Client-Server Design Alternatives: Back to Pipes but with Threads

Boris Roussev[1] and Jie Wu[2]

[1] Information Systems Department, Susquehanna University,514 University Avenue, Selinsgrove, PA 17870, USA
roussev@roo.susqu.edu
[2] Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, Fl 33431, USA
jie@cse.fau.edu

Abstract. In this paper we set out to theoretically explore and experimentally compare different client-server design alternatives implemented in Java. We introduce a new concurrent data structure, called concurrent hash table, for solving the synchronization problem in the classical producer/consumer model. The structure allows multiple reads and a single write to proceed concurrently. We look at the following TCP server designs: concurrent server–new thread per client; pre-threaded servers: locking around accept; socket passing through a shared buffer; socket passing through a concurrent queue; socket passing through a concurrent hash table; socket passing through pipes. The servers have been tested on a network of 35 workstations. The experimental results have shown that the server using pipes to pass tasks to the workers outperforms every other one. For all servers, better performance is achieved by using a number of worker threads in the range of one hundred rather than fifteen as commonly recommended.

1 Introduction

The third industrial revolution is all about what George Herbert Wells envisaged at the dawn of last century as "The World Brain." Today, we see the prophecy come true. The neurons of this brain constitute the interconnection of networked computers called the Internet where the processing elements interact with each other using the client-server pattern. In this paper we set out to explore and experimentally compare different client-server design alternatives implemented in Java. The application programming interface used is "Berkeley sockets" [9]. Threads, an abstract compute model, are used to structure the concurrent activities in the server programs.

The performance and correctness of the server programs is of crucial importance to the success of many network applications and distributed systems. With the explosion of the WWW, busy Web servers measure the number of connections per hour in the hundreds of thousands. Furthermore, many of these servers interact with backend database servers, which in turn have to process an even greater

workload. In other words, if we are to build robust Internet applications materializing Wells's metaphor, more than ever before the network applications should be based on efficient client-server architectures. Today, many new technologies, like the Java Virtual Machine (JVM), multiprocessor and multithreaded kernels have matured. In JVM the time required to spawn a new thread or to obtain an object's lock in most implementations is negligible. The use of multiprocessor computing machines is a norm. Mapping of threads to processors has been optimized tremendously. As a result, designs that have been impossible until recently become viable propositions.

Java [4], an object-oriented language, has become popular because of its platform independence and safety. It has greatly simplified network programming [5] by providing elegant TCP/IP API, object serialization, network class loading (code mobility), remote method invocation, Servlets and built-in concurrent constructs. This along with its phenomenally growing popularity entails a rapidly expanding body of projects: Atlas, Charlotte, Javelin, JPVM, Globus, IceT, JavaSpaces, MPIJ, Bayanihan to mention but a few, that use Java as a language for high performance computing on networks of workstations [11]. The Syracuse workshop [3] discussed Java's possible role as the basic programming language for science and engineering–taking the role now played by Fortran 90 and C++–and concluded that Java could become dominant by adding the necessary functionality to the basic Web loosely coupled distributed model.

In this paper we compare the following TCP server designs: (1) iterative server; (2) concurrent server–new thread per client; (3) pre-threaded server with locking (mutex) around accept; (4) pre-threaded server–connected socket passing through a shared buffer; (5) pre-threaded server–connected socket passing through a concurrent queue [8]; (6) pre-threaded server–connected socket passing through a concurrent hash table; and (7) pre-threaded server–connected socket passing through pipes.

For all the server architectures, we evaluate experimentally parameters taken for granted for quite a long time, for example, the number of worker threads a server should spawn, the buffer capacity of the shared buffer and techniques for synchronizing the work of the thread accepting the connections and the threads carrying out the service requests. To evaluate the servers, we run the same client on 35 hosts on the same subnet on which the server being evaluated is running. Each client spawns between 4 and 20 child clients to create multiple simultaneous connections to the server, for a maximum of 700 simultaneous connections. We consider the effect of having too many/few threads.

