The Multi-server Emulation System

Daniel P. Julin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
dpj@cs.cmu.edu

Mach Emulators

- Single-server BSD 4.3
- Single-server BSD 4.4
- Single-server OSF/1
- Single-server SVR4
- POE
- VMS
- DOS
- MacMach
- GNU
- Mach/MC
- OS/2
- Trusted Mach
- CMU Multi-server

Major Goals

modularity

- multiple system configurations - specialization
- third-party servers
- simpler maintenance

clean, extensible design

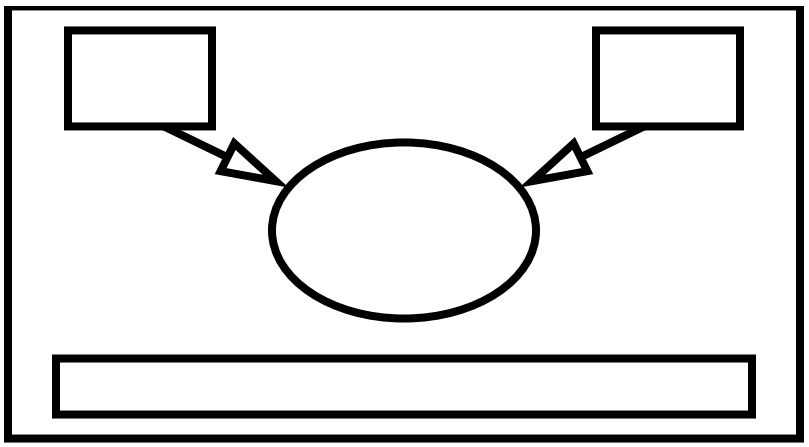
- add functionality
- integrate modified functionality
- support for multiple emulations / OS personalities

