Advanced Queues

Mark Sitkowski C.Eng, M.I.E.E
http://www.designsim.com.au

IBM's MQ Series

Following on from the discussion of some of the types of queuing mechanisms, available on Unix (see 'Basic_Queues.doc), it may be beneficial to examine the design of a queuing system, loosely modelled on that used by IBM, in its Websphere MQ Series.

The MQ paradigm is based on the concept of a Queue Manager, which controls a set of local queues, which are grouped into Channels, which latter are of the 'send' or 'receive' variety.
These queues are disk based files, each located under a directory, which bears the name of the queue.
Normally, there will be several inbound and several outbound queues combined into a receiving or sending Channel, respectively, where each Channel has, associated with it, the IP address of the remote machine.

In operation, each queue manager performs as a simple TCP/IP server, and listens for connections on its own separate port, whose number is around 1414.
Connections arrive from other machines and, in the traditional manner, the server forks to service each connection, transferring double-byte data from the socket to the disk.
Although this, in essence, appears to be a classic client-server system, the connections are, actually, from server to server, or queue manager to queue manager.
Most of the queue manager's time is spent listening for incoming connections, and polling its local outgoing queue directory, waiting for some local application to place data on a queue.
When this happens, the local queue manager forks, and makes a connection to the remote machine, whose IP address, and port number are available from the channel definition. While the child process is transferring data, the parent continues to check the directory, and listen to port 1414.

It may be instructive to follow this sequence through.

1. Machines A and B both listen on port 1414.
2. Machine A's queue manager is informed that an application has placed a message on the 'send' queue.
3. The queue manager forks a child process, which makes a socket connection to machine B, on port 1414.
4. Machine B's queue manager accepts the connection, and forks a child, to handle the connection. The child uses port 32768.
5. Data is transferred, from A to B.

Note, that there is no conflict, in terms of the port number, since both managers fork child processes, which use dup'd ports.

Unix shared memory-based queue functions

There are only four system calls associated with memory-based queuing functions:

msgget() – which creates or identifies queues, but does not get messages.
msgsend() – which does actually send a message to a queue
msgrecv() – which fetches messages from a queue
msgctl() – which either interrogates or deletes the queue

We create a queue like this:

```
int qd, qe;

   if((qd = msgget(IPC_PRIVATE, IPC_CREAT|0777)) == -1){
      perror("msgget");
   }
```

The IPC_PRIVATE parameter, has the same significance as it has for creating shared memory segments, and can actually, be any unique identifier contained in an int – or 'key_t', as it is defined in the header.
The IPC_CREATE | 777 flags are identical to those used with the open() system call for creating files, and have the same meaning.
The return value, qd, looks and works like a file descriptor, and is the queue identifier, which we will use, when accessing this queue.

Now that we have an identifier, we can read from and write to the queue but, first, we need to define a queue element structure.
As with other Unix queues, this structure has to look like this:

```
struct {
   long typ;
   char txt[length];
} msg;
```

Meaning that, when it is lying about in memory, or in flight down a TCP/IP connection, it looks like a packet, with a four-byte header, and a 'length' byte payload.
More importantly, the queue functions expect to read a piece of contiguous memory, which conforms to this definition.
The significance of this, is that we cannot dynamically allocate the memory, for the 'txt' member, since this would make our message look like an int, followed by a pointer. At best, we would enqueue or dequeue 8 bytes and, at worst, we would get a segmentation error.

As usual, there is a work-around for this apparent limitation.

Basically, we don't define a structure, at all, but concentrate on the packet concept.
If we declare a pointer to unsigned char

```
unsigned char *xsg;
```

then, we can send any kind of data, of any length. The only thing we need to remember, is to always allocate 4 bytes more memory, than we actually need, and to commence writing our data four bytes past the starting address of our allocated memory.