The remainder of the paper is structured as follows. In Section 2, we review the TCP/IP protocol stack and Java concurrent and multithreading constructs. In Section 3, we describe the architecture and the implementation of the server designs. Next, in Section 4 we give theoretical analysis of the performance of the concurrent hash table and synchronization using pipes. Then, in Section 5 we present experimental results. In the final section we outline plans for future work and conclude.

2 TCP/IP and Concurrent Java Programming

Most network applications are structured in two pieces: a client and a server. Clients normally communicate with one server at a time. On the other hand, servers commonly handle multiple client requests at any given point in time. Network protocols are involved in the client-server communication. In this paper we focus on the TCP/IP protocol suite. The TCP [9] protocol is a connection-oriented protocol that provides a reliable, full-duplex, byte stream for a user process or thread. TCP takes care of details such as acknowledgements, timeouts, and retransmissions. Most Internet application programs use TCP. TCP can use either IPv4 [10] or IPv6 [2].

Java is a shared memory thread-based language with built-in monitors [6] and binary semaphores as a means of synchronization at the object and class level [7]. The Java class libraries relevant to network programming, thread communication and synchronization are `java.net`, `java.io`, and `java.lang`.

```
class TCPClient {
   //instance variables
   public void execute () {
      for( int i = 0 ; i < nchildren - 1 ; i++ )
         {// Spawn a new client processe }
      for ( int i = 0 ; i < nrequests ; i++ ) {
         connect = new Socket( hostIP, port );
         // send request and get response
         connect.close() ;
}  }  }
```

Fig. 1. Pseudo code for the TCP client program testing the different server designs

3 Client-Server Design Alternatives

Below we describe each of the server designs we use. We examine how the main server thread and the worker threads can be synchronized. The client used to test all the servers is shown in Figure 1. When we execute the client,

```
java TCPClient IP port nchildren nrequest [millisec | filename]
```

we specify the host's IP address, the server's port, the number of children for the client to spawn, the number of requests each child should send to the server, and the number of milliseconds the server should delay its response to simulate the processing of a client request. The client closes the connection after receiving the server's response, so TCP's TIME_WAIT state occurs on the client, and not on the server.

All servers subclass the abstract class `TCPServer` shown in Figure 2. They implement its abstract method `handleRequest()`, processing a single client request. Each server extends the class constructor to kick off the required worker threads, monitors, locks, barriers and/or buffers used for synchronization and communication.

```
publicabstractclass TCPServer implements Runnable {
   public TCPServer( int port, int nreq, int nbytes, long time )
      { ...}
   public void run () {
      while ( true ) {
        try {
           connect = listenSocket.accept () ;
           handleRequest( connect ); processedRequests++;
           if ( processedRequests == nrequests )
              {listenSocket.close () ; break ;}
        } catch ( IOException excp ) { }
      } return ;
   }
   protected abstract void handleRequest( Socket connect ) ;
}
```

Fig. 2. Generic TCP server. Servers implements the abstract method `handleRequest()`

3.1 TCP Iterative and Concurrent Server Designs

Our first server is an iterative single-threaded server. It is used as a benchmark to measure the speed up of the concurrent servers. The server cannot process a pending client until it has served the current one.

Our next design is a classical concurrent server. It spawns a thread to handle each new client request. When a new connection is established, `accept()` method of `ServerSocket` returns, the server calls the worker thread constructor and then the spawned thread serves the client. The parent server thread goes back to wait for another connection.

