Arla

A Free AFS implementation from KTH
Edition 0.1, for version 0.34
1999 - 2000

Love Hörnquist-Åstrand
Assar Westerlund
Harald Barth
last updated $Date: 2000/10/01 19:18:41 $

Table of Contents

1 Introduction

> **Caution:** Parts of this package are not yet stable software. If something doesn't work, it's probably because it doesn't. If you don't have backup of your data, take backup.

1.1 What is Arla?

Arla is a free AFS implementation. Some of the goals are:

- to have an implementation that is free and can be used for adding and playing with cool stuff, like support for disconnected-mode. Implement features you can't get from commercial AFS.
- to provide an alternative to Transarc's AFS-client and server implementations.
- to add support for platfroms that don't have AFS support from Transarc today.

This release is known to work on the following platforms: NetBSD, OpenBSD, FreeBSD, Linux, Solaris, Darwin/MacOS X.

Earlier releases are known to work on current or earlier versions of the following platforms: SunOS, AIX, IRIX, Digital UNIX. Some fixes might be necessary to make Arla work.

There is or has been done work to support the following platforms: HPUX, Fujitsu UXP/V, . Some development is necessary to make Arla work.

There is work going on to support the following platform: Windows NT/2000. Contributions are very welcome.

1.2 Status

Arla has the following features (quality varies between stable and not implemented):

- a rxgen implementation called ydr (stable).
- a cache manager replacing Transarc's afsd. The cache managers quality depends on platform: *BSD, Linux i386 and Solaris are stable, others platforms are not as tested ane therefore not as stable.
- partly implemented fs, vos, pts commands. Commands typically issued by users are stable, commands issued by administrators may return unmotivated errors or are not implemented yet.
- an implementaion of rxkad written outside USA without any export restrictions (stable).
- a server implementation called milko, containing file server, volume server and protection server. The file server has an API to the storage backend(s). Milko is still unstable and not fit for production yet.

1.3 Bug reports

If you find bugs in this software, make sure it is a genuine bug and not just a part of the code that isn't implemented.

Bug reports should be sent to `arla-drinkers@stacken.kth.se`. Please include information on what machine and operating system (including version) you are running, what you are trying to do, what happens, what you think should have happened, an example for us to repeat, the

output you get when trying the example, and a patch for the problem if you have one. Please make any patches with `diff -u` or `diff -c`.

Suggestions, comments and other non bug reports are also welcome.

1.4 Mailing list

There are two mailing lists with talk about Arla. `arla-announce@stacken.kth.se` is a low-volume announcement list, while `arla-drinkers@stacken.kth.se` is for general discussion. There is also commit list `arla-commit@stacken.kth.se`. Send a message to `LIST-request@stacken.kth.se` to subscribe.

2 Description of AFS infrastructure

This is an overview of the AFS infrastructure as viewed from a Transarc perspective, since most people still run Transarc cells.

2.1 Server overview

AFS filespace is split up in smaller parts called cells. These cells are usually listed under '`/afs`'. A cell is usually a whole organization or an adminstative unit within an organization. An example is e.kth.se (with the path '`/afs/e.kth.se`'), that is the department of electrical engineering at KTH, which obviously has the '`e.kth.se`' domain in DNS. Using DNS domains for cell names is the typical and most convenient way.

All cells (and their db-servers) in the AFS world are listed in a file named '`CellServDB`'. There is a central copy that is maintained by Transarc at '`/afs/transarc.com/service/etc/CellServDB`'. In addition Arla can use DNS to find the db-servers of a cell. The DNS resource record that is used is the AFSDB. It tells you what machines are db servers for a particular cell. The AFSDB resource record is also used for DCE/DFS. An example (the 1 means AFS, 2 is used for DCE):

```
e.kth.se.               IN AFSDB     1 sonen.e.kth.se.
e.kth.se.               IN AFSDB     1 anden.e.kth.se.
e.kth.se.               IN AFSDB     1 fadern.e.kth.se.
```

Some cells use the abbreviated version '`/afs/<word-before-first-dot>`' (in the example above that would be '`/afs/e/`'. This might be convenient when typing them, but is a bad idea, because it does not create the same name space everywhere. If you create a symbolic link to '`/afs/e/foo/bar`', it will not work for people in other cells.

Note that cell names are written in lowercase by convention.

There are several servers running in an AFS cell. For performance and redundancy reasons, these servers are often run on different hosts. There is a built in hierarchy within the servers (in two different dimensions).

There is one server that keeps track of the other servers within a host, restart them when they die, make sure they run in the correct order, save their core-files when they crash, and provide an interface for the sysadmin to start/stop/restart the servers. This server is called bos-server (Basic Overseer Server).

Another hierarchy is the one who keeps track of data (volumes, users, passwords, etc) and who is performing the real hard work (serving files) There is the the database server that keeps the database (obviously), and keeps several database copies on different hosts relpicated with Ubik (see below). The fileserver and the client software (like the afsd/arlad, pts and, vos) are pulling meta-data out of the dbserver to find where to find user-privileges and where volumes resides.

