Networked File System Project Plan

Bob Lyon

Company Confidential

ABSTRACT

Herein lies a first cut of the networked file system (nfs) project plan. The document contains a brief product description, an implementation schedule and immediate assignments for the project members.

Introduction

This file is pumpkinseed:~blyon/nfs/memos/nfs_project_plan.txt

Product Description

The networked file system allows network clients to gain byte level access to (shared) files on a remote system.

The nfs differs from nd (Sun's "network disk") in that nd only makes a disk (not a file system) available via a simple protocol. The nd client then builds its own fs given the disk. Access control of disk areas is based solely on the requester's IP address. Since IP addresses are assumed to be unique, this disallows file sharing by the nd server. The use of IP address as the basis of access control has two other draw backs: first, a malicious or buggy piece of network software can easily trash a user's disk just by supplying an IP address; second, it violates the protocol layering concepts and necessarily makes changing a client's IP address or client's nd server difficult.

Since the server only emulates a disk and not a file system, there is no caching on the server side.

The nfs differs from rcp (4.2 based remote copy) in that rcp only allows data transfer in units of files. The client of rcp supplies a complete file-path name and receives a stream of bytes in return; it does not use the file system to walk down the directory tree to its desired file. Access control is based on the client's name and the client's host's name. Rcp is not transparent to the user of system; nfs will be transparent.

The nfs will allow a client to read or modify data based on opaque file handles, byte offsets, and byte lengths. The file handles will be acquired via a series of get handle calls whose inputs are *directory* file handles and file names. The nfs (for the first time) presents the network client with a complete file system.

Access control will be (must be) done on every nfs server operation, though its impact on performance will be kept to a minimum. Clients will be identified via client's host machine name, client's (Unix) uid, and an array of the client's (Unix) gids. Uids and gids are relative the client's machine when generated but are relative to the server when used in access control. This implies that all machines that access or implement the nfs on an internet must have a uniform Unix uid and gid space. (Unlike nd or rcp, the rpc based nfs has authentication parameters formally separated from the server protocols (in this case, the nfs) so that newer/better/different forms of network authentication can be introduced with little or no impact on the protocol or clients.)

The nfs will augment various Unix caching philosophies in the following manners:

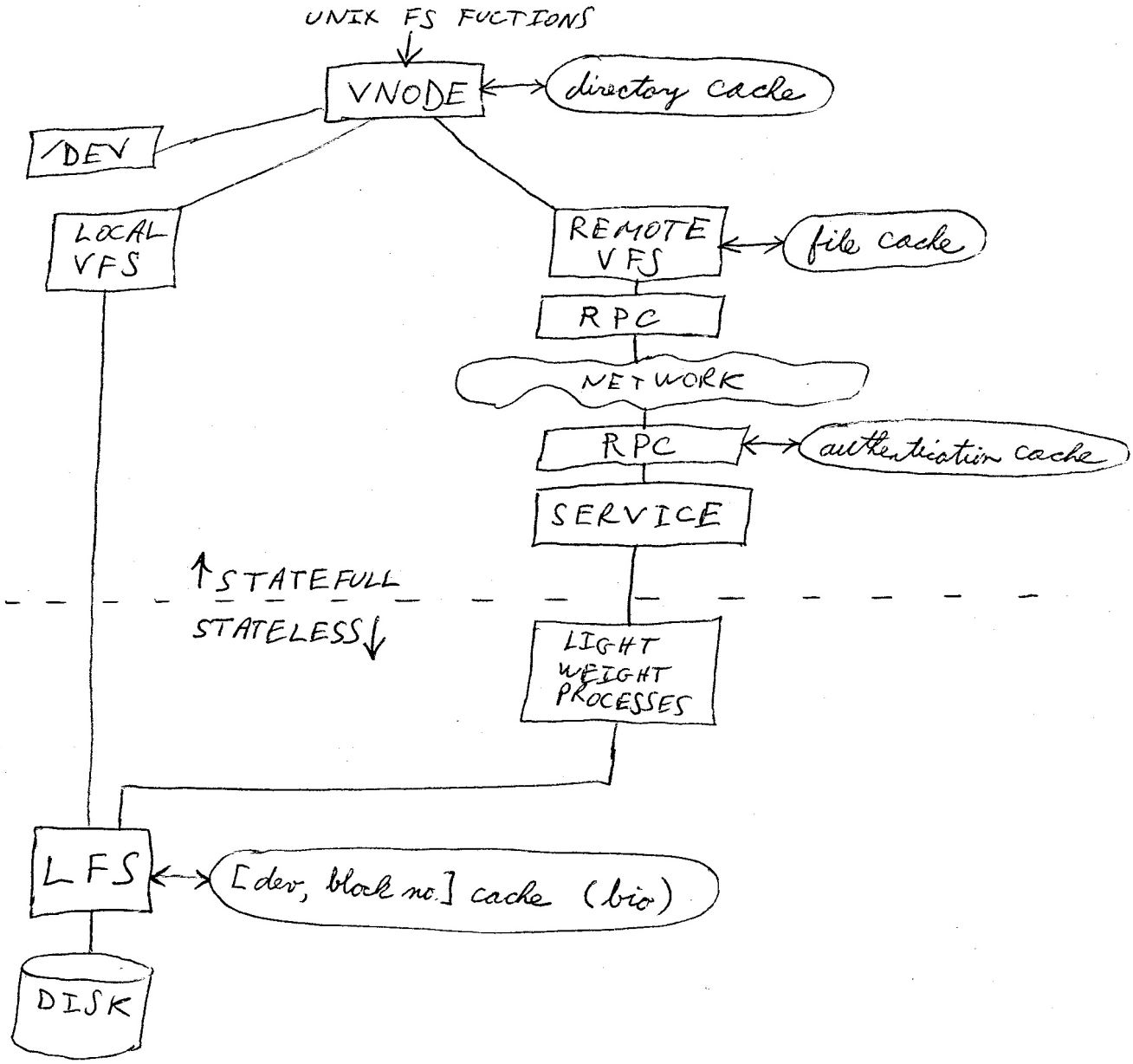
- The nfs client machines will continue to implement read ahead and write behind.