By way of example, let us assume, that we wish to enqueue an array of double-byte characters, which look like int's on a Solaris system, and shorts on most other Unix systems.

```
wchar_t  wchars[255];
```

For convenience, we would declare a dummy pointer to an int, so that we can fool the system, into letting us add our 'type' member, to the first 4 bytes of our unsigned char array. The 'type' member contains an arbitrary, user-defined integer, which we can use later, to identify our message in the queue.

```
Int *xint;
```

Now, we can send the array to a queue but, first, we need to allocate memory:

```
if((xsg = (char *)malloc(sizeof(wchars)+4)) == NULL){
    printf("Memory allocation error\n");
}
```

Note, how we ask for an extra 4 bytes, to cater for our 'type' member.

Next, we set our pointer to the first location in the array:

```
xint = (int *)&xsg[0];
```

Now, we can insert our integer:

```
xint[0] = 9;
```

and load the newly-acquired memory with our wide-character data:

```
memcpy((char *)&xsg[4], (char *)wchars), sizeof(wchars));
```

Finally, we send it to the message queue:

```
if(msgsnd(qd, (void *)xsg, sizeof(numbers), 0) == -1){
    perror("mq_send");
}
```

The syntax for msgsnd() is fairly obvious, and follows the pattern of write(), to a file, or that of send(), to a socket.
The first argument is our queue descriptor, the second is a pointer to the data, and the third is its length. Finally, we have the flag, which defines what to do with the message if the queue either contains the maximum number of bytes, or the maximum number of messages:

- If the flag is IPC_NOWAIT, the call returns immediately, and the message is not sent.
- If the flag is zero, msgsnd() will hang, until the queue is no longer full, or the queue gets deleted, or the current process catches an interrupt.

For our purposes, a flag of zero makes for more reliable delivery, so that is what we use.

Now for the receiving function, msgrcv().

The syntax is, as with msgsnd(), similar to the read() and recv() system calls. Given a receive buffer definition similar to

unsigned char data[8192];

and type and flag parameters defined as

Int, typ, flg;

We can see how familiar the code is:

```
If(msgrcv(qd, (void *)&data, sizeof(data), typ, flg)) ==  -1){
    perror("msgrcv");
}
```

The msgctl() system call returns information about the queue.

The syntax is:

msgctl(queue, flag, structure_pointer);

where the flag is either IPC_STAT, for retrieving data, or IPC_SET, for altering queue characteristics.
In practical terms, the only items we can alter are the permissions on the queue, unless we run as root, when we can change the maximum queue size. Accordingly, information retrieval is this function's greatest value.
The structure_pointer, mentioned above, points to the msqid_ds structure, defined in sys/msg.h as:

```
struct msqid_ds {
        struct ipc_perm msg_perm;       /* operation permission struct */
        struct msg      *msg_first;     /* ptr to first message on q */
        struct msg      *msg_last;      /* ptr to last message on q */
        msglen_t        msg_cbytes;     /* current # bytes on q */
        msgqnum_t       msg_qnum;       /* # of messages on q */
        msglen_t        msg_qbytes;     /* max # of bytes on q */
        pid_t           msg_lspid;      /* pid of last msgsnd */
        pid_t           msg_lrpid;      /* pid of last msgrcv */
        time_t          msg_stime;      /* last msgsnd time */
        time_t          msg_rtime;      /* last msgrcv time */
        time_t          msg_ctime;      /* last change time */
        short           msg_cv;         /* not used */
        short           msg_qnum_cv;    /* not used */
};
```

whose members are filled in by the system call.

Putting it all together, gives us a typical call as:

```
struct msqid_ds atr;

if(msgctl(qd, IPC_STAT, &atr) == -1){
    perror("msgctl STAT server");
}
```

from which we can get useful information, such as:

printf(“Current queue length: %d\n”, atr.msg_qnum);

Implementing some MQ functionality

The MQ model is very efficient, in terms of use of machine resources. The functionality is spread over many processes, and is a good example of a Multi-Dimensional Architecture.
However, since MQ needs to use persistent queues, it throws away all of its performance advantages, by doing very heavy disk I/O. This is largely irrelevant for short messages but, if it is used for performing large data extracts, from large databases, the delays become significant.

