Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1989

Efficient Implementation of Modularity in RAID

Charles Koelbel

Fady Lamaa

Bharat Bhargava
Purdue University, bb@cs.purdue.edu

Report Number:
89-893

Koelbel, Charles; Lamaa, Fady; and Bhargava, Bharat, "Efficient Implementation of Modularity in RAID" (1989). *Department of Computer Science Technical Reports.* Paper 760.
https://docs.lib.purdue.edu/cstech/760

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

EFFICIENT IMPLEMENTATION OF MODULARITY IN RAID

Charles Koelbel
Fady Lamaa
Bharat Bhargava

CSD-TR-893
July 1989

Efficient Implementation of Modularity in RAID*

Charles Koelbel
Fady Lamaa
Bharat Bhargava

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

Raid is a distributed database system that is very modular. This paper describes our design, implementation, measurements, and experiences in modifying the system to achieve an efficient implementation without sacrificing the original goals of modularity in the Raid system. This paper describes the rationale behind the modifications that lead to a facility by which several servers can be merged to run in a single operating system process. It includes an account of the changes that were needed in the different layers of the system. We feel that merging servers is a technique that can be applied to improve performance on any server-based system. However the modifications and alternatives for implementation are not easy to evaluate. This research study contributes in this direction. We have presented the data that was collected for database transaction processing using the old version (one server per process) vs. new version (multiple servers per process) in the Raid system.

1 Introduction

RAID is a robust and adaptable distributed database system for transaction processing [3]. Raid is a message passing system with server processes on each site. Raid divides the functions of transaction processing into software modules called servers. An operating system process can implement the capability of a single server or a collection of servers. The server design has facilitated the implementation effort by providing for flexibility, and by explicitly defining the interfaces between servers. This architecture provides for modularity and extensibility which in turn gives the capability to build an adaptable and dynamically reconfigurable system. A high level, layered communication package provides a clean,

*This research is supported by NASA and AIRMICS under grant number NAG-1-676, by UNISYS and AT&T Corporations.

location independent interface between servers. The Raid system is based on a server-processing model similar to the CAMELOT [10], SDD-1 [9], and R* [6] systems. Naive implementations of server based designs, however, often have high overheads for communications between servers. In RAID, for example, a single message round-trip typically takes 10 milliseconds; for comparison, a server's computation for a transaction may only take 40 milliseconds. System throughput could be improved if the interprocess communications time were reduced to a few milliseconds or hundreds of microseconds. This paper describes an approach to reducing this overhead which can be applied to any server-based system.

The current design provides for two versions of the Raid system. The first version runs with each server in an asynchronous process. The second version combines the servers that do not need to be asynchronous into a single process. Since our objective has been to conduct scientific experiments and measurements on various protocols for transaction processing and system configurations the first version has been very convenient. However, its performance was not satisfactory, particularly in terms of the contribution of communications overhead. We decided to tune this system and build the second configuration with merged servers to see how much improvements can be achieved. We wanted to experiment with various alternatives for the implementation of the merged server version and to gain experience that can be useful in the work of other research projects.

The original implementation of RAID had an attractive server model, but an inefficient implementation of that model. Our goals for the modification for the new version were to

- Decrease transaction latency in RAID by decreasing server overheads.
- Retain the same basic design.
- Retain as much modularity as possible.

We achieved these goals by merging conceptually separate servers into a single physical entity.

The original implementation of servers in RAID used one process for each server and a general-purpose communications protocol for interprocess communication. In addition to the communications overheads, this lead to excessive context switching, particularly since there are many servers in the system. There are several ways of reducing these overheads in RAID:

- *Use a special-purpose communications protocol between servers.* The protocol can be tailored to the needs of the current system, and need not pay a price for unused functionality. This reduces communications overhead, but does little for context switching. [1] discusses work on this idea done in RAID.
- *Write the system in an object-oriented language*, using the basic server design. This approach shifts the burden of message passing onto the language run-time environment, which is presumably more efficient than general-purpose operating systems. This is particularly attractive if the implementation language uses the same syntax for both local and remote object references, as Objective-C does [5]. Depending on the language, this may reduce communications overhead, context-switching overhead, or both. Emerald [4] is an example of a system that uses this method.