- The client nfs machines will keep an "internal node" directory cache in order to avoid repeated calls to lookup. The cache will be (in)validated when looking-up the last element of a path-name.
- The server will perform data read ahead and write through.
- The server will maintain a high performance authentication credentials cache that will allow fewer authentication bits to be passed per remote access.

Finally, it should be noted that the nfs is *not* a replacement for nd. In fact, nfs code must be able to coexist with nd.

Design (statements and issues)

The following is a fairly high level design of the nfs. The high level design is based upon Bill Joy's "inode evolves to vnode/vfs" file system proposal.



User view of deliverables

So far, this document has discussed the architecture of the nfs. This section discusses what the customer sees and how s/he uses it. Since the nfs is mostly transparent to C programs, using it should be trivial.

The delivered nfs will consist of a heavily modified Unix kernel (that implements *all* of the subsystems drawn in the figure above) along with various utility commands that enable the new functionality of the nfs. The functionality is best demonstrated via the following examples. The command names, syntax and parameters given below are only meant as examples are not to taken as the exact specifications of the end product.

Example 1: If a customer has booted his Unix machine and decides that the machine's file system should be made accessible to the internetwork, he would execute the following command:

```
exportfs 3
```

The command activates a nfs "server" and devotes three demons to the task of serving network clients.

The machine's kernel will now execute file system requests (reads, writes, ...) for network clients as if they were logged onto the machine. The clients are identified by uids and gids and access control will be based upon these ids. However, network clients with uids of zero ("root") will *not* be treated as the system's super user.

If the customer decides that his machine's file system should no longer be available to the internet, he would execute the command

```
unportfs
```

which (abruptly) turns off the nfs "server" on the machine. Any network clients who were accessing files on the machine will now fail. With respect to the network client, the effects of the unportfs command are very similar to those of a forced umount.

Some fine points are worth noting here:

- exportfs and unportfs are super user commands.

- It does not matter whether the machine is diskless (nd based) or diskfull; both configurations are compatible with the nfs server.

- It is believed that public, timeshared systems will export their file systems and that personal, high performance workstations will not, although the nfs design will not disallow the latter.

- Since personal machines (most likely) will not export their file system, (casual, non-transparent) access to their files can still be accomplished via rcp or rlogin.

- It is expected that the exportfs command will executed semi- automaticly following booting via init.

- The project will investigate and publish the optimal number of demons to devote to a dedicated nfs server, like Sun's titan machine.

