DI/ODE: The Distributed I/O Development Environment

James Larson, Nick Christenson, Scott Lystig Fritchie,
Philip Guenther, Jason Evans, Charles Murray
Sendmail, Inc.
Emeryville, CA 94608
{jim,npc,scott,guenther,jasone,cmurray}@sendmail.com

Abstract

This paper describes software that can be linked to an application which normally performs its I/O functions on a local server and makes them distributable and scalable in a manner which is extremely transparent to the application. Through the use of this system, an application normally confined to a single host can use a large number of back-end data repositories in a manner that is arbitrarily scalable, manageable, and which can be extended on-the-fly automatically and transparently to that application.

NOTE: Lots of this stuff is pure speculation at this moment. Many design decisions have been stated here, but they should serve as straw men. As details get implemented, this document should be updated to reflect the actual implementation.

1 Goals

Sendmail is in the business of providing email solutions to our customers and the Open Source community. Part of our customer base are the extremely high-end sites, such as Internet Service Providers, Application Service Providers, and other businesses that on a day-to-day basis test the capability of current systems to send and receive high volumes of electronic mail. In order to satisfy their needs, Sendmail has been working on a distributed system mail architecture. Its high-level goals can be stated as: Arbitrary scalability, extreme robustness, ultra-high performance, and maintainability.

Since Sendmail already has a stable full of high-quality existing applications, it didn't seem prudent to go reinventing them. We were well motivated to come up with a system that leverages our existing applications as much as possible, and requires minimal modifications to them. Also, time to market is very important, and we have a relatively small development team, so we wanted to create a generally useful solid foundation that was minimally complex, such that it could be built quickly and extended later.

We spent considerable time exploring mechanisms that have already been developed to adapt to our purposes. However, nothing seemed truly suitable to what we were trying to accomplish. Existing work, such as [Christ97], [Saito99], and [Horman99] didn't completely solve our problems.

DI/ODE, the subject of this paper, is our response to this challenge. DI/ODE is a comprehensive system which is designed to be linked in to existing applications with minimal, or perhaps no, source code modifications acting as a foundation for arbitrarily scalable Internet systems.

2 Architecture

2.1 The Big Picture

DI/ODE is a two-tiered system. The architecture is diagrammed in Figure 1. In this architecture there are machines that handle connections from the outside world, which we call DI/ODE Application Servers, and machines that store data, which we call DI/ODE Data Servers. These systems are connected via a high-speed low latency pocket LAN, which, in many ways, acts as the backplane for our "virtual computer". In keeping with the backplane metaphor, connections among the back-end machines are minimally authenticated and unencrypted. Therefore, network(s) over which the DI/ODE machines communicate should be dedicated to this purpose and not used for other purposes.

