Improving Linux resource control using CKRM

Rik Van Riel
Red Hat Inc.
Hubertus Franke, Shailabh Nagar
IBM T.J. Watson Research Center
Chandra Seetharaman, Vivek Kashyap
IBM Linux Technology Center
Haoqiang Zheng
Columbia University

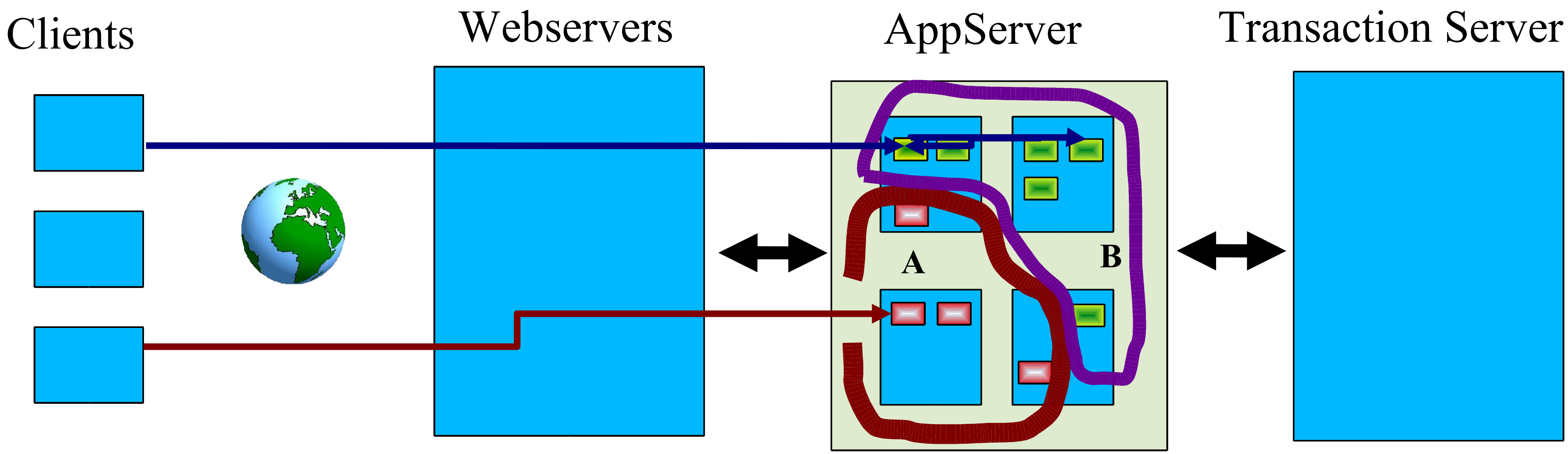
Outline

- Recap
  - Motivation
  - Architecture
- New since 2003
  - Core redesign
  - Resource Control Filesystem
  - Hierarchies
  - Schedulers
- Future Work

Workload Management Requirements

- Modified resource principal is a group of processes (class)
  - User-defined
  - Dynamic
  - Visible to OS kernel
  - Support for automatic classification of new processes
- Privileged user defines class entitlements/shares
  - Generally CPU, virtual/real memory
  - I/O, network less common but useful
- Role of OS Kernel
  - enforce shares
  - monitor, export class usage
- State of art for high-end Unixes and Windows (?)
  - HP-PRM/WLM, AIX WLM, Solaris, Tru64

Usage 1: Enterprise Servers



- Class determined by
  - who, what, where
  - any workload attribute (not all traditionally visible to kernel)
- Different QoS for each class:
  - Response time, bandwidth
- Class boundaries change rapidly

- Example Stock trading:
  - **Gold**: high volume trader initiating a transaction
  - **Silver**: all other stock trading
  - **Bronze**: mutual fund transactions quotes

Usage 2: Shell server

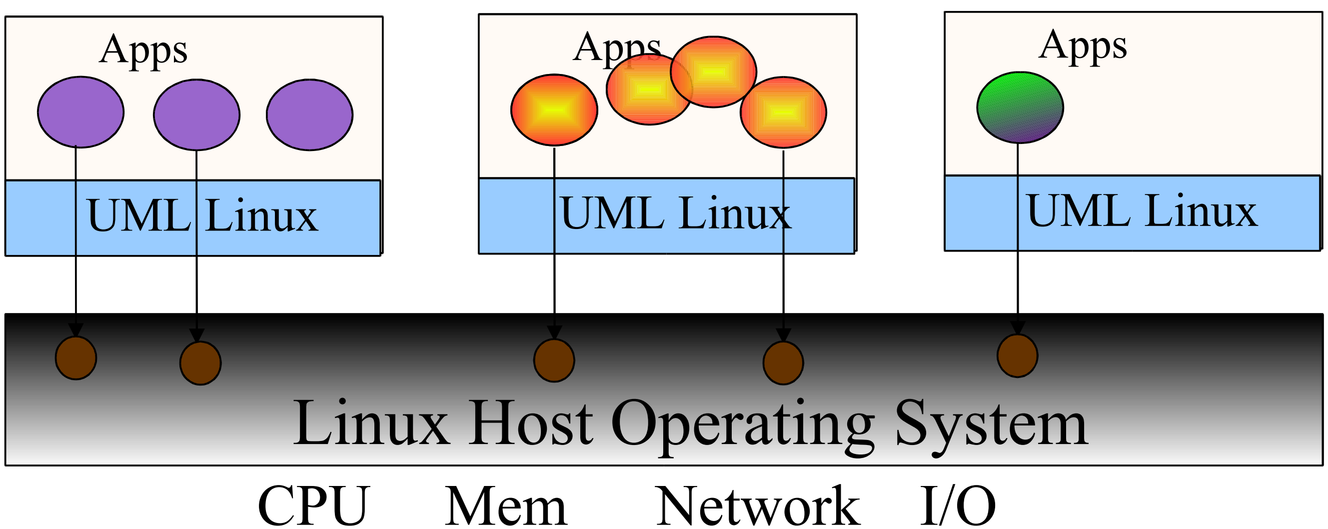
- University shell server with different users
  - Students: Low
  - Staff/postdocs : High
  - Accounts/Backup: Batch/Background
  - OS Class Projects, Physics simulations
- Resource shares set from PAM module at login
- Email processing
  - Charge to user being processed
  - Automatic classification based on uid/app name