3.2 Pre-threaded Server, Locking around `accept`

In this server design we use a technique called pre-threading. The server pre-spawns a pool of worker threads, see Figure 3. The worker threads are ready to serve the clients as each client connection arrives. As shown in Figure 4, all worker compete to obtain the `ServerSocket` lock passed to each of them when they were created. As a result, only one worker thread, the winner, is blocked in the call to `accept()`. The remaining workers are blocked trying to obtain

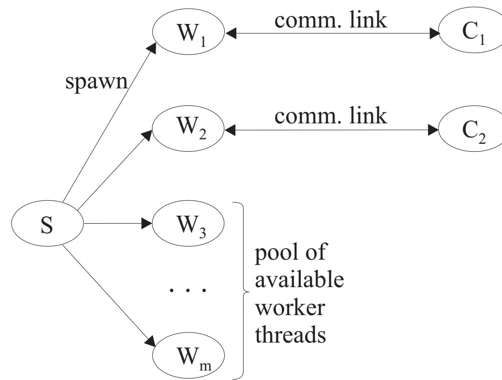

Fig. 3. The main thread S spawns worker threads, W_i, to serve the clients C_j

the server socket's lock. We use Java's block synchronization to implement this server design.

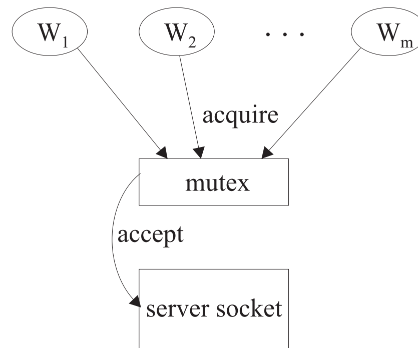

Fig. 4. TCP pre-threaded server, locking (mutex) around `accept`

Theoretically, the advantage of this technique is that a new client request can be handled without the cost of spawning a new thread. An important parameter of the design is the number of worker threads the server should spawn when it starts. 15 threads are recommended in [12]. In contrast, our experiments show that better results are obtained with 70 to 80 threads. If the number of clients at any time equals the number of worker threads, additional clients will notice degradation in response time. The kernel will still complete the TCP three-way handshake for the additional clients, up to the listen backlog specified when the `ServerSocket` is instantiated. Stevens [12] suggests that the main thread monitor the number of available workers, and if this value drops below or exceeds some thresholds, it must spawn or terminate some workers, respectively, to avoid

performance degradation. We found out experimentally that the main server thread can handle up to 200 worker threads without any performance penalty and there is no need to implement a sophisticated monitoring.

3.3 Pre-threaded Server, Connected Socket Passing through a Shared Buffer

Next, we modify the pre-threaded server to have only the parent server thread call `accept()` and then pass the connected socket to one of the worker threads through a shared buffer. The buffer is implemented as a fixed array of references to sockets along with two indices that circularly traverse the array, keeping track of the next position to `put` and `take` respectively. This is a classical producer/consumer solution where a monitor is used to synchronize the access to the shared buffer.

3.4 Pre-threaded Server, Connected Socket Passing through Concurrent Queue

In this server design the shared buffer described in Section 3.3 is implemented as a concurrent queue [8]. The queue uses the lock-splitting technique [7] to minimize access contention. A put lock ensures that only one `put` operation at a time can proceed. A take lock similarly ensures that only one `take` operation at a time can proceed. A `put` and a `take` can normally proceed independently, except when the queue is empty. The main server thread puts the connected socket on the queue. The worker threads take the connected sockets from the queue.

3.5 Pre-threaded Server, Connected Socket Passing through Concurrent Hash Table

Next, we introduce a new server design technique, where the shared buffer is implemented as a concurrent hash table, see Figure 5. A dynamic set that supports the operations `insert`, `search`, and `delete` is called a dictionary. A hash table is an efficient data structure for implementing dictionaries. Under reasonable assumptions, the expected time to search for an element in a hash table is $O(1)$. With hashing, an element x with key k is stored in slot $h(k)$, where h is a hash function used to compute the slot from the key k. When two keys hash to the same slot a collision occurs. Collisions in the concurrent hash tables are resolved using open addressing [1]. In open addressing, each table entry contains either an element or `null`. When searching for an element, we systematically examine table slots until an element is found. To perform insertion, we probe the hash table until we find an empty slot in which to put the element. To determine which slots to probe we extend the hash function to include the probe number, starting from zero, as a second input. The code is given in the Appendix. Each element in the concurrent hash table is of type `SynchronizedRef`. This class maintains a