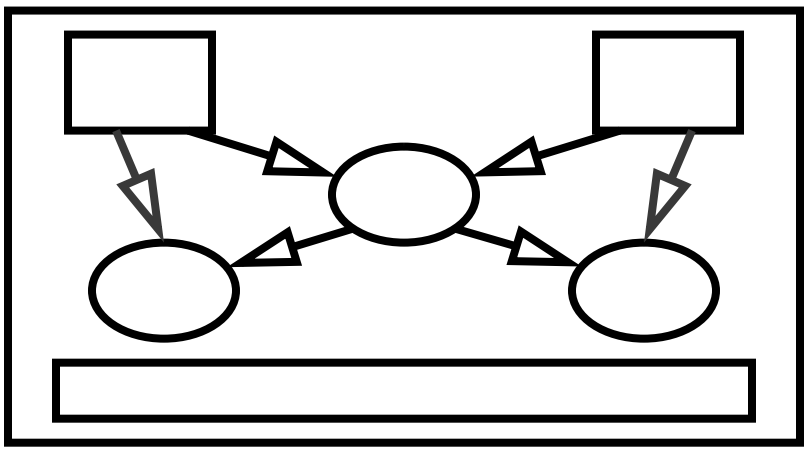
security

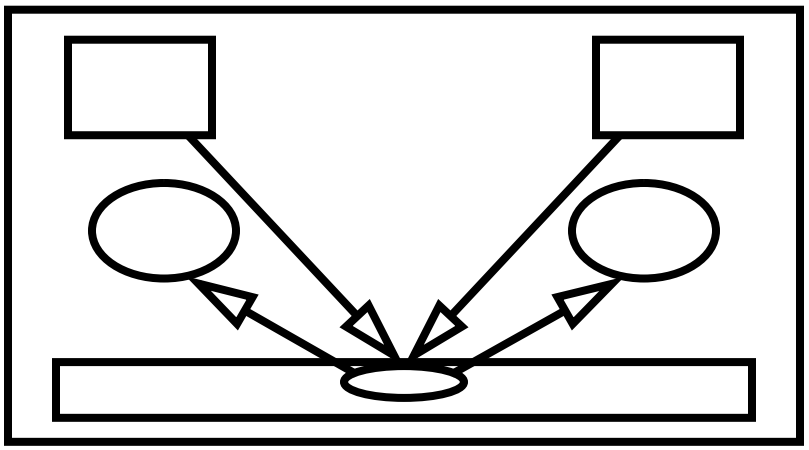
practical, useful system

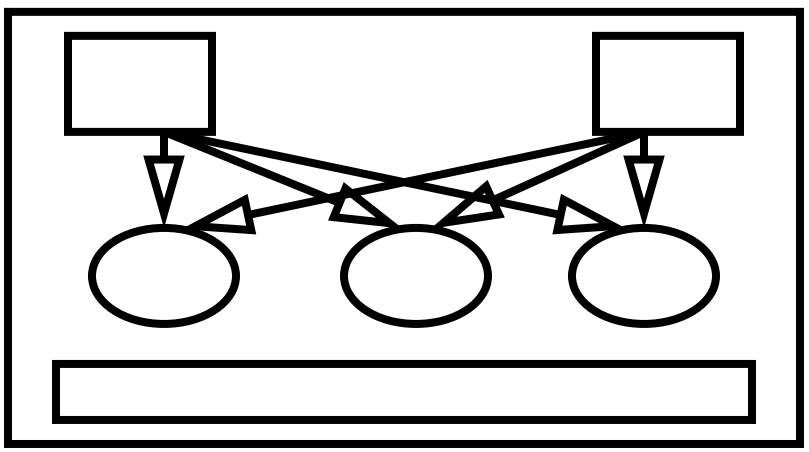
- performance
- sensible licensing

Possible Architectures









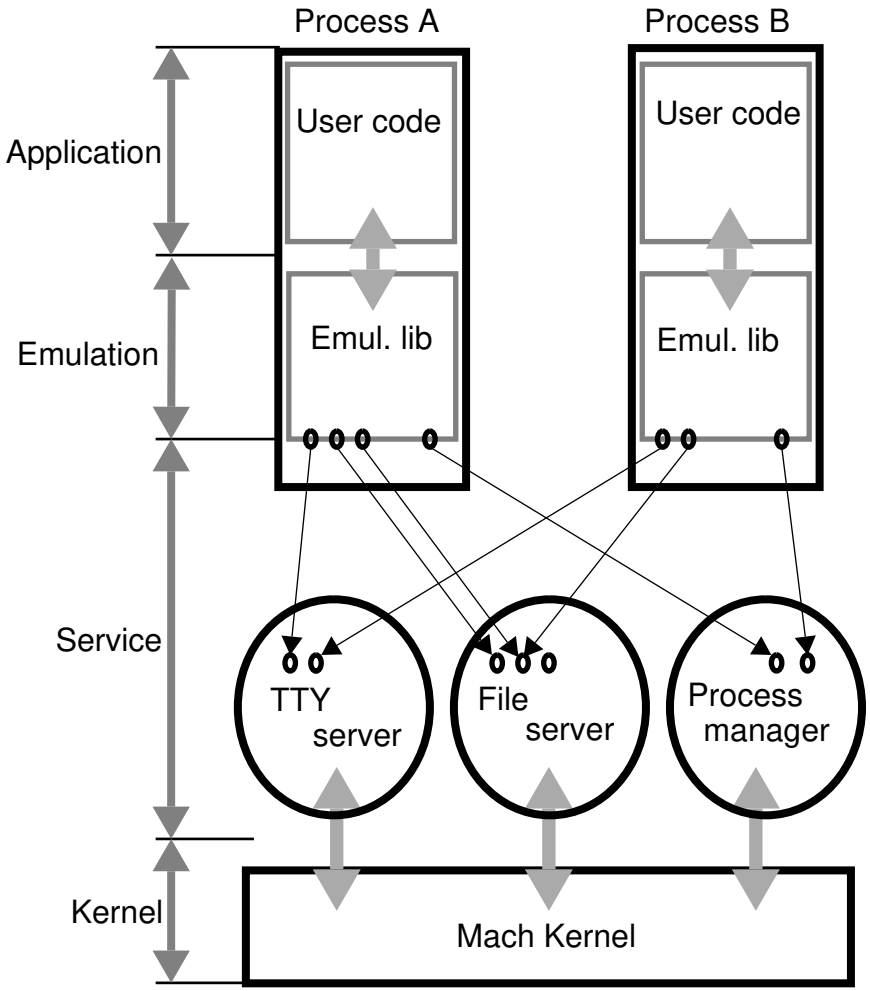
Issues

- separation of functions -- specialized servers
- coordination / shared state (all / some accesses)
- generic vs. OS-specific components and interfaces
- in-kernel vs. out-of kernel
- protected components (emulation library, etc.)
- ...

Solutions

- single-server
- 2-level multi-server
- 2-level + kernel support
- symmetric multi-server
- ...

Architecture



Revolution, not evolution

Key ideas

- everything out of kernel
- modular services
- coordination through emul lib.
- generic services / layering
- common facilities
- object-oriented approach
- build tools as needed

Approach

- general design
- BSD 4.3 prototype
- successive refinements

Design Overview

Components

- Servers
- Emulation library

System framework

- client-server interactions / object model
- common facilities
- common service interfaces

Low-level facilities / class library

Support tools

Major Servers

direct user service:

- file
- terminal (TTY)
- local IPC (pipes, sockets)
- process management
- network
- devices

support:

- root name service
- authentication
- shared state (blackboard)
- startup / admin
- diagnostics
- Mach services

Emulation Library

OS-specific

Inside the process address space

- performance gains (copyin/copyout, etc.)
- robustness concerns but NOT security (unforgeable Mach ports)

Local repository for almost all process state

- external mechanisms for shared info
- automatic state inheritance for UNIX *fork()*

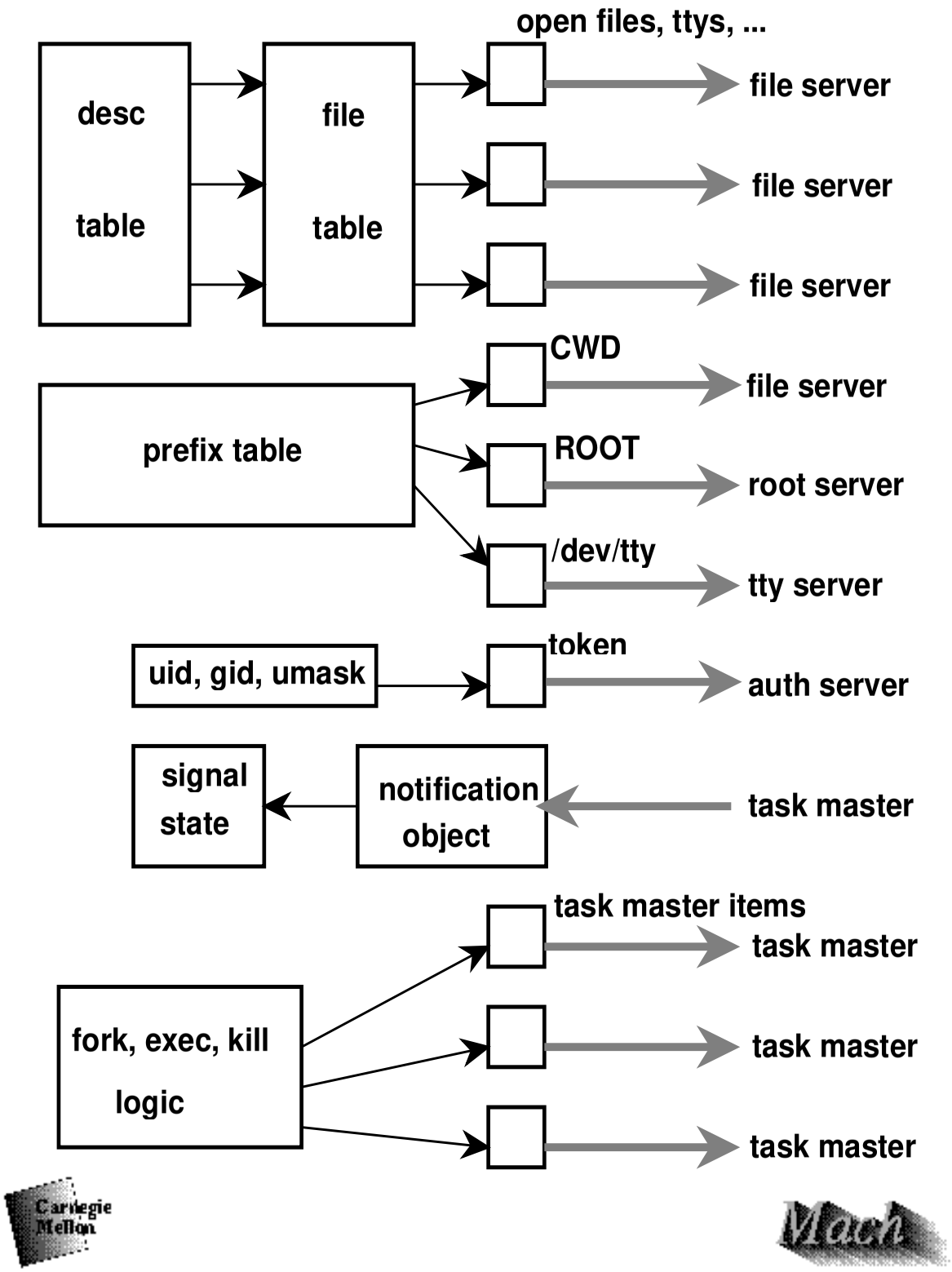
Direct access to each individual system entity (*items*) through separate Mach port

Active threads for signals, async I/O, etc.