Usage 3: Desktop

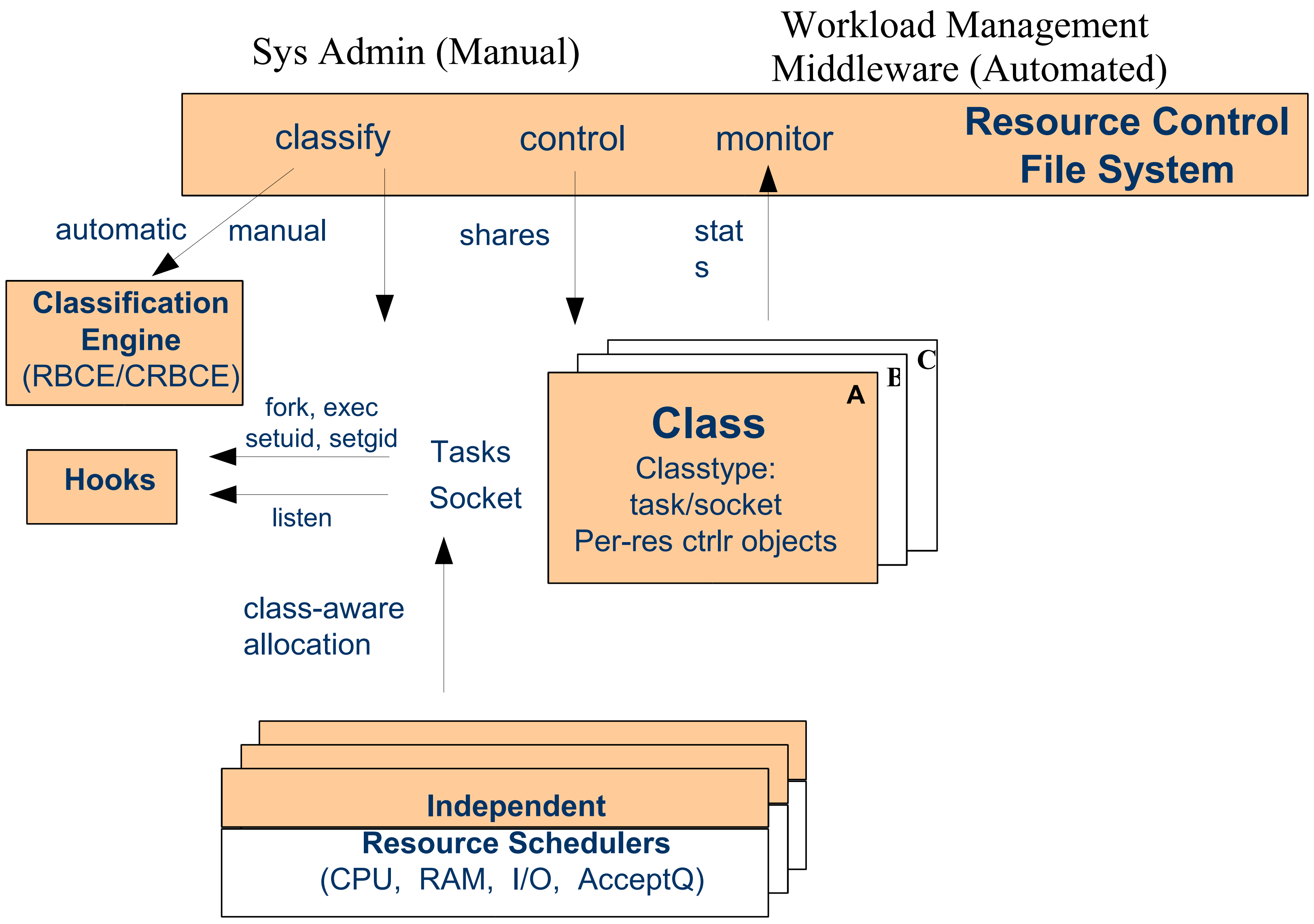
- Protect apps from each other
  - X
  - Xmms
  - Shell
  - Mozilla
- User level control over app-class shares
  - Done automatically by user's GUI
- Requirements
  - Simple interface
  - More tolerance for share enforcement inaccuracy
  - Little need for monitoring

Usage 4: UML/vserver Virtual Hosting

- Virtual Hosting using UML/vserver, apps run as processes under host system together with guest OS
- Every system resource needs to be regulated
- Service guarantees for each UML instance



CKRM Architecture



CKRM Main Components

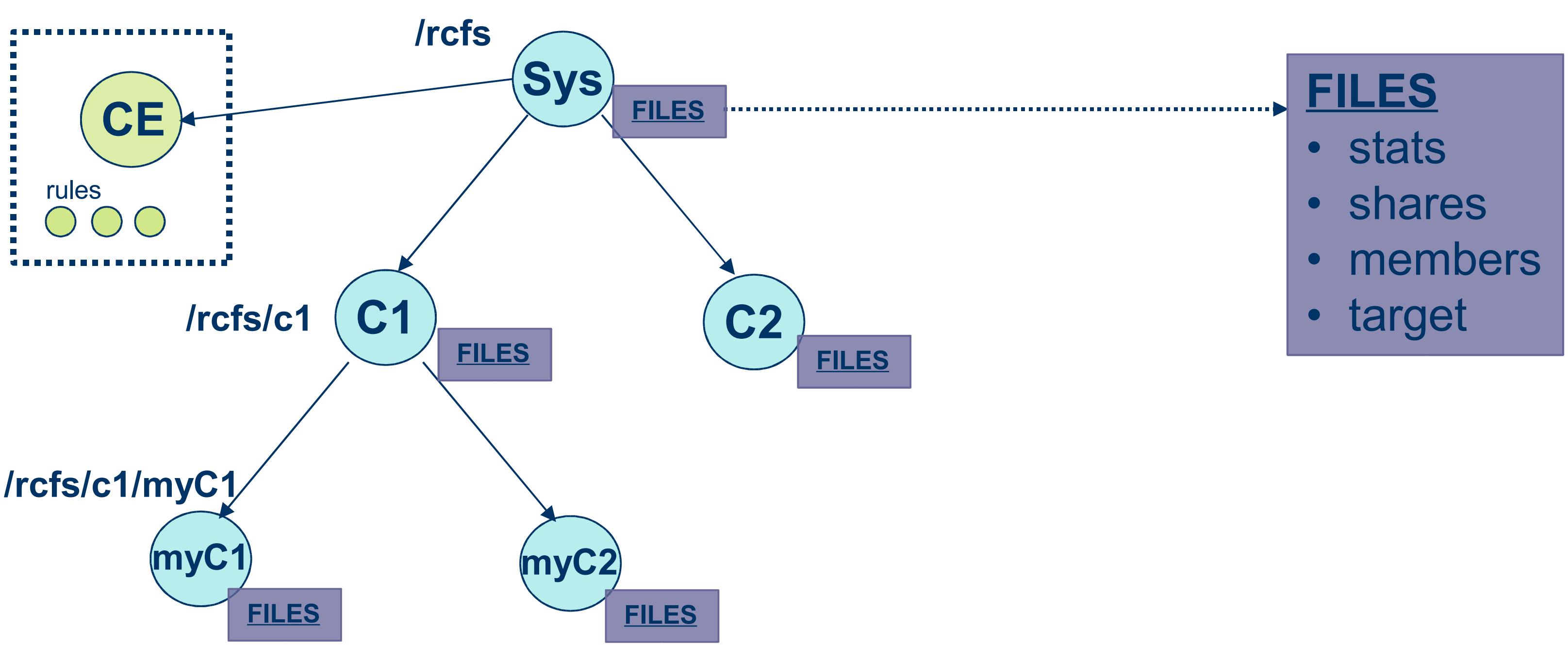
- Classtypes
  - Define kernel resource object to be grouped
  - Independent dimension for all other components
- Classes
  - Hierarchical grouping of kernel resource objects
  - Associated shares of managed resources
- Classification Engine
  - Policy-driven assignment of kernel objects to classes
  - Notifications of kernel events to user level
- Resource Control Filesystem
  - User API to CKRM
- Resource Controllers
  - Class-aware enhancements to existing Linux schedulers
  - Physical resources (CPU, Physical Memory, Disk I/O, Socket connections)
  - Virtual resources (number of tasks)