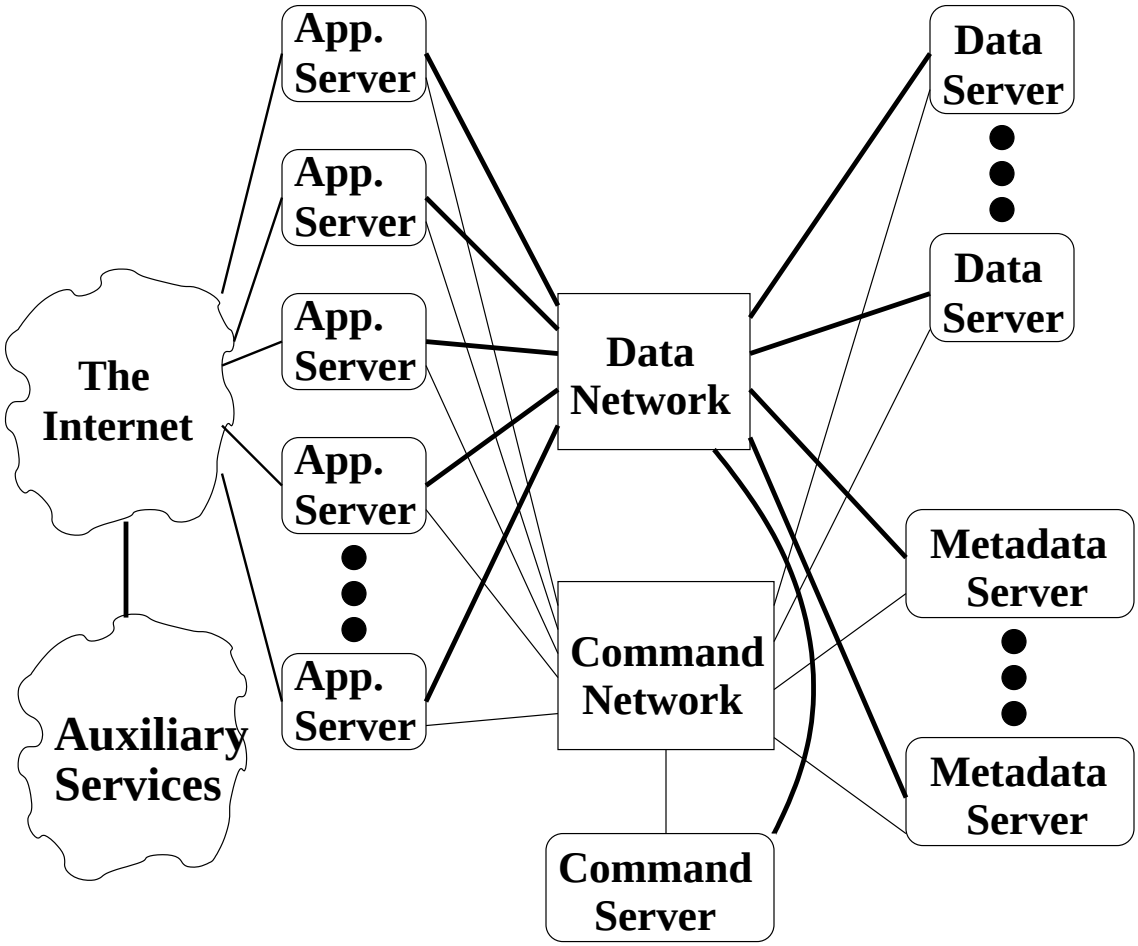

Figure 1: DI/ODE Architecture

The Application Servers are dataless computers that run applications. For running Internet services, they should have two ultra-high-speed network interfaces. On one, they receive and reply to requests from the outside world. Through the other, they send and receive information to and from the Data Servers. They may also use a lower speed network interface to connect to a physically distinct Command Network. Application Servers receive updates as to the state of the system from the Command Server. One Application Server will *never* talk to another. The intent of the system is to gain front-end (front defined as facing the Internet) scalability by the deployment of large numbers of Application Servers operating in parallel.

The Data Servers are, essentially, dumb disk on a network. They store and retrieve data for the Application Servers and occasionally process administrative requests from the Command Server. They do not initiate any communications, and one Data Server never talks to another. They must have a high-speed connection to the Data Network, but should not be reachable by machines which are not part of the DI/ODE system at all. Back-end capacity can be extended arbitrarily by the deployment of additional Data Servers. The additional storage and I/O capacity is transparently made available to the applications.

We also use a machine called the DI/ODE Command Server which is responsible for making certain that all the DI/ODE computers agree on the state of the network. The Command Server is non-operational. That is, if it crashes, all the other components continue to operate in their current mode, they just will not receive new updates about the state of the system as a whole.

The goal of the Command Server is to provide a centralized point for command and control functions of a DI/ODE system. This is beneficial in two ways. First, this provides a convenient point for human interaction with a system, both for affecting change in the system and for monitoring the health and status of the system. Second, the use of a single system eliminates the need for coding difficult distributed consensus algorithms to synchronize global state changes.

There may be times when we would like the Command Server to have a special channel to the other DI/ODE servers in case of network failure or congestion. Therefore, our design includes a second logical network, called the Command Network, which may be coincident with the Data Network or not, depending on the preference of those deploying the system. Because DI/ODE is intended for use in applications with extremely high I/O requirements, and most of the Network Interface Cards (NICs) that come stock with today's computers aren't up to the task, a site might decide that the built-in interfaces will be used for the Command Network, and higher-end NICs be purchased to connect components to the Data Network, and to connect Applicantion Servers to the Internet. Note, if the Command Network is physically separate from the Data Network, and it fails, the Command Server will attempt to perform its duties over the Data Network, but the converse is not true. Even if the Data Network is down, DI/ODE servers will not pass data over the Command Network.

