NFS implementation details

- **Hard vs. soft mounts**
- **Dealing with crashes and lost messages**
  - In what sense is NFS stateless?
  - What does it mean that NFS requests are *idempotent*?
  - Does NFS provide at-most-once semantics?
- **What happens if you have execute but not read perms on file?**
- **What if file permissions change after open?**
  - E.g., create file for writing with mode 0444
- **What happens if server restored from backup?**

NFS version 3

- **Same general architecture as NFS 2**
- **New access RPC**
  - Supports clients and servers with different uids/gids
- **Better support for caching**
  - Unstable writes while data still cached at client
  - More information for cache consistency
- **Better support for exclusive file creation**

File handles

```
struct nfs_fh3 {
  opaque data<64>;
};
```

- **Server assigns an opaque file handle to each file**
  - Client obtains first file handle out-of-band (mount protocol)
  - File handle hard to guess – security enforced at mount time
  - Subsequent file handles obtained through lookups
- **File handle internally specifies file system / file**
  - Device number, i-number, *generation number*, ...
  - Generation number changes when inode recycled

File attributes

```
struct fattr3 {
  ftype3 type;
  uint32 mode;
  uint32 nlink;
  uint32 uid;
  uint32 gid;
  uint64 size;
  uint64 used;
  specdata3 rdev;
  uint64 fsid;
  uint64 fileid;
  nfstime3 atime;
  nfstime3 mtime;
  nfstime3 ctime;
};
```

- **Most operations can optionally return** `fattr3`
- **Attributes used for cache-consistency**

Lookup

```
struct diropargs3 {
  nfs_fh3 dir;
  filename3 name;
};
```

```
struct lookup3resok {
  nfs_fh3 object;
  post_op_attr obj_attributes;
  post_op_attr dir_attributes;
};
```

```
union lookup3res switch (nfsstat3 status) {
case NFS3_OK:
  lookup3resok resok;
default:
  post_op_attr resfail;
};
```

- **Maps** $\langle \text{directory}, \text{handle} \rangle \rightarrow \text{handle}$
  - Client walks hierarch one file at a time
  - No symlinks or file system boundaries crossed

Create

```
struct create3args {
  diropargs3 where;
  createhow3 how;
};
```

```
union createhow3 switch (createmode3 mode) {
case UNCHECKED:
case GUARDED:
  sattr3 obj_attributes;
case EXCLUSIVE:
  createverf3 verf;
};
```

- `UNCHECKED` – **succeed if file exists**
- `GUARDED` – **fail if file exists**
- `EXCLUSIVE` – **persistent record of create**

Read

```
struct read3args {          struct read3resok {
  nfs_fh3 file;               post_op_attr file_attributes;
  uint64 offset;              uint32 count;
  uint32 count;               bool eof;
};                            opaque data<>;
                            };
union read3res switch (nfsstat3 status) {
case NFS3_OK:
  read3resok resok;
default:
  post_op_attr resfail;
};
```

- **Offset explicitly specified (not implicit in handle)**
- **Client can cache result**

Data caching

- **Client can cache blocks of data read and written**
- **Consistency based on times in `fattr3`**
  - mtime: Time of last modification to file
  - ctime: Time of last change to inode
    (Changed by explicitly setting mtime, increasing size of file, changing permissions, etc.)
- **Algorithm: If mtime or ctime changed by another client, flush cached file blocks**

Write discussion

- **When is it okay to lose data after a crash?**
  - Local file system
  - Network file system
- **NFS2 servers flush writes to disk before returning**
- **Can NFS2 perform write-behind?**
  - Implementation issues – blocking kernel threads
  - Issues of semantics – how to guarantee consistency