2.2 Basic overseer - boserver

The Bos server is making sure the servers are running. If they crash, it saves the corefile, and starts a new server. It also makes sure that servers/services that are not supposted to run at the same time do not. An example of this is the fileserver/volserver and salvage. It would be devastating if salvage tried to correct data that the fileserver is changing. The salvager is run before the fileserver starts. The administrator can also force a file server to run through salvage again.

2.3 Ubik

Ubik is a distributed database. It is really a (distributed) flat file that you can perform read/write/lseek operation on. The important property of Ubik is that it provides a way to make sure that updates are done once (transactions), and that the database is kept consistent. It also provides read-only access to the database when there is one (or more) available database-server(s).

This works the following way: A newly booted server sends out a message to all other servers that tells them that it believes that it is the new master server. If the server gets a notice back from an other server that tells it that the other server believes that it (or a third server) is the master, depending on how long it has been masterserver it will switch to the new server. If they can't agree, the one with the lowest ip-address is supposed to win the argument. If the server is a slave it still updates the database to the current version of the database.

A update to the database can only be done if more than half of the servers are available and vote for the master. A update is first propaged to all servers, then after that is done, and if all servers agree with the change, a commit message is sent out from the server, and the update is written to disk and the serial number of the database is increased.

All servers in AFS use Ubik to store their data.

2.4 Volume Location database server - vlserver

The vldb-server is resposible for the information on what fileserver every volume resides and of what kind of volumes exists on each fileserver.

To confuse you even more there are three types of support for the clients. Basically there is AFS 3.3, 3.4, and 3.6 support. The different interfaces look the same for the system administrator, but there are some important differences.

AFS 3.3 is the classic interface. 3.4 adds the possibility of multihomed servers for the client to talk to, and that introduces the N interface. To deal with multihomed clients AFS 3.5 was introduced. This we call the U interface.

The N interface added more replication-sites in the database-entry structure. The U interface changed the server and clients in two ways.

Now a server registers all its ip-addresses when it boots. This means that a server can add (or remove) an network interface without rebooting. When registering at the vldb server, the file server presents itself with an UUID, an unique identifier. This UUID will be stored in a file so the UUID keeps constant even when network addresses are changed, added, or removed.

2.5 Protection server - ptserver

The protection server keeps track of all users and groups. It's used a lot by the file servers.

2.6 Kerberos server - kaserver

The kaserver is a Kerberos server, but in other clothes. There is a new RPC interface to get tickets (tokens) and administer the server. The old Kerberos v4 interface is also implemented, and can be used by ordinary Kerberos v4 clients.

You can replace this server with an Heimdal kdc, since it provides a superset of the functionality.

2.7 Backup server - buserver

The backup server keeps the backup database that is used when backing up and restoring volumes. The backup server is not used by other servers, only operators.

2.8 Update server - upserver

With the update server its possible to automagicly update configuration files, server binaries. You keep masters that are supposed to contain the correct copy of all the files and then other servers can fetch them from there.

2.9 Fileserver and Volume Server - fs and volser

The file server serves data to the clients, keeps track of callbacks, and breaks callbacks when needed. Volser is the administative interface where you add, move, change, and delete volumes from the server.

The volume server and file server are ran at the same time and they sync with each other to make sure that fileserver does not access a volume that volser is about to change.

2.10 Salvage

Salvage is not a real server. It is run before the fileserver and volser are started to make sure the partitions are consistent.

It's imperative that salvager is NOT run at the same time as the fileserver/volser is running.

2.11 Things that milko does differently.

Fileserver, volumeserver, and salvage are all in one program.

There is no bu nor ka-server. The ka-server is replaced by kth-krb or Heimdal. Heimdal's kdc even implements a ka-server readonly interface, so your users can keep using programs like klog.

3 Organization of data

This chapter how data is stored and how AFS diffrent from, for example, NFS. It also describes how data kept consistent and what the requirements was and how that inpacted on the design.

3.1 Requirements

- Scalability
  It should be possible to use AFS with hundred-thousands of users without problems.
  Writes that are done to diffrent parts of the filesystem should not affect each other. It should be possible to distribute out the reads and writes over many file-servers. So if you have a file that is accessed by many clients, it should be possible to distribute out the load.
  If there is multiple writes to the same file, are you sure that isn't a database.
- Transparent to users
  Users should not need to know where their files are stored. It should be possible to move their files while they are using their files.
- Easy to admin
  It should be easy for a administrator to make changes to the filesystem. For example to change quota for a user or project. It should also be possible to move the users data for a fileserver to a less loaded one, or one with more diskspace available.
  Some benefits of using AFS is:
  - user-transparent data migration
  - an ability for on-line backups;
  - data replication that provides both load balancing and robustness of critical data
  - global name space without automounters and other add-ons;
  - @sys variables for platform-independent paths to binary location;
  - enhanced security;
  - client-side caching;

3.2 Anti-requirements

- No databases
  AFS isn't constructed for storing databases. It would be possible to use AFS for storing a database if a layer above provided locking and synchronizing of data.
  One of the problems is that AFS doesn't include mandatory byte-range locks. AFS uses advisory locking on whole files.
  If you need a real database, use one, they are much more efficent on solving a database problem. Don't use AFS.

3.3 Volume

A volume is a unit that is smaller then a partition. Its usually (should be) a well defined area, like a user's home directory, a project work area, or a program distribution.

