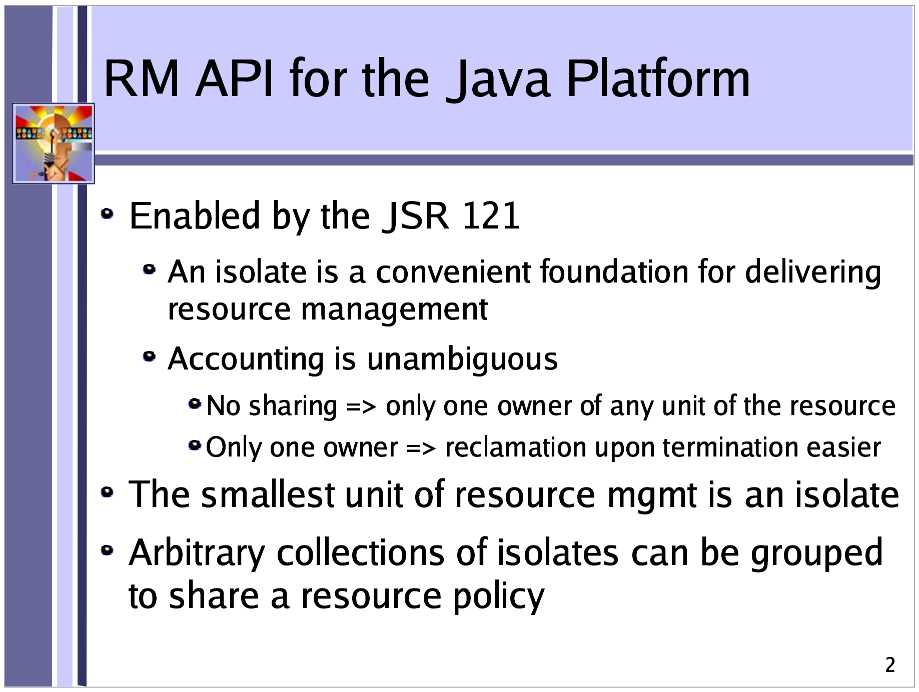
Isolate use case:
Resource Management Research

