Design of the BioSCAN Server Software

Doug L. Hoffman
hoffman@cs.unc.edu

Version 3.0
14 April, 1993

Department of Computer Science
University of North Carolina at Chapel Hill
Campus Box 3175, Sitterson Hall
Chapel Hill, North Carolina 27599-3175

Contents

1 Introduction

This paper is an exploration of the design goals for the Biological Sequence Comparative Analysis Node (BioSCAN) network server software and of the impact that these goals had on the overall structure and implementation of that software. The primary audience for this paper consists of computer scientists and computational biologists involved in developing similar server software. Biologists who are users of the BioSCAN computational node and have a desire for deeper understanding of how the server functions will also find this paper useful. It is assumed that the reader is familiar with UNIX[1] and with basic networking concepts. The peculiarities of implementing a network server for a batch resource will be identified and the solutions chosen by the BioSCAN design team explained. Research for the BioSCAN project, including the design of the server software, was supported in part by NSF grant MIP-9024585.

2 Background

2.1 Biological Sequence Matching: the Human Genome Initiative

Since the National Institutes of Health announced the Human Genome Initiative in 1990 [15] there has been a rapid increase in the volume of biological sequence data collected each year. The goal of the Human Genome Initiative is to map the entire genetic blueprint of mankind. The code in which this blueprint is written is deoxyribonucleic acid (`DNA`). `DNA` molecules are long strings made up of thousands of nucleotides. Each nucleotide is identified

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

by which base compound it contains: Adenine, Cytosine, Guanine, or Thymine. This makes it possible to record DNA sequences as character strings composed of the characters a, c, g, and t.

In the process of mapping the human genome as many as three billion characters of genetic sequence information will have to be isolated, identified, and cataloged. In addition to identifying DNA sequences, associated ribonucleic acid (RNA) sequences and protein sequences also need to be identified and recorded.

Databases of known biological sequences are being maintained at various sites around the world. Great efforts are made to ensure timely distribution of this information to sites involved in the cataloging effort. This is because, after the considerable task of isolating and identifying the sequence of bases in a nucleic acid or protein fragment, each new sequence must be checked against the existing catalog of sequences.

The GenBank database [8], maintained by the National Center for Biotechnology Information, contains more than eighty thousand DNA and RNA sequences, totaling over one hundred million characters. The SWISS-PROT database [8] contains over twenty-six thousand protein sequences with a total of eight million characters. These databases and others have been growing at exponential rates with doubling times between 18 and 24 months [4]. Given the large amount of data to be searched the time required for pattern matching becomes important.

At first glance, this does not sound like an onerous task—algorithms for pattern matching between text strings abound. The best of these algorithms are quite efficient (see [3] and [11]). There is, however, a complication; in biological sequence matching the patterns need not match exactly and in fact seldom do. Though efficient algorithms for exact pattern matching are well known there are no inexact matching algorithms with similar performance. There are several very different approaches to this problem [12].

Numerous software based approaches to this problem have been proposed, including the FASTA [16] and BLAST [2] algorithms. Though these methods are fast and useful a number of groups have implemented or proposed more ambitious inexact pattern matching algorithms as custom hardware designs [13, 14, 7, 9] . Among the groups building custom hardware is the BioSCAN project based at the University of North Carolina at Chapel Hill.

2.2 The BioSCAN Project

The BioSCAN project takes its name from the Biological Sequence Comparative Analysis Node that was first described by C. T. White in 1991 [17] and later expanded on by W. D. Dettloff, R. K. Singh, C. T. White and B. W. Erickson [6]. The foundation of the BioSCAN hardware is an algorithm based on work done by S. Karlin and S. F. Altschul [10, 1]. The BioSCAN project is a cooperative venture combining the efforts of the Department of Computer Science, the Department of Chemistry, and the Program in Molecular Biology and Biotechnology at the University of North Carolina at Chapel Hill with those of MCNC (formerly the Microelectronics Center of North Carolina). The remainder of this section briefly describes the BioSCAN hardware and introduces the design requirements for the BioSCAN server software.