Optimization: maximize client-side processing

- specification of server interfaces
- complex proxy objects

Multi-server: UNIX Emulation Library



Daniel P. Julin / Carnegie Mellon University / Mach Project

Major Facilities and Interfaces:

object model:

- servers export *items*

general-purpose:

- item management and access control
- naming
- I/O
- asynchronous notifications
- shared state management

specialized:

- network control
- process control
- TTY control
- system administration

Low-level Facilities

Common library of reusable objects (MachObjects, C++)

Support of high-level interfaces

- mapped file access
- data streams -- I/O buffers
- hierarchical name spaces
- inter-server name resolution

Internal functions

- authentication, access mediation
- asynchronous notifications
- file system glue code
- shared state -- blackboards

Support of emulation library

Tools

MachObjects

C++

RPC package

threads package w/ exception handling

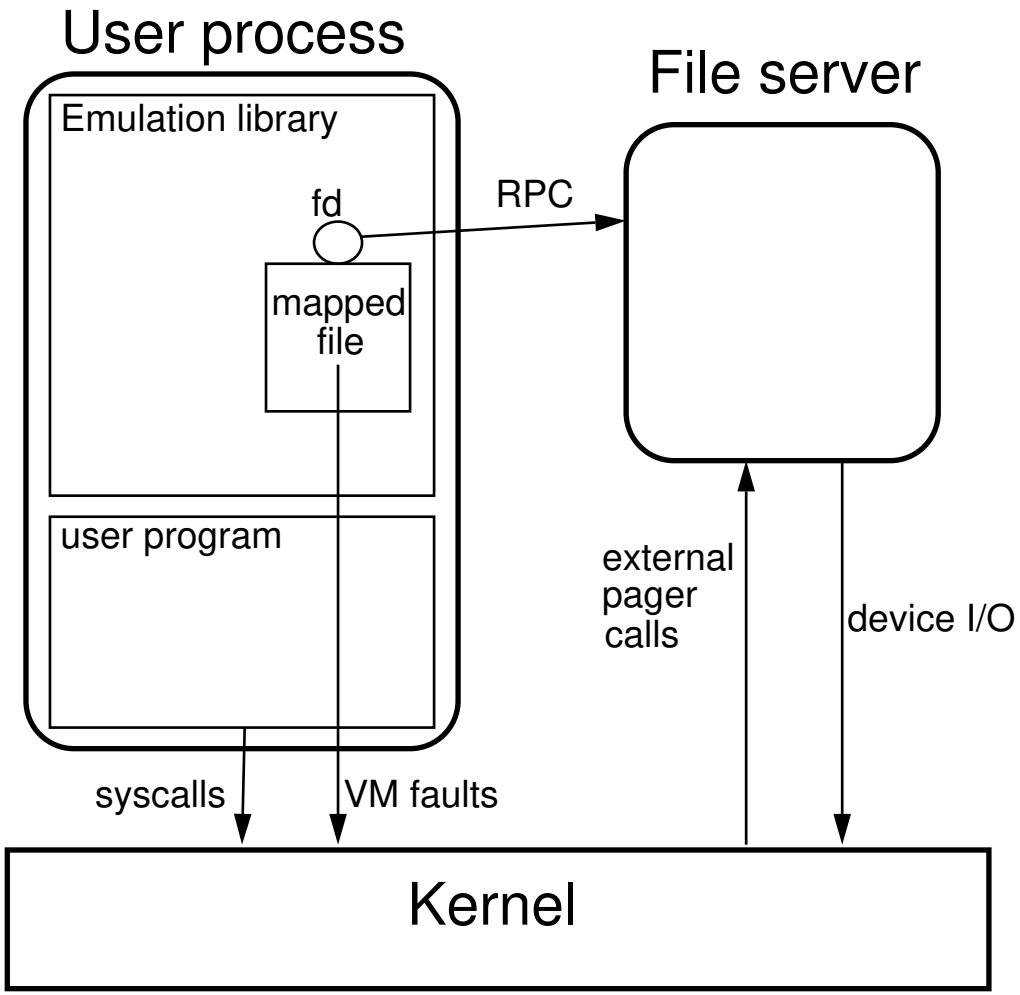
Sources

/afs/cs.cmu.edu/project/mach3/src/us

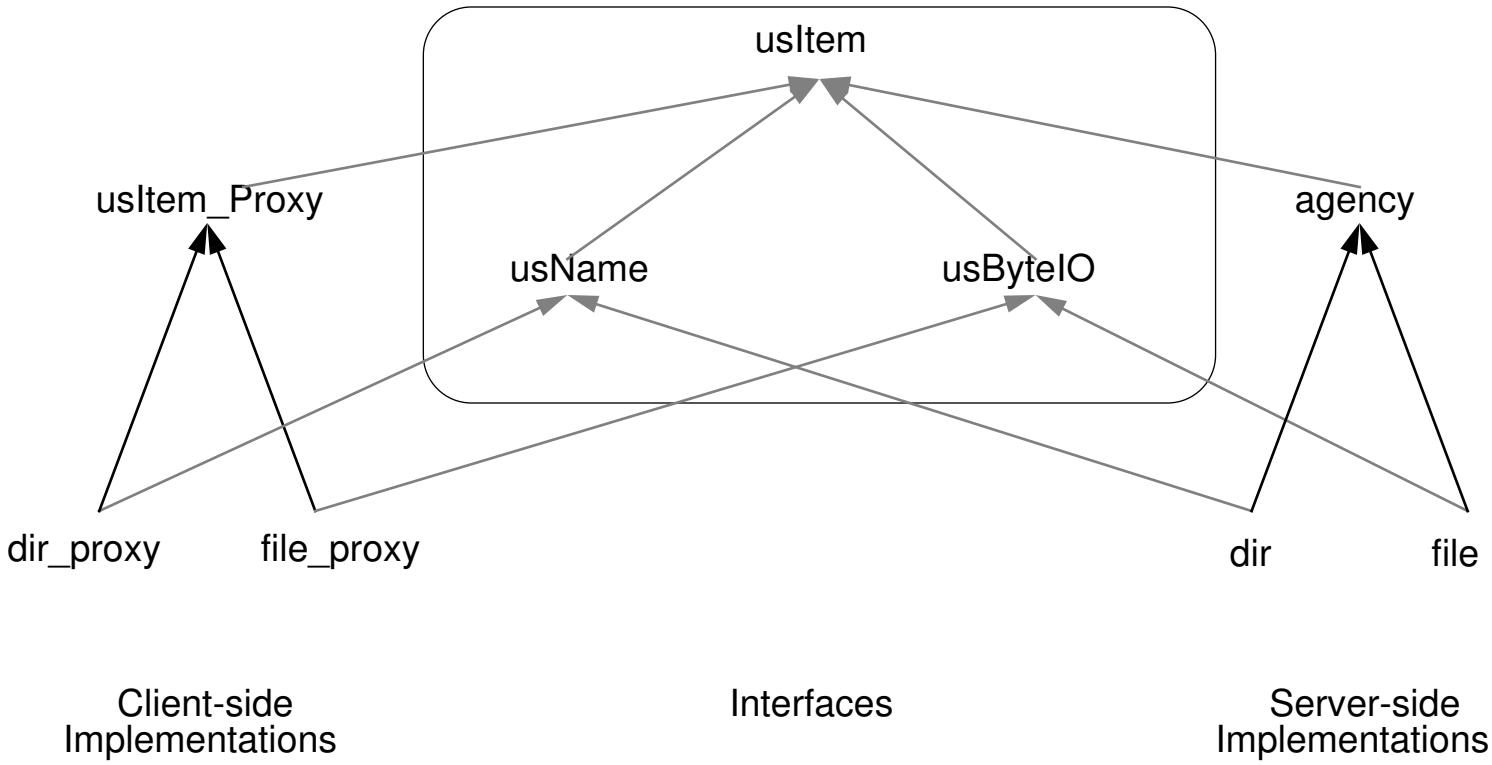
- include, local_include, unix_include
- lib/threads
- lib/machobj, lib/machobj++
- lib/us, lib/us++
- lib/ux
- lib/emul
- server/{as,config,diag,name,pipenet,tm,tty,ufs}
- pkg/xkernel

Example: simple UNIX file operations

```
int fd;

char buf[100];


fd = open("/tmp/foo",...);

read(fd,buf,100);

close(fd);
```



C++ Class Organization



Items

Common information

- type
- standard attributes

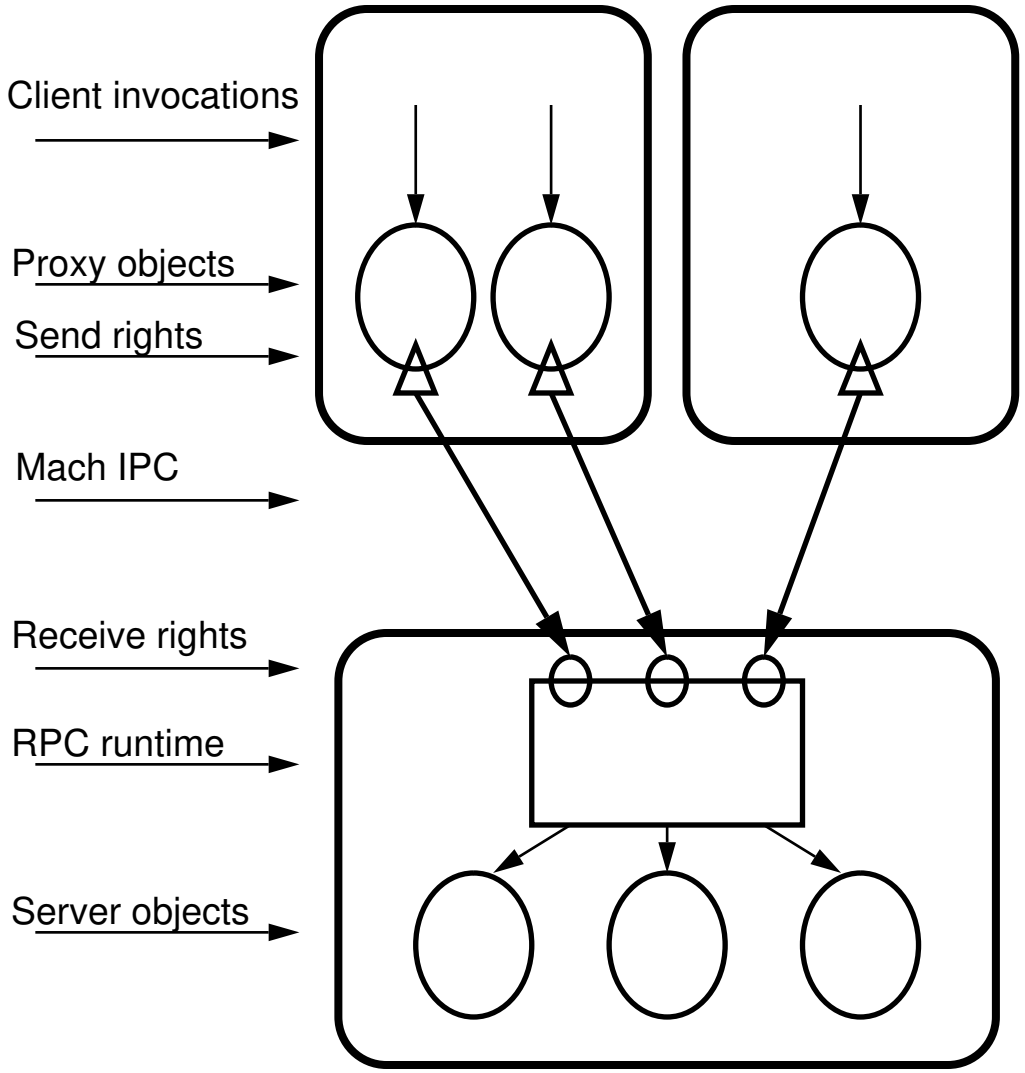
Classes

- common base for all items: *agency*

Main exported operations -- *usItem (us_item_ifc.h)*

- ns_get_attributes()
- specialized *set_attributes* operations

RPC System



- transparent C++ method invocations
- automatic object dispatching
- object references passed as arguments
- proxies can be arbitrarily complex
- garbage-collection through no-more-senders notifications