Example 2: After booting a machine (with a small disk), a customer wishes to be able to access (read and write) source files on a remote server that has exported its file system. The remote machine's name is "krypton" and it keeps source on /usr/src. Furthermore, assume the machine needs a huge amount disk for /tmp and that the area is allocated on another server, "titan", in a directory /tmp/client_temps/client_foo. The customer may invoke the following commands:

```
netmount krypton:/usr/src /usr/src
netmount titan:/tmp/client_temps/client_foo /tmp
cp /usr/src/bin/ls.c /tmp
```

The netmount command lets a customer associate a local directory (/tmp) with a remote directory on a named server (titan:/tmp/client_temps/client_foo). After the netmount commands, all file system access is transparent to the user programs. In this example, the cp command effectively copied bits from krypton and placed them on titan (provided the user had appropriate

access rights). The commands

```
vi /usr/src/bin/cat.c
:g/cat/s/cat/dog/g
:wq

or

touch /usr/src/bin/cat.c
```

effectively edits the file in-place on krypton, though processing is done on the local machine.

Again, some fine points need noting:

- The machine may be diskless (nd based) or diskfull.

- In the first release of nfs, swap areas cannot be netmounted.

- Only super user may invoke the netmount command and the effects apply to the whole machine.

- It is assumed that the netmount command will be invoked after booting via init, though later netmounts are allowable.

Limitations (and what's broken)

Unlinking an open file will cause access to that file to be revoked. There is some number of Unix programs that will break due to this feature.

If a remote server becomes unavailable (due to a crash or an unportfs) the client software will receive a hard io error.

Actual inumbers will be made visible from the server to the client. However, inumbers will never be passed from client to server. Therefore, programs that inspect inumbers (like ls -i or cp) should continue to work. However, since inumbers only contain device and block number (and not nfs server number), comparison of inumbers is meaningless (cp could break). Some kind of nfs server number may be (... may have to be) hacked into inumbers.

The Unix authentication parameters are unencrypted and are fairly easy to intercept or forge.

The special id "root" will be treated as "other" on the file server. In some cases this will disallow accesses for remote superusers that are allowed for the normal user.

Remote devices will not be accessible through the nfs. The nfs will only deal with real files and not with special files. (This also means that swap areas will not be supported by the first release of the nfs.)

Programs that make unique files by checking for a certain file's existence and making it if it is not there could fail when using the nfs. This is not considered important since most of these files are created in private, non-networked file systems.

The project recognises the need for a lock manager. However, a lock manager service is considered outside the scope of the nfs, proper. No lock manager service will be delivered with the first released of the nfs.

Nfs performance expectations should be tempered. Performance equal to nd is desirable. But given the added functionality of nfs, this goal is not very realistic. The goal could be reached if all tuning work on nd stops immediately and much effort is poured into nfs performance. The project expects to do the latter.

The nfs authentication and access control assumes a uniform Unix uid and gid space. This assumption can make administration of large internets very difficult. Connection of two previously disjoint internets will cause an administrative nightmare. This form of authentication was chosen so that the nfs can be delivered in a reasonable time frame in the absence of internetwork-wide authentication.

Schedule

A time-less schedule of major tasks and their dependencies is presented in this section. A time line for these tasks is presented at the end of this document. Although caches and caching strategies are very important in the end product cache design and implementation is *not* included in the various task. (Note that cache design is not explicitly excluded, either.) The schedule reflects the project members' desire to show feasibility ASAP without necessarily addressing performance at the same time.

a: Implement and debug vnode/local/lfs system in a Unix user process using a raw file system.

a1: Well defined interface to LFS must be specified.

a2: Well defined interface between vnode and *vfs must be specified.

b: (depends on a) Port vnode/local/lfs system to the kernel.

c: Specify and implement Unix style authentication parameters.

d: Port rpc to the kernel.

e: Choose a transport protocol for the nfs.

f: Specify client - server nfs protocol.

g: Design and implement kernel-level-light-weight / user-level-regular processes.

h: (depends on a1, c, e, f) Implement NFS Service as a user process.

i: (depends on h, g) Port NFS Service to the kernel and take advantage of light-weight processes.

j: (depends on a, a2, c, e, f) Implement NFS Client ("remote vnodes") in a user process.

k: (depends on b, j) Port NFS Client to the kernel.

w: Adapt to any nfs-project-external kernel system changes (like the kernel VM rework).

x: Fix parts of the kernel that break due to (a). For example, shared text uses bio which is bound to change.

y: On-going transport protocol performance improvements.

z: Implement and retro-fit any caches not implemented by any other tasks.