2.3 Hardware Description

The BioSCAN hardware is a massively parallel systolic array processor system built around an Application Specific Integrated Circuit (ASIC) VLSI chip. Each ASIC chip contains 812 individual processing elements (PE). The processor board can accommodate a maximum of 16 chips for a total of 12992 PEs. If all the processing elements are in use the processor board is capable of a theoretical maximum of 25.984 billion character comparisons per second.

Each PE is a bit-serial processor that performs the fundamental inner loop of the BioSCAN algorithm in 16 primary clock cycles. When running the hardware with a 32 MHz primary clock rate, 2 million characters are processed each second. The following C program performs the same operations that each PE does in silicon.

```
 #define M  28
 #define N  28

/* BioSCAN declaration */

 int            Th,             /* threshold */
                T[M][N],        /* similarity table */
                La, Lb;         /* length of A & B */
 char           *A,             /* sequence A[La] */
                *B;             /* sequence B[Lb] */
 int            *S,             /* similarity scores */
                i, j;           /* index A and B */

 extern report();

/* BioSCAN algorithm */

 main()
 {
    for (i=0; i<Lb; ++i) S[i] = 0;  /* initialize scores */
```

```
    for (i=0; i<La; ++i) {                          /* elements of A   */
      for (j=Lb-1; j>=0; --j) {                     /* elements of B   */
          if (S[j] < 0)
              S[j+1] = T[A[i]][B[j]];
          else if (S[j] >= Th)
              S[j+1] = S[j];                        /* stuck at >= Th */
          else
              S[j+1] = S[j] + T[A[i]][B[j]];        /* accumulate      */
          }
      if (S[Lb] >= Th) report(i);
    }
}
```

The sequence A represents the concatenation of all the database sequences to be searched for pattern matches. The sequence B represents the sequence to be matched. The table T contains values that represent the significance of a match between any two sequence characters. When a sequence of characters from A generates a sufficiently good match with B, detected by the sum of the T values exceeding the threshold value, the position of the A string in the database is reported.

The processing rate of the hardware is several orders of magnitude greater than that of a general purpose computer performing the same algorithm. Note that the sequences flagged by the BioSCAN hardware are simply interesting in terms of some statistical measure. Once these interesting sequences have been identified further processing is required to yield final results. This further processing is not performed on the BioSCAN hardware.

2.4 System Software Design Goals

From the start of the BioSCAN project it was assumed that some form of server software would be required to make the special purpose hardware available to a large number of users. The BioSCAN hardware is basically a batch type resource. When a job is started on the BioSCAN hardware it can run to completion or be aborted. It cannot be interrupted and restarted at a later time. The large amount of state maintained in the registers, tables, and processing elements of the processor hardware make context switching costly and undesirable.

If a system has only one user this batch orientation is not a great functional limitation as the user will usually need to run jobs sequentially. If multiple users wish to have access to the BioSCAN hardware some provision must be made for queueing user requests in an efficient and orderly manner. It would also be advantageous to allow access to the hardware from many workstations via a network. Since BioSCAN boards are scarce and expensive it is desirable to keep them as fully utilized as possible.

The nature of the underlying hardware, coupled with the desire of molecular biologists for an interactive research tool, helped to identify a number of primary design goals.

- Support for several concurrent client-server connections. Molecular biologists and chemists working with the software development team wanted to use the BioSCAN hardware as though it was attached to their own workstations.

- A single job queue to allow for optimal scheduling of the BioSCAN hardware resources. To get the greatest use out of the hardware it must be kept as busy as possible. What type of scheduling would be needed was an open question but a single scheduling mechanism was indicated.

- Support for multiple BioSCAN hardware devices in the same server node. The ongoing hardware development effort required that a single server be able to support more than one BioSCAN board.

These goals were augmented by the following secondary goals, goals driven by the needs and desires of the software developers.

- The ability to distribute server functions over multiple processors. From the beginning it was not certain that all of the server functions could be performed on a single processor while maintaining peak data flow to the custom hardware. Making provisions to distribute server functions across multiple processors seemed prudent.

- Avoid the need to maintain long duration network connections between clients and the server. Network congestion at other server sites indicated the need to keep connection time as short as possible.

- The use of generic UNIX program facilities to maintain maximum portability of the server software. We wanted to avoid the use of any commercial software packages that would require licensing fees and any public domain packages that were not in wide use.

- Simplify the coding requirements as much as possible. The tight project schedule and the limited number of software developers working on the project left little time to write large volumes of code.