Just for the sake of this exercise, we will assume that, for our purpose, speed of operation is of paramount importance, but the normal reliability of delivery is adequate, and a machine crash is tolerable.
Accordingly, we will design and partially code a simple queue manager, which implements most of the functionality of the MQ manager, but uses kernel queues. The implementation of any refinements is left, as they say, to the reader.

The Queue manager

First, let us remind ourselves of the two prime functions of our queue manager:

- Listen for incoming connections
- Check local queues for data needing transmission to remote sites

There are two possible solutions to this apparent dichotomy

1. Run two processes, one possibly being the child of the other
2. Run one process, within which we have two threads of execution.

Since MQ series was born at a time (and on a machine) where pthreads weren’t even a glimmer on the horizon, IBM uses multiple processes.
From a performance perspective, this would represent the best practice for us, too, but let us first specify what happens in a bit more detail.

If our server is sitting there, with its ear glued to port 1415, then how is it going to transmit data through that port?

The answer is, that it isn’t.

What happens is that, whatever checks the local queues, and decides to send the messages to the far corners of the earth, will make a connection to port 1414, in exactly the same way, as the server on the far side of the moon, with a message for us.

This means, that our server only has to differentiate between an inbound and an outbound connection, and take the appropriate action. Once the connection is established, and the server has forked a child, the child will have its own brand-new socket, and port number, and can send or receive data, at will..

The sequence of events in our server is now looking like this:

1 Listen to port 1414
2 If incoming connection is local, it is outbound.
   - Fork process for reading messages from queues
   - Inside this process, send outbound queue data to remote address
   - Child terminates
3 If incoming connection is remote, it is inbound data.
   - Fork process for enqueuing messages
   - Inside this process, enqueue incoming messages
   - Child terminates.

Now, we nearly have a complete picture, but we are pretending that we only have one queue manager, and one pair of queues at each end. In reality, we could have many, going to many destinations. Also, following the MQ policy of 'guaranteed delivery', we would need to re-send any messages which failed on the first attempt.

What about the user application?

The whole purpose of a queuing system, is to make data flow from one place to another. So, how does the user application fit in?

In an MQ environment, the application is always written such that it
- Calls CONNECT to connect to a queue manager
- Makes a request to GET inbound messages or PUT outbound messages
- Does its thing
- Quits.

Note that, given the above scenario, it is evident that the application doesn't need to directly execute the queuing system calls. The 'CONNECT()' function, from the MQ library, hides the mechanics of the TCP/IP connection to the queue manager, and the 'GET()' or 'PUT()' function calls merely pass the data down the socket connection, for the queue manager to enqueue or dequeue.

The Channel Manager

Everything seems to be automatic. Do we really need a Channel Manager?

It is the responsibility of the application to remove inbound messages, and the responsibility of the queue manager to forward outbound messages. This means, that, for inbound messages:
1. Server handles connection from remote site.
2. Server enqueues inbound message.
3. Server waits for application to connect with a GET request

For outbound messages:
1. Server handles connection from application, with a PUT request.
2. Server enqueues message.
3. Server forwards the message to the remote site.

It may be seen, that inbound messages are event driven, and need no external agent. Outbound messages are enqueued and then forwarded immediately, by the child process which enqueued them. Accordingly, unless there is a communications

failure, we may assume that the queue manager server will itself, immediately empty the outbound queue. It is this latter condition, which necessitates the use of a channel manager.
Each channel manager will be associated with a queue manager, and both will be handling traffic to and from a given destination, (the 'channel'). When the channel manager sees a message count of some number more than zero, it connects to its appropriate queue manager, on the appropriate port number, to tell it that the queue needs servicing, and the IP address where the data should be sent.

Its operation would follow this pattern:

1 Check length of outbound queue
2 If greater than zero, we need to send a message
3 Connect to port 1414
4 Give the queue manager server details of which queue, and where to send it
5 Go to sleep for predetermined period.