Modular design

- Classtypes can be independently included
  - One or more of task_classes, socket_classes
- Classification Engine completely optional
  - manual classification always available
- Resource Control Filesystem interface
  - replaceable with system call interface if necessary
  - Filesystem implemented as a loadable module
- Completely independent controllers
  - Independent data structures, kernel configuration
  - Independent in-kernel operation
    - May not be desirable in long term
    - Coupling possible through user-level WLM components
  - Decouples acceptance of scheduler patches in mainline kernel

User API (RCFS) Overview

- Directory = Class
  - Filesystem hierarchy ~= Class Hierarchy and namespace
  - /path/to/class represents the unique class name
- Virtual files = Class attributes
  - Created automatically
- Standard filesytem operations = CKRM functional API
  - mkdir/rmdir = create/delete class
  - read/write virtual file = get/set attributes (shares, stats, config,classification rules,…..)
  - File permissions/ownership used to restrict/delegate access to operations

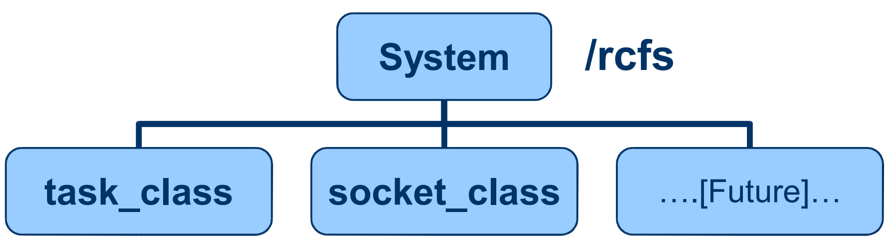


CKRM Core Overview

- Classtypes
  - Define kernel object being grouped
- Classes
  - Group of kernel objects
- Kernel hooks
  - CKRM code executed at significant kernel events such as fork, exec, setuid, setgid, listen

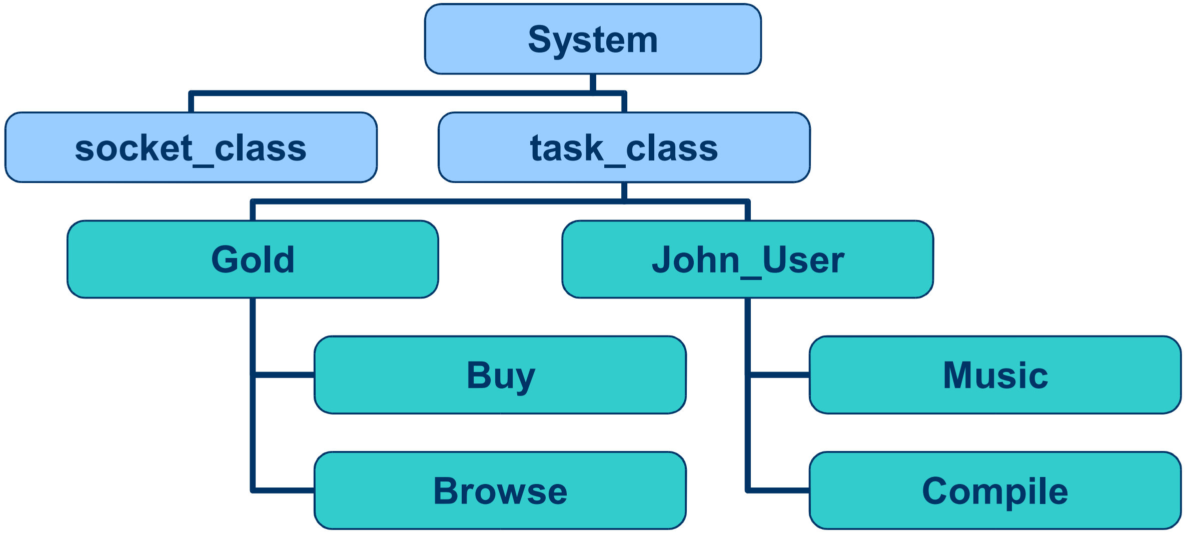
Classtypes

- Define kernel object being grouped
  - Currently tasks (task_class), listening sockets (socket_class)
- Independent dimension for other components
- Each classtype has an associated
  - Hierarchy of classes
  - Set of resource controllers
    - Mutually exclusive across classtypes
  - Classification engine rules
  - Directory in filesystem
    - Automatically created when classtype configured