Quota is controlled on volume-level. All day-to-day management are done on volumes.

3.4 Partition

In AFS a partition is what normally is named a partition. All partions that afs isusing is named a special way, '`/vicepNN`', where NN is ranged from a to z, continuing with aa to zz. The fileserver (and volser) automaticly picks upp all partition starting with '`/vicep`'

Volumes are stored in a partition. Volumes can't overlap partitions. Partitions are added when the fileserver is created or when a new disk is added to a filesystem.

3.5 Volume cloning and read-only clones

A clone of volume is often needed for the volume operations. A clone is copy-on-write copy of a volume, the clone is the read-only version.

A two special versions of a clone is the read-only volume and the backup volume. The read-only volume is a snapshot of a read-write volume (that is what a clone is) that can be replicated to several fileserver to distribute the load. Each fileserver plus partition where the read-only is located is called a replication-site.

The backup volume is a clone that typically is made (with `vos backupsys`) each night to enable the user to retrieve yestoday's data when they happen to remove a file. This is a very useful feature, since it lessen the load on the system-administrators to restore files from backup. The volume is usually mounted in the root user's home directory under the name OldFiles. A special feature of the backup volume is that inside it you can't follow mountpoint.

3.6 Mountpoints

The volumes are independent of each other. To clue the together there is a '`mountpoint`'s. Mountpoints are really symlink that is formated a special way that points out a volume (and a optional cell). A AFS-cache-manager will show a mountpoint as directory, in fact it will be the root directory of the target volume.

3.7 Callbacks

Callbacks are what enable the AFS-cache-manager to keep the files without asking the server if there is newer version of the file.

A callback is a promise from the fileserver that it will notify the client if the file (or directory) changes within the timelimit of the callback.

For read-only callbacks there is only callback given its called a volume callback and it will be broken when the read-only volume is updated.

The time range of callbacks range from 1 hour to 5 minutes depending of how many user of the file exists.

3.8 Volume management

All volume managment is done with the vos-command. To get a list of all commands 'vos help' can be used. For help on a specific vos subcommand, 'vos subcommand -h' can be used.

- Create

```
vos create mim c HO.staff.lha.fluff -quota 400000
```

- Move

  Volumes can be moved from a server to another, even when users are using the volume.
- Replicate

  Read-only volumes can be replicated over several servers, they are first added with `vos addsite`, and the replicated with `vos release` out over the servers.
- Release

  When you want to distribute out the changes in the readwrite volume.
- Remove

  Volumes can be removed

  Note that you shouldn't remove the last readonly volume since this make clients misbehave. If you are moving the volume you should rather add a new RO to the new server and then remove it from the old server.
- Backup and restoration of volumes.

  `vos backup` and `vos backupsys` creates the backup volume.

  To stream a volume out to a '`file`' or '`stdout`' you use `vos dump`. The opposite command is named `vos restore`.

3.9 Relationship between pts uid and unix uid

foo

4 Parts of Arla

> **Caution:** This text just tries to give a general picture. For real info read the code. If you have any questions, mail `arla-drinkers@stacken.kth.se`.

How does arla work

Arla consists of two parts, a userland process (arlad) and the kernel-module (xfs).

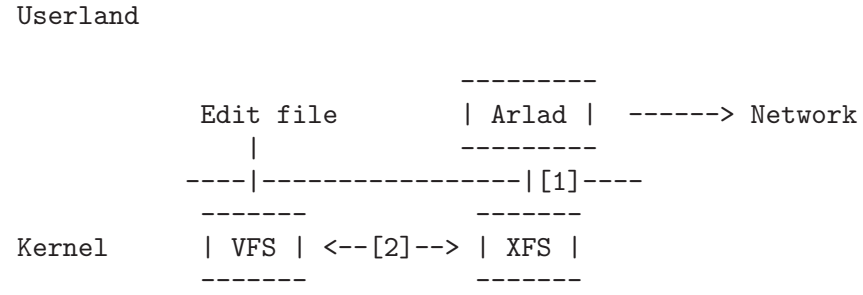
Arlad is written in user-space for simpler debugging (and less rebooting). As a uset space program arlad does not have the same limitations as if it would be written in the kernel. To avoid performance loss as much as possible, xfs is caching data.

xfs and arlad communicate with each other via a char-device-driver. There is a rpc-protocol currenly used specially written for this (‘`arlad/message.c`’)

xfs is written to be as simple as possible. Theoretically, xfs could be used by other user-space daemons to implement a file system. Some parts, such as syscalls, are arla-specific. These parts are designed to be as general as possible.

For example, xfs does not recognize which pioctl the user-level program calls, it just passes this information on to arlad.

The relation between Arlad and XFS



[1] A char device (/dev/xfs0)
[2] Xfs provides a filesystem for the vfs-layer in the operating system.

4.1 The life of a file

Step by step description of what happens during the creation of a file. The names are inspired of BSD-style VFS-layer but the idea is the same in most operating systems.

- The user decides to open a file.
- open(2) syscall is issued.
- The vfslayer sends a VOP_LOOKUP to xfs that is forwarded to arlad with a getnode() (seq-num 1).
- arlad tries to find the requested file and then, if found, sends an install_node to xfs by writing to the xfs character device.