The Protocol

We will also need to design a primitive protocol, contained within the message header, so that the server can differentiate between an inbound and an outbound connection and, for safety, can tell both of these from a spurious connection. Within the protocol, will have to be a slot for identifying the queue and the destination, if appropriate,

While keeping within the constraints of the basic queue structure, we will add some fields to it, to make it better suit our purpose.
We need the following information:

1. Type of operation - 'GET' or 'PUT', symbolised by '1' or '0', as 1 character.
2. Name of queue – 23 characters, although MQ permits 40.
3. Length of following message + length of an int.
4. Type – an integer, as defined in msg.h
5. Data

Which gives us a 32 byte header, which fits on either a 4-byte or 8-byte boundary, such that TCP/IP will not perform any padding.

We can deduce from all of the aforegoing, that multi-threading is not going to be of any use to us. We will have two families of processes, associated with the queue manager and channel manger, and it would be inefficient to attempt to run them as two threads. Readers who disagree are invited to code a threaded version and compare response times…

Server Code

Since this is a test program, we will begin it, by creating one outbound and one inbound queue. Ordinarily, these will have been created earlier, and their descriptors saved, for passing to the queue manager.

We will need the following definitions:

```
struct xqdata{
   long typ;         /* 0=1st on queue, n=1st of type n, -n=1st of <n */
   char msg[8192];
```

```
};
struct xqdata qdata;              /* data from msg queues */

struct xmsg {
   char mode;              /* 1=GET 0=PUT */
   char qname[23];
   int length;
   struct xqdata mesg;
};
struct xmsg *msg;

int qd, qe;           /* outbound, inbound queues */

int *xint;
char *xsg;

main()

{
/* and we can now create our queues: */

/* outbound queue */
   if((qd = msgget(IPC_PRIVATE, IPC_CREAT|0777)) == -1){
      perror("msgget");
   }

/* inbound queue */
   if((qe = msgget(IPC_PRIVATE, IPC_CREAT|0777)) == -1){
      perror("msgget");
   }

/* now the TCP/IP furniture */
   memset(&sa, 0, sizeof(struct sockaddr_in));

   if(gethostname(hname, sizeof(hname)) != 0){
      printf("Can't determine our own host name. Quitting\n");
      quit(-1);
   }
   if((hp = gethostbyname(hname)) == NULL){
      return(-1);
   }
   if((svc = getservbyname("ingreslock", "tcp")) == NULL){
      printf("%s doesn't exist\n");
      quit(-1);
   } else {
      portnum = svc->s_port;
   }
   printf("Server on machine %s, listening to port %d\n", hname, portnum);

   sa.sin_family = hp->h_addrtype;
   sa.sin_port = htons(portnum);
   if((s = socket(AF_INET, SOCK_STREAM, 0)) < 0){
      quit(-1);
   }
```

```
if(bind(s, (struct sockaddr *)&sa, sizeof(struct sockaddr_in)) < 0){
        printf("Can't bind to port %d\n", portnum);
        perror("bind");
        close(s);
        quit(-1);
    }
    listen(s, 20);
    while(1){
        inlength = sizeof(sa);
        if((xs = accept(s, (struct sockaddr *)&sa, &inlength)) < 0){
            break;
        } else {
             p = (char *)inet_ntoa(sa.sin_addr);
            printf("\nIncoming connection from >%s<\n", p);
            fflush(stdout);

            switch((ppid = fork())){          /* make our child process */
                case -1:
                    perror("fork");
                    exit(-1);
                break;
                case 0:
                    if(setsid() == -1){
                        perror("setsid");
                    }

                    i = 0;
                    memset(buf, '\0', sizeof(buf));
                    while((rval = recv(ds, buf, sizeof(buf), 0)) > 0){
                        printf("Server read (%d):>%s<\n", rval, buf);
                        /*
                         * Concatenate until we have it all
                         * data[] should be dynamically allocated!
                         */
                        memcpy(&data[i], buf, rval);
                        i += rval;
                        if(i >= 8192) break;
                        rval = 0;
                        memset(buf, '\0', sizeof(buf));
                    }
                     /* we only do 'GET' and 'PUT' */
                    if(data[0] == '0' || data[0] == '1'){
                        if(readtoken(ds, data) == 0){
                            printf("Server %d:Done\n", getpid());
                            if(close(ds) != 0){
                                perror("Server close socket error");
                            }
                            exit(0); /* child can quit now */
                        } else {
                            printf("Server %d:Finished with errors\n",
                                                        getpid());
                            if(close(ds) != 0){
                                perror("Server close socket error");
                            }
                            exit(-1);
```

```
                    }
                } else {
                    printf("Server %d:Received corrupt message\n",
                                                    getpid());
                }

            break;
            default:
                printf("Server %d:Listening for next connection\n",
                                                    getpid());
                waitpid(ppid, &status, 0);
            break;
        }


    }
  }
  /* we've finished with the queues, let's delete them */
  if(msgctl(qd, IPC_RMID, NULL) == -1){
    perror("msgctl RMID");
  }
  if(msgctl(qe, IPC_RMID, NULL) == -1){
    perror("msgctl RMID");
  }
  printf("Server done\n");

}                                   /* main */
```