- *Implement servers as lightweight processes (threads)*, rather than full UNIX processes. The threads' shared memory space can then be used for message passing, rather than costly operating system primitives. This saves both communications and context-switching overhead. The Argus project [7] took this approach by building their own threads package. Many modern operating systems such as Mach [8] also have thread primitives.

- *Merge several servers into the same process* rather than using a separate process for each server. This approach attempts to convert inter-process communication into simple data movement within the same process. Communications overhead is reduced by this conversion, while context switching is reduced because there are fewer processes. The rest of this paper will focus on this possibility.

All of these methods are generally applicable to server-based systems. The methods are not mutually exclusive; for example, it is reasonable to merge some servers and use a special-purpose protocol for communications between the other servers.

Performance gains for each of the above methods can be estimated. Using a special-purpose communications protocol may save 50% or more of the time spent on all communications. If 25% of the system execution time is spent in communications, this increases performance by 12.5%. Object-oriented languages have similar savings on communications, but tend to have higher overheads on other computations, so overall performance gains will tend to be lower. Good implementations of threads can reduce the time for a single context switch from hundreds of microseconds to tens of microseconds, and the time for a single communications call can fall from milliseconds to microseconds. If these two overheads together account for 50% of the system running time, performance gains can exceed 40%. The improvement for communications time using merged servers is similar to using threads for servers mapped into the same process; context switching between the merged servers is eliminated entirely. The total time for these overheads will be less than the 50% we quoted for using threads, since not *all* communications or context switches are affected (unless all servers are merged). Assuming 40% of the total time is used by overheads between the merged servers, the performance gains would be about 30%. Similar analyses could be done for other server-based systems. In general, the higher the system overheads are, the more these techniques can gain in performance.

Note that merging servers does not change the basic server model; only the implementation of that model changes. (This is also true of the other changes noted above.) This is similar to the difference between the definition of the ISO network layers and non-layered implementations of the ISO standard [11]. Retaining the same server model is important, since experience with the original system can then be transferred directly to the new one. Also, it enables clear comparisons between the two systems. Merging servers will also create a system with good modularity if the merging is done intelligently; the conceptual servers become the modules of the implementation. This was very important in our work, since experimental systems like RAID must be adapted and extended frequently. Monolithic systems are notoriously difficult to modify. A final advantage of the merged-server approach is that it requires little or no modification to the server codes themselves.

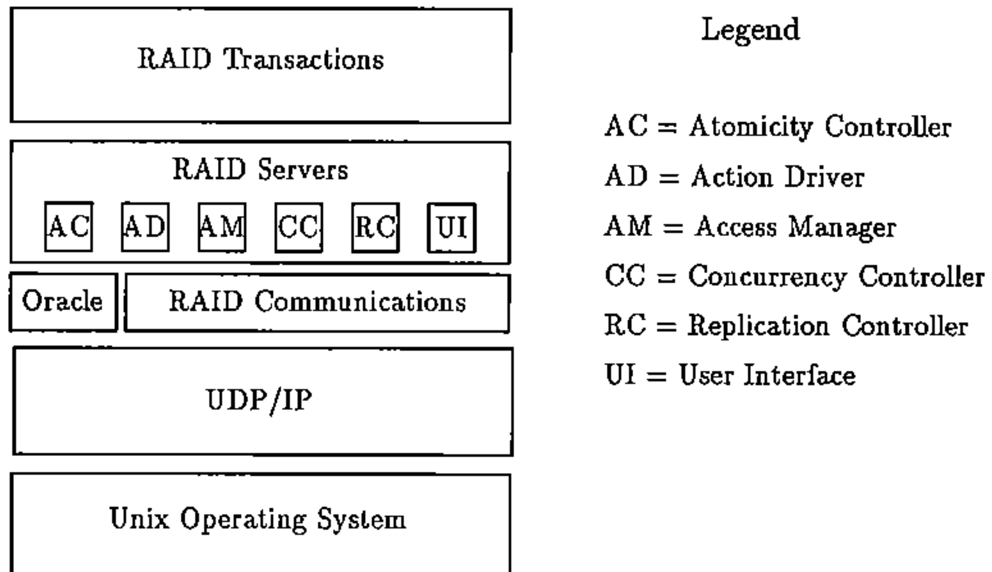

Figure 1: The RAID hierarchy.