- xfs inserts the node into the cache and returns from the device write.
- arlad sends a wakeup rpc message (seq-num 1) to xfs. If the return value is zero xfs tries to find the node in the cache, if not found it might have been flushed out of the cache and the whole thing is repeated.
- If a none-zero return value is returned, this value is sent as reply to the user. This way arla can decide what error message is returned, without xfs having support for each error.
- xfs now checks if it has the valid attributes. If the attributes are invalid, xfs will send a rpc message to arlad to refresh it.
- Since the user wanted to open the file, a getdata rpc message is sent from xfs to arlad. Now arlad fetches the files from the afs file server.
- Arlad stores the file in the file cache. All vnode operations will be done on this file. Now arlad sends a installdata to xfs.
- When xfs recives the installdata it looks up the node in the cache, and then it does a VOP_LOOKUP to find a vnode to the cachefile (and store it to keep it for future use).
- The same thing is done when the file is a directory, except that the directory is converted from the afs directory format to an operating system dependent format and stored in a file. xfs reads this file instead.
- If the directory is modified locally, write operations are done on the file obtained from the afs-server, and when done the newly changed file is converted and reinstalled.
- Now the user wants to read a file.
- read(2) system call is issued.
- A VOP_READ is sent to the from the vfs-layer to xfs.
- xfs checks if it has valid attributes/and data (and updates if needed). Now VOP_READ is simply performed on the stored vnode of the cachefile.

Tools and libs

What other tools does the arla suite consists of

libutil: `util/libutil.a` - A library for the most often used
modules like hashtable, double-linked list, logging functions, date-parsing, etc

rx: `rx/librx.a` - The library for the rx protocol
(see Section 4.2 [Rx protocol], page 11).

lwp: `lwp/liblwp.a` - The library for the lwp thread-package
(see Section 4.3 [LWP], page 11).

ydr: `ydr/ydr` - A stub generator that replaces rxgen.

rxkad: `rxkad/librxkad.a` - The rx Kerberos authentication package.

roken: `lib/roken/libroken.a` - The library that will unbreak
things that are missing or broken.

ko: `lib/ko/libko.a` - A library of functions that are arlad-core
related but also are useful for programs like vos, pts, fs, etc.

arlalib: `appl/lib/libarlalib.a` - A broken library that does all
the hard work with connections etc.

fs: `appl/fs/fs` - The fs util, extra feature
(amongst others): getfid.

vos: `appl/vos/vos` - The vos util.

pts: `appl/pts/pts` - The pts util, extra feature: dump.

udebug: `appl/udebug/udebug` - Debug your ubik server.

4.2 Rx protocol

Rx is run over UDP.

One of rxgen or ydr is used to generate stub-files, ydr is better since it generates prototypes, too.

The current implemetation of rx it not that beautiful.

4.3 LWP

LWP is a preepmtive thread package. It does it's context-switching by creating a private stack for each thread. The heart of the package is select(2).

The core of the package is '`process.S`'. Its currenty built with a bourne-shell file. '`Process.s`' is preprocessed with `cpp` and then given to (g)as. The magic how to build this file a portable way is not fun.

The stack is checked for overruns in context-switches, but that is often too late. It might be an idea to add a *red zone* at the top of the stack to be able to detect overruns.

For architectures not supported by LWP, it can be exchanged with the pthreads pakage.

4.4 The files in arlad/

This is a short describtion of the files to bring new deveplopers up to speed.

4.4.1 The core of arlad

'`adir.c`' - contains all functions needed to to operations
on afs-directory files.

'`afsdir_check.c`' - check if an AFS-directory looks sane.

'`arla.c`' - The startup and the standalone (-t) code.

'`arladeb.c`' - The logging code specific to arla, like aliases
for debugging masks.

'`cmcb.c`' - The callback-server that is contacted by the
server when a callback expires or a server wants to send an InitCallBackState.

'`conn.c`' - The connection cache, responsible for caching connection
based on pag and security index. It will also create new connection when needed.

'`cred.c`' - Keep track of all credentials that all users have
inserted. Indexed on pag.

'`fbuf.c`' - An interface between rx and filedescriptors. It is also
used to mmap files. Used by '`adir.c`'.

'`fcache.c`' - Responsible for keeping track of files in the cache.
Also fetches files from the afs-server.

'`fprio.c`' - Tries to give files priority. These files are
therefore not garbarge-collected as fast as they would be otherwise. If you wonder
what this is for, think of the disconnected mode.

‘inter.c’ - An interface to hide all junk in fcache, just give
the items a VenusFid and you can access them this way.

‘kernel.c’ - The interface between arlad and the char-device.

‘messages.c’ - The rpc interface between arlad and xfs.

‘volcache.c’ - Cache for all volumes.

4.4.2 Operating system specific files

These are the files that contain operating specific functions. Today it’s just conv_dir().