Flashback: NFS2 write call

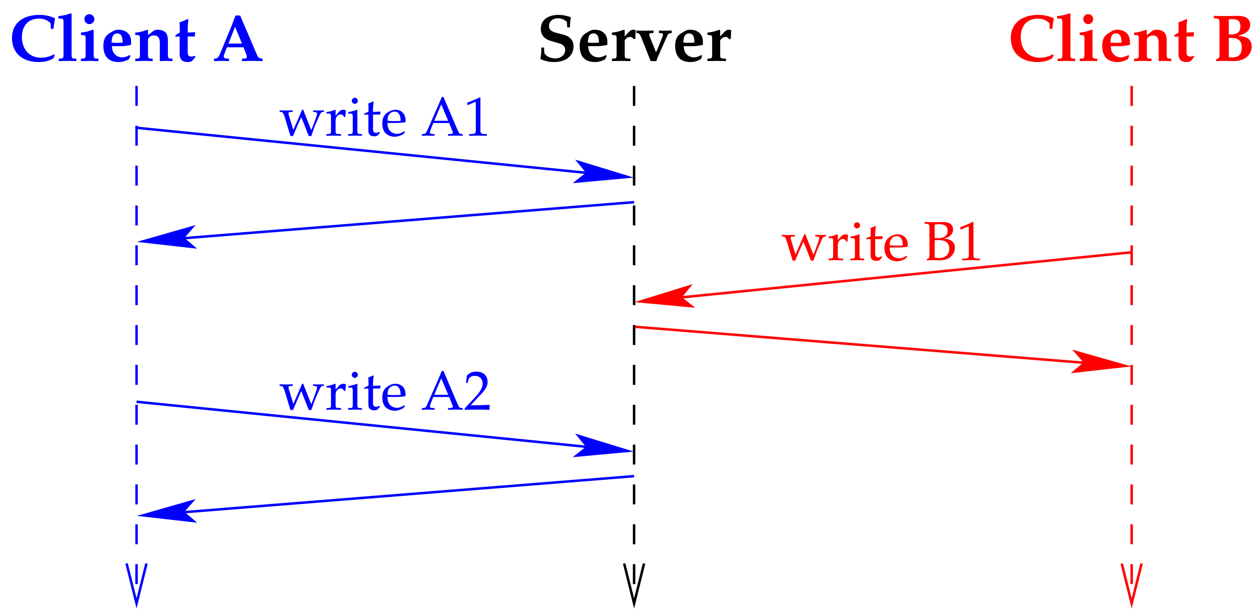
```
struct writeargs {
  fhandle file;
  unsigned beginoffset;
  unsigned offset;
  unsigned totalcount;
  nfsdata data;
};
```

```
union attrstat
    switch (stat status) {
case NFS_OK:
  fattr attributes;
default:
  void;
};
```

```
attrstat NFSPROC_WRITE(writeargs) = 8;
```

- **On successful write, returns new file attributes**
- **Can NFS2 keep cached copy of file after writing it?**

Write race condition



- **Suppose client overwrites 2-block file**
  - Client A knows attributes of file after writes A1 & A2
  - But client B could overwrite block 1 between the A1 & A2
  - No way for client A to know this hasn't happened
  - Must flush cache before next file read (or at least open)

NFS3 Write arguments

```
struct write3args {
  nfs_fh3 file;
  uint64 offset;
  uint32 count;
  stable_how stable;
  opaque data<>;
};
```

```
enum stable_how {
  UNSTABLE = 0,
  DATA_SYNC = 1,
  FILE_SYNC = 2
};
```

- **Two goals for NFS3 write:**
  - Don't force clients to flush cache after writes
  - Don't equate cache consistency with crash consistency
    I.e., don't wait for disk just so another client can see data

Write results

```
struct write3resok {
  wcc_data file_wcc;
  uint32 count;
  stable_how committed;
  writeverf3 verf;
};
union write3res
    switch (nfsstat3 status) {
case NFS3_OK:
  write3resok resok;
default:
  wcc_data resfail;
};
```

```
struct wcc_attr {
  uint64 size;
  nfstime3 mtime;
  nfstime3 ctime;
};
struct wcc_data {
  wcc_attr *before;
  post_op_attr after;
};
```

- **Several fields added to achieve these goals**

Data caching after a write

- **Write will change mtime/ctime of a file**
  - "`after`" will contain new times
  - Should cause cache to be flushed
- **"`before`" contains previous values**
  - If `before` matches cached values, no other client has changed file
  - Okay to update attributes without flushing data cache