Immediate assignments

The following is a list of sub-tasks, their short description, who the task is assigned to and for how long, and what (if any) issues prevent the task from being completed. Time estimates are best guesses and assume that the person devotes 100% of his time to the task.

1) DFS Protocol Specification (pugs). This document describes all "remote procedures" implemented by a nfs server and used by a client. The spec is written in English and Sun-NDR. It describes all procedures' arguments, results and semantics. Time: 2 weeks. Dependency: Due to performance considerations, the choice of kernel-level rpc transport protocol will affect this higher level protocol (see # 5). (The transport protocol choice should only affect the specifications of two procedures - read and write.)

2) RPC kernel port (dan and rusty with help from blyon). The implementation of RPC/UDP/IP must be ported from Unix user space to kernel space. (RPC is composed of three connected packages - ndr, authentication and rpc_message. Each package has a client side and a service side.) After the port some performance measurement must be done to discover any performance crocks and to temper our final performance expectations. Issue: The user - kernel interface to the kernel-based RPC is not well understood. Writing a performance measurement program should force this issue. Time: 4 weeks. Dependencies: None; however, "Unix authentication" will eventually affect some of the code (see # 4).

3) Unix user level / raw fs, vnode implementation (srk and pugs). The first cut of the inode evolves to vnode implementation will be written and debugged at Unix user level; it will access a raw file system. The implementation will then be ported to the kernel. The task has three major pieces - vnode level (replaces inode and knows about various file system implementations), lfs (a

local disk file system), and device access (a local device file system). Time: 13 weeks, include kernel port. No dependencies.

4) Unix Authentication (blyon). Unix uids and gids must be passed from client to server in order for the server to perform file access control. The server then returns to the client a short-hand handle which the user can use on subsequent calls. This just involves writing a new flavor of authentication for rpc clients and rpc servers. As usual it will first be implemented at user level, then ported to the kernel (see task # 2). Time: 2 weeks. No dependencies.

5) Alternative transport for nfs rpc (pugs, dan, rusty, blyon). UDP may not be the best way to move data among clients and servers. Since this affects # 1, the issues must be resolved in due haste. Time: 1 weeks. No dependencies.

6) dg place holder. David Goldberg has no tasks directly connected with the networked file system. We hope to use David's talents in related areas such as (a) using user-level rpc and providing blyon with feed-back, (b) producing better rpc/ndr documentation, (c) providing the yellow pages which is needed long term, (d) investigating remote system performance monitors, (e) on-going internetwork routing issues, and (f) provide nfs help wherever David or management deem necessary.

Project Requirements

1) Managing source changes to the Unix kernel will be hard enough just within the nfs project. We do not wish to coordinate with other projects (such as the memory management rework). Therefore an additional integration (timesharing / filesharing) machine, "Dufus" is needed. Dufus source will be copied from Krypton (the current integration machine) *after* the merge of frozen-out source code with 1.1 source code. The "join" of Dufus and Krypton is not well understood at this point.

Anticipated disk storage requirement for Dufus is one 330 OR two 168's OR one Eagle.

2) When the vnode - lfs implementation is ported to the kernel, the porters will require machines with their own local disks that are *trashable*. It is believed that Steve Kleiman will require his own disk-full machine. Pugs and dan currently have disk-full machines; blyon, dg and rusty have diskless machines.

3) Some stages of testing and performance measurement will require "lab" machines. The project members assume that two SQA lab machines will always be available to them. Lab conflicts with other projects can be resolved by acquiring more hardware or delaying one or both projects.