‘aix-subr.c’ - AIX

‘bsd-subr.c’ - FreeBSD 2.2.6, OpenBSD 2.2, 2.3, NetBSD 1.3.x

‘hpux-subr.c’ - HPUX

‘irix-subr.c’ - Irix

‘linux-subr.c’ - Linux 2.0.x, 2.1.x, 2.2

‘solaris-subr.c’ - Solaris 2.5.x, 2.6, 7

‘sunos-subr.c’ - SunOS

‘unknown-subr.c’ - Stub used when compiled on a unknown OS.

pioctl and kafs

The pioctl interface is the only part of xfs that is afs related.

pioctl is a ioctl but called with a path instead of a filedescriptor. When you probe if there is a live afsclient you first run `k_hasafs()` that probes if there is an afsclient around. It also sets up some static variables in the library. So if you start to do `pioctl()` w/o running `k_hasafs()`, you’re up to funny errors, and/or get a corefile.

`k_hasafs()` does an `AFSCALL_PIOCTL` with opcode `VIOCSETTOK` and insize == 0, ie you try to set a token (ticket) that is 0 bytes long. This is cleary invalid and kafs expects to find an `EINVAL` returned from `syscall(2)`.

The pioctl is used more then just for `AFSCALL_PIOCTL`, an other use is `AFSCALL_SETPAG` (setting pag). It has also been in use for setting xfs debugging levels.

When xfs discovers that a path is given in the `pioctl()` it does a `VOP_LOOKUP` on the path and if the returned value is a vnode that resides in afs then it extracts the xfs-handle for that node (that just happens to be the VenusFid) and passes that on to arlad.

The only ugly thing about the current implentation is that the syscall code assumes that the arlad on "xfs-fd" is the arlad that should get this syscall.

An example of using `pioctl()`:

```
int
fs_getfilecellname(char *path, char *cell, size_t len)
{
    struct ViceIoctl a_params;

    a_params.in_size=0;
    a_params.out_size=len;
    a_params.in=NULL;
    a_params.out=cell;
```

```
    if (k_pioctl(path,VIOC_FILE_CELL_NAME,&a_params,1) == -1)
        return errno;

    return 0;
}

int
main (int argc, char **argv)
{
    char cell[100];

    if (!k_hasafs())
       errx (1, "there is no afs");

    if (fs_getfilecellname (".", cell, sizeof(cell)))
       errx (1, "fs_getfilecellname failed");

    printf ("cell for `.' is %s", cell);
    return 0;
}
```

5 Debugging

This chapter of the manual includes tips that are useful when debugging arla.

Arla and xfs contains logging facilities that is quite useful when debugging when something goes wrong. This and some kernel debugging tips are described.

5.1 Arlad

If arlad is run without any arguments arlad will fork(2) and log to syslog(3). To disable forking use the –no-fork (-n) switch. In the current state of the code, arlad is allways to be started with the recover (-z) switch. This will invalidate your cache at startup. This restriction may be dropped in the future.

To enable more debuggning run arla with the switch –debug=module1,module2,... One useful combination is

```
--debug=all,-cleaner
```

The cleaner output is usully not that intresting and can be ignored.

A convenient way to debug arlad is to start it inside gdb.

```
datan:~# gdb /usr/arla/libexec/arlad
(gdb) run -z -n
```

This gives you the opportunity to examine a crashed arlad.

```
(gdb) bt
```

The arla team appreciates cut and paste information from the beginning to the end of the bt output from such a gdb run.

To set the debugging with a running arlad use `fs arladeb` as root.

```
datan:~# fs arladeb
arladebug is: none
datan:~# fs arladeb almost-all
datan:~#
```

By default, arlad logs through syslog if running as a daemon and to stderr when running in the foreground (with *--no-fork*).

5.2 Debugging LWP with GDB

For easy tracing of threads we have patch (http://www.stacken.kth.se/projekt/arla/gdb-4.18-backfr) for gdb 4.18 (a new command) and a gdb sequence (think script).

The sequence only works for i386, but its just matter of choosing different offset in topstack to find $fp and $pc in the lwp_ps_internal part of the sequence.

You should copy the '`.gdbinit`' (that you can find in the arlad directory in the source-code) to your home-directory, the directory from where you startat the patched gdb or use flag -x to gdb.

Your debugging session might look like this:

```
(gdb) lwp_ps
Runnable[0]
 name: IO MANAGER
  eventlist:
  fp: 0x806aac4
  pc: 0x806aac4
 name: producer
  eventlist: 8048b00
  fp: 0x8083b40
  pc: 0x8083b40
Runnable[1]
[...]
(gdb) help backfrom
Print backtrace of FRAMEPOINTER and PROGRAMCOUNTER.

(gdb) backfrom 0x8083b40 0x8083b40
#0  0x8083b40 in ?? ()
#1  0x8049e2f in LWP_MwaitProcess (wcount=1, evlist=0x8083b70)
    at /afs/e.kth.se/home/staff/lha/src/cvs/arla-foo/lwp/lwp.c:567
#2  0x8049eaf in LWP_WaitProcess (event=0x8048b00)
    at /afs/e.kth.se/home/staff/lha/src/cvs/arla-foo/lwp/lwp.c:585
#3  0x8048b12 in Producer (foo=0x0)
    at /afs/e.kth.se/home/staff/lha/src/cvs/arla-foo/lwp/testlwp.c:76
#4  0x804a00c in Create_Process_Part2 ()
    at /afs/e.kth.se/home/staff/lha/src/cvs/arla-foo/lwp/lwp.c:629
#5  0xfffefdfc in ?? ()
#6  0x8051980 in ?? ()
```

There also the possibility to run arla with pthreads (run configure with –with-pthreads).