Write stability

- **Server write must be at least as stable as requested**
- **If server returns write UNSTABLE**
  - Means permissions okay, enough free disk space, ...
  - But data not on disk and might disappear (after crash)
- **If DATA_SYNC, data on disk, maybe not attributes**
- **If FILE_SYNC, operation complete and stable**

Commit operation

- **Client cannot discard any UNSTABLE write**
  - If server crashes, data will be lost
- **COMMIT RPC commits a range of a file to disk**
  - Invoked by client when client cleaning buffer cache
  - Invoked by client when user closes/flushes a file
- **How does client know if server crashed?**
  - Write and commit return `writeverf3`
  - Value changes after each server crash (may be boot time)
  - Client must resend all writes if verf value changes

Attribute caching

- **Close-to-open consistency**
  - It really sucks if writes not visible after a file close (Edit file, compile on another machine, get old version)
  - Nowadays, all NFS opens fetch attributes from server
- **Still, lots of other need for attributes (e.g., `ls -al`)**
- **Attributes cached between 5 and 60 seconds**
  - Files recently changed more likely to change again
  - Do weighted cache expiration based on age of file
- **Drawbacks:**
  - Must pay for round-trip to server on every file open
  - Can get stale info when statting a file

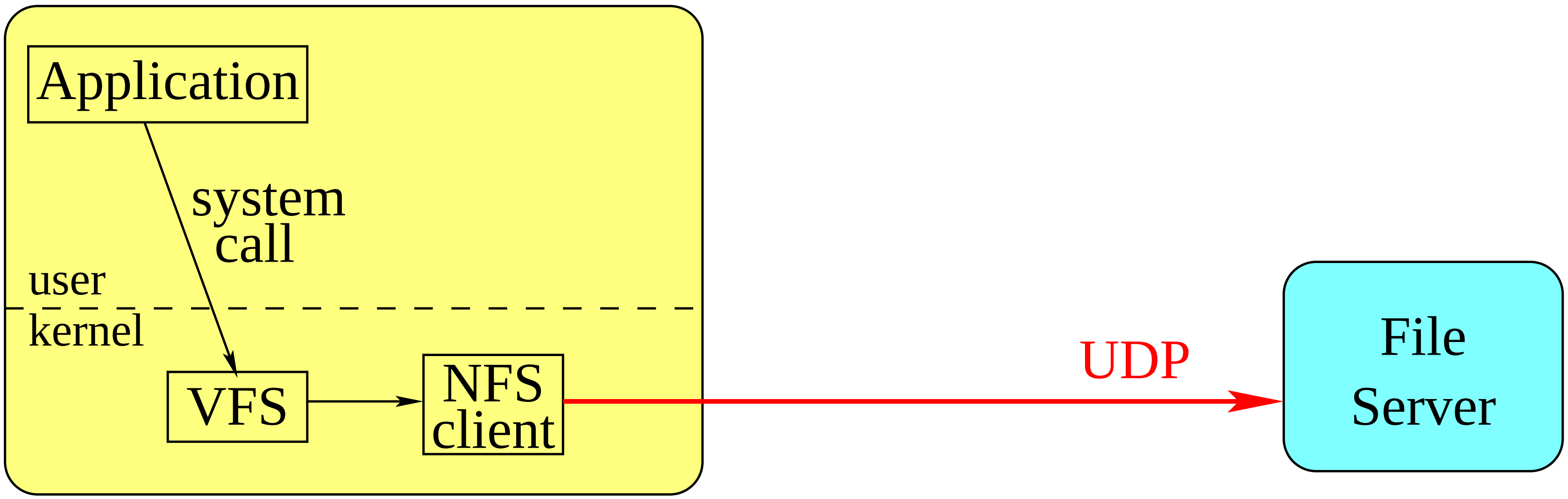
NFS Optimizations

- NFS server and block I/O daemons
- Client-side buffer cache (write-behind w. flush-on-close)
- XDR directly to/from mbufs
- Client-side attribute cache
- Fill-on-demand clustering, swap in small programs
- Name cache

