KORBit: A Kernel Space CORBA ORB
CS 423 Project Design Document

Chris Lattner (lattner@uiuc.edu)
Andy Reitz (areitz@uiuc.edu)
Fredrik Vraalsen (vraalsen@uiuc.edu)
Keith Wessel (kwessel@uiuc.edu)

November 2, 2000

1 Introduction

Providing a pervasive distributed object system at the kernel level opens a number of possibilities to to both users and application developers. From the user's perspective, a distributed object system allows the use of remote machines with the same interface as provided by the local machine. This makes access to remote systems transparent, so that the user doesn't need to waste any brain power (a most precious resource!) to take advantage of remote resources. From a developer's point of view, placing a distributed object system in the kernel means that it will become a standard system service, which is present on every machine (running that brand of kernel, anyway). Furthermore, an in-kernel system provides a fixed API for which to develop.

Because there are numerous advantages, we have decided to place a CORBA-compliant Object Request Broker (ORB) into the Linux Kernel. We plan to take an existing ORB, refactoring it as a Linux kernel module, so that it may reside in kernel space. We believe that this system will provide the advantages described above. This document will lay out the major design obstacles and features of this idea. In Section 2, we go through our choices of ORB and kernel versions. Section 3 mentions various use cases for the kernel ORB, while section 4 describes the structure of the chosen ORB and identifies pieces that need to change to fit into the kernel. Then in section 5 we discuss the specifics of how we will integrate the ORB into the kernel. Section 6 presents an example application that demonstrates the usefulness of the kernel ORB.

2 ORB and Kernel Details

2.1 ORB

While writing a CORBA ORB from scratch would have been fun, it also would have diverted significant effort from the main goals of this project. Therefore, we decided upon the more focused task of "porting" an existing ORB into the Linux kernel. Fortunately, there are many *Open Source* ORBs in existence, so we had many to choose from.

We had several properties that we required from an implementation, and these requirements served as a guide when making our ORB selection. First, we desired an ORB that was relatively small, and with code that was easy to understand. This eliminated ORBs like TAO. Second, we wanted something that would interface well with the kernel. Third, we wanted something that was modular, so that we could strip out unnecessary functionality easily.

Based on these requirements, **ORBit**[1], the ORB designed for the *Gnome project*[2], fit the bill perfectly. ORBit is a CORBA 2.2-compliant ORB that was designed for speed and to C language CORBA bindings. The first objective helps us because it means that ORBit is rather lean. The second point is important because not only does ORBit provide bindings for the C language (which is important for interfacing with system libraries, which are almost exclusive coded in C), but it also means that ORBit itself was *programmed* entirely in C! This makes integrating ORBit into the Linux kernel much more tractable. We have standardized on version `5.3` of ORBit for this project.

2.2 Kernel

The Linux kernel[3] is the most popular Open Source UNIX-like kernel available today. Because the source code is open, it is easy to make extensions, and experiment with new modules. Furthermore, because of its popularity, it is

[1] http://www.labs.redhat.com/orbit/
[2] http://www.gnome.org/
[3] http://www.kernel.org/

easy to find support and documentation about the Linux kernel. Unfortunately, there are many different versions of the Linux kernel available, and choosing one to standardize on wasn't an easy choice.

The two main options were the `2.2.x` series kernel, which represents the current *"stable"* branch, and the `2.4.0-test-x` series of development kernels. These kernels represent where Linux will be moving to in the next year. However, because they are in development, they are subject to substantial bugs, as well as evolving APIs.

Despite these details, we decided upon the `2.4.0-test9` kernel for our project. We feel that developing for this kernel will ensure that our code stays viable for a longer period of time. Furthermore, we feel that the development kernels are mature enough at this point for our uses, and that doing module development of this nature is feasible.

3 Use Cases

1. **Object-Oriented Layer:** Using an in-kernel CORBA implementation, it is possible to expose an Object-Oriented view of the kernel and all of its services. This would elevate the operating system beyond the 30+ year old *procedural* model, into a more modern object-based approach.

2. **Network Transparency:** The ORB can route method invocations to local or remote objects, all without explicit intervention from the user. This enables normal applications to work in a distributed manner without changes. For example `ls` and `cp`) are automatically forwarded to the correct machine giving us a transparent distributed file system almost for "free" (see figure 1).