Classes

- Group of kernel objects
- Associated shares (lower and upper bounds)
- Hierarchical to allow further subdivision of resources
  - Top Level shares controlled by privileged user, lower levels can be delegated
- Manifest as directories in /rcfs
  - Filesystem hierarchy under classtype mirrors class hierarchy



Classification

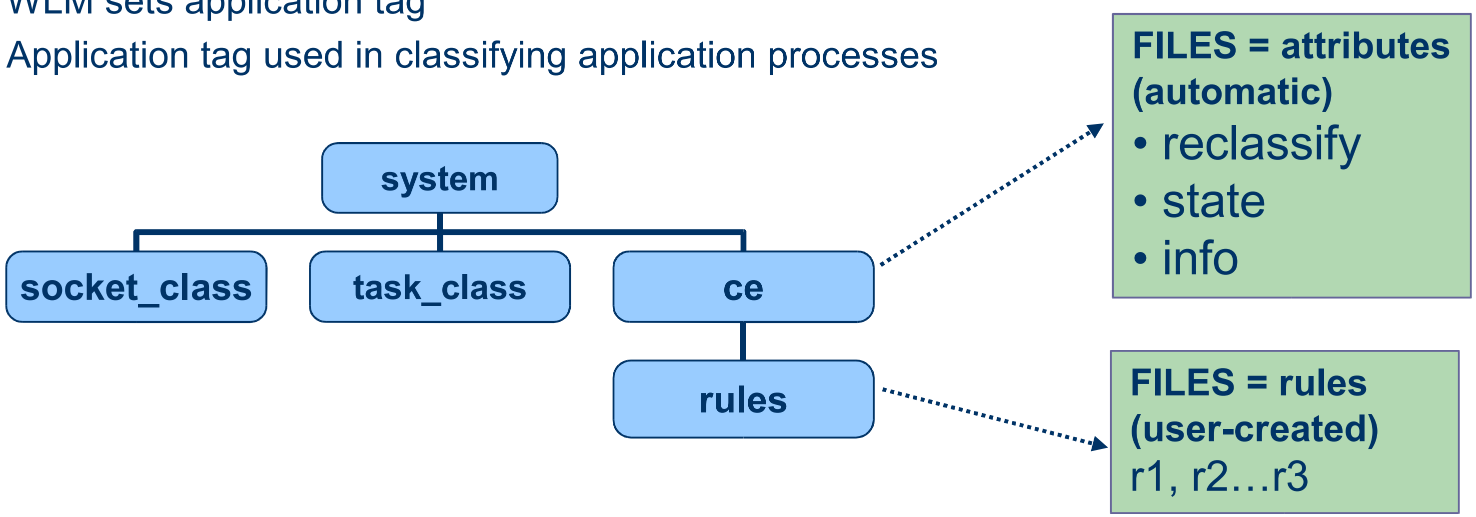
- All kernel objects managed by a classtype need to be in some class
  - Default class always present for each classtype
  - Objects inherit parent's classification unless manual/automatic classification done
- Manual classification
  - echo "<object identifier>" > /path/to/class/target
  - echo "1324" > /rcfs/taskclass/tc1/target
    - Classifies task with pid=1324 into tc1
  - echo "127.0.0.1/80" > /rcfs/socket_class/nc1/target
    - Classifes port 80 of ipv4 address into nc1
- Classification Engine (CE) assists in automatic classification
- Automatic classification points
  - Conceptually any point where the kernel object's attribute changes
    - CKRM implements a useful subset which can be extended as need arises
  - Tasks: fork(), exec(), setuid(), setgid()
  - Sockets (for connection control): listen()
- Manual classification overrides CE, if latter present, until automatic classification explicitly reenabled
  - re-enablement by writing object id to /rcfs/ce/reclassify

Classification Engines

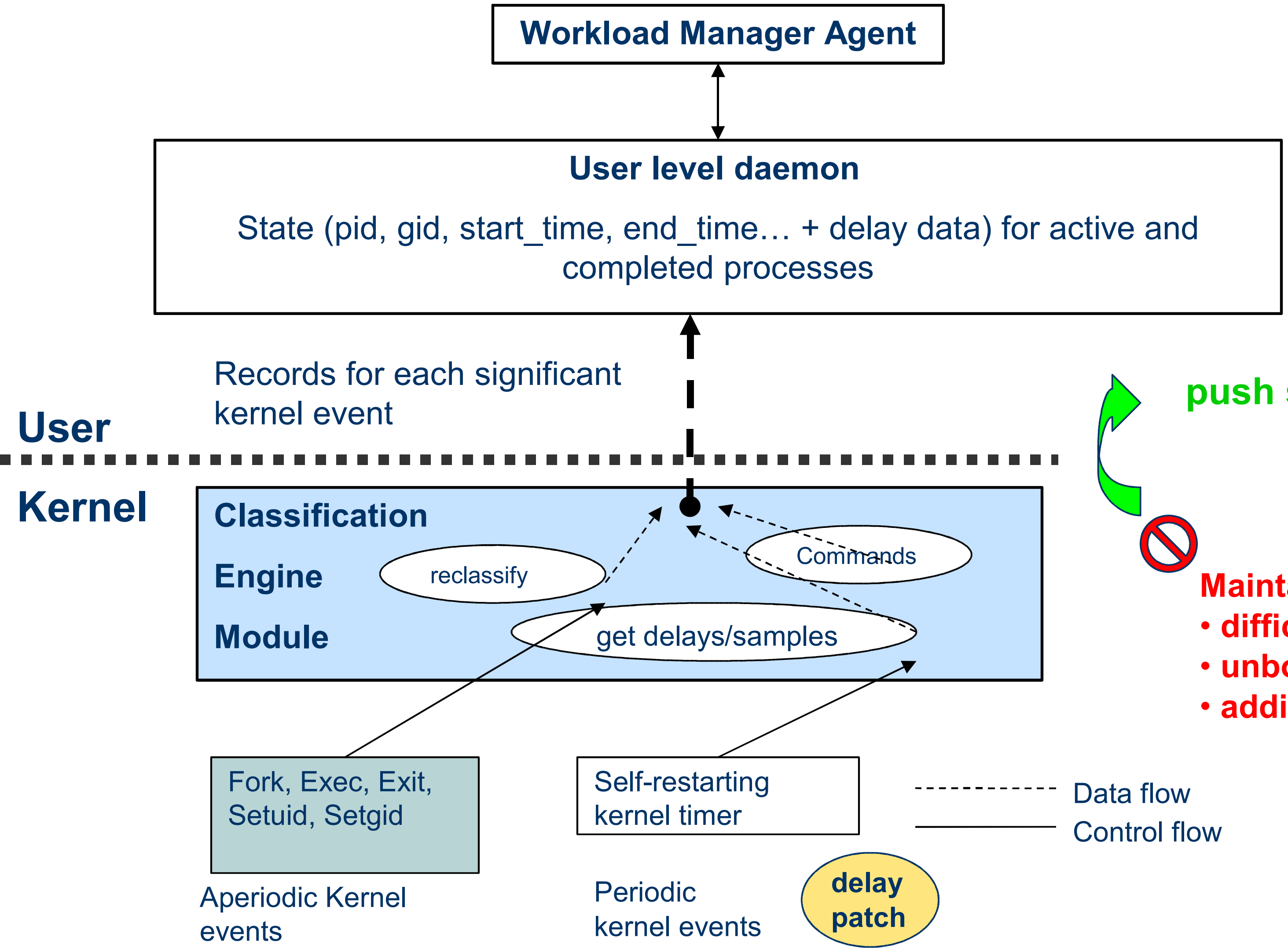
- Optional module for CKRM operation
- Can be custom-built outside CKRM project
  - Only needs to adhere to CKRM's "return classification" interface
  - Module's output is a recommendation that may be rejected by CKRM core
- CKRM provides two rule-based classification engines
- RBCE (Rule-Based Classification Engine)
  - Flexible classification using rule matching
  - Expected to meet manual system administration needs
- CRBCE (enhancements to RBCE)
  - Supplies user space with data useful for goal-oriented workload management
  - Expected to meet WLM middleware needs