User-level file systems

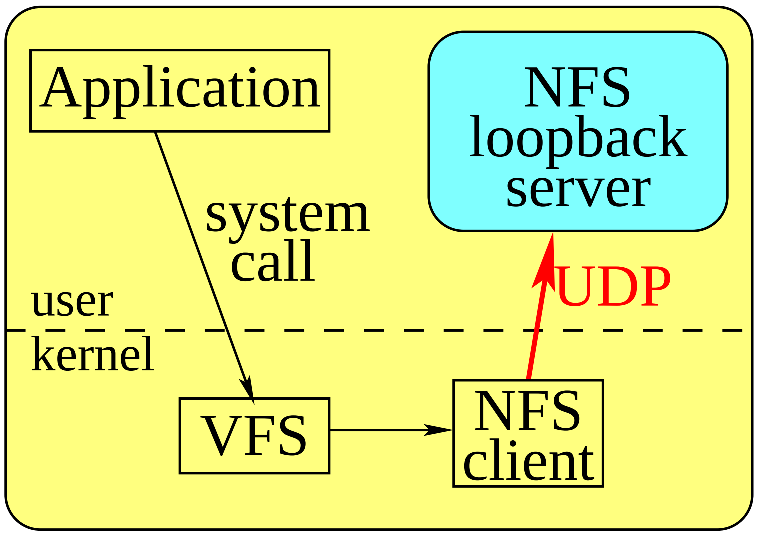
- **Developing new file systems is a difficult task**
  - Most file systems implemented in the kernel
  - Debugging harder, crash/reboot cycle longer
  - Complicated kernel-internal API (VFS layer)
- **File systems are not portable**
  - Kernel VFS layer differs significantly between OS versions
- **NFS can solve these problems…**
  - C++ toolkit greatly simplifies the use of NFS

NFS overview



- **NFS is available for almost all Unixes**
- **Translates file system accesses into network RPCs**
  - Hides complex, non-portable VFS interface

Old idea: NFS loopback servers



- **Implement FS as an NFS server in a local process**
- **Requires only portable, user-level networking**
  - File system will run on any OS with NFS support

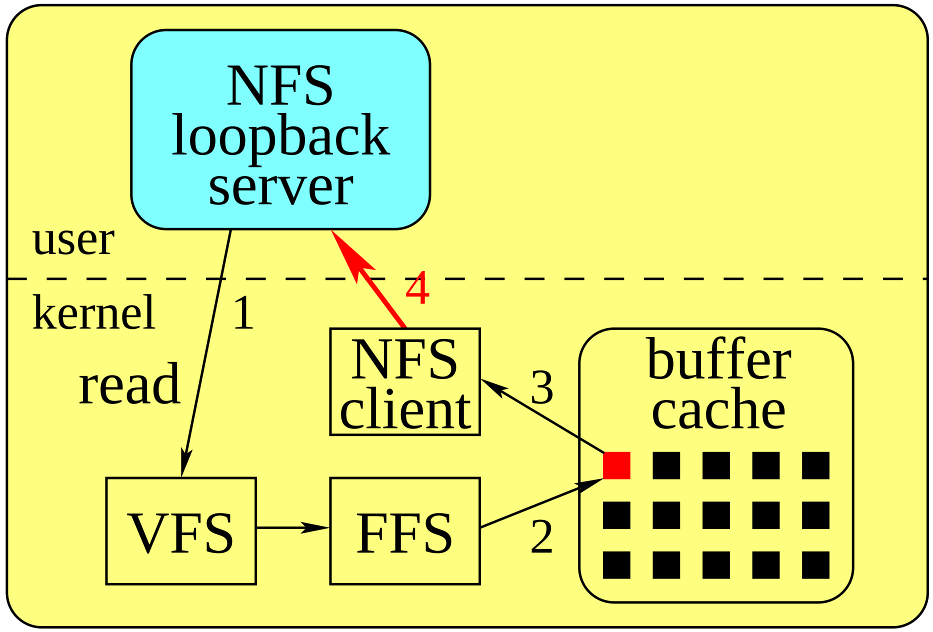
Problem: Performance

- **Context switches add latency to NFS RPCs**
- **Must service NFS RPCs in parallel**
  - Overlap latencies associated with handling requests
  - Keep disk queue full for good disk arm scheduling
- **If loopback server blocks, so do other processes**
  - E.g., loopback for `/loop` blocks on a TCP connect
  - *getcwd()* and "`ls -al /`" will block, even outside of `/loop`
- **One slow file can spoil the whole file system** [a]
  - If one RPC times out, client decides server is down
  - Client holds other RPCs to avoid flooding server
  - Example: Alex FTP file server