3. **Object Services:** New kernel services could be written in any language with ORB bindings, and then integrated into the Kernel. This allows device drivers to be written in perl, for example. Furthermore, these new services could be *"pervasively object-oriented"*: represented both internally and externally as objects.

4. **User Space Services:** With an in-kernel ORB, services that are traditionally provided by the kernel can be exported to user space. As easily as a userland process can invoke methods on an object that resides in kernel space, a kernel space client can invoke methods on an object that

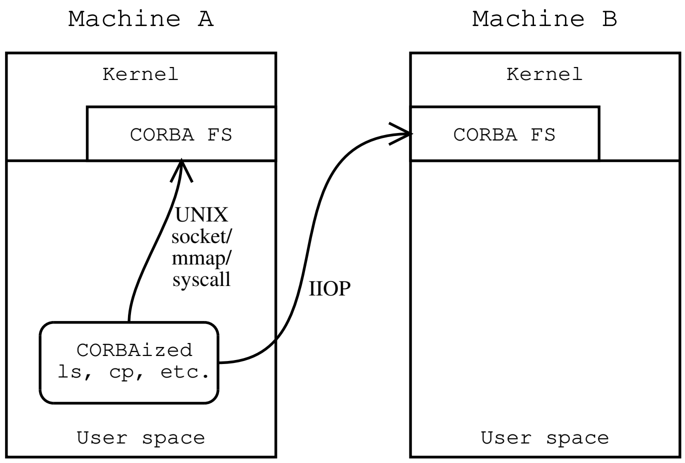

Figure 1: Distributed filesystem

resides in user space. For example, a serial port driver could be written in user space, and the user could then dynamically change driver at will (to change from a wheel mouse to a tablet, for example). Another example of possible applications is a file system implemented as a user space object (see figure 2).

4 ORB Design and Refactoring

4.1 ORBit

4.1.1 Core modules

orb The code in this directory implements the core of the ORB. There are quite a number of files in this directory, all of which are a part of the central ORB functionality. This code breaks down the specifications of the ORB, piece by piece, to form a complex yet organized lay-out of the ORBs structure and functionality.

IIOP This module channels requests appropriately, according to whether the requests are local or remote requests. Local requests are delivered via “shared library object activation”. Remote requests are transformed into IIOP streams, delivered to the remote system, decoded, and passed to the

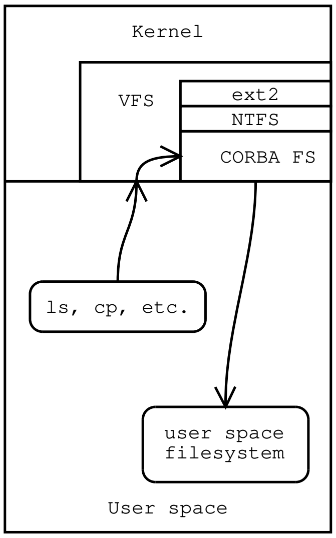

Figure 2: Filesystem implemented in user space

caller. This is obviously an important part of functionality over all.

ORBitutil This code contains utility functions to ease the portability of ORBit across different platforms. It also includes some support functions to facilitate debugging.

4.1.2 Support modules

libIDL This library is an IDL parser that is used by the IDL compiler.

orbit-idl-compiler This is a stand-alone program that compiles IDL files into the requisite stubs and skeletons. The ORBit IDL compiler comes with a back-end for C, and there are also back-ends available for other languages such as C++ and Perl.

services Contains code for the Event service, Name service, and Time service.

daemons right now, this only contains one daemon – the *interface repository*.

4.2 External dependencies

ORBit depends on a number of external libraries.

libc ORBit uses many standard C library functions to do memory allocation, string handling, hostname lookup, general networking, and so on.

libm ORBit uses the pow() function from the math library. However, the it is being used is to compute powers of ten, so we wrote a small replacement function to remove this dependency.

glib Glib is a support library initially developed for the GTK+[4] widget library. Glib contains a lot of application support code, such as event loops, thread handling, memory allocation and various data types such as strings, lists and hashtables.