Additionally, we're going to need some other functions, to do the housekeeping for us.
First, the message parser:

```
/*****************************************************************************
 * Parse the message header, and extract the message. The raw format is:
 * ---------------------------------------------------------------------
 * | GET or PUT | Queue name | Length of msg + type |  type    | msg.....     |
 * |   1 byte        |  23 bytes       |  4 bytes                  |4 bytes | 8192 bytes...|
 * |     0           | 1             23| 24                      27 | 28  31  | 32 .....     |
 * ---------------------------------------------------------------------------
 * Which is defined as:
 * struct xmsg {
 *    char mode;                -> '1' = GET, '0' = 'PUT'
 *    char qname[23];
 *    int length;
 *    unsigned char mesg[8192];
 * };
 *
 * Note that the 'mesg' member is what we actually need to enqueue the msg,
 * and is in the format:
 *
 * struct {
 *    int type;
 *    char txt[8188];
 * } qdata;
```

```
 *
 ****************************************************************************/
readtoken(s, token)                 /* readtoken */

int s;
unsigned char *token;

{

pid_t pid;
unsigned int priority;
int rval;
int nwrite;
char name[25];
char mode;
int length;
int i;

    printf("\nChild server PID %d running\n", getpid());

    msg = (struct xmsg *)token;

    strcpy(name, msg->qname);
    mode = msg->mode;

    length &= 0x00;             /* make a number out of chars */
    rval = 0x00 | token[24];
    rval = rval << 24;
    length |= rval;

    rval = 0x00 | token[25];
    rval = rval << 16;
    length |= rval;

    rval = 0x00 | token[26];
    rval = rval << 8;
    length |= rval;

    length |= token[27];

    printf("Token data: queue >%s< mode >%c< length %d\n", name, mode, length);
    for(i = 28; i < rval; i++){
        printf("%c", token[i]);
    }
    printf("\n");
/*
     * This is artificial, since we need to examine the channel
     * details, to find out which queue goes where
     */

    if(mode == '1'){                        /* GET (De-queue) */
        if(strcmp(name, "MQ1") == 0){       /* Inbound queue */
            dequeue(qd, s);
        }
        else if(strcmp(name, "MQ2") == 0){  /* Outbound queue */
```

```
            printf("Doing GET from outbound queue\n");
            dequeue(qe, s);
        }
    } else {                            /* PUT (Enqueue) */
        if(strcmp(name, "MQ1") == 0){       /* Inbound queue */
            printf("Doing PUT to inbound queue\n");
            enqueue(qd, length, &token[28]);
        }
        else if(strcmp(name, "MQ2") == 0){   /* Outbound queue */
            enqueue(qe, length, &token[28]);
        }
        /* We now need to forward this message to the address at the other
         * end of this channel
         */
        forward(token, name);
    }
    return(0);

}                       /* readtoken */
```