---

[a] `NFS3ERR_JUKEBOX` can help, but has problems

Problem: Any file I/O can cause deadlock



1. **Loopback server reads file on local disk**
2. **FFS needs to allocate a buffer**
3. **Kernel chooses a dirty NFS buffer to recycle**
4. **Blocks waiting for reply to write RPC**

Problem: Development and debugging

- **Bugs must be mapped onto NFS RPCs**
  - Application make system calls
  - Not always obvious what RPCs the NFS client will generate
  - Bug may actually be in kernel's NFS client
- **When loopback servers crash, they hang machines!**
  - Processes accessing the file system hang, piling up
  - Even `umount` command accesses the file system and hangs
- **Repetitive code is very error-prone**
  - Often want to do something for all 20 NFS RPC procedures (e.g., encrypt all NFS file handles)
  - Traditionally requires similar code in 20 places

SFS toolkit

- **Goal: Easy construction of loopback file systems**
- **Support complex programs that never block**
  - Service new NFS RPCs while others are pending
- **Support multiple mount points**
  - Loopback server emulates multiple NFS servers
  - One slow mount point doesn't hurt performance of others
- **Simplify task of developing/debugging servers**
  - `nfsmounter` daemon eliminates hangs after crashes
  - RPC library supports tracing/pretty-printing of NFS traffic
  - RPC compiler allows traversal of NFS call/reply structures

`nfsmounter` **daemon**

- **`nfsmounter` mounts NFS loopback servers**
  - Handles OS-specific details of creating NFS mount points
  - Eliminates hung machines after loopback server crashes
- **To create an NFS mount point, loopback server:**
  - Allocates a network socket to use for NFS
  - Connects to `nfsmounter` daemon
  - Passes `nfsmounter` a copy of the NFS socket
- **If loopback server crashes:**
  - `nfsmounter` takes over NFS socket
  - Prevents processes accessing file system from blocking
  - Serves enough of file system to unmount it

Asynchronous I/O and RPC libraries

- **Never wait for I/O or RPC calls to complete**
  - Functions launching I/O must return before I/O completes
  - Bundle up state to resume execution at event completion
- **Such event-driven programming hard in C/C++**
  - Cumbersome to bundle up state in explicit structures
  - Often unclear who must free allocated memory when
- **Alleviated by two C++ template hacks**
  - `wrap`—function currying: bundles function of arbitrary signature with initial arguments
  - Reference counted garbage collection for any type:

    ```
    ptr<T> tp = new refcounted<T> (/* ...  */);
    ```

`rpcc`: A new RPC compiler for C++

- **Compiles RFC1832 XDR types to C++ structures**
- **Produces generic code to traverse data structures**
  - RPC marshaling only one possible application
- **Can specialize traversal to process particular types**
  - Encrypt/decrypt all NFS file handles for security
  - Extract all file attributes for enhanced caching
- **Outputs pretty-printing code**
  - Environment variable makes library print all RPC traffic
  - Invaluable for debugging strange behavior

Stackable NFS manipulators

- **Often want to reuse/compose NFS processing code**
- **SFS toolkit provides stackable NFS manipulators**
  - NFS server objects generate NFS calls
  - Most loopback servers begin with `nfsserv_udp`
  - Manipulators are servers constructed from other servers
- **Example uses:**
  - `nfsserv_fixup`—works around bugs in NFS clients
  - `nfsdemux`—demultiplex requests for multiple mount points