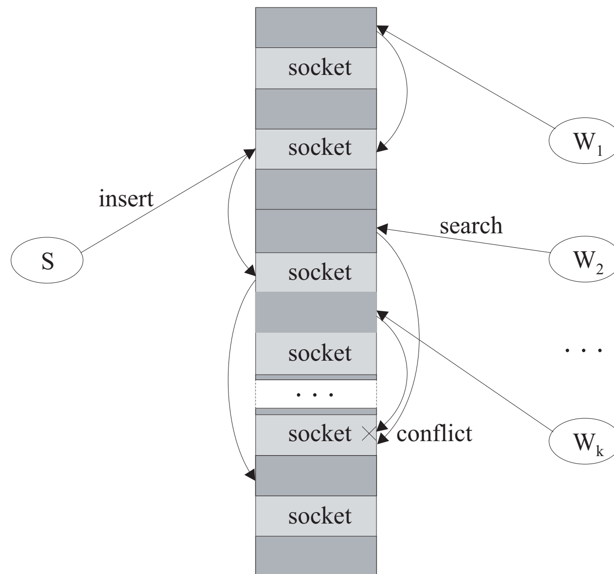

Fig. 5. Concurrent hash table. The main server thread inserts connected sockets in the table. The prethreaded worker threads retrieve these sockets

single reference variable that is always accessed and updated under synchronization. Using `SynchronizedRef` objects as elements in the hash table, allows us to loosen the synchronization among the worker threads, and between the main server thread and the worker threads using an optimistic control strategy. Each thread can access any slot without possessing a synchronization lock as a precondition. We have two different cases to consider corresponding to the methods `insert` and `search`: (i) The main server thread calls `accept()` and then inserts the returned connected socket in the table using the `insert` method of the concurrent hash table object. The `insert` method uses the `hashCode()` method of the `Object` class to compute the key of the connected socket and double hashing [1] to compute the slot in the table. (ii) Worker threads retrieve connected sockets from the table using the `search` method of the concurrent hash table object. They pass as an argument to the search method a pseudo randomly generated number. This number is used to calculate the first slot of the table to be checked for a connected socket. If the slot is not `null`, the connected socket is retrieved and the slot is set to `null`. Otherwise, the next slot in the probe sequence is calculated and checked. If a collision occurs, i.e. two or more workers access the same slot, because the table slots are objects of type `SynchronizedRef` only one of the workers will retrieve the connected socket and set the slot to `null`. The rest will see the `null` value and try to retrieve a connected socket from other slots.