Now, the enqueuing and dequeuing functions:

```
enqueue(qdd, length, token)             /* enqueue */

int qdd;
int length;
unsigned char *token;

{

unsigned char *xsg;
struct msqid_ds atr;

    printf("Server enqueuing messages...\n");

    if(msgsnd(qdd, (void *)token, length, 0) == -1){
        perror("msgsnd");
    }

    memset((char *)&atr, '\0', sizeof(struct msqid_ds));

    if(msgctl(qdd, IPC_STAT, &atr) == -1){
        perror("msgctl STAT server");
    } else {
        if(atr.msg_qnum > 0){
            printf("Placed %d messages (%d total bytes) on q %d\n", atr.msg_qnum
, atr.msg_cbytes, qdd);
        }
    }


}                       /* enqueue */
```

```
dequeue(qqd, s)                 /* dequeue */

int qqd;
int s;

{

struct xmsg ndata;        /* data from TCP/IP */

int i;
int rval;
int nwrite;
long typ = 0;  /* get 1st available msg */
int flg = 0;   /* block until msg arrives */
struct msqid_ds atr;

    printf("Client collecting messages from queue...\n");
    memset((char *)&qdata, '\0', sizeof(qdata));
    while((rval = msgrcv(qqd, (void *)&qdata, sizeof(qdata), typ, flg)) != -1){
        printf("Server read %d bytes of type %d queue %d data:\n",
                                            rval, qdata.typ, qqd);

        sprintf(ndata.qname, "MQ%d", qqd);
        ndata.length = rval;
        ndata.mode = 0;
        ndata.mesg.typ = 9;
        memcpy(ndata.mesg.msg, qdata.msg, rval);

        for(i = 0; i < rval; i++){
            printf("%c", qdata.msg[i]);
        }
        printf("\n");

        if(rval > 0){    /* no point writing nothing */
if((nwrite = send(s, (void *)&ndata, rval, 0)) < rval){
                perror("Server: write to socket");
                free(xsg);
                return(-1);
            } else {
                printf("Server %d: sent %d byte msg to client\n",
                                        getpid(), nwrite);
            }
        } else {
            printf("Strange: zero-length message on queue...\n");
        }

        memset((char *)&qdata, '\0', sizeof(qdata));
        memset((char *)&ndata, '\0', sizeof(ndata));
        memset((char *)&atr, '\0', sizeof(struct msqid_ds));
        if(msgctl(qqd, IPC_STAT, &atr) != -1){
            if(atr.msg_qnum == 0){
                printf("No more messages\n");
                break;
            } else {
                printf("%d messages left (%d bytes) on q %d\n", atr.msg_qnum, at
```

```
r.msg_cbytes, qqd);
            }
        } else {
            perror("msgctl STAT dequeue");
        }
    }
    printf("Server de-queued q %d\n", qqd);
    free(xsg);

}                       /* dequeue */
```

The Client Code

Our queue manager, as coded above, will work stand-alone, and GET and PUT messages to the two queues it created. What it won't do, is to forward the messages to remote sites. This is because it needs to behave as a client, in order to do that. The following code creates a stand-alone client, which we can use in place of the 'user application' and which the reader is encouraged to incorporate, as a set of functions, into the queue manager.

