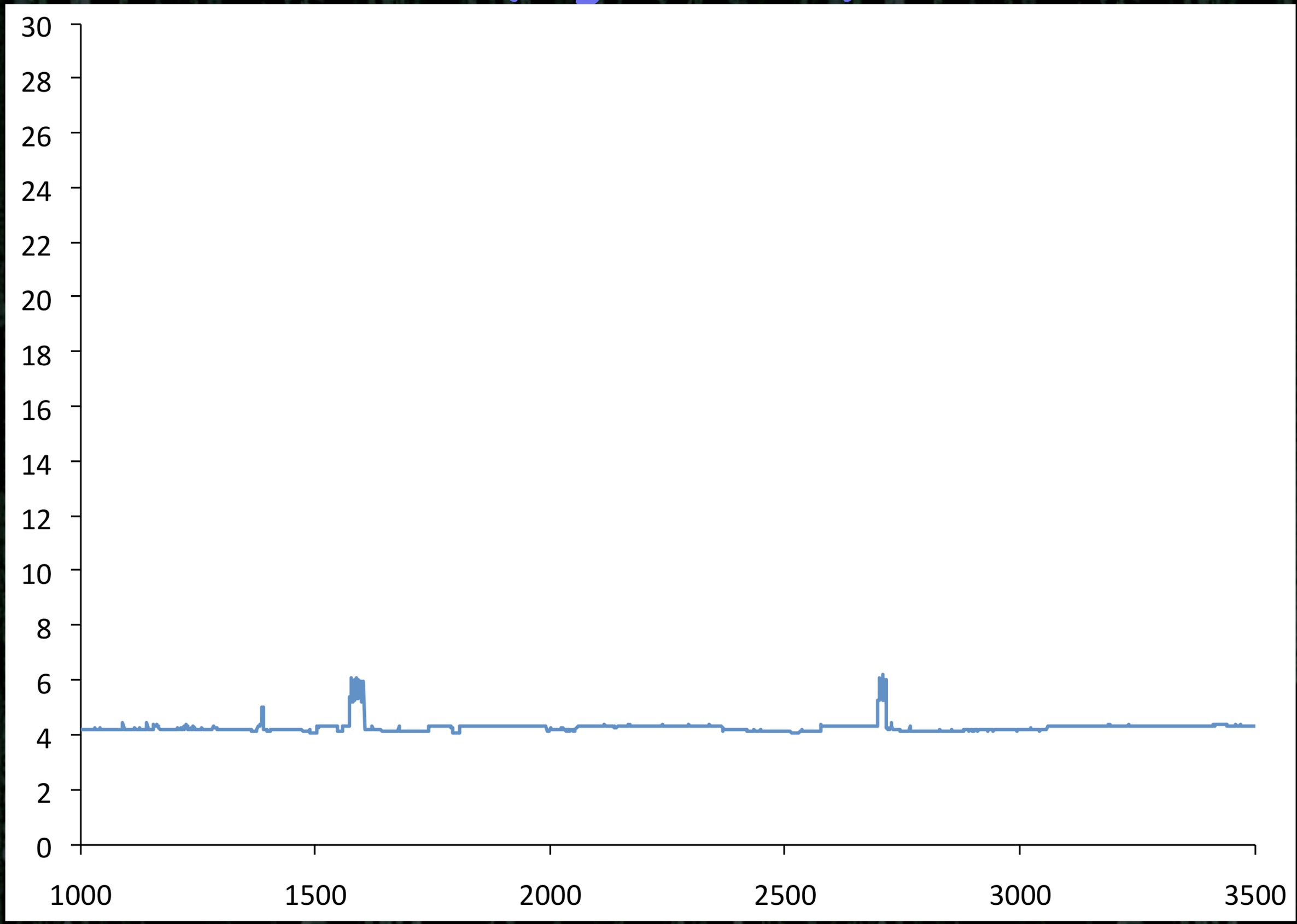
CD with Plain Java



CD with RTGC

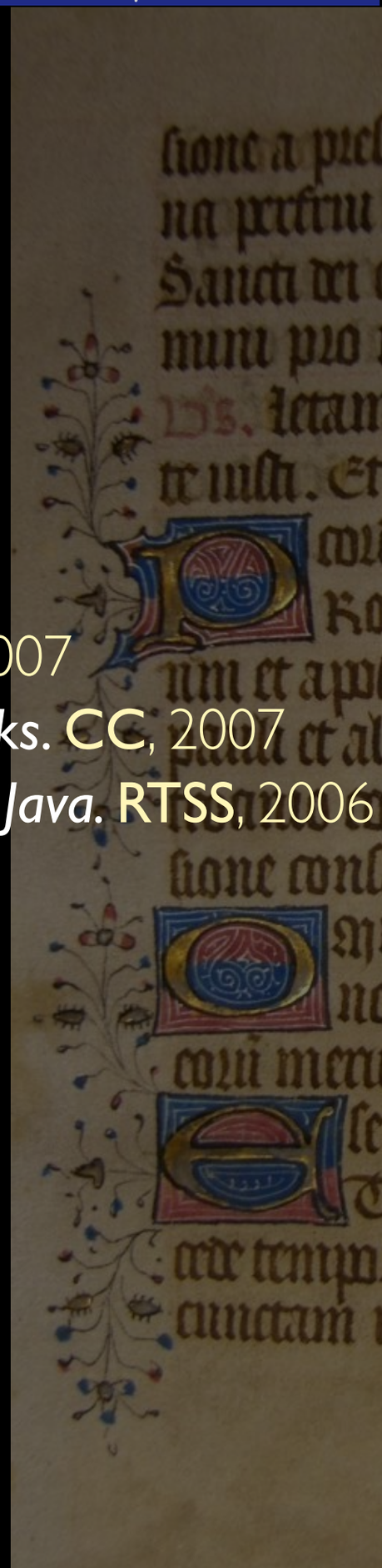

30
28
26
24
22
20
18
16
14
12
10
8
6
4
2
0
1000
1500
2000
2500
3000
3500

CD with Flexotasks (region tasks)



Paper trail

- *A Unified Restricted Thread Programming Model for Java.* LCTES, 2008
- *StreamFlex: High-throughput Stream Programming in Java.* OOPSLA, 2007
- *Garbage Collection for Safety Critical Java.* JTRES, 2007
- *Hierarchical Real-time Garbage Collection.* LCTES, 2007
- *Reflexes: Abstractions for Highly Responsive Systems.* VEE, 2007
- *Scoped Types and Aspects for Real-time Java Memory management.* RTS, 2007
- *Accurate Garbage Collection in Uncooperative Environments with Lazy Stacks.* CC, 2007
- *An Emprical Evaluation of Memory Management Alternatives for Real-time Java.* RTSS, 2006
- *Scoped Types and Aspects for Real-Time Systems.* ECOOP, 2006
- *A New Approach to Real-time Checkpointing.* VEE, 2006
- *A Real-time Java Virtual Machine for Avionics.* RTAS, 2006
- *Preemptible Atomic Regions for Real-time Java.* RTSS,2005
- *Transactional lock-free data structure for Real Time Java.* CSJP, 2004
- *Real-Time Java scoped memory: design patterns, semantics.* ISORC, 2004
- *Subtype tests in real time.* ECOOP, 2003
- *Engineering a customizable intermediate representation.* IVME, 2003



Conclusions

- Flexible tasks graph have a simple semantics and an efficient implementation
- Integration between real-time and non-real-time code allows for gradual adoption
- Implementation in a production quality virtual machine shows good performance