After identifying the major goals of the server software, and being mindful of the limited development resources available, the following areas required major design decisions.

- The low level communication protocol. The act of exchanging data between a client and the server requires certain fundamental capabilities regardless of the nature of the service being provided. These capabilities center on reliability of transmission and detection of errors.

- The high level communication protocol. The division of labor between the client and server software, data transmission volume, and ease of future modification all impact the design of the high level protocol.

- The number and nature of the server processes. Ease of implementation, efficiency, security, and logical decomposition of server functions dictate the overall structure of interacting server processes.

In the sections that follow, each of these subjects will be addressed in turn.

3 Low Level Protocol Issues

In designing server software for the BioSCAN project we were faced with a fundamental choice of communication protocol: use reliable stream transport and accept whatever overhead comes with it or use connectionless datagram delivery and code the routines necessary for reliable delivery ourselves. Recently, protocols layered on top of datagram delivery have become popular, particularly in network wide servers such as ARCHIE and PROSPERO. The classic stream based server has been somewhat eclipsed as the standard architecture for implementing new server software. This section will investigate the strengths and weaknesses of these approaches to server implementation with particular emphasis on suitability for the BioSCAN project.

3.1 User Datagram Protocol

The User Datagram Protocol (`UDP`) has two main properties that differentiate it from the Transmission Control Protocol discussed in the next section. These properties are *unreliability* and *connectionless delivery*. It is the combination of these two properties that allow `UDP` to have a very simple implementation with consequently low transmission overhead.

The first of these, unreliability, simply means that delivery of a datagram is not guaranteed. No notification is provided regarding message arrival or loss and no feedback is provided about data rates between machines. A packet may be lost, duplicated, or delivered out of sequence and it is left to the applications involved to handle any errors that this causes. Packets may arrive at the server faster than they can be processed causing packets to be lost. No message is sent to the sender regarding the packet's fate.

Connectionless delivery, on the other hand, is a more positive feature. Essentially each packet is treated independently from all other packets. This scheme works well for servers that operate in a transaction oriented manner, where each message sent represents a complete operation that is independent from any prior or subsequent messages. This implies a *stateless* server where each request can be immediately processed and no record need be kept of the outcome of the request. The `rwho` daemon is a simple example of such a server.

For a server like `rwho` the handshake between client and server is almost nonexistent. The client fires off its packet blindly and then waits for a reply. While waiting for a response from the server the client runs a timer. If the targeted server is not functioning, or the packet gets lost somewhere in the network, the client has no way of knowing what happens to the packet. If no reply is received before the timer expires the client may tell the user that the "server is not responding" or something equally un-informative. The client may attempt a retry, sending the packet again some predetermined number of times, but how the client handles the situation is its own responsibility. The only way a client knows its request was received is by getting a packet in reply from the server.

Such a server is quite simple to implement. The server listens to a port waiting for incoming packets. Once a packet is received, it is decoded and the proper response is sent back to the requesting client. Packets that don't arrive are not a problem; the server never knows about them. If the volume of arriving packets is greater than can be processed the excess packets are simply lost. The server processes what it can and that's that. If duplicate packets arrive they are handled as any other request would be. If the client doesn't handle the duplicated replies they can just be discarded.

This rather simple scenario rapidly becomes more complicated if multi-packet requests are required. If a request consists of more than one packet all packets in the requesting sequence must be received and placed in the proper order before the request can be processed. The reliable delivery of packets implies some form of acknowledgment by the server as each packet is received, possibly with some form of timeout on the interval between packets so the server can request that the client retransmit a lost packet. The server must now keep the state of the client's request until the request is completed or the server detects that the client has canceled the request. Having to wait for an acknowledgment before transmitting the next packet is very inefficient so some form of *sliding window* scheme [5] is required. Multi-packet requests greatly complicate server design.

As has been seen, unreliability is an inherent property of `UDP` based server software. Under the appropriate circumstances this is not a major detraction and the protocol's other advantages can make using `UDP` attractive. The simplicity and low overhead of a `UDP` implementation is quickly lost if reliable delivery of multi-packet sequences is required.