4.3 KORBit

The kernel ORB (KORBit) will consist of the core ORBit modules. The support modules are not needed for the ORB itself, they are just needed for external services (e.g. naming), IDL compilation, and so on.

To take care of the external dependencies we have map the function calls to their kernel counterparts or port the functionality to the kernel.

Many of the C library functions already have counterparts in the Linux kernel, so handling these will be as simple as using `#define` to redirect function calls to their kernel counterparts (e.g. `printf` to `kprintf`) or writing short wrapper functions. For the functions that cannot be handled in this way we have to either port them to the kernel or rewrite ORBit to use the functions that are available in the kernel.

It turns out that ORBit only needs a subset of the functionality in Glib, so we decided to separate out the required functions into a new library (called `KGlib`) that we will port to the kernel. KGlib has many of the same dependencies on the C library as ORBit does, and we will solve this in the same way as mentioned above.

5 ORB Integration into Kernel

This section will attempt to establish, in concrete terms, which functions KORBit will need to leverage from the Linux kernel. Code needs to be written so that KORBit can establish itself as a well-behaved kernel module. Code must

[4] http://www.gtk.org/

also be written in order to translate the `libc`-style system calls made in the ORBit code to the kernel-style calls that must be made in the KORBit code.

5.1 Linux Kernel Module API

Conceptually, writing a Linux kernel module is as simple as supporting two functions: `init_module()` and `cleanup_module()`, and then compiling into object code (with the appropriate defines). However, when considering what is actually necessary in order to initialize KORBit, and the full ramifications of being a Linux kernel module, things become a bit stickier.

In a nutshell, initializing KORBit requires doing enough initialization so that an IOR can be written to `/proc`, and launching a new kernel thread in order to service requests to the ORB. This first step requires creating and binding a TCP socket, and then performing the gyrations to actually write a file into `/proc`. This process is best described by the document "Linux Kernel Module Programming Guide"[5], by Ori Pomerantz. This document provides some sample code for initializing the requisite data structures, and setting up the callback function necessary for `/proc`.

While all of the above is simply a matter of driving the Linux kernel, we encounter problems when attempting to support the nuances of kernel modules. Linux kernel modules support the notion of *"reference counting"*. But considering the timeframe for our project we will leave as a future enhancement.

When KORBit is loaded, it will set it's reference counter to some non-zero value, and mark its `VISITED` flag. From this point on, these values will not be changed. The only way to unload KORBit will be manually (via `rmmod`).

5.2 Linux Networking API

Generally speaking, the network functions[6] available in the Linux kernel for dealing with the TCP/IP stack are quite similar to the standard `libc` functions. Thus, porting the ORBit networking code to kernel space should be quite straight-forward.

[5] http://www.linuxdoc.org/LDP/lkmpg/node16.html
[6] "The Linux Kernel: Networks" - http://www.linuxdoc.org/LDP/tlk/net/net.html

5.3 Memory Allocation

The Linux kernel provides the functions `kmalloc()` and `kfree()` for allocating memory. These functions are basically equivalent to their `libc` counterparts, with one small exception: `kmalloc()` can only be used to allocate less than 128Kb of memory. If more memory than what is provided by `kmalloc` is needed, then the allocation must be done directly, by requesting memory pages from the kernel. But all things being equal, this isn't especially difficult.

5.4 Standard C library

Whenever we encounter a function that doesn't have a direct kernel counterpart (`gethostbyname()`, for example), we will attempt to duplicate the needed functionality within KORBit. This can be done by copying the requisite code out of the GNU C Library[7], and either in-lining it within KORBit, or by providing it in a support file. We foresee this process as being akin to manual labor – it doesn't require much thinking, it just requires some hard work.

6 Transparent Distributed File System Example

KORBit provides a powerful way for the kernel to interact with user space processes. It also allows services to be built on top of KORBit functionality. One example that is greatly simplified with KORBit is a transparent distributed file system.

Appendix A.1 gives a listing of our prototype IDL for specifying a set of object oriented file system access interfaces. These interfaces are designed with the following goals in mind:

1. *Allow object oriented programs to use a higher level API than traditional file descriptors:* New applications can take advantage of a more modern and uniform API for file access. Explicit exceptions allow for structured error handling and eliminates issues when using `errno` and multithreaded programs together.