Merging servers does not change the servers' internal processing, only their packaging with respect to each other. Our implementation of RAID made no change to the server codes at all; only a high-level controller was added.

The next two sections describe the original and new RAID organizations in more detail. Section 4 then describes our implementation of the new organization. Section 5 compares the performance of the old and new systems. Finally, Section 6 gives some conclusions.

2 The Original RAID Implementation

The RAID system is organized in the hierarchy shown in Figure 1. In this paper we will focus on the server and communications layers. Each of the servers shown in the figure responds to service requests from the other servers in a clearly defined manner. The RAID communications package uses UDP to implement communications between servers. It provides a number of extensions to UDP, including

- A high-level naming scheme
- Location independence of servers
- Arbitrary sized messages
- Multicast support

The first two extensions are major advantages of the RAID communications package, and are closely related. Each server has a RAID address consisting of its type, site number, RAID number (several independent instances of RAID can be running simultaneously), and sequence number (several servers of the same type may be running on one site).

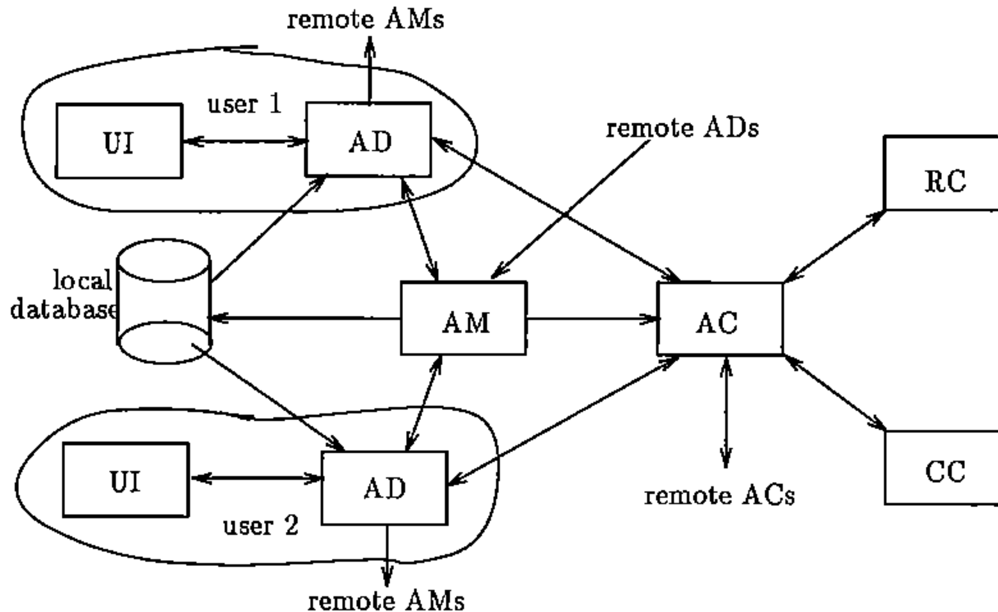

Figure 2: The conceptual organization of a RAID site.

This provides a natural way to send a message to another server and creates location transparency in the server codes. A server need not know the physical locations of other servers, only their RAID addresses. This had important implications when merging the server codes. An oracle server registers RAID servers and their UDP addresses as they begin execution and distributes this information to other servers. The information is stored by the communications routines and used to translate between RAID addresses to UDP addresses when messages are sent and received.

Figure 2 shows the pattern of server communications in RAID. Each box in the figure represents a RAID server. Arrows represent service requests from one server to another (some arrows represent more than one service request). Unboxed server names represent servers on other sites. The roles of the servers in the RAID system are

- **User Interface (UI)**: a front end invoked by the user to process relational calculus queries.
- **Action Driver (AD)**: accept a parsed query from the UI, format the query as a transaction (read and write actions), and execute the transaction.
- **Access Manager (AM)**: provide write access to the local database, ensuring that updates are posted atomically to stable storage.
- **Atomicity Controller (AC)**: manage the two commit phases of transaction processing to ensure that a transaction commits or aborts globally.
- **Replication Controller (RC)**: maintain consistency of replicated copies of the database in the event of multiple site failures.