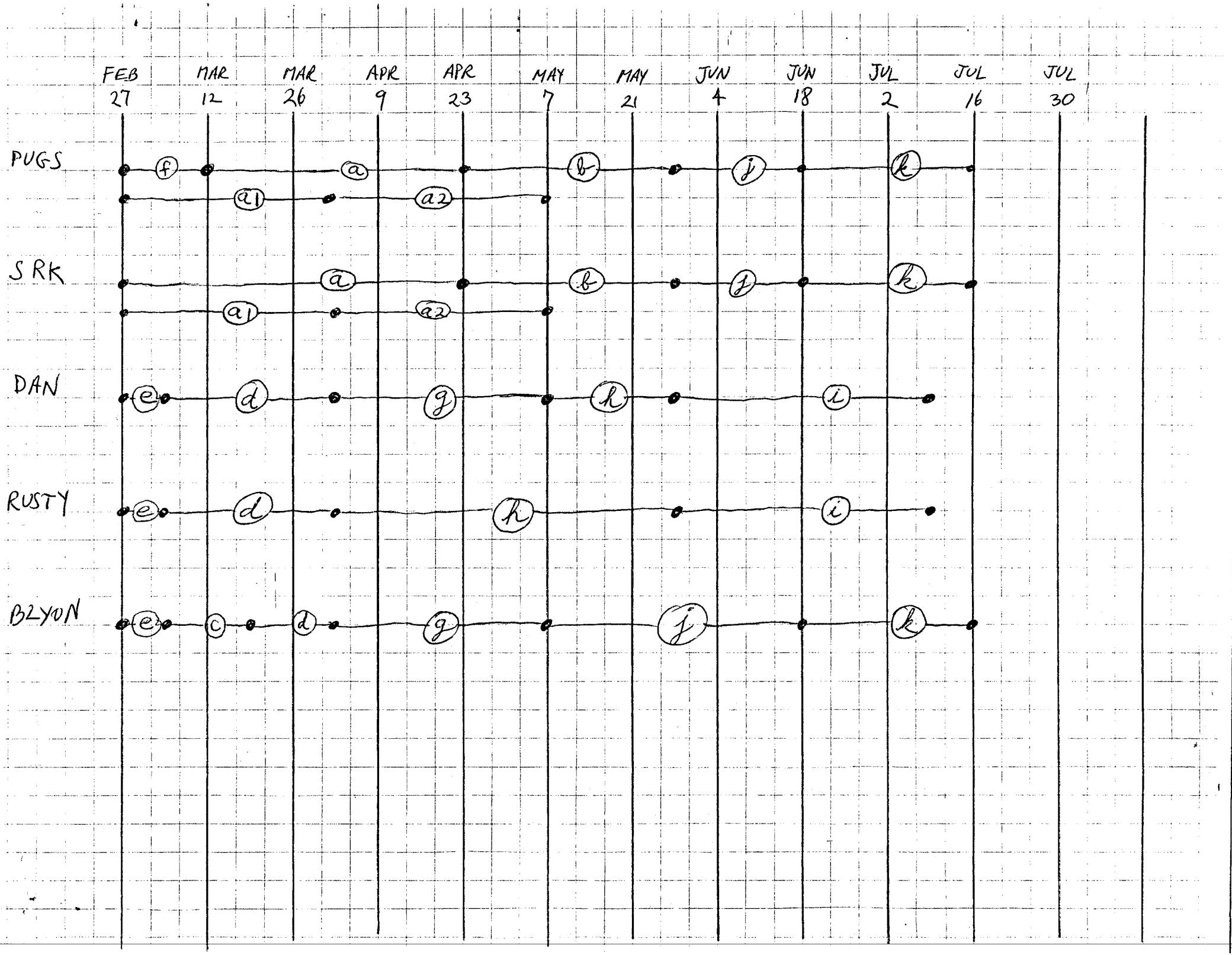
4) As more pieces of the nfs get implemented performance measurement and analysis become increasingly more important. In many cases, "gprofing the kernel" will be adequate. However, this has the unfortunate side effects of consuming a non-trivial amount of cpu time and providing an all-or-nothing approach to monitoring. The project members strongly recommend that Sun acquire a hardware monitor so that critical legs of the nfs kernel can be measured without interference.

A Modest Proposal

Blyon suggests that each project member take time once a month to send a "progress report" to management and all project members. The report should be an electronic message that takes no more than a page of paper to print and should not be composed via *troff. No more than one hour should be devoted to the message.

The message has three sections - Progress, Plan, and Problems. The composer should state what he has accomplished since his last report, what he intends to accomplish in the next month, and what prevented (or prevents) him from making progress.

Digestion of several month of reports typically uncovers problem areas that are never immediately apparent.


FEB 27
MAR 12
MAR 26
APR 9
APR 23
MAY 7
MAY 21
JUN 4
JUN 18
JUL 2
JUL 16
JUL 30
PUGS
SRK
DAN
RUSTY
BZYON