3.2 Transmission Control Protocol

The other major IP protocol implements reliable stream delivery and is known as Transmission Control Protocol (`TCP`). `TCP` guarantees reliable, non-duplicated, properly sequenced data reception between two communicating processes. It is a *stream oriented* protocol that passes data between client and server as a sequence of bytes (note that this implies transparent buffering of data). When two processes have a significant amount of data to exchange `TCP` is the protocol of choice.

Processes using `TCP` communicate over a *virtual circuit*, a temporary connection similar in concept to a telephone call. Before communication can begin the client process must *call* the server and establish a connection. Once the connection is established, the client and server processes communicate by exchanging streams of bytes. The protocol software ensures error free transmission of the data. If, for some reason, the circuit is broken both processes will detect the break.

Aside from reliable delivery of data the other main feature of `TCP` is adaptive transmission rate control. `TCP` implements *sliding window* control for data packet transmission. This scheme alleviates some of the efficiency problems associated with reliable delivery by trying to saturate the bandwidth of the transmission channel. This is done by allowing the transmission of multiple packets before requiring an acknowledgment of receipt. `TCP` measures the time it takes for each packet to be received and acknowledged, and uses this information to adjust its transmission rate. In this way network congestion is automatically compensated for without any action on the part of the application processes. This stands in contrast to the normal `UDP` timeout and retransmit strategy that responds to network congestion with lost packets and multiple retransmissions.

To summarize, TCP provides reliable stream delivery of data via temporary virtual circuit connections. Once a connection is established the communicating processes are guaranteed delivery of all transmitted data or notification of a break in the connection. Actual packet transmission rates are transparently adjusted to compensate for channel bandwidth and network congestion. Because the system software that implements the protocol provides all of these services, writing application software is greatly simplified.

3.3 The BioSCAN Low Level Protocol

As has been seen, both the UDP and the TCP protocols have advantages and disadvantages. Different types of applications are better suited to one protocol or the other. UDP, with its bare minimum of overhead, is well suited to transaction oriented client-server software where each request and reply are contained in a single packet and the size of the packets is relatively small. If the application in question requires sequences of packets to specify a single request things rapidly become more complex, with the burden falling on the application software itself.

TCP, on the other hand, implements a host of reliability features that are independent from the application software. There is a price to be paid in terms of administrative overhead since all the features mentioned above are implemented regardless of whether they are needed or not. If an application is limited to small, succinct messages, UDP may be a better choice. If large amounts of data must be transmitted to form a single processing request, TCP is the better choice.

BioSCAN is not a pure transaction based system. The amount of data that needs to be transmitted to define a user job can vary greatly and can grow to be several kilobytes.

The output data can be even more voluminous. BioSCAN will be used in a interactive manner, the same basic job parameters submitted for processing multiple times with only small modifications. To minimize the amount of data passing over the network the server needs to maintain the state of each client session. Though it is possible to implement this type of server with either protocol it is most easily implemented using `TCP`.

The BioSCAN server requires the implementation of most, if not all, of the features already found in `TCP`. That a custom protocol implementation would outperform existing `TCP` implementations in any significant way is not assured and the complexity of the task would certainly result in a long period of debugging time. Given the nature of BioSCAN's data requirements, we decided that `TCP` is the proper protocol for the implementation of the BioSCAN server.

We still had a desire not to maintain a large number of concurrent network connections as there are a limited number of connections available on a given system. There is also some small overhead associated with maintaining each connection, overhead that is a burden on all machines in the transmission path from the client to the server. This adds unnecessarily to network traffic and adversely affects all systems on the network.

These considerations lead to a server that limits each client session to short transactions but uses `TCP` as its underlying communications protocol. By restricting the duration of a session to a brief exchange between client and server the following advantages are gained.

- Network congestion is reduced by minimizing the time each client remains connected to the server.
- The number of processes needed to support a given number of users is fewer than the number of active clients.

- Data transmission reliability is handled by the underlying `TCP` system software, greatly simplifying the server code.

Of these advantages, the last, reduction in the amount of code needed, was the most important. The way in which the client and server exchange information during one of the sessions is the subject of the next section.