Grzegorz Czajkowski
Sun Research

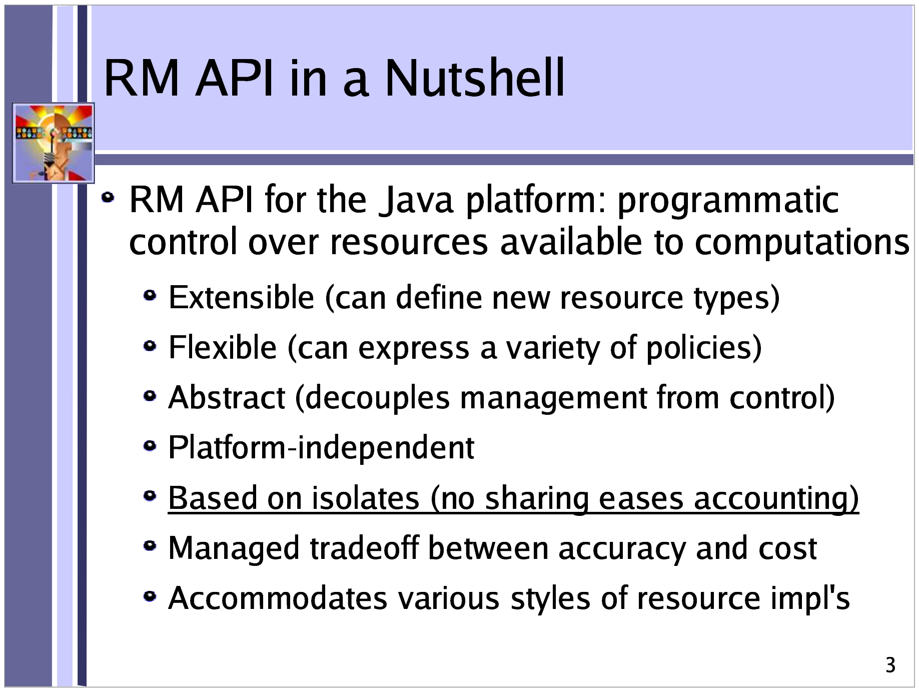


RM API for the Java Platform

- Enabled by the JSR 121
  - An isolate is a convenient foundation for delivering resource management
  - Accounting is unambiguous
    - No sharing => only one owner of any unit of the resource
    - Only one owner => reclamation upon termination easier
- The smallest unit of resource mgmt is an isolate
- Arbitrary collections of isolates can be grouped to share a resource policy

2

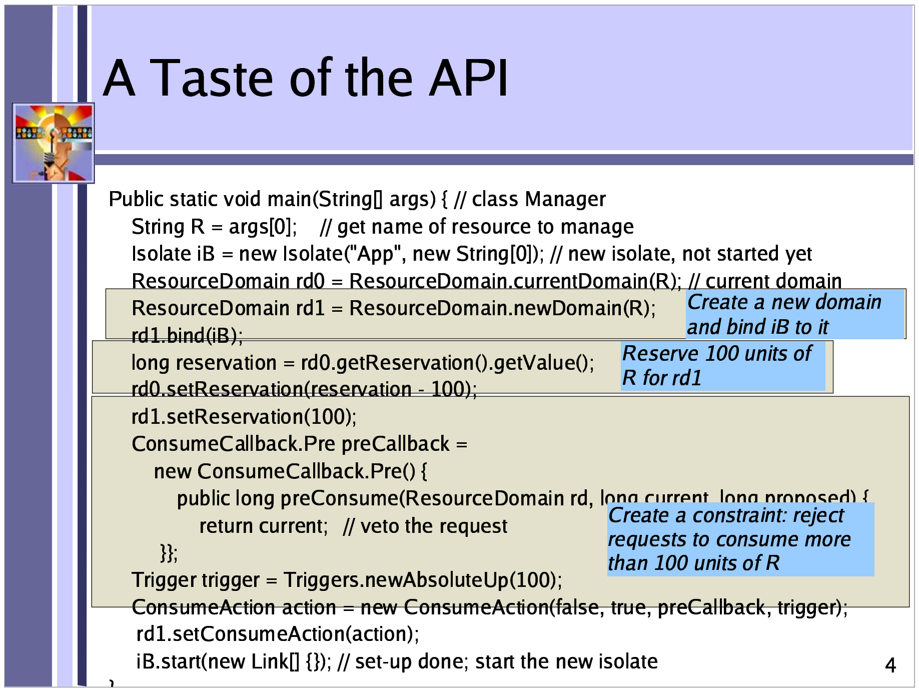
…



RM API in a Nutshell

- RM API for the Java platform: programmatic control over resources available to computations
  - Extensible (can define new resource types)
  - Flexible (can express a variety of policies)
  - Abstract (decouples management from control)
  - Platform-independent
  - <u>Based on isolates (no sharing eases accounting)</u>
  - Managed tradeoff between accuracy and cost
  - Accommodates various styles of resource impl's

…



A Taste of the API

```
Public static void main(String[] args) { // class Manager
   String R = args[0];    // get name of resource to manage
   Isolate iB = new Isolate("App", new String[0]); // new isolate, not started yet
   ResourceDomain rd0 = ResourceDomain.currentDomain(R); // current domain
   ResourceDomain rd1 = ResourceDomain.newDomain(R);
   rd1.bind(iB);
   long reservation = rd0.getReservation().getValue();
   rd0.setReservation(reservation - 100);
   rd1.setReservation(100);
   ConsumeCallback.Pre preCallback =
      new ConsumeCallback.Pre() {
         public long preConsume(ResourceDomain rd, long current, long proposed) {
            return current;   // veto the request
      }};
   Trigger trigger = Triggers.newAbsoluteUp(100);
   ConsumeAction action = new ConsumeAction(false, true, preCallback, trigger);
    rd1.setConsumeAction(action);
    iB.start(new Link[] {}); // set-up done; start the new isolate
```