2.2 Components and Interfaces

The relationship between the components of the DI/ODE system is revealed in Figure 2. On this topic, we start with the application. This can be any application which is able to be linked with libraries written in C and uses, directly or indirectly, the STDIO and/or system calls which access files in some manner. This application runs on an Application Server. There is no limit on the number of Application Servers which may be part of a DI/ODE system. However, certain architectural decisions in the application may limit the system's scalability in practice. For example, if the application can occasionally lose track of the data it's writing such that the whole data space needs to be swept and repaired, and the application cannot run while this is taking place, then for the repair process to proceed, all invocations of the application on all Application Servers would need to be shut down.

The next piece of the system we will discuss are the DI/ODE Client Libraries. Currently, only one of these exist. This library mimics the STDIO and syscall interfaces to a `vnode`-based file system. Upon instantiation, the Client Library reads the file `/etc/diode/filesock.conf` to retreive a list of sockets (may be IP or Unix domain sockets) on which Client Daemons will listen for requests. More than one Client Daemon may run on a given machine. The Client Library will determine which Client Daemon to communicate with by hashing a portion of the data item's namespace. In the file case, this means hashing a portion of the path to the file itself. In practice, we use Consistent Hashing [Kanger97] which costs us very little (except in conceptual simplicity) and helps us a great deal when we have to migrate data from one Data Server to another. The first Client Daemon that starts listens on a known port, our implementation uses

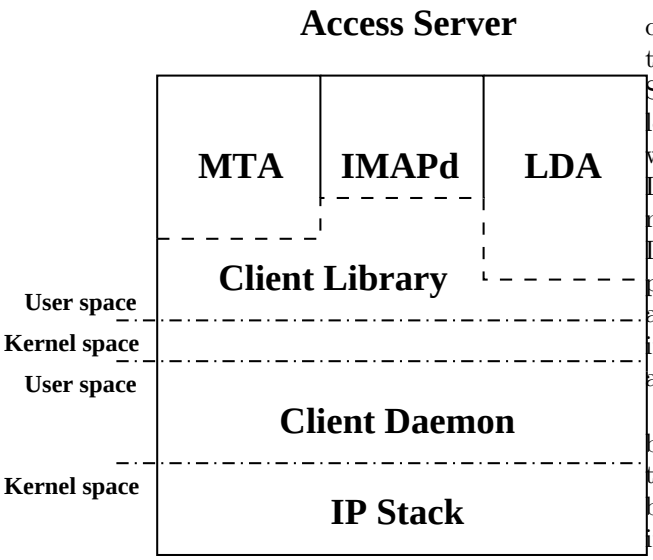

Figure 2: DI/ODE Interfaces

TCP/XXXX, and client daemon number n will listen on port TCP/(XXXX $+n$). Note that all I/O activity happening in the same logical directory will pass through the same Client Daemon on each Application Server.

Additionally, the file /etc/diode/paths.conf contains the list of the logical file paths that are considered to be in the "DI/ODE space" and not local to the server in question. For example, the `paths.conf` file might contain:

```
/var/spool/mqueue//%h
/var/mail//%i/%i/%h
```

The "//" indicates the boundary between the local file system and the DI/ODE space. Everything to the right of this seperator is stored on the Data Servers. Everything to the left of the seperator is local to the Application Server. The "the path that will be used by the hashing algorithm in the Client Library to determine which Client Daemon to communicate with. It will also be hashed by the Client Daemon to determine on which Data Server that particular directory tree resides. The "records that are not used in the hash calculation but still reside in DI/ODE space. These directories must exist on all Data Servers.

When an application linked with the Client Library performs an `open()` of a file, the `open()` that is executed is the version in the Client Library. This function checks to see whether the file in question is under any path name specified in the `/etc/diode/paths.conf` file. If so, then the I/O request is encapsulated in an RPC request and sent to a local Client Daemon to be sent over the Data Network. If not, then the arguments are passed to the "real" `open()` call. From then on, the Client Library tracks which file descriptors are associated with local or remote files and routes calls which operate on data in each space appropriately.