3.6 Pre-threaded Server, Connected Socket Passing Through Pipes

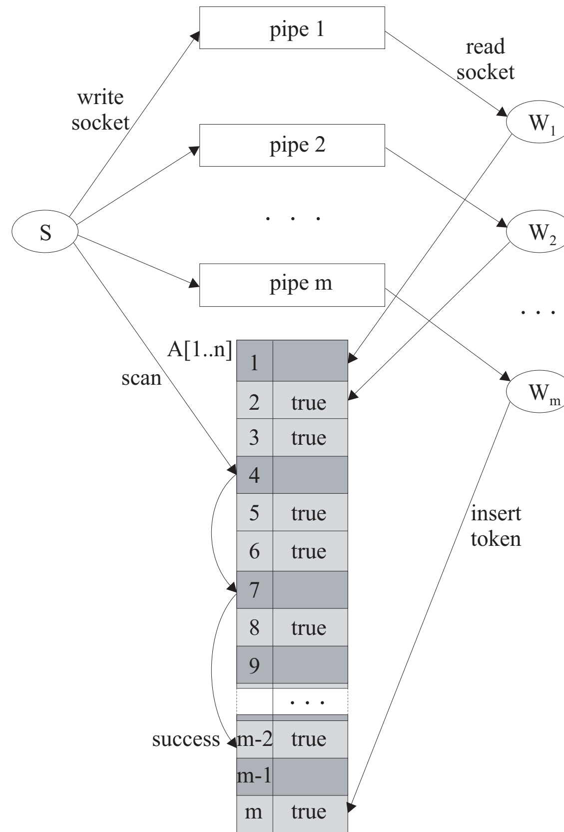

Fig. 6. TCP pre-threaded server, connected socket passing through pipes

The final modification, shown in Figure 6, of the pre-threaded server gets around the need for using a common buffer and synchronization between the main thread and the worker threads. The main server thread calls `accept()`. It keeps track of the worker threads being free to pass a new connected socket to a free worker through a pipe. We had to write our own pipe class since class `Socket` does not implement the `Serializable` interface and therefore objects of this class cannot be passed through Java communication pipes. When a new request arrives, the main server thread finds the first available free worker by scanning the array of `WorkerStatus` elements denoted as $A[1..n]$ in Figure 6 and passes the socket to that worker through its own pipe. Being finished with a client, the worker thread changes the status of its pipe back to ready state by writing `true` in its `WorkerStatus` element.

4 Theoretical Analysis

4.1 Analysis of Concurrent Hashing

Given a concurrent hash table T with m slots that stores n elements, we define the load factor α for T as n/m, $\alpha \leq 1$. We make the assumption of *uniform hashing*: each key considered is equally likely to have any of the $m!$ permutations of $\{0, 1, \ldots, m-1\}$ as its probe sequence. In our implementation we use double hashing which is a suitable approximation to uniform hashing. Double hashing uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$$

where h_1 and h_2 are auxiliary hash functions. The initial position probed is $T[h_1(k)]$; successive probe positions are offset from previous positions by the amount $h_2(k)$ modulo m. The value of $h_2(k)$ must be relatively prime to the concurrent hash table size m for the entire concurrent hash table to be searched. Otherwise, if m and $h_2(k)$ have greatest common divisor $d > 1$ for some key k, then a search for key k would examine only $1/d$th of the table.

Theorem 1. *Inserting an element into a concurrent hash table with load factor $\alpha < 1$ requires at most $1/(1-\alpha)$ probes on average, assuming uniform hashing.*

Proof Inserting an element requires an unsuccessful search followed by the placement of the element in the first empty slot found. In an unsuccessful search, every probe but the last accesses an occupied slot, and the last slot probed is empty. Let $p_i = \Pr\{\text{exactly } i \text{ probes access occupied slots}\}$ for $i = 0, 1, 2, \ldots$. For $i > n$, we have $p_i = 0$, since we can find at most n slots already occupied. Thus the expected number of probes is

$$1 + \sum_{i=0}^{\infty} ip_i \tag{1}$$

To evaluate (1) we define $q_i = \Pr\{\text{at least } i \text{ probes access occupied slots}\}$ for $i = 0, 1, 2, \ldots$. Since i takes on values from the natural numbers

$$\sum_{i=0}^{\infty} ip_i = \sum_{i=1}^{\infty} q_i$$

The probability that the first probe accesses an occupied slot is n/m. Thus

$$q_1 = \frac{n}{m}.$$

A second probe, if necessary, is to one of the remaining $m-1$ unprobed slots, $n-1$ of which are occupied, thus,

$$q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right).$$

The ith probe is made only if the first $i-1$ probes access occupied slots. Thus,

$$q_i = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-(i-1)}{m-(i-1)}\right) \leq \left(\frac{n}{m}\right)^i = \alpha^i,$$

since $(m-n-j)/(m-j) \leq (m-n)/m$ when $m-n \leq m$ and $j \geq 0$. Now, we can evaluate (1).

$$1 + \sum_{i=0}^{\infty} i p_i = 1 + \sum_{i=1}^{\infty} q_i \leq 1 + \alpha + \alpha^2 + \alpha^3 + \ldots = \frac{1}{1-\alpha} \qquad \blacksquare$$

If α is a constant, Theorem 1 predicts that inserting an element runs in $O(1)$ time.