2. *Provide all of the functionality of the existing file system API:* We would like to provide a mapping library that converts traditional requests into

[7]http://www.gnu.org/software/libc/libc.html

CORBA requests. This requires complete coverage of the existing file system API. This is described below.

3. *Wrap existing file system functionality in an object oriented way:* We don't plan on extending kernel functionality at all, just making access to it nicer.

6.1 System Architecture for a Transparent Distributed File System

When an application starts up, a global `KernelAccessAPI::FileSystem` is initialized by a shim library static intitalizer section. This object is named '`CurrentFileSystem`' and bound remotely based on the following conditions:

- If the '`CORBA_FS_IOR`' environment variable is set, then the specified IOR is used to initialize the '`CurrentFileSystem`' object.
- If not, the IOR found in '`/proc/fs/corba_fs_ior`' is used. This IOR corresponds to the local kernel's FileSystem object.

This very simple system allows great power. This allows a user to have a program transparently access a remote file system, without the application knowing what is going on at all.

6.2 Support for legacy applications (i.e. X-Emacs)

It is very desirable to tap the large application base that already exists that uses the current file system API (either directly or indirectly). Because of this, the shim library will also contain routines that remap calls to `open`, `close`, `read`, `write`, etc. to method invocations on the `CurrentFileSystem` object. Because KORBit has C language bindings, we should have relatively few headaches to worry about here.

6.3 Implementing Kernel functionality with KORBit

KORBit facilities also provide ample opportunities to implement kernel services in CORBA, allowing the object to run in the kernel, or potentially in user space. Appendix A.2 contains IDL for a CORBA file system API to write file systems with KORBit. This provides a straightforward wrapping of the Linux virtual

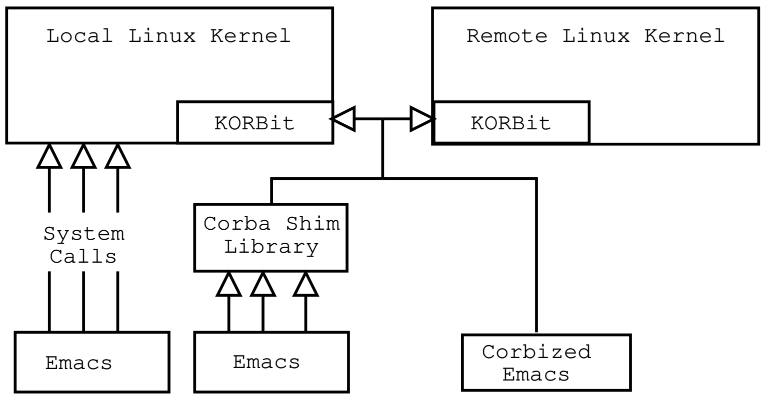

Figure 3: Transparent Remote File System

file system API. This API is already written in an object oriented style, which makes CORBAization much easier.

Another example of kernel services is drivers. Many hardware drivers could be initially written in userspace (where debuggers are much easier to use) and then later be pushed 'in kernel'. This separation promotes the use of modular code that is well structured. Although, being able to move code from userspace to kernel space transparently sounds wonderful in theory, it may be difficult to port services that have complex dependencies (for example a perl interpreter).