One of the promises of DI/ODE is that it will serve to stripe data from a single logical directory across a number of Data Servers. Therefore, we need some mechanism to determine which files end up on which Data Server. Our level of granularity here is the directory. That is, all files in a single logical directory will end up in the same physical directory on the same Data Server. The Client Daemon runs a hash function over the "by the number of Data Servers available (defined in the `/etc/diode/mount.conf` file) The remainder determines which Data Server in the list (numbered starting with one) on which that particular directory resides. These calculations are all performed by the Client Daemon, and the details of these transactions are completely hidden from the application.

A sample `/etc/diode/mount.conf` file is included below:

```
ds1 1 /var/spool/mqueue /data/queue/as1
nfs3:udp
ds2 2 /var/spool/mqueue /data/queue/old
nfs3:udp
ds1 1 /var/mail /data/mail nfs3:udp
ds2 2 /var/mail /data/mail nfs3:udp
```

The first column lists the Internet host name of the Data Server. The second column is the hash-bin

number corresponding to that Data Server (mapping the output of the Consistent Hash algorithm to an individual entity). The third column is the local file system entry point into DI/ODE space. The fourth column is the remote file system export point. The fifth column lists the Data Protocol and transport mechanism.

As the Data Network in a DI/ODE system is always assumed to be very fast and very local, we can set timeouts to be very low. Packet loss and/or significant response delays are never due to some random event between a client and server, but are local events which must have been caused by a serious incident or server overload. The Client Daemon has a fixed sized request queue. If the number of outstanding requests exceeds a certain threshold, any new requests will be immediately acknowledged back to the Client Library (and, hence, to the application) as failed. However, if this happens the queue will not be immediately flushed. If the queue remains full after a second timer expires, then the Data Server is declared dead by the Appliation Server, the Command Server is notified of this, that Data Server's request queue is flushed, all outstanding requests from applications are replied to as failed. From this point, the Application Server will never send another request to this Data Server, until the Command Server has notified it that this Data Server has been returned to duty. Note: Applications using DI/ODE for I/O must check return values from their I/O operations, and deal with failures in a sane fashion.

The Server Daemon is a high-performance request broker that takes data requests from the network, applies the path to files as a relative path against a top level data repository directory, and then acts on those files in that directory space as requested. The Server Daemon never initiates a conversation with any DI/ODE component. As a corollary of this, no Data Server ever contacts another Data Server.

A program called the Command Server Daemon runs on the Command Server. It keeps in contact with all the DIO/DE nodes. A text based administrative interface called `csterm` runs on the Command Server communicating requests between a human administrator and the Command Server Daemon. The Command Server Daemon listens to a socket and can therefore is the appropriate command-and-control interface between other applications, e.g. automated processes, a GUI, etc. and the DI/ODE system.

The actions issued by the `csterm` are turned into Command Protocol requests and are passed to Client Daemons, or other sorts of actions (for example: `rsh data-server-0 reboot`) as appropriate.

3 Implementation

Now that we have described what the various pieces do, we proceed with some details of our implementation.

Our implementation of the File Client Library is written in C and is thread safe. The only STDIO/syscall functions that we have not implemented are `mmap()` and `syscall()`. There are a few others that have only partially been implemented. For us to implement `mmap()` would either be brutally complex and negate all the performance advantages of using this call, therefore we insist that applications not use it. We don't support the use of `syscall()` because that is our mechanism for actually performing local file operations, so we could not both emulate it and perform local file I/O. Some pieces we have not implemented include `fcntl()` and `lockf()` style locking, device files, and symbolic links. The latter two could be incorportated, but since our applications don't require them, we haven't bothered. Because we don't implement symbolic links, we assign a lexical meaning to the UNIX ".." directory, that is, a path like: `/this/is/a/PATH/../to/a/file` is interpreted as being identical to: `/this/is/a/to/a/file`. This simplifies things a great deal for us.