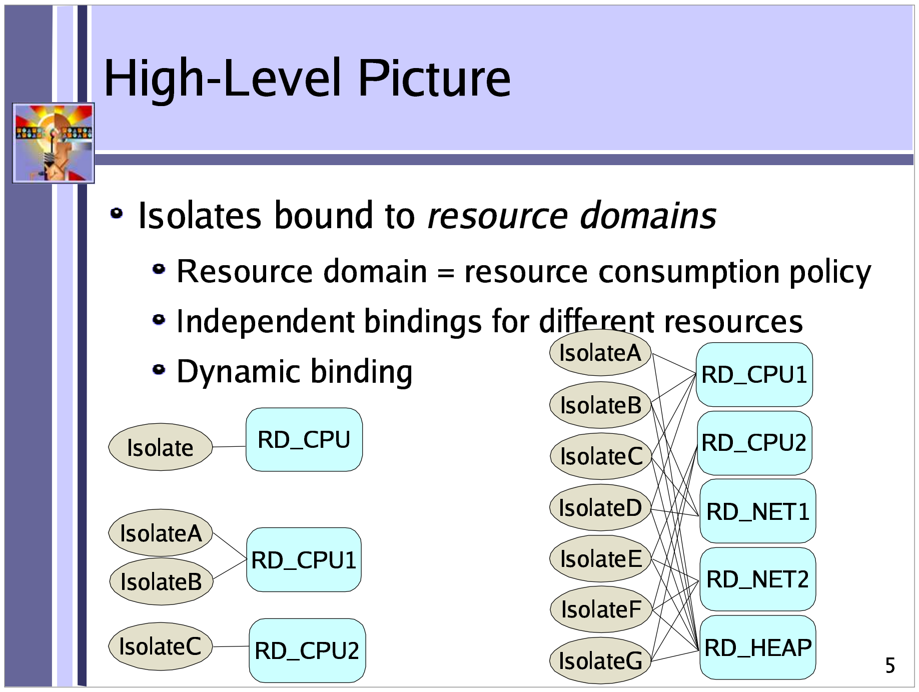
Create a new domain and bind iB to it

Reserve 100 units of R for rd1

Create a constraint: reject requests to consume more than 100 units of R

4

Privileged applications would be written this way: create other isolates and set up policies controlling resource consumption.