5.3 xfs

XFS debugging does almost look the same on all platforms. They all share same debugging flags, but not all are enabled on all platforms.

Change the debugging with the *`fs xfsdebug`* command.

```
datan:~# fs xfsdebug
xfsdebug is: none
datan:~# fs xfsdebug almost-all
datan:~#
```

If it crashes before you have an opportunity to set the debug level, you will have to edit '`xfs/`*your-os*`/xfs_deb.c`' and recompile.

The logging of xfs ends up in your syslog. Syslog usully logs to /var/log or /var/adm (look in /etc/syslog.conf).

5.4 xfs on linux

There is a problem with klogd, it's too slow. Cat the '`/proc/kmsg`' file instead. Remember to kill klogd, since the reader will delete the text from the ring-bufer, and you will only get some of the message in your cat.

5.5 Kernel debugging on BSD

Kernel debugging on NetBSD, OpenBSD, and FreeBSD are almost same. You get the idea when when looking at the exampel from NetBSD below.

```
(gdb) file netbsd.N
(gdb) target kcore netbsd.N.core
(gdb) symbol-file /sys/arch/i386/compile/NUTCRACKER/netbsd.gdb
```

If you want to use the symbols of xfs, there is a gdb command called add-to-symboltable that is useful. The symbol file is obtained by loading the kernel module xfs with kmodload -o /tmp/xfs-sym. This in only used for NetBSD and OpenBSD. FreeBSD have symolparser in the kernel module (elf)loader, no one of this magic is needed. The addres can you get from lkmstat (it's the area field).

You should also source the commands in /sys/gdbscripts to get commands like xps (ps inside gdb).

```
: lha@nutcracker ; modstat Type Id Off Loadaddr Size Info Rev Module
Name DEV 0 29 ce37f000 002c ce388fc0 1 xfs_mod [...]
[...]
(gdb) add-symbol-table xfs.sym ce37f000
```

One of diffrencies between the BSD's are the `proc`, a command that enables you do to a backtrace on all proc. On FreeBSD you give the `proc` command a '`pid`', but on NetBSD and OpenBSD you give a pointer to a `struct proc`.

An important thing to know is that you can debug a live kernel too, this can be useful to find deadlocks. To attach to a kernel you use a command like this:

```
(gdb) file /netbsd
(gdb) target kcore /dev/mem
(gdb) symbol-file /sys/arch/i386/compile/NUTCRACKER/netbsd.gdb
```

5.6 Darwin/MacOS X

You'll need two computers to debug arlad/xfs on darwin since one use remove kernel debugger over ip.

First you need to publish the arp-address of the computer that you are going to crash.

We haven't found any kernel symbols with MacOSX Public Beta, so you should probably build your own kernel. Use Darwin xnu kernel source with cvs-tag: Apple-103 (not xnu-103).

```
gdb xfs.out
target remote-kdp
add-symbol-table ...
attach <host>
```

6 Porting

The largest part of the work needed to port Arla to a new operating system is in porting xfs, as kernel programming always is harder, less portable and messier than user-space dito. Arla in test mode (*arla-cli*) should work without any porting on any system that's not very far away from Unix and that provides berkeley sockets (including cygwin32). The hard part is porting the XFS kernel module, and we will spent most of this text on how to do that.

6.1 user-space

The user-space parts should work on basically any system that is reasonably Posix and has berkeley sockets. The build uses autoconf and should adapt itself to most forseeable circumstances. If it fails to consider something that is missing or not working on the particular OS you are porting to, hard-code it to make sure that is what is missing and then try to create an autoconf test for it. If you fail to do so, or have no autoconf experience, send us the patches anyway and tell us where you are having the problem.

6.1.1 LWP

The only thing that might take a little bit more effort in porting is the context-switch in the LWP user-level threads package. There are assembler versions for most of the common architectures in '`lwp`'. Part of the problem is getting this code assembled properly. There is unfortunately no easy and portable way of preprocessing and assembling code. There is a script '`lwp/make-process.o.sh`' that tries to do in some different ways, but it may fail for you. Next problem is that assembler syntax can vary a lot even on the same CPU. The source files are written in such a way that they should be acceptable to almost any syntax, but if it fails you have to find out what particular syntax has to be used and adapt the source file for that.

The more interesting problem is if there is no support for your CPU. The first thing to try then is the *--with-pthreads* option that uses the pthreads library. If that fails or you want LWP working you have to figure out enough details on your CPU to write two functions in assembler, '`savecontext`' and '`returnto`' that save and restore the processor context.

6.2 XFS

1. It helps to have source code for your operating system.

   In theory, if stuff was documented well enough, you wouldn't need it. In practice it never is, so you find out interfaces specs and how stuff works by reading the source code. If you're unable to find source code for your OS, try finding source for the closest match. If your OS is based on BSD, try the appropriate version of BSD, for example.

2. If you don't have source, try second best, include files.

   You can usually gather quite a lot of information on the workings of the kernel by reading the includes files in '`<sys/*.h>`'.