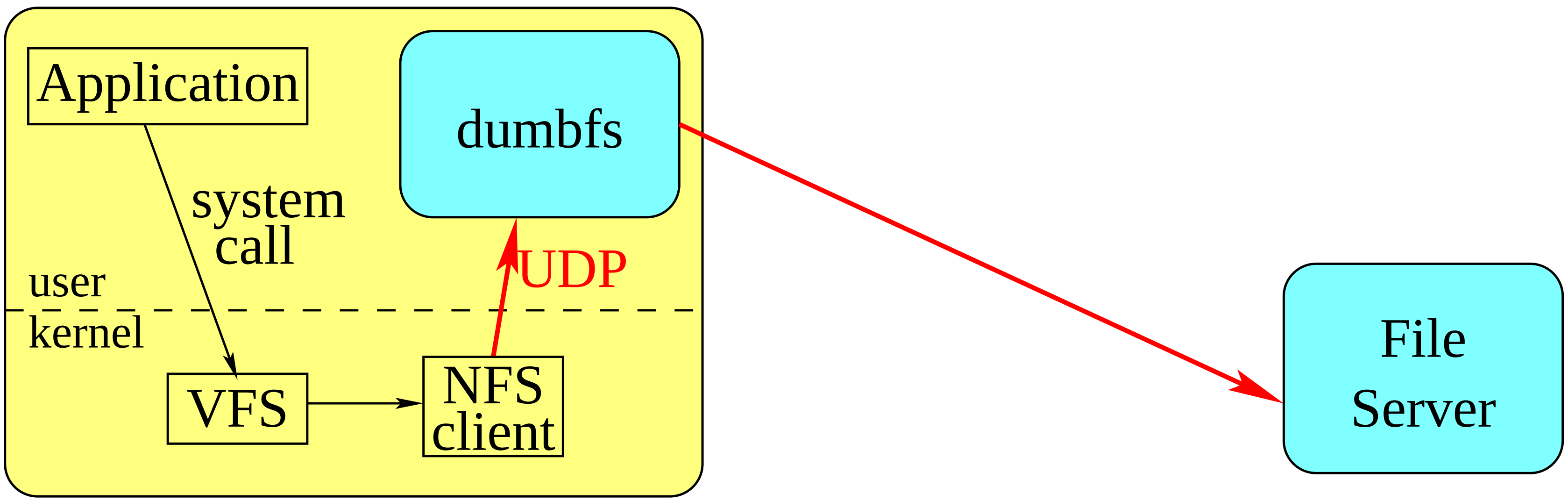
Creating new mountpoints

- **Hard to create mountpoints in-place and on-the-fly**
  - If user looks up `/home/u1`, must reply before mounting
  - Previous loopback servers use links: `/home/u1`→`/a/srv/u1`
- **SFS automounter mounts in place with two tricks**
  - `nfsmounter` has special gid, differentiating its NFS RPCs
  - SFS dedicates "wait" mountpoints under `.mnt/{0,1,...}`
- **Idea: Show different files to users and `nfsmounter`**
  - User sees `/home/u1` as symlink `u1`→`.mnt/0/0`
  - `.mnt/0/0` is symlink that hangs when read
  - `nfsmounter` sees `/home/u1` as directory, can mount there
  - When mount complete, `.mnt/0/0`→`/home/u1`

Limitations of loopback servers

- **No file close information**
  - Often, FS implementor wants to know when a file is closed (e.g., for close-to-open consistency of shared files)
  - Approximate “close simulator” exists as NFS manipulator
  - NFS version 4 will include closes
- **Can never delay NFS writes for local file system**
  - E.g., CODA-like cache hard to implement