RBCE

- Classification rule
  - { [(attr,value)]+ -> class }
  - Attributes of task: uid, gid, executable name, application tag
  - Created by echoing terms to /rcfs/ce/rules/<rulename>
- Classification rules ordered
  - Matched in order at classification point by CE module
  - “Catch-all” rule advisable for no-match case
- Application tags
  - Additional flexibility for grouping based on application specific criteria
    - Application informs WLM of transaction start
    - WLM sets application tag
    - Application tag used in classifying application processes



CRBCE and Resource monitoring



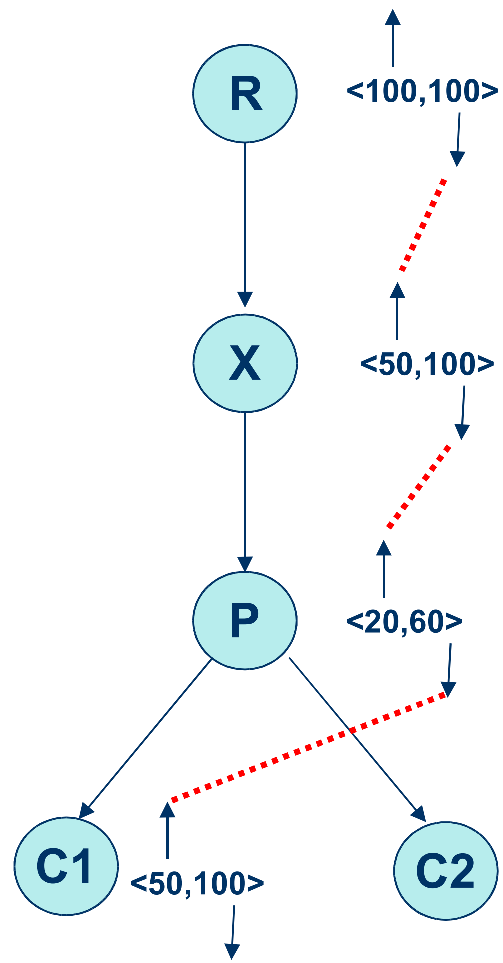
push state to user space

Maintaining state in kernel

- difficult to do ..
- unbound in requirements
- additional complexity

Shares

- Distinguish for each resource
  - limit (upper bound)
  - guarantee (lower bound)
    - No oversubscription, no starvation !
- Parent provides a base (think 100%)
  - max_limit, total_guarantee
- Child gets a relative fraction
  - limit < max_limit(parent)
  - guarantee/total_guarantee(parent)
- Actual Shares received
  - determined by path…
- Changing shares
  - Possible without touching siblings' values



50/60 * 20/100 * 50/100 = 8.3%

echo "res=cpu, guarantee=50, total_guarantee=100" \
> /rcfs/taskclass/R/X/shares

Stats

- cat /path/to/class/stats
- Multiple lines from each active controller
  - Prefer one statistic per line a la vmstat
  - Data, interpretation is controller specific
- Leaf nodes updated accurately
  - Parents updated lazily

Resource Controllers

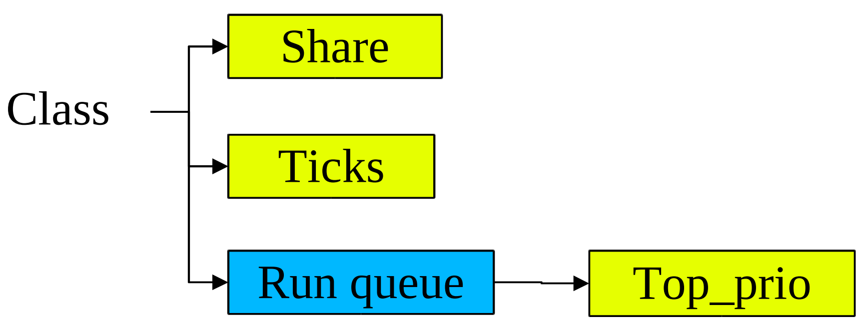
- Each task associated with a class
  - Task resource requests queued by class
  - Explicit or implicit per-class queues
- Control
  - Share based preferential service to class queues
- Monitoring
  - Additionally maintain class statistics
- Network classes
  - Not tied to tasks but ipaddress:port
  - controlled similarly

Resource Controller Status

- Controllers provided
  - CPU
  - Physical Memory (preliminary port available)
  - Disk I/O bandwidth (CFQv2 based port expected 8/04)
  - Inbound Socket Connections
- Virtual resource controller for number of tasks
  - template
  - Prevent fork bombs
- Other controllers can be developed as needed