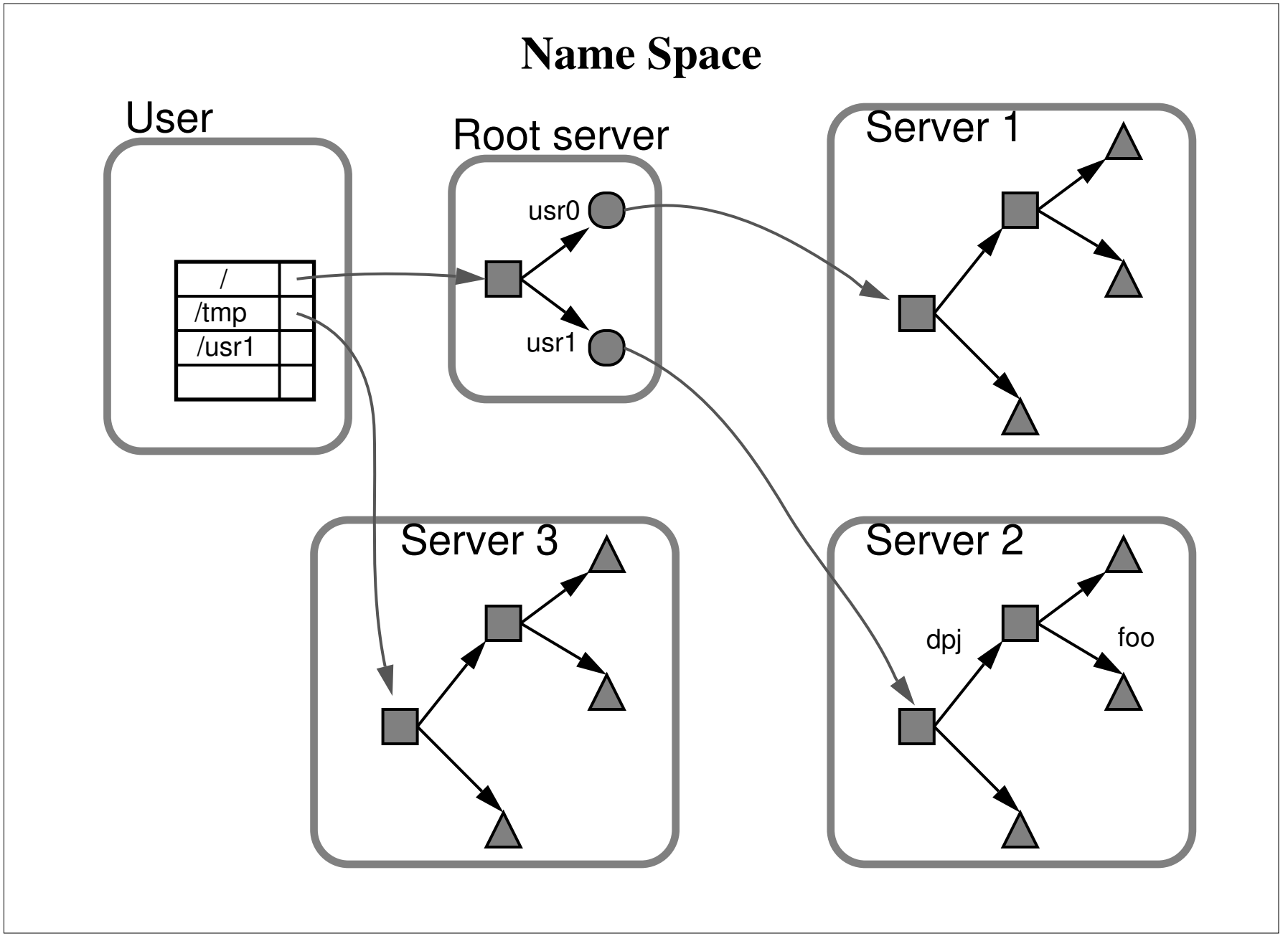
Name Service

objectives:

- uniform access to all items
- easy integration of new, foreign components
- customize name space: per-user / per-system, per-OS
- support distributed and centralized operation
- issues of scale, network distribution

key ideas:

- item manipulation: type, attributes
- access mediation: credentials, access rights, access control lists
- uniform name space: resolving protocol + glue components (root name server, directories, symlinks, mount points)
- user controlled operations: prefix table, parsing of path names
- item creation through the name space


Name Space
User
Root server
Server 1
usr0
/
/tmp
/usr1
usr1
Server 3
Server 2
dpj
foo