Application: DumbFS



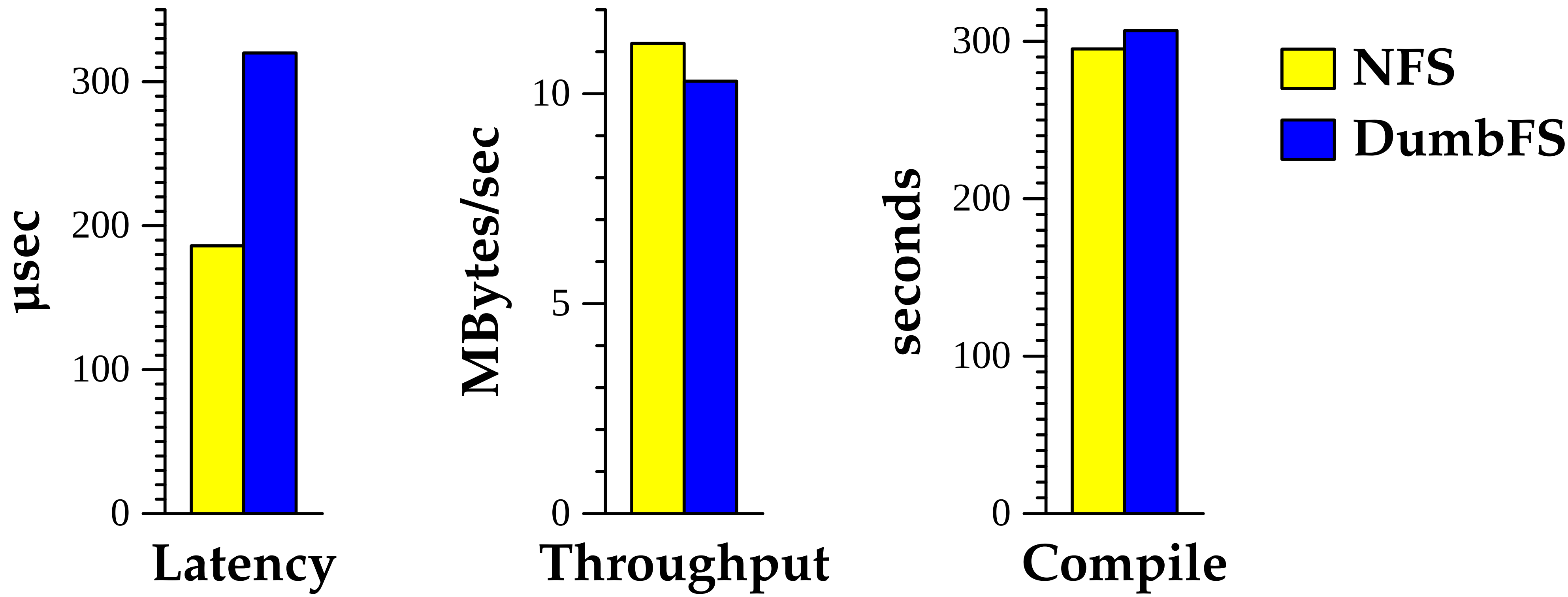
- **Simplest loopback server—just forwards requests**
  - 119 lines of code, no cleanup code needed!
- **Isolates performance impact of toolkit**

DumbFS NFS RPC forwarding

```
void dispatch (nfscall *nc)
{ // ...
  nfsc->call (nc->proc (), nc->getvoidarg (),
              nc->getvoidres (), wrap (reply, nc) /* ... */);
}
static void reply (nfscall *nc, enum clnt_stat stat)
{
  if (stat == RPC_SUCCESS) nc->reply (nc->getvoidres ());
  else // ...
}
```