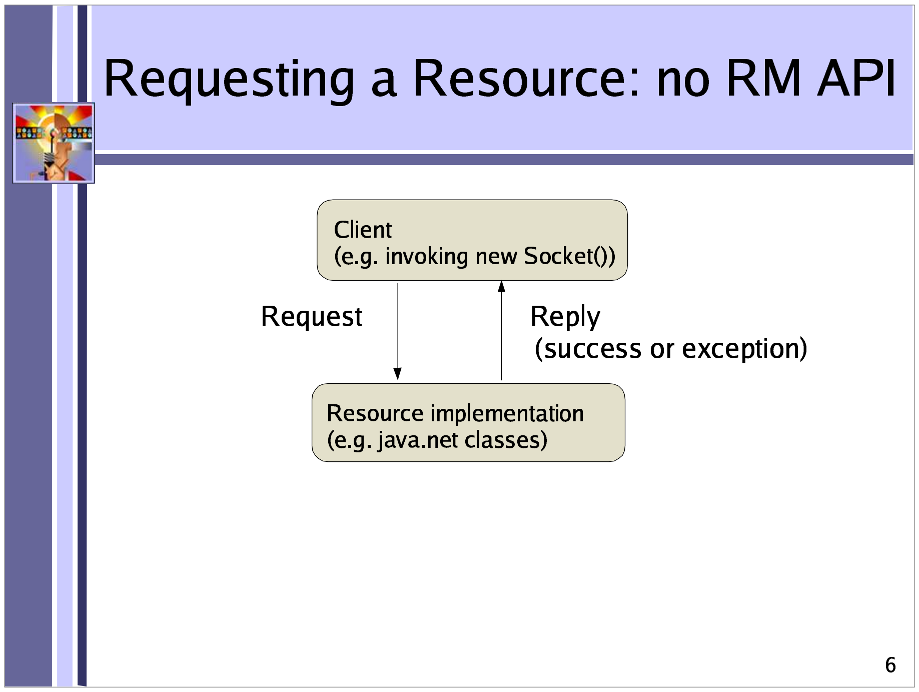
High-Level Picture

- Isolates bound to *resource domains*
  - Resource domain = resource consumption policy
  - Independent bindings for different resources
  - Dynamic binding

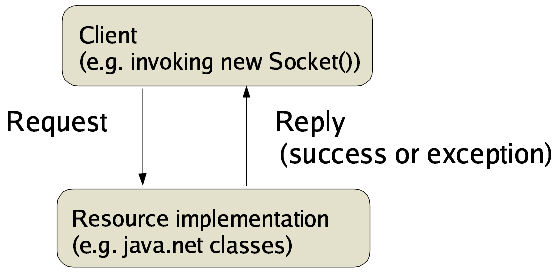
5

We'll discuss resource domains in details later; let's just say for now that they encapsulate a resource usage policy.

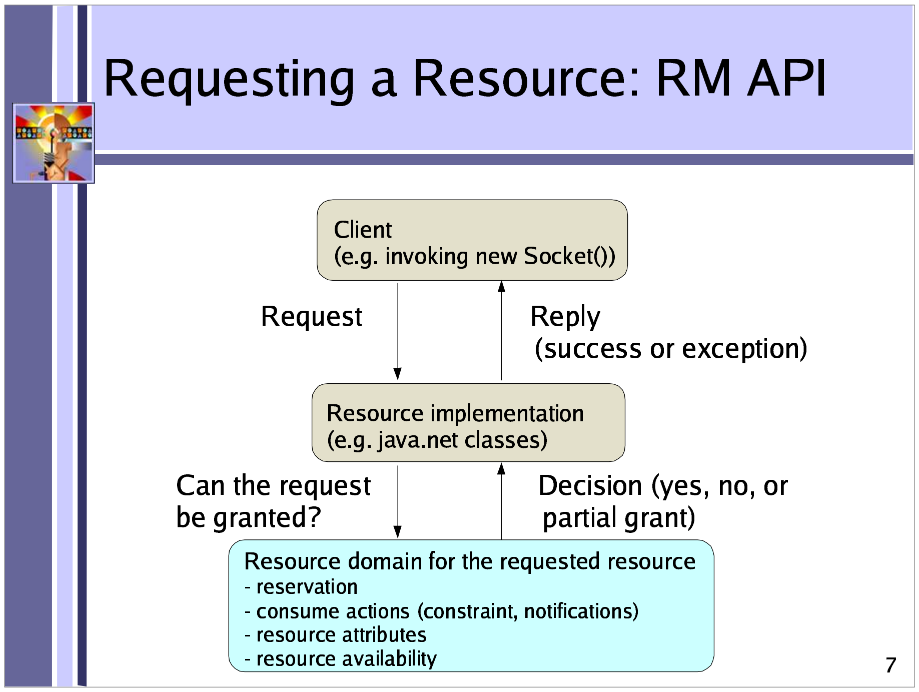
Left top – single isolate, single domain.
Left bottom – three isolates, two domains.
Right – seven isolates, five domains for three resources.
A, B, and C bound to the same domain for CPU (RD_CPU1), so they share the same policy; independent of what G's policy wrt CPU is. A and B share the same domain for CPU, but not for Net. In fact, A is not bound to any domain for Net.