Theorem 2. *In a concurrent hash table with load factor $\alpha = n/m < 1$, the expected number of probes in a successful search is at most $1/\alpha$.*

The proof is similar to that of Theorem 1. If α is a constant, Theorem 2 predicts that searching an element runs in $O(1)$ time. For example, if we have a table with $\alpha = 0.95$, then the average search will take 1.05 probes, meaning that when the server is overloaded, a worker thread can quickly retrieve a connected socket and service a pending request.

4.2 Analysis of Pipe Synchronization

Here, the roles of producer and consumer are reversed w.r.t. the concurrent hash table. The worker threads insert synchronization tokens (`true` values) while the main server thread searches for an empty slot to find a free worker. Similarly to hash table, given an array $A[1..m]$ with m elements n of which are set to `true`, we define the load factor β for A as n/m, $\beta \leq 1$.

Proposition 1. *Insertion of `true` by a worker thread takes $O(1)$ time.*

Theorem 3. *Assuming uniform hashing, searching for an element into array $A[1..m]$ with load factor $\beta < 1$ requires at most $1/\beta$ probes on average by the server thread S.*

The proof is similar to that of Theorem 1. If β is a constant, Theorem 3 predicts that searching an element runs in $O(1)$ time.

5 Experimental Results

To evaluate the server design alternatives, we run the same client on 35 hosts running Windows NT against each server, measuring the server wall-clock time required to process a fixed number of requests. We summarize all our CPU timings in Table 1. The readings recorded in column 3 (column 4) correspond to 250 milliseconds (500 milliseconds) server delay before sending back the response.

Table 1. Timing comparisons of the various server designs

#	Server description	CPU time	CPU time
1	Iterative server	700000	-
2	Concurrent server, one thread per child	12552	21499
3	Locking around accept with 50 threads	21153	33602
4	Shared buffer with 50 threads, 101 capacity	16562	28960
5	Concurrent queue with 70 threads	13048	-
6	Concurrent hash table with 50 threads, 101 capacity	14562	24265
7	Pipes with 150 threads	13195	19253

Each client spawns 4 child clients to create 4 simultaneous connections to the server, for a maximum of 140 simultaneous connections at the server at any time. Further, each client makes 20 connections to the server amounting to 2800 connections altogether. For the tests involving multithreaded servers, the server creates up to 200 worker threads when it starts. We consider also the effect of having too many/few threads. The clients send 1 byte to the server, which responds with 4000 bytes after waiting for a predefined interval of time specified in milliseconds. The number of worker threads and the buffer capacities recorded in the second column of Table 1 were found experimentally to give the best results for the corresponding server design. Table 2 shows how we have arrived at the figures for the server using a concurrent queue.

The small CPU time obtained for the concurrent server indicates that spawning a new thread may be less expensive than synchronizing a great number of pre-spawned threads. The smallest CPU time is for the server using pipes. In this design, not only is there no synchronization among the worker threads, but there is no additional cost for spawning worker threads. We were interested to find out the threshold after which the server using pipes denies service, i.e., the effect of having too many threads to synchronize. We ran experiments with up to 700 simultaneous clients, and the server was still providing a satisfactory service.

Table 2. Effect of threads number on the performance of the pre-threaded concurrent server using concurrent queue as a shared buffer

# threads	10	30	40	50	60	70	80
CPU time	70312	23781	20828	17549	17047	13048	18203

6 Conclusion

In this paper we compared the performance of seven server designs by running them against the same client. The experimental results show that only one pre-threaded server, the one using pipes, outperforms the classical concurrent server

where the server spawns a new thread to handle the client connection. This leads us to the conclusion that spawning a new thread is less expensive than synchronizing a great number of threads. We introduced a new concurrent data structure called concurrent hash table. Although theoretically sound, in practice the performance of the concurrent hash table is not so good as the performance of the server using pipes. We also considered the number of worker threads the server should spawn in order to get maximum performance. We found out that better performance is achieved by using a greater number of worker threads, in the range of 100 rather than 15 as is commonly recommended.