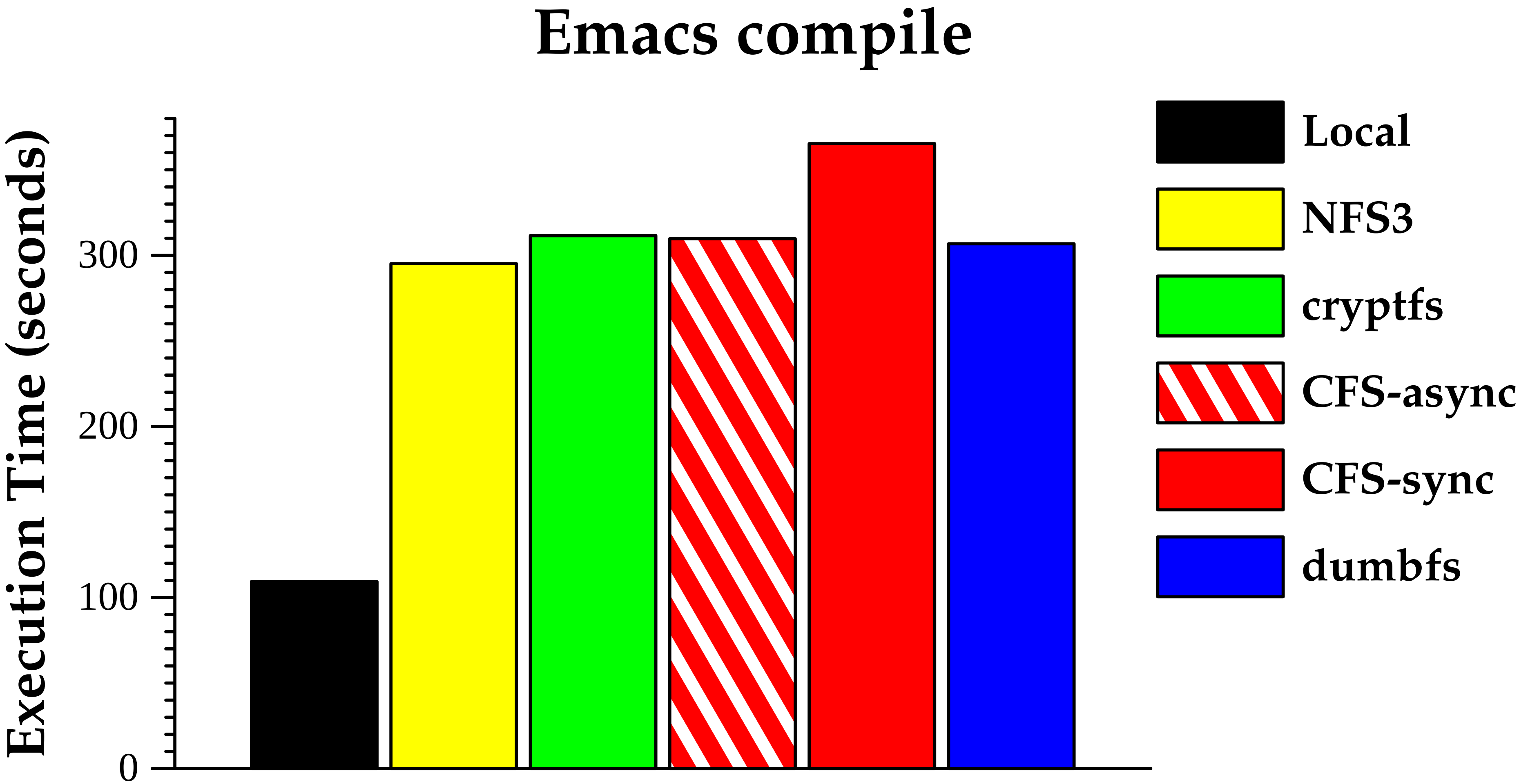
- **Single dispatch routine for all NFS procedures**
- **RPCs to remote NFS server made asynchronously**
  - `dispatch` returns before `reply` invoked

DumbFS performance



Application: CryptFS

- **Acts as both NFS server and client (like DumbFS)**
  - Almost 1–1 mapping between NFS calls recevied and sent …encrypt/decrypt file names and data before relaying
  - Bare bones "encrypting DumbFS" <1,000 lines of code, Complete, usable system <2,000 lines of code
- **Must manipulate call/reply of 20 RPC proceedures**
  - Encrypted files slightly larger, must adjust size in replies
  - All 20 RPC procedures can contain one more file sizes
  - RPC library lets CryptFS adjust 20 return types in 15 lines



Emacs compile
Execution Time (seconds)
0
100
200
300
Local
NFS3
cryptfs
CFS-async
CFS-sync
dumbfs

Conclusions

- **NFS allows portable, user-level file systems**
  - Translates non-portable VFS interface to standard protocol
- **In practice, loopback servers have had problems**
  - Low performance, blocked processes, deadlock, debugging difficulties, redundant, error-prone code,…
- **SFS toolkit makes most problems easy to avoid**
  - `nfsmounter` eliminates hangs after crashes
  - `libasync` supports complex programs that never block
  - `rpcc` allows concise manipulation of 20 call/return types
  - Stackable manipulators provide reusable NFS processing