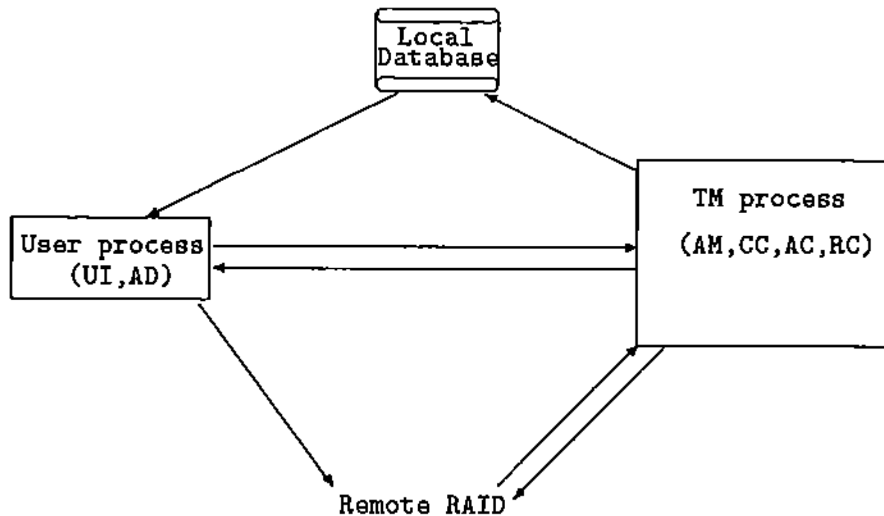

Figure 3: The physical organization of a RAID site.

- **Concurrency Controller (CC)**: check whether a transaction history is locally serializable at a given site.

One of the main goals of the RAID system was to provide modularity and reconfigurability to allow experimental studies into new methods of distributed processing. This was accomplished by making each server on a site reside in a different UNIX process. The servers communicate with each other using the RAID communications routines. Because of the location transparency provided by those routines, the site and its servers are not tied to any particular host on the network. The design is very attractive because of its adaptability, but the performance of the system suffers because of high communication costs and excessive context switching between the multiple processes competing for CPU time. The study in [2] shows that only a small fraction of the wall-clock time used on a given transaction is directly attributable to server processing; the rest must be attributed to system overhead. We therefore investigated alternatives for reducing this overhead.

3 The New RAID Structure

Our decision to reduce system overheads in RAID by merging conceptually separate servers into one process forced other choices. Merging servers did not change the conceptual organization of RAID from the hierarchy of Figure 1 and the communications of Figure 2. Only the packaging of servers into processes changed. We decided to merge the AC, AM, CC, and RC into one process, the *Transaction Manager* (hereafter referred to as the TM). The UI and AD were merged into the *User* process. This new organization of a site is shown in Figure 3. The only part of this design that was peculiar to RAID was the choice of servers to merge. The general strategy of merging servers can be applied to any server-

based design. It is profitable whenever the system and communications overheads are high.

The division of servers between the TM and User processes was chosen for pragmatic reasons. The AC, AM, CC, and RC servers at a RAID site execute as long as the site is up, while the UI and AD appear and disappear as users come and go. It does not make sense to package permanent servers like the AC with temporary servers like the UI because the server lives are so different. The UI and AD are very closely associated with each other, so packaging them together was a natural decision. It is less clear that having all four servers packaged in one TM process is an advantage, since several transactions might be processed concurrently if the servers were in different processes. This would be possible if there were long latencies in the servers (as there are in the AM) or if the processes could run in parallel (for example, if RAID were ported to a multi-processor). For future investigation of these possibilities, we designed the TM to be configurable at run time with any combination of the four servers. For example, the AC, CC, and RC could be grouped together, with the AM being run in a separate process. This allows us to test and compare various configurations of servers in processes directly.