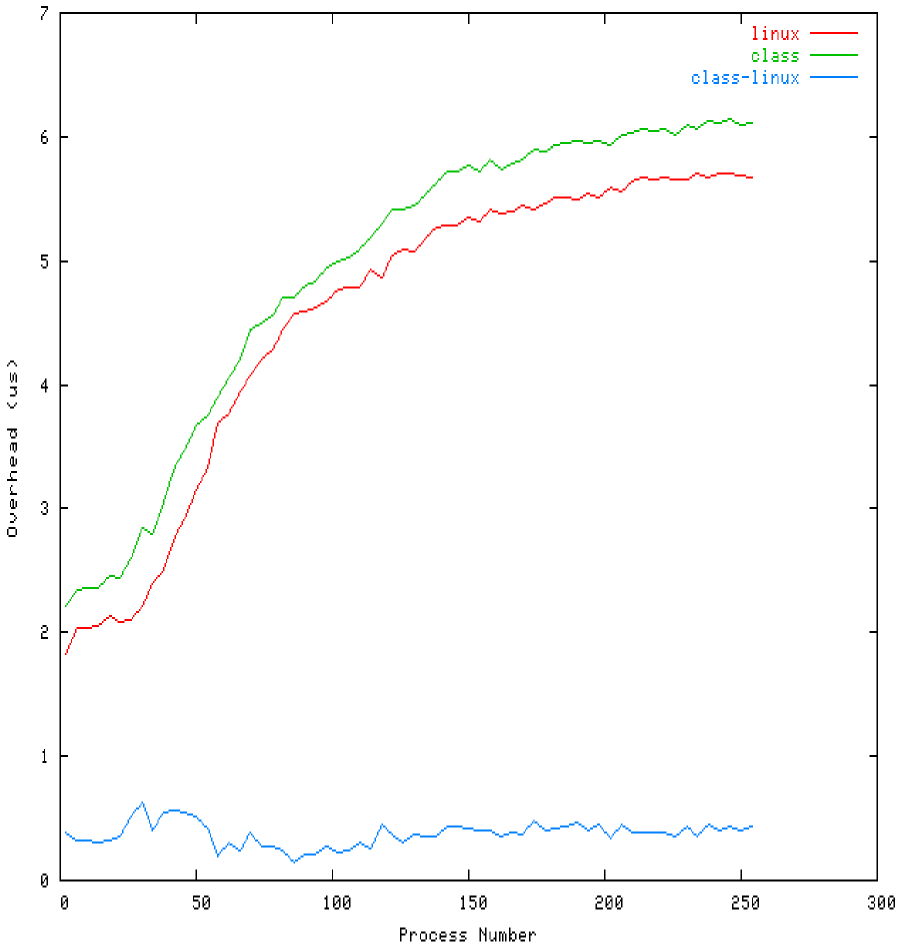
CKRM CPU Scheduler

- Each class has its own runqueue
- Minimal changes to the existing scheduler:
  - same runqueue structure
  - same way to calculate time_slice, sleep_average and prio, etc.
  - same O(1) behavior within class
- `get_next_task()` now makes **2** decision
  - First selects the next class to run
  - Then, within that class select the top priority task just as today



- **Class Selection**:
  - Based on accumulative normalized time per class
    - **ecp(C) = Σ ticks(C)/share(C)**
    - monotonic increasing function
  - Select class C with **min(ecp(C))**
  - Consider finite sliding window CWIN [min..min+WS]
  - min=min(ecp(C)); WS ~ 128,256
  - When a class is reactivated (task is rescheduled)
    - **`if`** (min <= ecp(C) < min+WS)
      `then` insert C at WIN[ecp(C)]
      `else` insert C at WIN[min].
  - Provides fairness (shares) only
- **Urgency** (Interactivity)
  - **ecp(C) = (Σticks(C)/share(C)) * scale + top_prio**
  - High priority in class gives a short term boost
- **Scheduler maintains O(1) characteristics**

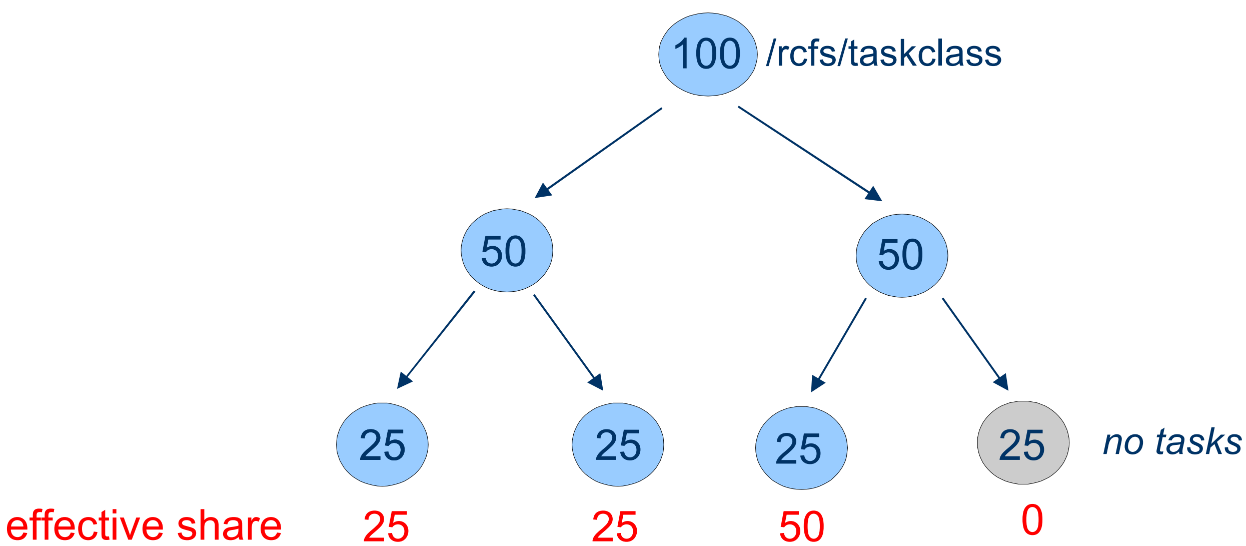
Scheduling Overhead



- Measured using Lmbench
  - lat_ctx -s 0 $N,   N=(2..256)
- Scalability: the overhead of Class Fair Scheduler increases at about the same pace as Linux 2.5 Scheduler
- The static overhead (class – linux) varies from 0.14us to 0.63us during the measurement
- Since class selection is O(1), i.e. Independent of #classes, there are no scalability concerns with #classes
- Code optimization might further reduce the static overhead