A File system IDLs

A.1 CORBA bindings to Kernel VFS (Virtual FileSystem) interface

```
// ----------------------------------------------------------------------------
// FileSystem.idl
// ----------------------------------------------------------------------------
//
// This file is used to define the Kernel CORBA API for accessing the filesystem
// on a machine.  This defines mappings both to access the files in the kernel
// and to implement a filesystem for the kernel.  This should probably be split
// into two files eventually.
//
// Currently unimplemented: KernelAccessAPI::FileSystem::ReadDirectory
//                          KernelAccessAPI::FileSystem::Select
//                          KernelImplementationAPI::SuperBlock::getDiskQuotaOps
//
// ----------------------------------------------------------------------------


// These are the exceptions that may be thrown and what they map to in the
// Linux kernel.  This interface is extended by other interfaces so that the
// names don't have to be typedef'd into each interface that wants to use these
// errors.
//
interface Errors {
    exception IsDirectory {};             // EISDIR
    exception PermissionDenied {};        // EACCES
    exception FileExists {};              // EEXIST
    exception FileNotFound {};            // ENOENT
    exception IsNotDirectory {};          // ENOTDIR
    exception ReadOnlyFile {};            // EROFS, ETXTBSY
    exception RecursiveSymlink {};        // ELOOP
    exception IsBusy {};                  // EBUSY
    exception OtherError{};               // Misc other ones...
};


// ----------------------------------------------------------------------------
```

```
// KernelAccessAPI Module - Allow user level programs to call into the kernel
// -------------------------------------------------------------------------------

module KernelAccessAPI {
  struct FileStatus {           // Corba equilivant of struct stat
    long  DeviceNum;            // st_dev
    long  InodeNum;             // st_ino
    short Mode;                 // st_mode
    short NumLinks;             // st_nlink
    long  UserID;               // st_uid
    long  GroupID;              // st_gid
    long  DeviceType;           // st_rdev
    unsigned long Size;         // st_size
    unsigned long BlockSize;    // st_blksize
    unsigned long NumBlocks;    // st_blocks;
    unsigned long AccessTime;   // st_blocks;
    unsigned long ModifiedTime; // st_blocks;
    unsigned long ChangeTime;   // st_blocks;
  };

  // -----------------------------------------------------------------------------
  // FileSystem Interface - Access to filesystem and File object factory
  // -----------------------------------------------------------------------------

  interface File : Errors {
    void Read(out string buffer) raises (IsDirectory, OtherError);
    void Write(in string buffer) raises (OtherError);
    void Close();

    long FileControl(in long command) raises (OtherError);

    FileStatus GetStatus() raises (OtherError);

    void ChangeDirectoryTo()    // This implements fchdir...
      raises (IsNotDirectory, PermissionDenied, OtherError);

    enum SeekDirection { FromStart, FromCurrent, FromEnd };
    long Seek(in long Offset, in SeekDirection Direction) raises (OtherError);

    // DuplicateAs is very gross (FD #), but we need it for dup2
```

```
  File Duplicate() raises (OtherError);
  File DuplicateAs(in long NewFD) raises (OtherError);
};



// ---------------------------------------------------------------------------
// FileSystem Interface - Access to filesystem and File object factory
// ---------------------------------------------------------------------------

interface FileSystem : Errors {

  // -------------------------------------------------------------------------
  // File Manipulation Routines
  // -------------------------------------------------------------------------

  File Open(in string Filename, in long openFlags, in short mode)
    raises (FileExists, IsDirectory, PermissionDenied, FileNotFound,
    IsNotDirectory, ReadOnlyFile, RecursiveSymlink, OtherError);

  File Create(in string Filename, in short mode)
    raises (FileExists, IsDirectory, PermissionDenied, FileNotFound,
    IsNotDirectory, ReadOnlyFile, RecursiveSymlink, OtherError);

  void Link(in string FromPath, in string ToPath)
    raises (PermissionDenied, IsNotDirectory, RecursiveSymlink, FileExists);

  void Unlink(in string Filename)
    raises (PermissionDenied, FileNotFound, IsNotDirectory, IsDirectory);

  void Rename(in string OldName, in string NewName)
    raises (IsDirectory, FileExists, IsBusy, IsNotDirectory, PermissionDenied,
    RecursiveSymlink);

  void ReadLink(in string Linkname, out string LinkValue)
    raises (FileNotFound, PermissionDenied, RecursiveSymlink, OtherError);


  FileStatus GetStatus(in string Filename)
    raises (FileNotFound, PermissionDenied, RecursiveSymlink, IsNotDirectory,
```

```
    OtherError);

  FileStatus GetLinkStatus(in string Filename)
    raises (FileNotFound, PermissionDenied, RecursiveSymlink, IsNotDirectory,
    OtherError);


  // ----------------------------------------------------------------------
  // Directory Manipulation Routines
  // ----------------------------------------------------------------------

  void MakeDirectory(in string PathName)
    raises (FileExists, PermissionDenied, FileNotFound, IsNotDirectory,
    ReadOnlyFile, RecursiveSymlink, OtherError);

  void RemoveDirectory(in string PathName)
    raises (PermissionDenied, FileNotFound, IsNotDirectory);

  // ChangeDirectory returns CWD so that you can implement getcwd as
  //  { return ChangeDirectory("."); }
  string ChangeDirectory(in string PathName)
    raises (IsNotDirectory, PermissionDenied, FileNotFound, RecursiveSymlink);


  // ----------------------------------------------------------------------
  // Special Purpose Routines...
  // ----------------------------------------------------------------------

  void MakeNode(in string FileName, in short Mode, in long DeviceNum)
    raises (PermissionDenied, FileExists, FileNotFound, IsNotDirectory,
    RecursiveSymlink);

  void Mount(in string DeviceFile, in string Location, in string FSType,
     in long Flags)
    raises (PermissionDenied, FileNotFound, IsBusy, IsNotDirectory);

  void Unmount(in string Filename)
    raises (PermissionDenied, FileNotFound, IsBusy);
};
```

```
};
```