4 High Level Protocol Issues

Client application programs communicate with BioSCAN server processes though `TCP` socket streams. The messages that pass through these sockets, their form and meaning, define the high level BioSCAN protocol. The low level protocol, `TCP`, provides the method of connection and ensures reliable delivery of messages between the client and the server. The major goals for the high level protocol are that it be easy to implement and easy to extend. We made two major design choices regarding the high level protocol: messages would not be transmitted in a binary format and client-server sessions would be as brief as possible.

As an aid to debugging we decided to encode, in `ASCII`, the messages passed between the client and the server. All the high level protocol messages are transmitted as human readable character strings. Though this encoding required the writing of a lexical analyzer to decipher the messages, the analyzer needed to be written only once. The same analyzer is used in both the client and the server software and, because of its general nature, can be used to decode message formats that are added in the future.

The second design choice, keeping client-server sessions as brief as possible by using simple transactions, was complicated by the nature of the data that needed to be communicated.

To understand why all the protocol sessions could not be simple transactions we will look at the types of messages that are actually needed.

4.1 Types of Client Requests

It has already been stated that short client-server sessions are desirable. To see whether such a short session implementation is possible we must examine the operations that the client can ask the server to perform. The list of functions appears below.

- Request server configuration information.
- Submit a job for processing.
- Request the status of outstanding jobs.
- Cancel a previously submitted job.

Three of these four basic functions—request information, request status, and cancel a job—are well suited to short client-server sessions. In each case the client establishes a connection with the server and, after the exchange of identifying messages, submits the request. If there are no errors in the syntax of the request, the server responds by sending the requested information or, in the case of a cancellation request, verification that the request was processed. The session is then terminated.

4.2 A Not So Brief Transaction

As mentioned previously, transactions need to be brief. This implies the transmission of a relatively small volume of data during a client- server session. The client requests discussed so far are accomplished with the exchange of a few hundred characters. The volume of data required to define a job is quite a bit larger. These data are identified below.

- The sequence to be matched (the query sequence).
- A similarity table.
- A threshold value.
- The data to be searched.

The sequence to be matched may contain as many characters as there are `PE`s in the BioSCAN hardware configuration. This implies a limit of 12992 characters. The similarity table consists of 784 integer values and the threshold value is a single integer. A single job request may contain any number of sequence, similarity table, and threshold value tuples.

To select the data to be searched the client submits a list of names that identify databases maintained by the server. Included in the data returned by the configuration request is a list of databases. In all, the size of the list is no more than a couple of hundred characters but the list does vary in size.

One of the unstated goals of the server design was to isolate the client from the details of the server's hardware configuration. This is desirable since future BioSCAN hardware may have significantly different characteristics from the current sixteen-`ASIC` board. If the underlying hardware were visible to the client a maximum of sixteen tuples could be defined, one per chip. An idealized BioSCAN machine would place no limit on the number of tuples in a job request.

The upshot of all this is that a job submission request can be big, tens of thousands of characters in length. Though the server responds only with a verification of submission or an error it is hard to think of a job submission request as a transaction. Also note that the desired result of submitting a job is to receive information on possible matches with strings in the database. In this sense the job request is not a transaction either. Transactions are self contained and terminate without the expectation of a future response. A job request implies the future receipt of result data.

Despite these contradictions it is still useful to think of the BioSCAN server as transaction oriented. The effect of processing client requests in a transaction like manner reduces network connections, lessens the number of active server processes, and segments the server code into small functional units.

5 Process Interaction Issues

The way in which client processes interact with the BioSCAN server software is determined primarily by the server protocol. To some extent the structure of the server is itself determined by the protocol specification. The choice of the `TCP` transmission protocol dictates the method in which connections are listened for and how an incoming connection request is handled. Beyond the socket interface that the client process sees there is another layer of interaction between processes.

There is nothing that would have prevented implementing the entire BioSCAN server as a single, monolithic process, simultaneously maintaining multiple client sessions while at the same time scheduling processes and returning results. We chose not to follow this route for a number of reasons. These reasons will be revealed shortly.

5.1 The Traditional Server

There are two types of traditional network server programs: those that are transaction based and those that are session based. The BioSCAN server has features of both types of server so it is relevant to discuss the two traditional types before examining the marriage of the two.

Transaction based servers typically respond to simple, one packet queries by returning some resulting data to the requester. This type of server is referred to as connectionless because no persistent bidirectional data path is established between the client and the server. The client simply fires off a request to the server and hopes to receive a reply some time in the future. There is no acknowledgment of the request's receipt by the server other than the data packets sent in response. This type of server is most often implemented using `UDP`.