Name Space

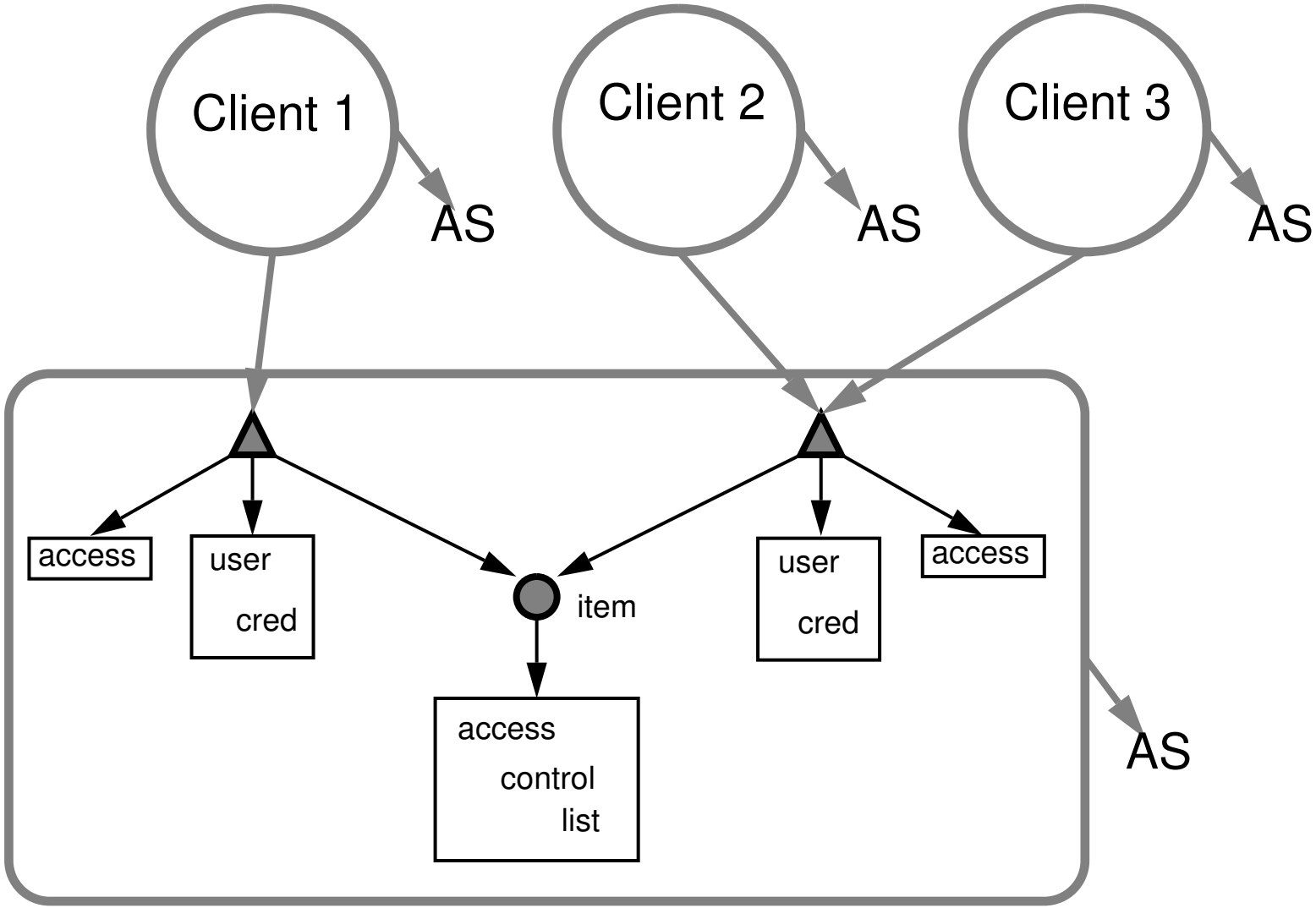
Classes

- building blocks: *dir, symlink, tsymlink, mountpt*
- common base for terminal entries: *vol_agency*
- temporary entries: *tmp_agency*
- client-side navigation: *std_name*

Main exported operations -- *usName (us_name_ifc.h)*

- ns_resolve()
- ns_create()
- ns_insert_entry()
- ns_remove_entry()
- ns_list_entries()

Access Mediation



Access Mediation

Classes:

- mediation proper: *agent, agency*
- user credentials: *std_cred*
- access control list: *std_prot*
- connection with authentication server: *std_auth*

Main exported operations -- *usItem (us_item_ifc.h)*

- ns_authenticate()
- ns_duplicate()
- ns_get_access()
- ns_get_protection()
- ns_set_protection()
- ns_get_privileged_id()

I/O

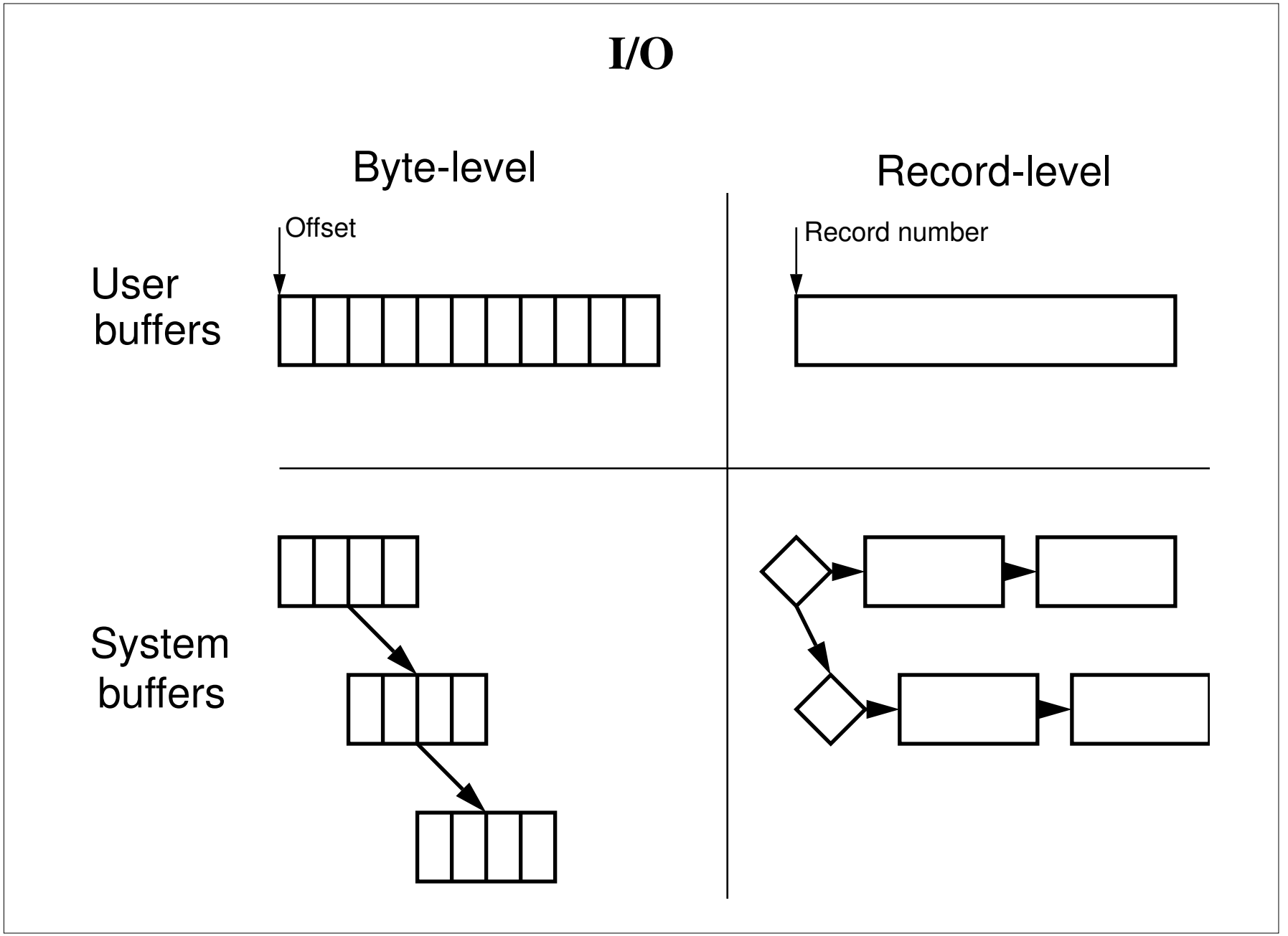
objectives:

- allow (but not require) use of shared memory
- compatibility with existing servers
- minimize server overhead / complexity
- support sophisticated users
- extensions for complex I/O
- independent access by multiple users

key ideas:

- data representation: bytestreams and records (64-bit keys)
- data manipulation: user buffers and system blocks
- all operations under user control (async operations, random access)
- service options: wait, truncate, probe
- explicit and transparent shared memory

I/O


Byte-level
Record-level
Offset
Record number
User buffers
System buffers

I/O

Classes:

- mapped-file management: *mf_user, mf_mgr, mf_mem, pager*
- system buffers: *default_iobuf_mgr*
- streams: *bytestream, recordstream, stream_base*

Main exported operations -- *usByteIO (us_byteio_ifc.h), usRecIO (us_recio_ifc.h)*

- io_read(), io_write()
- io_read_seq(), io_write_seq()
- io_map()
- io_read1rec(), io_write1rec()
- io_getbytes(), io_putbytes()
- io_getrec(), io_putrec()

Other Modules

Shared state management

Asynchronous notifications

Process management

Network control

TTY control

VNODE glue code

Shared State Support

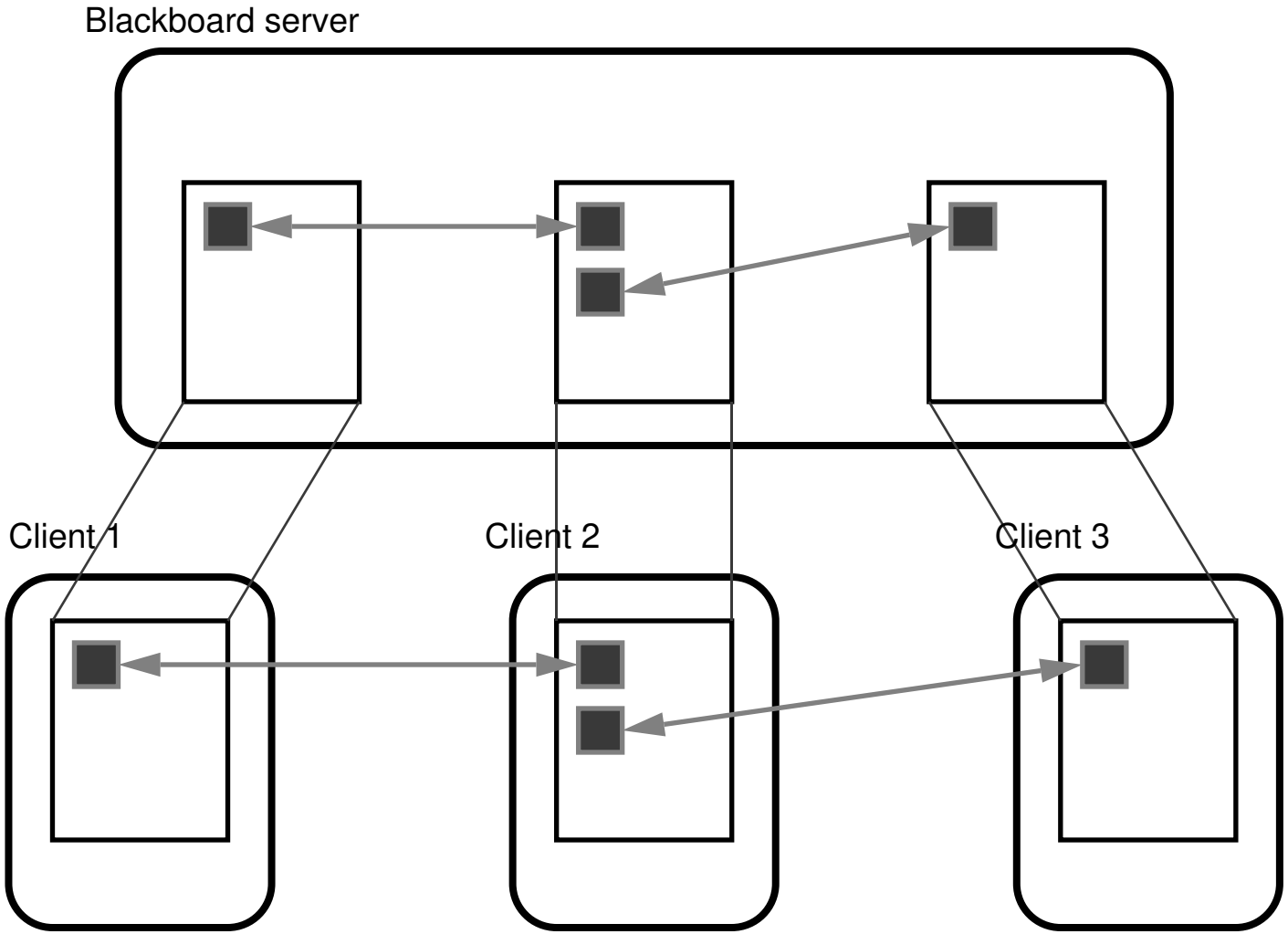
objectives:

- arbitrary sharing between multiple clients
- decoupled from individual servers
- management of client-side cached information
- very low overhead possible
- flexible tradeoffs between space, time, trust, complexity, ...

key ideas:

- *item callbacks*: synchronous boolean notifications -- also used for access revocation
- *blackboards* and *shared records*: arbitrary data, direct user access
- shared memory or messages
- central *blackboard server*