We have also adapted DI/ODE to provide a similar interface for the Sleepycat [Olson99] Database. For the Sleepycat interface, the regular Sleepycat DB library serves as the Client Library. However, the application must set up the DB environment to use RPC as the communications mechanism to the Database. This feature became available in Sleepycat 3.1.17. The Client Daemon acts as a multiplexor for connections from the applications to several remote instantiations of the Sleepycat DB. We stripe the data across multiple back-end Sleepycat Data Servers in a similar manner as we do for files. We perform the Consistent Hash function on a portion of the DB key. The patterns for hash matching are stored in the `/etc/diode/meteamount.conf` file. [**NPC: Guess we should have an example here.**]

Note, that there are significant restrictions on the operation of this database. First, as of Sleepycat 3.2, while the API is present to perform two-phase commits using the `txn_prepare` statement, this is currently a noop function and there is no way to recover a failed database in this manner. Therefore, all transactions, as well as cursor operations

for which ordering is important, must be confined to a single Sleepycat Data Server. Even more restrictive, Sleepycat uses page-level locking and has no mechanism to detect deadlocks which span multiple distinct databases. We provide this mechanism by having the Client Daemon implement timeouts to the back-end servers, but this is a crude mechanism at best. Applications which deadlock in this manner often will not be able to achieve reasonable performance on this system.

As already mentioned, the Client Protocol for communications between the Client Libraries and the Client Daemons is just ONC RPC.

The Client Daemon is written in a language called Erlang [Armst96]. Erlang is a pragmatically functional language that runs in a virtual machine environment. Although that environment currently has no mutli-processor support, multiple very light weight threads (which Erlang calls processes) and are implemented in the VM using `select()`. This restriction is the reason that we provide the capability for running more than one Client Daemon on each Application Server. Erlang has some very powerful features which make its use in this environment advantageous. They are better chronicled in [Fritch00].

For our file Data Protocol, we elected to use NFS. We made this decision for several reasons. First, the reasons we felt that we couldn't use regular kernel based NFS to solve our data storage problems were primarily due to deficiencies in and lack of control we had over NFS clients, a problem we have rectified in DI/ODE. Second, by using NFS, we wouldn't have to build a File Server Daemon to run on the Data Servers. There exist (a small number of) well designed, high performance NFS servers out there that we can leverage. Third, there just aren't that many ways to remotely do network equivalents of `read()`, `write()`, etc., and the mechanisms used by NFS were good enough. By default, we use NFS v3 over UDP, although our code supports NFS v2 as well as both versions running over TCP.

One problem with NFS is that its locking isn't sufficient for our needs. Therefore, the DI/ODE-ified `flock()` creates lock files on the File Data Servers. These files are actually leases which are kept refreshed transparently by the Client Daemon. If an application with open locks closes its connection with the Client Daemon, it knows that it is safe to delete all that application's lock files. This is another case where the fact that we have control over our NFS client implementation allows us to overcome one of the protocol's deficiencies for these sorts of systems. This is also the reason that we do not support inode-based locking mechanisms like `lockf()` and `fcntl()` at the present time.

Simply because our applications don't need them, there are some NFS commands that we have not implemented, most notably SYMLINK, MKNOD, READLINK, and PATHCONF. These would be easy to add to the Client Daemon, but we have no compelling reason to at the present time.

The DB Data Protocol is merely an RPC encapsulation of the Sleepycat API, and is identical to the mechanism already implemented in Sleepycat [Sleepy00] except that the Client Daemon provides a centralized control point for the system as well as multiplexing features.

On the Server Daemon side, for file I/O one can use any high-quality NFS server, although these are rarer than one might at first expect. At the very least, the server should implement NFS commands like CREATE, RENAME, REMOVE, and LINK atomically, or the applications might not perform correctly.

For the Metadata Server Daemon, we have rewritten the Sleepycat Database server to better meet our needs. The one that ships with the Sleepycat software is designed for debugging, not high performance I/O. Consequently the stock server is synchronous, single threaded, and has no capability to receive requests from multiple clients simultaneously. We have threaded the daemon, added significant buffering, made it asynchronous, and added other improvements. We are contributing this code back to Sleepycat so that they can maintain it and share it with the Internet development community.