We plan to run more experiments to fine-tune the behavior of the concurrent hash table. We are going to test this data structure on a multiprocessor and to gather data about the number of collisions. Similarly to hash tables this concurrent data structure is highly sensitive to the choice of the hash function and the capacity of the table. Theoretically, this server design should give good performance when the load factor α approaches 1.

References

1. T. Cormen et al., *Introduction to Algorithms*, The MIT Press, 1994.
2. S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, 1998.
3. G.C. Fox and W. Furmanski, "Java for parallel computing and as a general language for scientific and engineering simulation and modeling," *Concurrency: Theory and Practice*, Vol 9(6), pp.415-425, 1997.
4. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Sun Microsystems, Inc., Palo Alto, CA, 1996.
5. E.R. Harold, *Java Network Programming.* O'Reilly & Associates, 1997.
6. C.A.R. Hoare, "Monitors, An Operating System Structuring Concept," *Communication of the ACM*, Vol.17, pp.549-557, Oct. 1974; Erratum in *Communication of the ACM*, Vol.18, p.95, Feb. 1975.
7. D. Lea, *Concurrent Programming in Java*, 2nd ed., Addison-Wesley, 1999.
8. M. Michael and M. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," *In Proc. of the 15th ACM Symposium on Principles of Distributed Computing*, Philadelphia, Pennsylvania, pp. 267-276, May 1996.
9. J. Postel, ed., "Transmission Control Protocol," RFC 793, 1981.
10. J. Postel, "Internet Protocol," RFC 760, 1980.
11. B. Roussev and J. Wu, "Lottery-based scheduling of multithreaded Java applications on NOWs," *Annual Review of Scalable Computing*, Vol.3, 2001.
12. W.R. Stevens, *Unix Network Programming*, Vol.1, 2nd ed., Prentice Hall, 1998.

Appendix

```
class HashBuffer {
   protected SynchronizedRef[] table ;
   protected int capacity ;
   public HashBuffer( int capacity ) {
      this.capacity = capacity ;
```

```
      table = new SynchronizedRef[capacity] ;
      for ( int i = 0; i < capacity ; i++ )
         table[i] = new SynchronizedRef( null ) ;
   }
   protected void insert( Object obj ) {
      int pos = 0 ;
      int key = obj.hashCode() ;
      if ( key < 0 ) key = -key ;
      int hash1 = key % capacity ;
      int hash2 = 1 + key % (capacity - 1) ;
      while ( true ) {
         pos = hash1 ;
         if ( table[pos].get() == null ) {
            table[pos].set( obj ) ;
            return ;
         }
         for ( int i = 0; i < capacity ; i++ ) {
            pos = (pos + hash2) % capacity ;
            if ( table[pos].get() == null ) {
               table[pos].set( obj ) ;
               return ;
            }
         }
         synchronized( obj ) {
            try { obj.wait( 500 ) ; }
            catch ( InterruptedException e ) {}
         }
      }
   }
   protected Object search( int key ) {
      Object temp = null ;
      int pos = 0 ;
      if ( key < 0 ) key = -key ;
      int hash1 = key % capacity ;
      int hash2 = 1 + key % (capacity - 1) ;
      while ( true ) {
         pos = hash1 ;
         if ( ( temp = table[pos].set( null ) ) != null )
            return temp ;
         for ( int i = 1; i < capacity ; i++ ) {
            pos = (pos + hash2) % capacity ;
            if ( ( temp = table[pos].set( null ) ) != null )
               return temp ;
         }
         synchronized( obj ) {
            try { obj.wait( 500 ) ; } catch(InterruptedException e){}
         }
      }
   }
}
```