Blackboard Service



Asynchronous Notifications

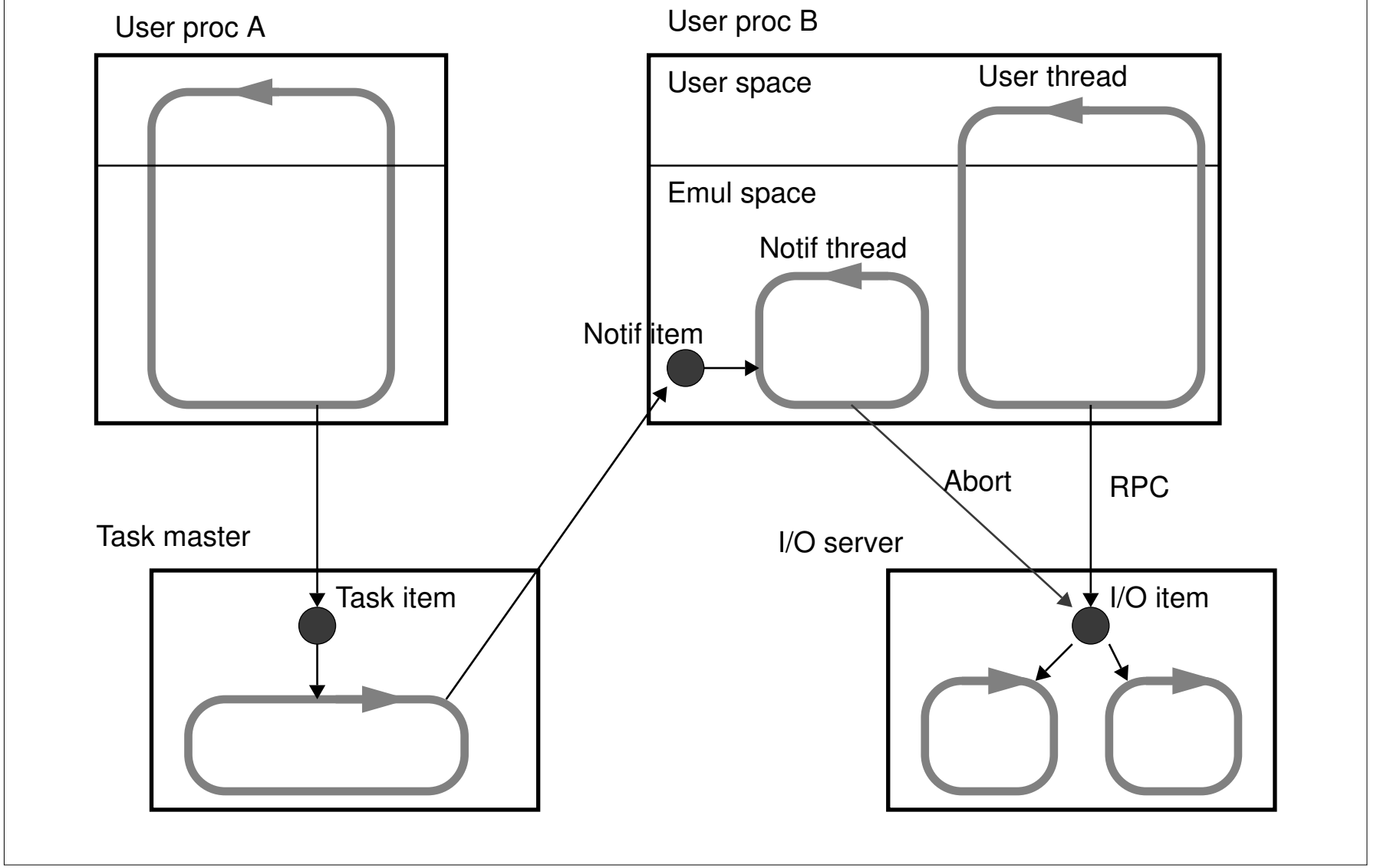
objectives:

- UNIX signals, VMS events, cheap external asynchronous service
- integrated in the uniform name space
- minimize overhead when not in use

key ideas:

- generic: event number, code, subcode
- special item in the emulation library
- remote invocation through the task_master
- alert and exception handling mechanism in the emulation library

Signal Delivery



Startup

boot on POE or other single-server

- share TTY's, network
- separate disk partition

config_server

- netname service

startup script (shell)

- start all the servers

setup the name space (fsadmin script)

emul_init

- first emulated process

debugging: GDB

Status

BSD prototype is now self-hosting:

- stays up indefinitely
- supports all common UNIX “shell-level” activities except networking (TFTP only)
- performance acceptable for everyday use of prototype
- short-term task: finish the C++ conversion

Analysis of design and documentation underway

More work:

- functionality
- performance
- evaluation / experimentation

Performance

Already decent and getting better

Suspected/possible bottlenecks:

- MachObjects invocations: not really
- object instantiation and returning objects to clients: yes
- RPC: maybe
- inheritance of state during fork()
- complex generic operations: fork(), item lookups
- paging

Avenues of action:

- optimize the RPC package
- optimize selected high-level functions (fork(), stat(), file open)
- new mechanisms: blackboard server
- experiment with inheritance of state (memory + ports)
- streamline common code paths and object creation (C++, new threads)

Functionality / Design

Relatively straightforward:

- networking + network file systems
- devices
- implement more system calls and/or options
- X
- system startup, core dumps, etc...

More interesting:

- asynchronous I/O, select()
- item callbacks -- mount()/umount()
- shared state management
- set-uid exec()
- resource control
- experiment with other OS / features

Size and Complexity

Source code:

- 9 servers + 1 BSD emulation library + utilities
- 1244 files: 139(C++) + 382(C) + 511(headers) + 31(assembler) + 212
- 6.8Megs, 250000 raw lines, 128000 effective lines, 60000 semicolons
- 1173 effective lines of machine-dependent code in 19 files (+ support libs)

Complexity:

- 78 classes in common library
- 84 exported methods in 7 logical interfaces (18 interface classes)

Concerns:

- bigger stress on the kernel
- need shared libraries to avoid code replication
- need more maturity in the common library + standards

Other Aspects

complex emulation libraries:

- simpler servers, optimizations, specialization
- very hard to write
- possible fork() overhead
- shared state is still a challenge

languages support:

- compiler is not mature
- good RPC support is hard
- multiple inheritance
- runtime checking
- why hasn't all this been solved already?

Summary of Experiences

positive:

- the overal architecture (servers + GP facilities) works
- modularity is a big advantage
- object-oriented technology + complex tools do help
- common service library is a success so far

issues:

- some OS features are too obscure to standardize
- beware of code size and replication
- standard interfaces increase complexity
- “jury still out” on complex emulation libraries

Future Work

Refine the framework, experiment with new ideas...

Develop better language support (better languages?)

Build production systems -- get rich!

Distributed systems -- scaling

New high-level features (transactions, multi-media, ...)

Second pass through other emulation system projects