The important characteristic of this type of server, from the process interaction point of view, is that a single process can handle all requests. The server simply handles requests in the order in which they arrive, letting the operating system handle the queue of incoming requests. If requests arrive faster than they can be responded to requests are simply lost. The `rwho` server is typical of this type of server.

If the transaction based server takes the minimalist approach to client-server interaction, the session based server takes the opposite approach. Upon receiving a connection request this type of server establishes a reliable, bidirectional data path to the caller. This connection is maintained until processing of the client's requests is complete. It is usually the case that each connection request spawns a new process on the server machine. `Telnet` is typical of this type of server.

5.2 The Structure of the Server Processes

The software that supports the BioSCAN hardware in its interactive mode of operation is divided into two major parts, the client application and the server. The client application is linked to the Application Subroutine Library (`ASL`) which communicates to the server via the BioSCAN Application Program Interface (`API`). The server itself consists of several distinct software processes that provide for job creation, job scheduling, data formatting, hardware interface, and the return of the resultant data to the client application.

The BioSCAN server provides network users with a way of interfacing to the BioSCAN hardware in a shared environment. In its quiescent state the server consists of two UNIX processes; the BioSCAN server daemon, *bioscand*, and the BioSCAN job scheduler daemon, *bioschedd*. The server daemon provides the connection to the client application through an Internet domain socket connection and interacts with the client application to create a job request.

A job request contains all the information needed to execute a database sequence search: test sequence(s), similarity table(s), threshold value, and database name(s). Once a job request is complete it is sent to the scheduler daemon for queueing and execution by the BioSCAN hardware. The scheduler translates the job request into a form, called a *board run*, that reflects the actual hardware configuration in use.

Information contained in the board run is then used to load the registers in the BioSCAN hardware and perform the actual sequence scan. The following is a description of the major components (processes) of the BioSCAN server.

5.2.1 Bioscand: the Server Process

The server process, *bioscand*, is structurally a traditional `TCP` type server. The first server process started binds the public `IP` port and awaits incoming service requests. On the arrival of a request a new process is *forked* and the parent process continues waiting for new requests.

The child server process opens a communication socket to the client and the exchange of messages begins. The client process identifies itself and states which type of transaction it would like to conduct. If an illegal request is made or an error is detected in the format of the data sent by the client an error message is sent to the client. The connection between the client and the child server process is broken, and the child server process then terminates.

If the client's request is valid the child server process takes the appropriate action. This requires the child to interact with the scheduler process, *bioschedd*, to queue a job request, abort a job request, or ascertain the status of a previously queued job. Only a request for the list of available databases is handled entirely by the child process.

After successful completion of the requested action the child server returns any requested data and verifies that the transaction is complete. The socket connection is closed and the child server process terminates.

5.2.2 Bioschedd: the Scheduler Process

The scheduler process, *bioschedd*, is the focal point of all data flow in the BioSCAN server software. The scheduler's function is to queue the client job requests and to run them on the BioSCAN hardware. All requests for processing are passed by the server processes to the scheduler. The scheduler must take into account individual job priority and the possibility of combining jobs for simultaneous execution.

Once a job is ready to be dispatched the scheduler *forks* a child process that becomes a *run process*. The run process actually communicates with the BioSCAN hardware, loading the processor board registers and passing the database characters to the board for processing. The run process calls back to the requesting process before starting the hardware processing. If the requester doesn't respond to the call back the run is aborted and the scheduler notified that the associated processor board is free (the scheduler is informed by getting a `SIGCLD` signal generated by the run process's termination).

Once in contact with the requesting process, and having prepared the BioSCAN hardware, the run process passes the requested database files to the hardware. As results are sent back from the hardware the run process formats them and sends them back to the requesting process. When the job ends there is a final handshake with the requester, the socket connection is closed, and the run process terminates.

5.2.3 Biomond: the Monitor Process

The monitor process, *biomond*, is a recent addition to the server design. It provides a way for authorized users to monitor the scheduler's queue and processing statistics while the server is running. Its functions are not visible to the user and are not important for understanding the server software design. I am mentioning this process for the sake of completeness. To clarify the relationships among the various types of constituent processes I will now describe the sequence of events that occur during server initialization.