Several options were open to us for how the servers would be merged.

- *Implement servers as subroutine libraries in the same process.* The major advantage of this approach is that it is very fast, since it can completely avoid copying data. Thus, the only cost of communicating with a server is a subroutine call, which is on the order of microseconds. The disadvantage is that not all servers act as subroutines. The AC, for example, often can run asynchronously with its caller, so it would be a poor candidate for a subroutine library. If the AM used asynchronous disk writes, it would also be a poor choice as a subroutine. We also felt that modularity and extensibility might suffer, under this design. Because these features are important to the RAID project, we did not use this approach for the TM. In the User process, the disadvantages are much less severe than for the TM. The AD does operate as a set of subroutines, and we did not foresee problems with modularity or extensibility. The User process is currently being written implementing the AD as a subroutine library.

- *Call server routines directly from the communications routines on internal messages*, and use the same interface for messages to internal and external servers. A server wishing to communicate with another server would call the appropriate communications routine. If the message destination were in the same process, the server routine would be called; otherwise, the message would be sent via UDP. This approach has the advantage that server codes need not be rewritten, and it is almost as fast as the first alternative. It has the same disadvantages as the previous approach, however, and it severely violates the layering hierarchy of RAID (Figure 1). Routines at one level of the RAID hierarchy should only call routines at a lower level; thus, the communications routines should not call server codes. Such calls would compromise modularity by making the RAID communications routines less general. Since we give modularity a high priority, we rejected this option.

- *Copy internal messages to a queue*, and use the same interface for messages to internal and external servers. The use of the communications routines is similar to the last alternative. Like that alternative, it requires no changes to the server codes. The problem of servers not behaving as subroutines does not apply here, however, nor is the layering scheme violated. The major disadvantage of this method is that it is not as fast as simple subroutine calls, since it requires copying data. (Enqueuing the data without copying is not feasible, since the data can then be overwritten or even deallocated before the message is received.) This is the method that we used to write the TM.

4 Implementation

This section describes the changes made to the different parts of the RAID code to implement the design in Section 3. The next two sections will describe separately the implementation details of the TM and User processes.

4.1 Implementation of the Transaction Manager (TM)

Most of the work involved is related to this part of the RAID software. Changes had to be made to the communication package, the RAID server code, and the system interface. The next three sections describe all these changes in detail.

4.1.1 Changes to the oracle and communications library

Two basic considerations drove most of our changes to the communications library:

- Multiple processes can now be present in the same process. This was not true under the original implementation; in fact, parts of the communication library depended on having a one-to-one correspondence between processes and servers.

- Internal communication (i.e. messages between servers in the same process) must be efficient. Using UDP to send a message from a process to itself is wasteful.

The first consideration forced us to use a more general addressing translation than the old version of RAID used, while the second forced us to add more code checking for the internal communication case. We will describe the latter change first.

Shared queues are now used for communication among servers residing in the same process. The sending routine, SendPacket, now enqueues the messages if the destination is internal to the process and use UDP otherwise. This requires just one test of the destination address and a short section of code putting the message on the queue. The receiving routine, RecvMsg, now checks for both internal and external messages when called. Priority is given to internal messages by first checking whether the queue is empty. If an internal message is found, it is returned immediately. Otherwise, RecvMsg listens at the UDP socket for an external message. We chose this priority because internal messages

are more likely to be related to the currently active transaction; also, testing the queue is faster than listening at a UDP socket.

It is important to note that these changes were only made to the internals of the SendPacket and RecvMsg routines, not to their interface with the rest of the program. Because this is true, the packaging of servers into processes is transparent to the servers themselves. A server can use the same procedure to deliver a message to either an internal server or an external server. Similar techniques could be used in any server-based system that used virtual server addresses. In general, this allows the servers to be merged making few, if any, changes to the servers themselves.

Having merged servers causes an interesting problem when receiving messages. When a single server resides within each process, there is no need to check which server should receive the message since there is only one possibility. With merged servers, the target server for a given message is not always known. Some way must be available to determine the correct server. Our solution was to include the recipient's RAID address in the message header and have the *receiving* routine read it. We added a new routine, RecvMsgAddr, that reads the destination field and returns it via pointer parameters. The sender's RAID address was already included in messages in this way to facilitate return messages. For backwards compatibility, we kept the old routine, RecvMsg, that ignores the destination.