A.2 CORBA bindings for User-space filesystem implementations

```
// ------------------------------------------------------------------------------
// KernelImplementationAPI Module - Implement Kernel functionality in Korba
// ------------------------------------------------------------------------------
//
// Main source of information:
//    http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html
//
module KernelImplementationAPI {
  interface FileOps;
  interface InodeOps;
  interface DEntry;
  interface DEntryOps;
  interface SuperBlock;
  interface SuperBlockOps;
  interface FileSystem;

  interface File {};   // struct file
  interface Inode{};   // struct inode

  interface InodeOps {  // struct inode_operations
    FileOps getDefaultFileOps();
    long    create (in Inode i, in DEntry d, in long flags);
    DEntry  lookup (in Inode i, in DEntry Parent);
    long    link   (in DEntry d, in Inode i, in DEntry d2);
    long    unlink (in Inode i, in DEntry d);
    long    symlink(in Inode i, in DEntry d, in string name);
    long    mkdir  (in Inode i, in DEntry d, in long flags);
    long    rmdir  (in Inode i, in DEntry d);
    long    mknod  (inout Inode i, in DEntry d, in long major, in long minor);
    long    rename (inout Inode i, in DEntry d, in Inode i2, in DEntry d2);
    string readlink(in DEntry D);
    DEntry followlink(in DEntry d1, in DEntry d2, in long flags);

    // get_block, readpage, writepage, flushpage, truncate, permission
    // smap, revalidate
```

```
};

interface DEntryOps { // struct dentry_operations
  long revalidate(inout DEntry d);
  long hash(in DEntry d);
  long compare(in DEntry d, in string a, in string b);
  void delete(in DEntry d);
  void release(in DEntry d);
  void inodePut(in DEntry d, in Inode I);
};

interface DEntry {    // struct dentry
  long       getDCount();
  Inode      getInode();
  DEntry     getParent();
  DEntry     getMounts();
  DEntry     getMountPoint();
  long       getDTime();
  DEntryOps  getDEntryOps();
  SuperBlock getSuperBlock();
};

interface FileOps {  // struct file_operations
  enum SeekDirection { FromStart, FromCurrent, FromEnd };

  long llseek(inout File f, in long offset, in SeekDirection Direction);
  string read(inout File f, in long size);
  void write(inout File f, in string buffer);
  void readdir(inout File f, inout string buffer);
  long flush(inout File file);
  long release(in Inode inode, in File file);
  long fsync(in File f, in DEntry dentry);

  // poll, mmap, check_media_change, revalidate, lock, fasync
};

interface SuperBlockOps {  // struct super_operations
  void readInode(inout Inode WhichOne);
  void writeInode(inout Inode WhichOne);
  void putInode(inout Inode WhichOne);
```

```
    void deleteInode(inout Inode WhichOne);
    void clearInode(in Inode WhichOne);

    void notifyChanged(in DEntry d);
    void putSuper(in SuperBlock SB);
    void writeSuper(in SuperBlock SB);
    void umountBegin(in SuperBlock SB);
  };

  interface SuperBlock {  // struct super_block
    long getBlockSize();
    FileSystem     getFileSystemType();
    SuperBlockOps  getSuperBlockOps();
    //DiskQuotaOps   getDiskQuotaOps();
  };

  interface FileSystem {  // struct file_system_type
    string getName();
    long getFlags();
    SuperBlock ReadSuperBlock();
  };
};
```