3. Be lazy

   Try to find out what other XFS port is most similar to your OS and start with that code.

4. Figure out how your kernel works.

   You need to figure out how a few things work in your kernel:

1. Loading/unloading kernel modules

   That varies quite a lot but it's probably easy to figure out if you have the source code for some other loadable module. Sometimes you can get the kernel to add your cdev, system call and file system automatically but usually you have to write code in your 'entry-point' to add these to the appropriate tables.

2. Adding a new character device driver

   The kernel has a table of all known device drivers, ordered by major number. Some kernels have one for block devices and one for character devices and some have a common one. That entry usually consists of a number of function pointers that perform the operations (open, close, read, write, ...), and possible a name and some flags. It could look something like the following:

   ```
   struct cdevsw {
   int (*d_open)();
   int (*d_close)();
   ...
   };

   struct cdevsw cdevsw[];
   ```

   These are then usually stored in a table 'cdevsw' indexed by the major device number. If you're really lucky there's a new way to get the kernel to add your 'struct cdevsw' to the global table when loading the module or a function that does the addition for you. Otherwise there might be functions for adding/removing devices to the global table. If not, you'll have to fallback on looking for a free slot in the table and putting your struct cdevsw there. In some cases, this is not stored in a table but then there'll be a way of adding entries to the new data structure so you don't need to worry about it.

3. Adding a new system call

   This is quite similar to adding a new cdev but the table is usually called 'sysent' instead.

4. Adding a new file system

   Once again, quite similar in principle. The names of the structures tend to vary quite a lot more.

5. Finding out how the VFS/Vnode switch works

   The structure vfsops contains function pointers for all of the file system operations. You need to figure out what operations you need to implement (usually at least mount, unmount, root, sync, and statfs).

   The operations that are performed on files are vnode operations (usually stored in a struct vnodeops), and you need to figure which of these you need and how they should work. Also, which is not as explicit, how vnodes are supposed to be allocated and freed and such.

5. Suggested plan of action

   1. Start by writing a minimal hello-world module and make sure you can load and unload it properly.
   2. Then add a device driver to the module which dummy functions and verify that works.
   3. Try to fit the device driver functions in '`xfs_dev.c`' into the device driver.
   4. Do a dummy module with a system call and verify that you can call it.
   5. Start trying to add enough of the vfs/vnode operations from '`xfs_vfsops.c`' and '`xfs_vnodeops.c`' so that you can build it.
   6. Debug it.
   7. Send us patches

7 Oddities

7.1 AFS

- Directories - UnixModeBits are ignored when the vnode is a directory.
- Errnos are sent over the network. Like Solaris ENOTEMPTY(93) doesn't even map to an error on sunos4 where ENOTEMPTY is 66.
- Mountpoints have the mode-bits 0644, if they don't they are symlinks (and have the mode-bits 0755).

7.2 Operating systems

- On Irix 6.5 you have to build the dirents depending on what ABI of the binary you are currently running.
- . and .. need to be first in directories, this is needed since some programs (like make) "knows" that the two first entries are . and .. and thus can be skiped.
- Reclen (in struct dirent) shouldn't be too large. When its larger then the buffer used in opendir/readdir/closedir, you loose.

8 Arla timeline

Arla have existed for quite some years.

Development started around XXX by Björn Grönvall bg@nada.kth.se, quick followers was Assar assar@sics.se (at that time assar@pdc.kth.se>) and Johan Danielsson <joda@pdc.kth.se. The platform that was chosen was Sparc SunOS4 (the OS that NADA, KTH was using).

Some work was being done by Patrik Stymne patriks@e.kth.se in porting arla to Ultrix, but this work was never finished.

At this time there was no free rx, lwp or rxkad. A basic rx implementation was written, and the threading problem was solved by using pthreads.

The Arla development started to slow down around 11 April 1995.

In about Mar-Jun 1996 rx and lwp was released by Transarc, this was made possible by Jim Doyle jrd@bu.edu, and Derrick J. Brashear shadow@dementia.org.

In September 1997, an rxkad implementation was written by Björn. At the same time, a need for an AFS client for OpenBSD rose at the Stacken, the local computer club at KTH. Other free OS:es, as NetBSD, FreeBSD and Linux(primarily sparc) were also in need of AFS clients.

In TokKOM, a local communications system using LysKOM (http://www.lysator.liu.se/lyskom/), Assar suggested to some club members that it would be a nice thing to resume the arla development.

Some people suggested that it would be less trouble having someone with access to the Transarc AFS source code port the code to the relevent platforms. Assar then ported xfs to FreeBSD 2.2.x in notime, just to show the high portability.

People started to understand that arla was a concept that would work, and first out was Love Hörnqvist-Åstrand lha@stacken.kth.se to join. Development was primarily aimed at OpenBSD and NetBSD at the moment, and Arla lived for at least 2-3 weeks in /var/tmp on a host named yakko.stacken.kth.se.

Magnus Ahltorp map@stacken.kth.se joined shortly thereafter, spending the rest of the year reading about the Linux VFS, and after a while, Artur Grabowski art@stacken.kth.se also started to work on arla, concentrating on OpenBSD kernel stuff.