One minor change was made to the oracle. As part of its initiation, each RAID server registers with the oracle and requests addresses of other servers from the oracle for use in future communications. Under the old RAID implementation, there was a one-to-one correspondence between servers and UDP sockets. Clearly, this is inefficient if several servers are in the same process and can share a socket. The new system allows more than one server to register using the same socket. All RAID communications are sent to that socket; the destination address field is used to dispatch the message to the correct server. The new initialization also excludes internal servers from the list of oracle requests, since their location is already known.

4.1.2 Changes to the server codes

The main structure of each individual server was a main loop that continuously received messages and called the appropriate routines to handle the requests. Figure 4 shows the pseudo-code for the AC as an example. With all the servers merged together, there is only one main loop for all servers, shown in pseudo-code in Figure 5. Messages to the process are demultiplexed to the appropriate server using the destination address field. The server processing routines are unchanged. This new design implies that the servers will execute synchronously. We discuss the loss of concurrency from this decision in Section 6.

4.1.3 Changes to the system interface

We decided to design the TM so that it could be configured with various combinations of merged servers, rather than always containing the AC, AM, CC, and RC. The intent was to allow experimentation with different combinations of merged servers, either to optimize the grouping or simply to observe performance effects. To allow such reconfiguration, we

```
main( )
{
    InitializeAC( );
    while ( TRUE ) {
        MsgType = RecvMsg( MsgBody, &RAIDAddr );
        ProcessACMsg( MsgType, MsgBody, RAIDAddr );
    }
}

ProcessACMsg( MsgType, MsgBody, RAIDAddr );
{
    switch ( MsgType ) {
      case AD_REQUEST:
        ProcessADRequest( MsgBody, RAIDAddr );
        break;
            ⋮
      default:
        ReportError( "Unknown messge type" );
        break;
    }
}
```

Figure 4: Pseudo-code for AC server.

```
main( )
{
    InitializeAll( );
    while ( TRUE ) {
        MsgType = RecvMsgAddr( MsgBody, &SendAddr, &RecvAddr );
        switch ( RecvAddr.ServerType ) {
          case AC_TYPE:
            ProcessACMsg( MsgType, MsgBody, SendAddr );
            break;
          case AM_TYPE:
            ProcessAMMsg( MsgType, MsgBody, SendAddr );
            break;
          case CC_TYPE:
            ProcessCCMsg( MsgType, MsgBody, SendAddr );
            break;
          case RC_TYPE:
            ProcessRCMsg( MsgType, MsgBody, SendAddr );
            break;
          default:
            ReportError( "Unknown server type" );
            break;
    }
}
```

Figure 5: Pseudo-code for Transaction Manager.

decided to load all the compiled servers into one program and use command line options to instantiate the appropriate servers within the TM. In addition to the new program, we wrote a new version of the RAID "start-site" shellscript. This shellscript starts the servers for an entire site. Formerly, it invoked separate programs for each of the servers; now it only invokes the TM. Some new options were also added to the shellscript to control the grouping of servers into processes. The default configuration is to have all servers running in one process.

4.2 Implementation of the User Process

The UI and AD in the original RAID implementation were written very early in the research project, before the RAID communications library was designed. Because our priorities were in the distributed processing aspects of RAID, the UI and AD were never thoroughly integrated with the rest of the system. In particular, they do not exploit the location independence offered by the RAID communications routines. This made merging the servers into the User process more complex than the TM process. We report here on a first attempt at that merging.

In the original version, the UI forked an AD process and communicated with it using UNIX pipes. The UI then forked another program to parse the user transaction, read that program's output from another pipe and copied it to the AD's "read" pipe. The UI read the final result from the AD's "write" pipe and copied it to standard output. The AD *continuously* read from one pipe, processed the transaction sent, and wrote the result to the other pipe. We merged the UI and AD servers into one process to avoid the UI-AD pipe. The AD is now a subroutine that takes its input directly from the parser output and writes its result directly to standard output. This is a slight improvement on the original design, but is still inelegant. A new version of the User process is currently being written which merges the servers by implementing the AD as a subroutine library. We hope this will make the new program both more efficient and more maintainable.