5.3 Process Initialization

The BioSCAN server is started by executing the program that will become the scheduler daemon, *bioschedd*. This program first reads the specified configuration files and checks them for validity. Then, several simple diagnostics are run to verify that the specified hardware is present and functional. At this point *bioschedd* launches the *bioscand* process. This process becomes the listener for incoming user connection requests and is referred to as the server daemon. During initialization, the scheduler daemon provides the server daemon with the address of a socket. This socket will provide the path for job requests to travel from the server daemon to the scheduler daemon. Both the server daemon and the scheduler daemon must be running for the server to be operational.

5.4 Servicing User Requests

When the server is fully active many processes will exist: server daemon processes, board run processes, and the scheduler daemon process. Each client that is connected to the server communicates through its own copy of `bioscand`. These are child processes of the initial `bioscand` that are launched as incoming connection requests are received from user applications.

The server is called by a client application using the `API` which, in turn, is built on the `TCP/IP` suite of networking protocols. The server daemon creates and then tries to read from an Internet domain (`AF_INET`) stream type socket (`SOCK_STREAM`). Using a stream type socket instead of a datagram type socket (`SOCK_DGRAM`) simplifies the listen loop. Stream style socket communication guarantees message delivery and handles any transmission errors for the server.

On receiving a request for connection from a client application the server forks. Part of the `accept()` call creates a new socket connection to the calling process. The child process uses this new socket connection to communicate with the client process after closing its copy of the original socket. The parent `bioscand` closes its copy of the new socket and continues to listen for incoming connection requests on the original socket. This connection mechanism implies that, if there are n connected users, there will be $n + 1$ `bioscand` processes active; the listener and one child for each connection.

The connected server daemon handles the `API` calls passed to it via the socket connection. The `API` calls echo the `ASL` calls made by the client application program. Data regarding the server's configuration can be requested, job parameters passed, the status of previously submitted jobs checked, and outstanding job requests canceled. The goal of a client process is to submit a valid job request. Once the job request is completed the user application requests that the job be run. This causes the server daemon to pass the request to the scheduler daemon for execution on the BioSCAN hardware.

5.5 Running a Job

Once a job request reaches the scheduler daemon the request is placed in the job queue. At this point there are several things that the scheduler might do to the client's job request in an effort to improve system performance. These include merging jobs that can run in parallel, splitting large jobs to run on multiple BioSCAN hardware boards, and ordering jobs by priority. Though the first implementation of the BioSCAN hardware is a single board device, the scheduler is capable of controlling multiple independent devices. This is because each actual run execution is performed by an independent process called a *run* process.

When a job comes to the head of the queue, and the needed hardware resource becomes available, the job request is formatted for running on the BioSCAN hardware. From the configuration information, read during initialization, the job request is transformed from a format targeted at an idealized BioSCAN machine to a run request that is targeted at the actual hardware available. Once the run request has been formatted the scheduler daemon forks, creating a child *run* process to execute the run request on the BioSCAN hardware. The scheduler returns to the task of listening for job requests and to performing optimizations on the job requests already queued. The child process meanwhile attempts to establish a `TCP` socket connection to the client that submitted the job request. If the client does not respond to the call back the job request is discarded, the scheduler notified that the hardware is no longer in use, and the child process terminates.

If the client responds, the *run* process is ready to proceed with processing the run. The *run* process has possession of the formatted run request and the hardware on which to run it. The board is initialized, the appropriate database objects accessed and the run commences.

During the run, result data are received from the BioSCAN hardware asynchronously. As these data become available they are passed to the user application via the socket connection. At the end of the run the run process notifies both the client and the scheduler and terminates.

6 Summary

6.1 Design Decisions

During the design of the BioSCAN server software a number of often contradictory design requirements were combined with a tight schedule and limited resources to produce a fully functional network server. I will review major design choices and their consequences.

The choice of `TCP` as the underlying, low level communication protocol allowed rapid implementation of the server software by simplifying the coding requirements. Development time could be spent on problems unique to the server software, not implementing yet another low level communication protocol. `TCP` also has the advantages of widespread acceptance, almost universal availability, and excellent reliability. Most `TCP` system software is stable and highly optimized.