First, we need much the same definitions, as we did with the server:

```
struct sockaddr_in sa;
struct hostent *hp;
struct servent *svc;
int a, s;
char hostname[2048];
unsigned short portnum;
char buf[8192];
unsigned char *token;
int nread;
int nwrite;
char mode;
char queue[23];
char srcfile[255];
struct qdata {
    long typ;          /* 0=1st on queue, n=1st of type n, -n=1st of <n */
    char txt[8192];
};

int qd, qe;            /* forward, reverse queues */

struct xmsg {          /* data from TCP/IP */
    char mode;         /* 1 = GET, 0 = PUT */
    char qname[23];    /* MQ1=inbound MQ2=outbound */
    int length;
    struct qdata mesg;
};

struct xmsg msg;

int *xint;
char *xsg;
```

```
main(argc, argv)              /* main */

int argc;
char **argv;

{

unsigned char *p;
int i;
int flag = 0;
int slen = 1000000;      /* socket buffers */

    if(argc < 2){
        printf("Usage: mqclient <host> <queue> <1=GET|0=PUT> [file]\n");
        exit(-1);
    } else {
        strcpy(hostname, argv[1]);
        strcpy(queue, argv[2]);
        mode = argv[3][0];
        if(argc == 5){
            if(mode == '0'){
                strcpy(srcfile, argv[4]);
            } else {
                printf("Wrong mode >%s<\n", argv[3]);
                exit(-1);
            }
        }
    }

    printf("Client connecting to host %s\n", hostname);

    if((hp = gethostbyname(hostname)) == NULL){
        perror("gethostbyname");
        exit(-1);
    }

    if((svc = getservbyname("ingreslock", "tcp")) == NULL){
        printf("%s doesn't exist\n");
        exit(-1);
    } else {
        portnum = svc->s_port;
    }

memset(&sa, '\0', sizeof(sa));
    memcpy((char *)&sa.sin_addr, hp->h_addr, hp->h_length); /* set address */
    sa.sin_family = hp->h_addrtype;
    sa.sin_port = htons((u_short)portnum);

    if((s = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0){
        perror("socket");
        exit(-1);
    }
    if(setsockopt(s, SOL_SOCKET, SO_RCVBUF, (void *)&slen, sizeof(slen)) < 0){
        perror("Client setsockopt");
```

```
    }
    if(setsockopt(s, SOL_SOCKET, SO_SNDBUF, (void *)&slen, sizeof(slen)) < 0){
        perror("Client setsockopt");
    }
    if(connect(s, (struct sockaddr *)&sa, sizeof(sa)) < 0){
        printf("Unable to connect to %d\n", portnum);
        perror("Connect");
        close(s);
        exit(-1);
    }

    /* assemble the token */

    strcpy(msg.qname, queue);
    msg.mode = mode;
    msg.length = 0;              /* this gets set in readmsg() */
    msg.mesg.typ = 9;            /* arbitrary identification number */

    if(mode == '0'){             /* get that which we wish to PUT */
        readmsg(&msg, srcfile);
    }

    token = (unsigned char *)&msg;
if((nwrite = send(s, token, sizeof(struct xmsg), 0)) < sizeof(struct xmsg)){
        perror("Write to socket");
        exit(-1);
    } else {             /* the write succeeded */
        printf("Client sent %d byte token:>%s<\n", nwrite, token);
        if(mode == '1'){                        /* we asked to read msgs */
            printf("Client waiting to read queue...\n");
            memset(buf, '\0', sizeof(buf));
            flag = 0;
            while((nread = recv(s, buf, sizeof(buf), 0)) > 0){
                printf("\nRead %d byte message:\n", nread);
                for(i = 0; i < nread; i++){
                    printf("%c", buf[i]);
                    if(buf[i] == EOF && buf[i+1] == '\0') flag = 1;
                }
                if(flag == 1) break;
                memset(buf, '\0', sizeof(buf));
            }
            printf("\nReceived messages\n");
        } else {                                /* no reply needed */
            /* do nothing */
        }
    }
    close(s);
}                           /* main */
```

Now, we need to have our application read in arbitrary data (like double-byte XML) and set it into our data structure for transmission. The following trivial function accomplishes this:

```
/*************************************************************************
```

```
 * Reads a double-byte XML message from a file, and appends it to the
 * token array.
 ****************************************************************************/

readmsg(where, file)                    /* readmsg */

struct xmsg *where;
char *file;

{

int fd;
int i;

    if((fd = openfile,, O_RDONLY)) == -1){
        printf("Can't open %s\n", file);
        return(-1);
    }
    if((nread = read(fd, buf, sizeof(buf))) > 0){

        /* we assume the message is never bigger than the buffer */
        where->length = nread;

        printf("Read %d byte message:\n", where->length);
        memcpy(where->mesg.txt, buf, nread);

        for(i = 0; i < nread; i++){
            printf("%c", buf[i]);
        }
        printf("\n");
    } else {
        perror("Read XML");
        return(-1);
    }
    close(fd);

}                           /* readmsg */
```