5 Measurements of Original and New RAID Systems

We made several measurements on the new RAID system to compare it with the old version. The first set of measurements compares the costs of the new internal communication routines. Since the implementation of the new system involved creating two independent merged servers (the TM and the User process), two additional sets of measurements for transaction processing time were collected. One was gathered with only the TM being active, i.e. the UI and AD were still separate processes. The other set was collected with both merged servers being active. The transaction benchmarks and measurements are as described in [3,2].

Bytes:	64	512	2048	8192
External (using UDP)	11.9	20.1	46.7	153.3
Internal (using queues)	1.1	2.7	8.3	30.7

Table 1: Round trip external and internal communication times by packet length (in milliseconds)

Transaction	1 site	2 sites	3 sites	4 sites
select one tuple	0.3	0.3	0.4	0.4
select eleven tuples	0.4	0.4	0.4	0.4
insert twenty tuples	0.6	0.6	0.8	0.8
update one tuple	0.4	0.4	0.4	0.4

Table 2: Transaction execution time for original RAID system (in seconds).

5.1 Measurements of Communication Times

In order to explain the performance times obtained from the new design, some measurements were collected on the new communications used. Table 1 compares the time to send messages of different sizes using the old RAID communication routines (built on top of UDP) with the time for the new internal message queues. Communication between servers in different processes still uses UDP; the times for those messages is essentially unchanged. Our new routines, however, require 80 to 90 percent less time for internal messages.

5.2 Measurements of Transaction Execution Times

Table 2 (taken from [3]) shows the times taken for transaction processing for several database queries using the original RAID system. The times include only the cost of committing the transaction; cost of parsing the query is ignored. Table 3 shows the times for the same queries using the TM process, but with the old UI and AD. The configuration for the TM packaged the AC, AM, CC, and RC together. For one site, the new system is 33 to 55 percent faster than the old. For four sites, the figures show a speedup of 7.5 to 20 percent. The lower speedup for multiple sites is to be expected, since they require external communications for site-to-site messages.

These improvements approximate our expectations based on the improvement in message-passing times and the numbers of messages converted from external to internal. The amount of time saved for each transaction with the new design is proportional to the reduction in the number of UDP messages. The first two rows of the table only involve reading items from the database, which saves 4 UDP messages out of 6 total messages (in the single-site case). Thus, if the average packet length were 512 bytes, Table 1 shows that the time savings from communications alone would be $4 \times (20.1 - 2.7) = 69.6$ milliseconds.

Transaction	1 site	2 sites	3 sites	4 sites
select one tuple	0.18	0.19	0.19	0.32
select eleven tuples	0.18	0.19	0.22	0.34
insert twenty tuples	0.40	0.40	0.45	0.64
update one tuple	0.22	0.22	0.22	0.37

Table 3: Transaction execution time for RAID system with new TM and old UI and AD (in seconds).

Transaction	1 site	2 sites	3 sites	4 sites
select one tuple	0.12	0.13	0.13	0.25
select eleven tuples	0.13	0.16	0.17	0.31
insert twenty tuples	0.35	0.37	0.37	0.57
update one tuple	0.16	0.17	0.18	0.32

Table 4: Transaction execution time for RAID system with new TM and User Processes (in seconds).

This is just over half the actual speedup of 120 milliseconds for selecting one tuple on one processor, which is reasonable for such a simple estimate. The last two rows involve writing to the database, saving 3 more UDP messages of an additional 5 messages (again, in the single-site case). Here, the expected communications speedup is $7 \times (20.1 - 2.7) = 121.8$ milliseconds, again in reasonable agreement with the experimental data. In both cases, more messages are needed in the multiple-site case to coordinate with remote sites; these must be sent via UDP, so no further speedups can be expected.

Table 4 shows similar transaction execution times in RAID systems containing both a TM and a User process. The speedups range from 42 to 68 percent for one site and 20 to 38 percent for 4 sites.