The decisions to use short, transaction like sessions and to pass messages between the client and server in `ASCII` also simplified the writing of the server software. Each of the four transaction types could be implemented independently, allowing modular code. Debugging was simplified because all messages exchanged between the client and the server were in human readable form (when displayed on a terminal). The generalized lexical analyzer that was developed allows easy extension of the high level protocol. New statements can be implemented using simple `C` function calls.

The use of multiple child processes to establish one on one sessions with client programs allows the isolation of the scheduling and processing portions of the server. This provides greater system integrity by precluding any direct communication between client processes and the scheduler.

The single scheduler process enables the use of run queue optimization to maximize system performance. Job splitting and job combining are possible ways of improving system performance. The single scheduler also makes monitoring performance and controlling job submission easy.

Finally, the submission and call back scheme employed by the server provides the impression of an interactive system without the attendant network and system overhead. Client to server connections are of short duration and network resources are not required while a job is waiting in the queue.

6.2 Some Performance Data

In tests run directly on the BioSCAN hardware, data rates as high as 1.95 million characters per second have been observed. This compares favorably with the hardware's theoretical maximum rate of 2.0 million characters per second. Some degradation in data throughput is to be expected when the server software is added as a layer between the user application and the actual hardware but preliminary measurements indicate that the slowdown experienced is minimal. Data rates as high as 1.934 million characters per second have been observed.

6.3 Future Work

Future work on the BioSCAN server software falls into two categories: enhancements to the basic server functions and improvements in overall system performance. Enhancements to basic functions include adding remote monitor and control functions, improving system event logging, caching of standard similarity tables, and enhancing the job priority mechanism. These features will extend the capabilities of the existing server by making the server more flexible and easier to administer.

The inclusion of more advanced scheduling algorithms in the server awaits the availability of actual usage data. The server's current workload is not heavy enough for throughput improvements to be observed. Future efforts will utilize theoretical models and synthetic work loads to allow performance predictions.

References

[1] S. F. Altschul. Amino acid substitution matrices from an information theoretic perspective. *Journal of Molecular Biology*, 219:555–565, 1991.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[3] R. S. Boyer and J. S. Moore. A fast string search algorithm. *Communications of the ACM*, 20:762–772, 1977.

[4] C. Burks. How much sequence data the data banks will be processing in the near future. In R. R. Colwell, editor, *Biomolecular Data*, pages 17–26. New York: Oxford University Press, 1989.

[5] Douglas E. Comer. *Internetworking with TCP/IP, Volume 1.* Englewood Cliffs, N.J. : Prentice-Hall, 1991.

[6] W. D. Dettloff, R. K. Singh, C. T. White, and B. W. Erickson. A 50 mhz 1.5m transistor asic for biosequence analysis. *International Solid State Circuit Conference Digest of Technical Papers*, pages 40–42, Feb. 1991.

[7] M. Gokhale et al. Splash: A reconfigurable linear logic array. *Proceedings of the International Conference on Parallel Processing*, page 526, 1990.

[8] M. Gribskov and J. Devereux, editors. *Sequence Analysis Primer.* New York: Stockton Press, 1991.

[9] R. P. Hughey and D. P. Lopresti. B-sys: A 470-processor programmable systolic array. *Proceedings of the International Conference on Parallel Processing*, pages 580–586, 1991.

[10] S. Karlin and S. F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Sciences*, 87:2264–2268, 1990.

[11] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. Technical Report CS-74-440, Department of Computer Science, Stanford University, Stanford, Calif., 1974.

[12] J. B. Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review*, 25(2):201–237, 1983.

[13] D.P. Lopresti. P-nac: A systolic array for comparing nucleic acid sequences. *Computer*, 20(7):98–99, July 1987.

[14] A. Mukherjee. Hardware algorithms for determining similarity between two strings. *IEEE Transactions on Computers*, 38:600–603, 1989.

[15] National Institutes of Health. The U.S. human genome project: The first five years fy 1991-1995. Technical Report NIH 90-1580, National Center for Human Genome Research, 1990.

[16] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85:2444–2448, 1988.

[17] C. T. White et al. Bioscan: A vlsi-based system for biosequence analysis. *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, pages 504–509, 1991.