Requesting a Resource: no RM API



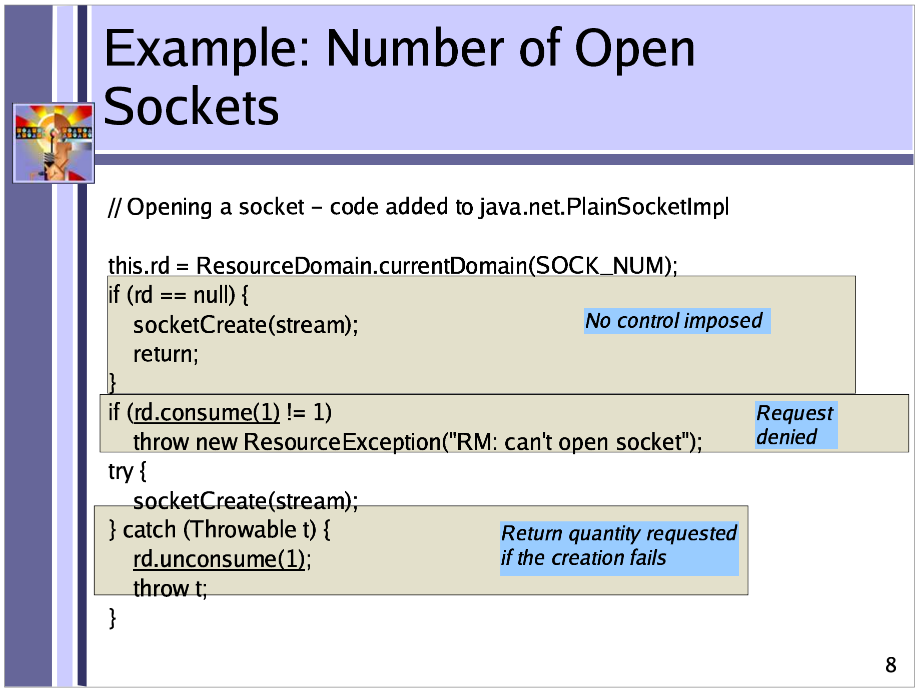
The picture is the same regardless of whether the request is mediated by the JDK classes, the runtime (heap), OS (CPU), middleware
(JDBC connections), etc.



Requesting a Resource: RM API

Client
(e.g. invoking new Socket())
Request
Reply
(success or exception)
Resource implementation
(e.g. java.net classes)
Can the request be granted?
Decision (yes, no, or partial grant)
Resource domain for the requested resource
- reservation
- consume actions (constraint, notifications)
- resource attributes
- resource availability

...



Example: Number of Open Sockets

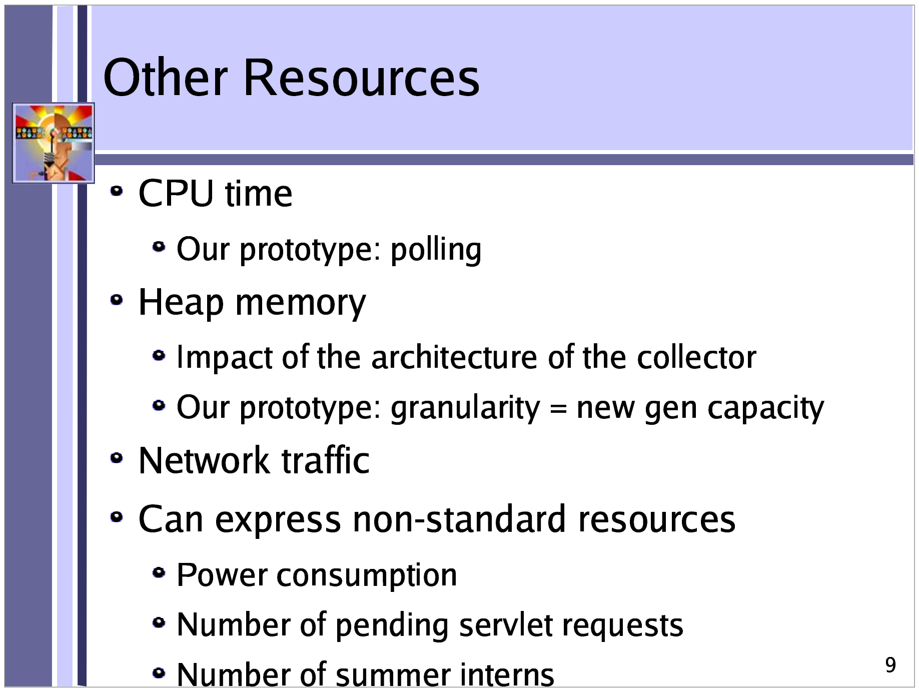
```
// Opening a socket – code added to java.net.PlainSocketImpl

this.rd = ResourceDomain.currentDomain(SOCK_NUM);
if (rd == null) {
   socketCreate(stream);
   return;
}
if (rd.consume(1) != 1)
   throw new ResourceException("RM: can't open socket");
try {
   socketCreate(stream);
} catch (Throwable t) {
   rd.unconsume(1);
   throw t;
}
```