Almost all of the improvement between Tables 3 and 4 is due to fixing a performance bug in the original AD (file descriptors were being closed without first being opened). We cannot credit this improvement to our merged server design; it was simply a case of stumbling over an old bug and fixing it. If the time for parsing the transaction were also included in the above numbers, there would be a further time savings for the User process. This is because one level of indirection (the UI-AD pipe) has been eliminated.

6 Analysis and Conclusion

We have accomplished our goals in this project: to improve the performance of the RAID system while maintaining its modularity. We achieved this by creating a difference between the virtual system architecture (the communications links in Figure 2) and the actual

implementation (the process packaging in Figure 3). In addition, we think that future changes to the RAID system can be made within our "virtual server" implementation. This idea is currently being tested, as some changes to the virtual system architecture are being implemented at the server level without modifying the communications routines or the TM main loop. Similar merging of conceptual servers can be done on any server-based system. If implemented carefully, it can eliminate overhead without sacrificing modularity or redesigning the conceptual system. In fact, it is possible to do the merge without changes to the server routines themselves.

While merging all four servers certainly minimizes communications cost, it also forces the servers to run synchronously. This may be a disadvantage if many transactions are run concurrently or if RAID is ported to a multi-processor machine. Since the RC and CC only communicate with their local AC, the best configuration on a single processor should include all three servers in one process. Concurrency is not an issue in this case, since the RC and CC cannot run in parallel (for the same transaction) and only one process can be executing. The AM also communicates with the AC so we do save one UDP message by including the AM in the same process. However, the AM's main job is to write data to disk. Since we only use synchronous I/O, it may be an advantage to run the AM in a separate process even on a uniprocessor to avoid blocking the entire TM. If the machine running RAID is a multi-processor, we may have to redesign the system to exploit its concurrency. As a simple example, it would be useful to have each server working on a different transaction in parallel on such a system. We are currently experimenting with different configurations of the TM to determine how much effect this loss of concurrency has.

We close with some lessons we've learned that we think are applicable to other distributed systems:

- Modularity is important to accommodate new algorithms and techniques in research systems.
- It is possible to have both modularity and reasonable performance.
- The modularity of the implementation (process packaging) need not reflect the modularity of the design (servers).
- Overhead in a distributed system can be reduced by collecting conceptual servers into a single physical entity.

Acknowledgements

We would like to thank Dr. Jim Gray of Tandem Corporation for suggestions that led to the merged server design. John Riedl was helpful in the initial stages of our design. We thank all all the students at Purdue University and the University of Pittsburg who have contributed to RAID over the years. Our work would not have been possible without theirs.

Bibliography

References

[1] Bharat Bhargava, Enrique Mafla, John Riedl, and Bradley Sauder. Implementation and measurements of an efficient communication facility for distributed database systems. In *Proceedings of the 5th IEEE Data Engineering Conference*, Los Angeles, CA, February 1989.

[2] Bharat Bhargava and John Riedl. Implementation of RAID. In *Proc. of the 7th IEEE Symposium on Reliability in Distributed Systems*, Columbus, Ohio, October 1988.

[3] Bharat Bhargava and John Riedl. The RAID distributed database system. *IEEE Transactions on Software Engineering*, 15(6), June 1989.

[4] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-12(12), December 1986.

[5] Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 1986.

[6] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in R*: A distributed database manager. *ACM Transactions on Computer Systems*, 2(1), February 1984.

[7] Barbara Liskov and R. Scheifler. Guardians and actions: Lingustic support for distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[8] Richard F. Rashid. Threads of a new system. *Unix Review*, 4(8):37–49, August 1986.

[9] J.B. Rothnie et al. Introduction to a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 5:1–17, March 1980.

[10] *Guide to the CAMELOT Distributed Transaction Facility.* Carnegie Mellon Computer Science Department, 0.98(51) edition, May 1988.

[11] Richard W. Watson and Sandy A. Mamrak. Gaining efficiency in transport services by appropriate design and implementation choices. *ACM Transactions on Computer Systems*, 5(2):97–120, May 1987.