One other implementation that's worth mentioning is that we have not implemented any sort of access control, even a trivial model like Unix users and groups, into our system. The Client Daemon "user" owns all the files stored on the File Data Server, and the Metadata Server Daemon doesn't care who sends it requests as long as they come from an Application Server that is known to the Command Server. Adding access control would be straightforward, but we haven't yet found a need to do so.

4 Features

It might appear that this system doesn't have much going for it over the use of kernel-based NFS, JINI, CORBA, or other I/O mechanisms. However, we think this system has some very compelling advantages.

First, we believe that the ease of integration of the

DI/ODE system into existing applications is compelling. After updates are made to the Client Library or Client Daemon we will often test it by linking the system with some familiar applications to see if they will work unmodified. The GNU `fileutils` package is a favorite for this. Of course, neither the GNU `mknod` command nor `ln -s` will work, but we don't consider modifications to our programs to be acceptable unless a suite of familiar utilities work as expected through the DI/ODE system.

Second, we believe that with this system we have better I/O control than other data distribution mechanisms. For example, kernel-based NFS clients are overly generic when it comes to processing client requests. They do not provide fine-grained control over timeouts, don't use information about client exiting to assist in data clean up, and send too many unnecessary events, like LOOKUPs, over the network. Even more dramatically, for all practical purposes we can obtain hard mount-like robustness with failure semantics that don't cause the machine to seize whenever an NFS server becomes temporarily unavailable.

Third, we have a point for centralized monitoring and control. Since we have a process monitoring the whole system end-to-end, we can better control the flow of data to reduce the likelihood and seriousness of local resource deficiencies in the system.

Fourth, we can perform some interesting data gymnastics behind the scenes. In the current release, the most compelling one is our ability to perform on-the-fly data migration from one Data Server to another completely transparently to the application. If a Data Server's I/O or storage capacity is becoming exhausted, we can add a new Data Server, inform the Client Daemons, and they will recalculate their hashes for data locations. At this point, references to the data themselves cause the transfer to occur automatically. Client Daemons in this state are aware that data may be found in either location and understand how to deal with data that might be moved out from under it. At any time, we can also start a background sweeper process which walks the data tree on the Data Server locking and moving data as it pleases.

As one might expect, there are significant restrictions on what sorts of file access we can provide during migration time. One thing that it's especially difficult to deal with is file and directory renaming during the migration of data from one directory to another. Therefore, because it's not required by the applications we're most concerned with, we do not allow file and directory renaming to move a file system entity out of its original directory. That is, a file or directory may be renamed in its current parent directory, but may not be moved to a new parent directory. Also, to avoid confusion among applications, we require that an application have a file open if it is going to rename that file.

The migration process is multiphase. This is so that there are never points at which I/O is halted. Data can continue to be stored and retreived as the system gradually learns about the new location. The one exception to this is directory renaming, which is temporarily stalled during a brief migration period. However, our critical applications do not need to rename directories themselves, so this doesn't concern us.

The data migration scheme used in DI/ODE is very complex, and we can't possibly do it justice here. However, it will be chronicled more thoroughly in [Guenth01].

5 Evaluation

[NPC: Obviously, this section is mostly predictions now, but will need updating later.] Overall, our system works quite well. We're very happy with how easy it is to integrate the DI/ODE Client Libraries with existing applications. So far, we feel that has been its strongest point.

The migration mechanism works, but it has its quirks. Once a migration is started, it must be completed. There is no mechanism to abort a migration and back out of the transfer at this time. It might be nice to have one.

Performance of the system is respectable, but it could stand some improvement. What we really need is a way to get data in and out of the Client Daemon's Erlang virtual machine without copying the data across the user/kernel boundary. We have some ideas on how we might be able to attack this problem.