The first entry in ChangeLog is dated Fri Oct 24 17:20:40 1997. Around this time arla was given a CVS tree, to ease development. Now you could also mount the xfs-device and get the root-directory out of it.

The Linux port was done in a few weeks in the beginning of 1998. Only the Linux 2.0 kernel was supported at this time.

In April 1998 Assar hade a Arla paper presented at Freenix. Linux 2.1 support was written also written around this time. This was a major work since there was a lot of stuff that had changed (namely the dcache).

The first milko entry is dated Thu Oct 30 01:46:51 1997. Note that this milko in a sense "worked". You could get files out from it and store them.

There was from this point a lot of work being done and quite a lot of studies was "wasted". We learned a lot, but not the stuff we were expected too.

In Mars 2000 preliminary support for MacOS X/Darwin 1.0 was merged in by Magnus and Assar.

Around the same time there we hacked in support for Solaris 8 (beta2) There was also some work being done on Windows 2000 native driver at same time.

In June 2000 there was a presentation on MADE2000 in Gothenburg, Sweden.

This just includes some milestones, for more information se Changelog.* and NEWS files in the distribution.

9 Authors

Currently writing on arla are

Assar Westerlund, Everything
Magnus Ahltorp, Everything
Artur Grabowski, BSD xfs, OpenBSD maintainer
Love Hörnquist-Åstrand, Everything
Robert Burgess, fs, Solaris xfs
Johan Danielsson, OSF/1 xfs
Hans Insulander, pts, OpenBSD maintainer
Mattias Amnefelt, vos, milko
Harald Barth, doc
Tomas Olsson, milko
Mikael Vidstedt (userland, some milko stuff)
Jimmy Engelbercht (bos)

Rhapsody xfs port was contributed by Alexandra Ellwood <lxs@MIT.EDU> Previously, Rhapsody named Darwin.

Disconnected code is written by:

WUWEI SHEN <wwshen@engin.umich.edu>
Cheng Jin <chengjin@eecs.umich.edu>
Paul Callaway <pcallawa@umich.edu>

For more contributors look in the THANKS file in the distribution on ftp://ftp.stacken.kth.se/pub/arl
.

Appendix A Acknowledgments

lwp and rx are copyrighted by IBM. We're grateful to Derrick J Brashear `shadow@dementia.org` and Jim Doyle `jrd@bu.edu` for making them available.

the `rxkad` implementation was written by Björn Grönvall `bg@sics.se` and is also part of the kth-krb distribution.

Some of the files in '`libroken`' come from Berkeley by the way of NetBSD/FreeBSD

`editline` was written by Simmule Turner and Rich Salz.

The code for gluing these together were written by ourselves.

Bugfixes, documentation, encouragement, and code has been contributed by:

Aaron M. Ucko
`amu@MIT.EDU`

Alec Wolman
`wolman@cs.washington.edu`

Alexandra Ellwood
`lxs@MIT.EDU`

Brad Keryan
`keryan@andrew.cmu.edu`

Constantine Sapuntzakis
`csapuntz@openbsd.org`

Dan Winship
`danw@MIT.EDU`

Derrick J Brashear
`shadow@dementia.org`

Harald Barth
`haba@pdc.kth.se`

Jim Doyle `jrd@bu.edu`

John Davison
`davisoja@clarkson.edu`

John Hawkinson
`jhawk@MIT.EDU`

Karl Ramm
`kcr@MIT.EDU`

Mark Eichin
`eichin@kitten.gen.ma.us`

Per Boussard T/ED
`Per.Boussard@era-t.ericsson.se`

Dima Ruban
`dima@best.net`

Max `davros@cyclone.Stanford.EDU`

Andrzej Filinski
`andrzej@daimi.aau.dk`

Chuck Lever
chuckl@netscape.com

WUWEI SHEN
wwshen@engin.umich.edu

Cheng Jin chengjin@eecs.umich.edu

Paul Callaway
pcallawa@umich.edu

Brandon S. Allbery
allbery@ece.cmu.edu

Ken Raeburn
raeburn@raeburn.org

Jeffrey Hutzelman
jhutz+@cmu.edu

Hee-Seok Heo
hsheo@postech.ac.kr

Paul Ewing Jr.
ewing@ima.umn.edu

Niklas Hallqvist
niklas@appli.se

Marko Asplund
aspa@hip.fi

Chris Wing
wingc@engin.umich.edu

Simon Josefsson
jas@pdc.kth.se

Magnus Lindström
magnus.lindstrom@s3.kth.se

Greg Stark
gsstark@mit.edu

Matt deberg@mit.edu

Björn Grönvall
bg@sics.se

Tino Schwarze
tino.schwarze@informatik.tu-chemnitz.de

David Sanderson
david@transarc.ibm.com

Roman Hodek
Roman.Hodek@informatik.uni-erlangen.de

Michael Sperber
sperber@informatik.uni-tuebingen.de

Dr. Lex Wennmacher
wennmach@geo.Uni-Koeln.DE

Janne Johansson
jj@dynarc.se

Dr A V Le Blanc
LeBlanc@mcc.ac.uk

Dave Morrison
dave@bnl.gov

If you have done something and are not mentioned here, please send mail to arla-drinkers@stacken.kth.se

If you are mentioned here and have not contributed, that's because we expect you to.