Managing hierarchies

- Traversing hierarchies costly
- Lazy monitoring and control
  - Update parent usage
  - Redistribute effective share
  - Kernel thread or user space
  - Reusable for different controllers



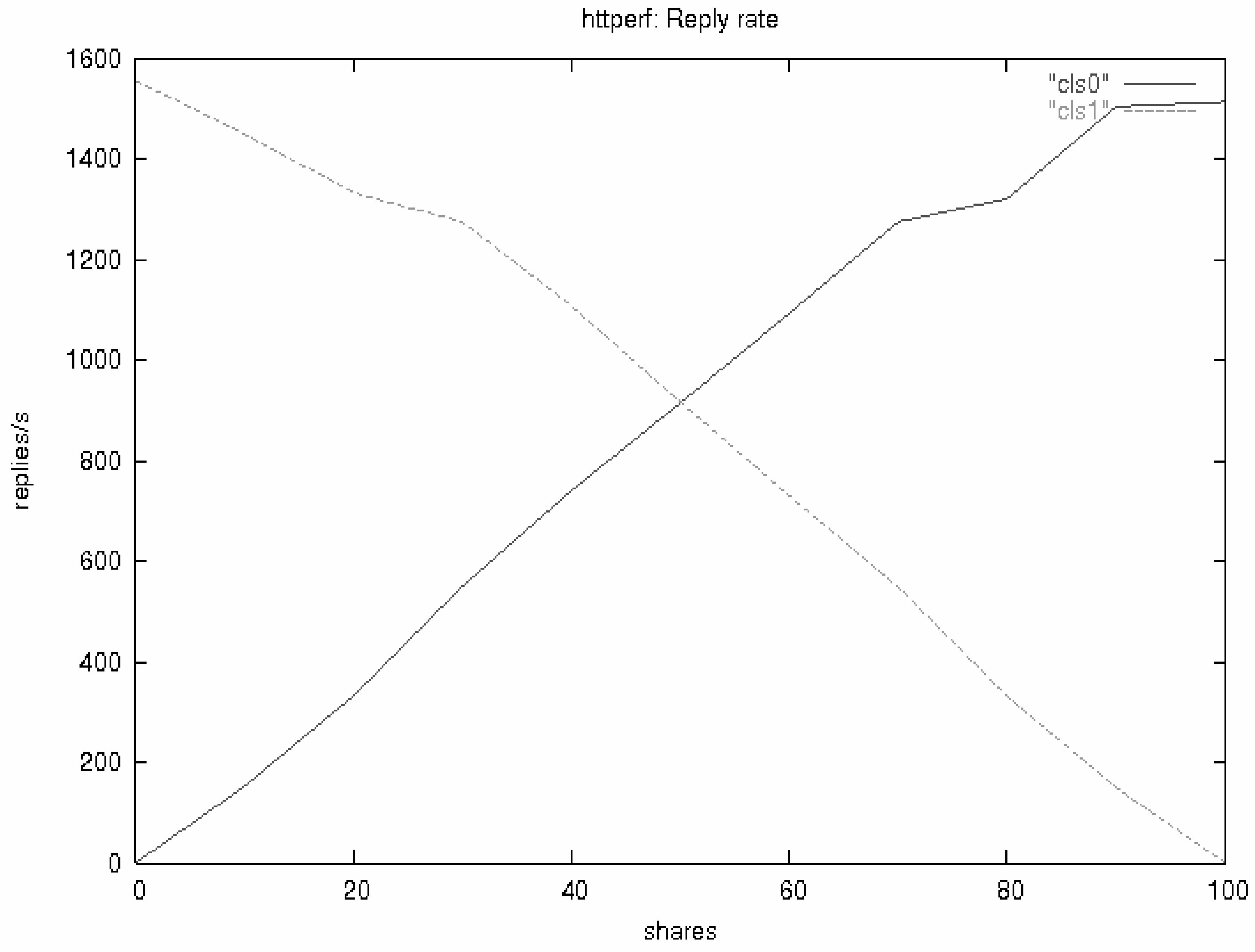
Inbound Connection Control

- Using Accept Queue classes
- Classify using (local, remote) * (IP, port)
  - Iptables rules to MARK SYN packets
- Split single accept queue into multiple queues
  - Assign shares to classes/queues
  - Use weighted round robin to accept packets
- Inbound connections accepted in proportion to shares assigned
  - Response time proportionally reduced
- Drawbacks
  - Classification hard in presence of proxies and multiple classes on same remote host