6 Future Plans

There are a number of improvements we know we want to make for DI/ODE. We're considering whether providing Support for NFS v4 would be worthwhile or not. NFS v4 has a number of compelling features [Pawlow00] which would be useful to us, including real in-band locking. However, it's an order of magnitude more complex than NFS v3. Maybe it would be worthwhile to bypass NFS v4 and implement a DAFS [something] client? The fact that DAFS was designed to be implemented

in user-land, plus the fact that it was designed as a shared-bus technology makes it compelling. We're not sure either direction is worth the effort it would take. Perhaps we should discard NFS altogether and come up with our own Data Protocol? We're also considering more DI/ODE Client Libraries. An ODBC interface is possibly the most compelling.

The biggest area of interest to us are improvements in the form of data redundancy. With control of the Data Protocol client it is fairly straightforward to support a redundant Data Network. Even more intriguing for us is the notion of redundant Data Servers of all varieties. This is *much* more difficult than one might at first think, but we have a design that we're itching to implement. The methods are complex enough, though, that details of its implementation is more appropriate for a separate paper.

A third area that we're interested in is increasing the system's autonomy by improving the Command Server's understanding of the functioning of the system as a whole. We would like the Command Server to continue to improve in alerting administrators to crises, both arrived and impending. As an extension to this, the system will have a great deal of data available to it in order to map out its own resource utilization. It should be possible for the system to autonomously track an increase in the consumption of its own resources, and to compute a best and worst case scenario as to when various components will need to be upgraded or expanded.

We'd like to improve the performance of the system, increasing the number of transactions/second and bytes/second that can be supported on a given Application Server. We have some lines of attack on this problem that we believe will be fruitful, although we're unlikely to ever be able to completely reach performance parity with the use of kernel-based NFS as an access method.

DI/ODE really uses the Data Network as a shared bus running IP as the bus protocol. However, there exist shared bus protocols that are considerably more efficient, such as the Virtual Interface Architecture [VI97]. We find these technologies to be very appealing for our applications.

Other possible interest areas would be to improve the security of the Data and Command Networks, and possibly even to try to extend the DI/ODE framework to make the system workable across a wide-area network.

7 Conclusion

Gee, I don't really know what I want to say here. That's odd.

8 Acknowledgments

Thanks to our management team all the way up the ladder for letting us work on cool stuff. We definitely will be thanking Linda, Allen, and probably consulting folks before this is all over.

References

[something] DAFS reference

[Armst96] J. Armstrong, R. Virding, C. Wikström, M. Williams, *Concurrent Programming in ERLANG, 2nd Ed.*, Prentice Hall International (UK) Limited, London, UK, 1996.

[Christ97] N. Christenson, T. Bosserman, D. Beckemeyer, A Highly Scalable Electronic Mail Service Using Open Systems, *Proceedings of the First USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, 1997, pp. xx–xx.

[Fritch00] S. Fritchie, J. Larson, N. Christenson, D. Jones, L. Öhman, Sendmail Meets Erlang: Experiences using Erlang for Email Applications, *Proceedings of the Sixth International Erlang/OTP User Conference*, Stockholm, Sweden, Oct., 2000.

[Guenth01] P. Guenther, N. Christenson, S. Fritchie, J. Larson, J. Evans, C. Murray, Active Data Migration in the DI/ODE System, To Be Published.

[Horman99] S. Horman, High Capacity Email, `http://www.us.vergenet.net/linux/mail_farm/html/`

[Kanger97] D. Kanger, et. al., Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web, *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, El Paso, TX, May, 1997, pp. 654–663.

[Olson99] M. Olson, K. Bosic, M. Seltzer, Berkeley DB, *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, CA, June, 1999, pp. xx–xx.

[Pawlow00] B. Pawlowski, et. al., The NFS Version 4 Protocol, *2nd International SANE Conference*, Maastricht, Netherlands, May, 2000.

[Saito99] Y. Saito, B. Bershad, H. Levy, Manageability, Availability and Performance in Porcupine: A Highly Scalable, Cluster-Based Mail Service, *Operating Systems Review*, **34**(5):1–15, Dec., 1999.

[Sleepy00] Berkeley DB Reference Guide: RPC Client/Server. Available at: `http://www.sleepycat.com/docs/ref/rpc/intro.html`

[VI97] Virtual Interface Architecture Specification, Version 1.0, December 16, 1997. Available at `http://www.viarch.org/html/Spec/vi_specification_version_10.htm`