No control imposed

Request denied

Return quantity requested if the creation fails

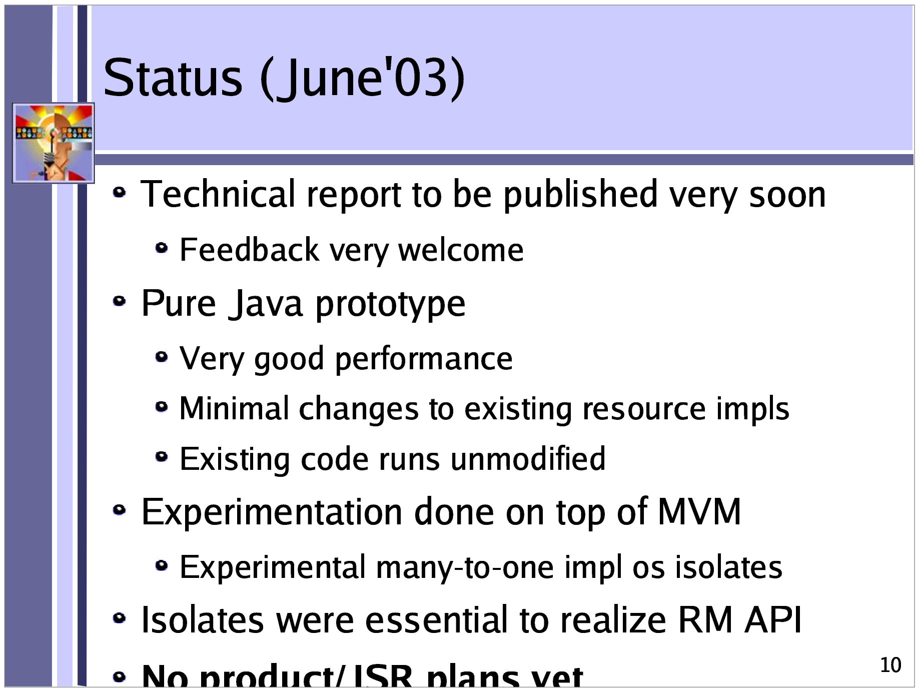
Note that the client's API to using the resource hasn't changed; all RM API work is done under the hood.



Other Resources

- CPU time
  - Our prototype: polling
- Heap memory
  - Impact of the architecture of the collector
  - Our prototype: granularity = new gen capacity
- Network traffic
- Can express non-standard resources
  - Power consumption
  - Number of pending servlet requests
  - Number of summer interns

Note: first time we've mentioned “implicit” resources; main point is that CPU is a bit different than the other resources we've mentioned (no API to request units of).



Status (June'03)

- Technical report to be published very soon
  - Feedback very welcome
- Pure Java prototype
  - Very good performance
  - Minimal changes to existing resource impls
  - Existing code runs unmodified
- Experimentation done on top of MVM
  - Experimental many-to-one impl os isolates
- Isolates were essential to realize RM API
- **No product/JSR plans yet**

...