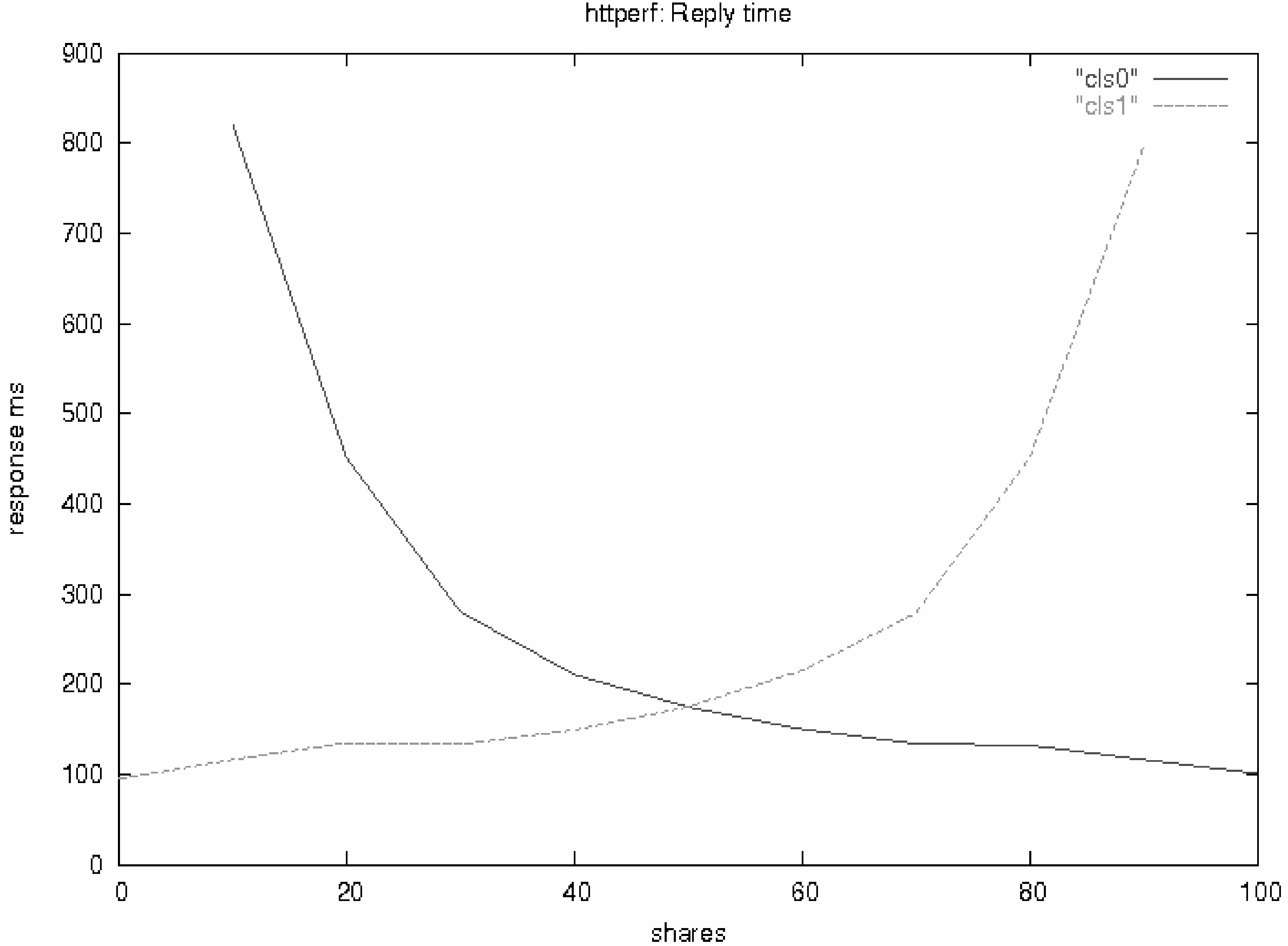
AcceptQ Experimental Setup

- Server
  - Httpd (apache) webserver with default config
  - ckrm with 'ckrm_listenaq' controller
- Two clientsrunning httperf
- Use iptables on the server to assign MARK values to connection requests from the client machines. The MARK values are assigned based on the client machines' IP addresses
- httperf run multiple times on each client against the server with each run corresponding to a different share setting for the clients.
  - default class = 90, class 1 = 10
  - default class = 40, class 1 = 60

AcceptQ: Reply Rate



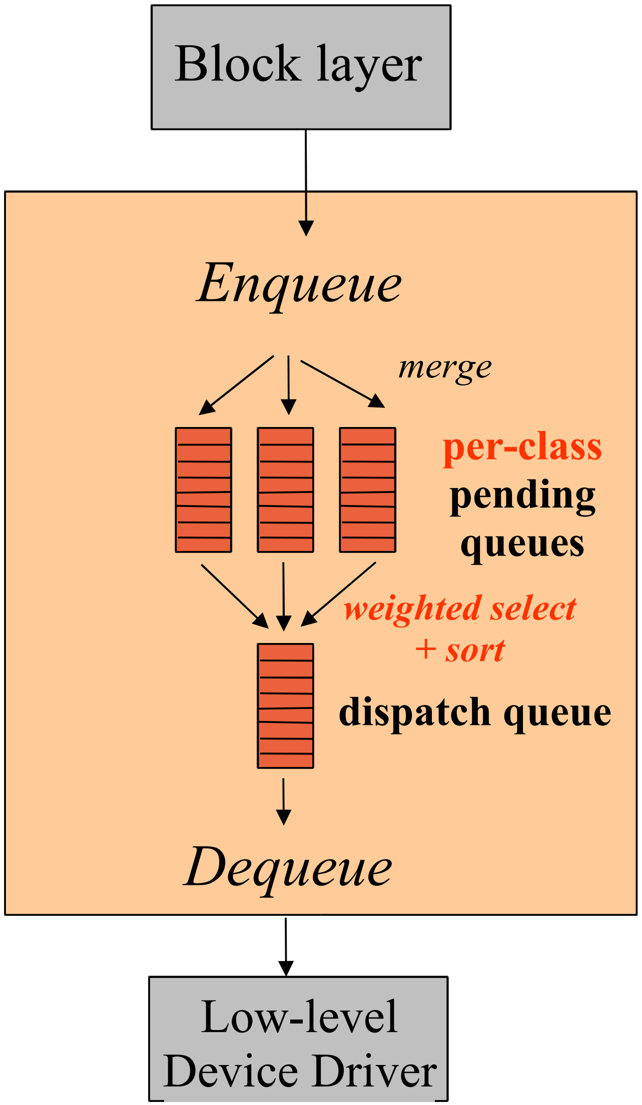
AcceptQ: Response Time



CKRM Memory Control

- Share is #maximum physical pages used per class
  - hard/soft, min/max variants also possible
- Only control page reclaimation
  - classes can exceed shares if no memory pressure
- No distinction between over-share classes
  - reclaim as many pages as needed by shrink_cache()
- Use global active/inactive lists
  - maintains global LRU order
  - overhead of repeated scans of under-share pages
- Experimenting with alternate schemes

CKRM I/O Scheduler



- CFQ v2 variant
  - Kernel enforcement of non-hierarchical shares
  - User-level, lazy enforcement of hierarchical shares
- Future experimental work
  - Add anticipation
    - Wait and service next request from same task until class share exceeds high-water mark
  - Add deadlines
    - Sort/fifo lists for each class

Future Work

- Complete port of remaining schedulers to RCFS API
  - PlanetLab vserver + CKRM prototype
- Testing, optimization of hierarchical control
- Investigate suggestions from Kernel Summit'04
  - Separate per-controller classes
    - Visible to user ?
  - Reuse kernel data structures like struct user ?
- Explore merger of monitoring with related projects
  - ELSA/CSA

Class-based Kernel Resource Management

http://ckrm.sf.net

Rik Van Riel
Red Hat Inc.
Hubertus Franke, Shailabh Nagar
IBM T.J. Watson Research Center
Chandra Seetharaman, Vivek Kashyap
IBM Linux Technology Center
Haoqiang Zheng
Columbia University