1991

Design and Implementation of a Distributed System with Dynamic Resource Allocation Based Upon Bidding Schemes

Dan C. Marinescu

Kenneth Wenstrup

Patrick Davis

Report Number:
91-046

DESIGN AND IMPLEMENTATION OF A DISTRIBUTED SYSTEM WITH DYNAMIC RESOURCE ALLOCATION BASED UPON BIDDING SCHEMES

Dan C. Marinescu
Kenneth Wenstrup
Patrick Davis

CSD-TR-91-046
June 1991

Design and Implementation of a Distributed System with Dynamic Resource Allocation Based Upon Bidding Schemes

Dan C. Marinescu, Kenneth Wenstrup and Patrick Davis

Computer Sciences Department
Purdue University
West Lafayette, IN 47907 USA

June 10, 1991

1 Background

A fairly large number of distributed systems have been proposed and built in the last decade; see for example [Bar81], [Jon79], [Laz81], [OuS80], [Pop81], [PoM83],[TsL83], [WiV80].

A critical issue in designing a distributed system is the resource allocation problem – namely, how to share available resources among the set of consumers. This problem is considerably more difficult than resource management in a centralized system. The resource management in a centralized system is a scheduling problem [CoD73], [Con77], [VaD76] related to the classical notion of job sequencing [CoM67] in the study of production management [Buf77]. In this case the resource management is performed by a scheduler which enforces a policy subject to a set of constraints. The resource allocation policy e.g. First Come First Serve, FCFS, priority scheduling and so on, is a set of rules for the scheduler to decide which customer will get access to the resource. The constraints dictate the objective function to be optimized. Possible approaches are: to attempt to optimize the overall resource utilization with no regard to customer satisfaction, or to attempt to minimize the response time with no regard to resource utilization, or some middle ground policy.

The problems are considerably more difficult in a distributed system because the overall system status is not available to any one party. Both the resources and the consumers are scattered throughout the system and a finite communication delay exists from the time a change of state occurs and the time a remote agent responsible for resource allocation registers this change. A critical issue is to have a load balanced system, to ensure that each resource has an equal contribution to the overall system load. Different strategies exist to acomplish this goal. Another non-trivial problem is to ensure that explicit consumer requests e.g. the time by which a task is completed, or the need to assign to the task a group of resources can be met.

The mechanism being discussed here is a global, dynamic, physically distributed, cooperative, heuristic scheme. It is based upon the idea of bidding. In a bidding scheme, a producer/consumer agent places a bid by informing the others about its availability/needs and specifying a cost associated with the services. Several types of bidding schemes are possible. For example one could have resources which advertize themselves and the consumers place bids to acquire resources. An alternative scheme is to have the consumer advertize their needs and have the resources placing bids to be used. A scheme which allows both consumers and producers to advertize is also possible.

The basic mechanism of bidding was discussed in [Smi80] using a different terminology. In [Smi80] each node is responsible for two roles with respect to the bidding process: manager and contractor. The manager represents the task in need of a location to execute, and the contractor represents a node which is able to do work for other nodes.

In this scheme a broker is used as an intermediary both by the producer and the consumer agents, it acts as a *matchmaker*. A broker runs in every node of the net and has multiple functions. It is used by the agent which adverizes to make known its needs/availability to all or to a subset of other brokers. In turn these brokers contact the local scheduling agents to determine if the node can provide/use the services being requested/available. This scheme has been discussed at least in name in [Tay89] and a commercial product based upon the idea is available from Hewlett Packard. All nodes have full autonomy in the sense that the broker decides to answer to a bid or to ignore it.

An application of this scheme to co-scheduling computing intensive tasks in a network of workstations is presented in [Ata91].

2 Design and Implementation of a Broker System

2.1 Design Considerations

The broker system described here was developed from an earlier bidding system consisting of client daemons and server daemons running on Sun workstations on the Ethernet LAN [Mar90]. In this earlier system, two daemons ran on each host. A client daemon was responsible for receiving service requests from clients on the local host, broadcasting a bid request over the Ethernet, receiving and evaluating the bids made in response to this request, sending accepts/rejects back to the bidding workstations, and sending the results back to the local client. The server daemon was responsible for receiving bid requests from remote client daemons, making bids on behalf of servers running on the local host, and confirming or removing these bids once an accept/reject was returned.

In the new system, the functions of the client daemon and server daemon are combined into one process, the broker. This not only lowers the number of processes which need to run on each workstation, but it also allows certain data structures and procedures to be shared by both functional parts of the broker. For example, both the client serving part of the broker and the server serving part share the resources needed for communication over the network.

In addition to combining the server daemon and client daemon into one process, the new system was designed with several other goals in mind. These goals included:

- To design and support client-broker and broker-broker protocols for communication in the bidding system.

- To support multiple bidding algorithms and create a flexible design which allows new bidding algorithms to be easily added to the existing system.

- To make the system highly portable, so that it does not need to run on the Sun workstations or over the Ethernet LAN.

- To support a dynamic network configuration so that the workstations to which a bid request is broadcast and those to which a workstation will make a bid can be easily changed.

2.2 The Broker System Protocols

Most communication between client and broker and among the brokers is done using UDP sockets in the Internet domain.[1] This unreliable, connectionless communication is preferred for several reasons. It allows the broker to communicate with multiple clients and other brokers without needing to continually create and break connections. It also enables the use of one well-known socket within each broker, rather than needing a socket to listen for connections and one for creating connections to other processes.

The protocols described below were developed to create a quasi-reliable mode of communication over the unreliable UDP protocol. This is done through the use of acknowledgement packets, and by retrying unacknowledged sends multiple times. In the following protocols, constants such as TO_SRQ and RT_SQR are used to tell how long to wait for acknowledgement of a packet and how many times to resend an unacknowledged packet; these constants are defined within the system. The names in parentheses, such as **servreq**, are the structure types of each kind of packet within the broker system.

Client-Broker Protocol

1. Client sends a service request (**servreq**) to broker.

2. The broker upon receiving the service request, returns a service request acknowledgement (**servreqack**) and begins the bidding process (see broker-broker protocol below).

3. If the client does not receive an acknowledgement back within TO_SRQ microseconds, it does one of two things. If it has not sent the request RT_SRQ times, it returns to step one. Otherwise, it informs the user that the broker on this workstation is unreachable and quits.

4. After the broker has accepted a subset of bids (see broker-broker protocol), it returns to the client a service reply (**servrep1**) which tells the client how many packets are to follow. The broker then sends a service bid packets (**servbid**) for each accepted bid.

5. The client returns a service reply acknowledge packet (**servreplack**) or a service bid acknowledge packet (**servbidack**) for each received packet from step four.

6. The broker waits TO_SRP and TO_SRPB microseconds to receive a service reply acknowledge or a service bid acknowledge packet respectively. If the broker has not tried the service reply RT_SRP times or the service bid RT_SRPB times, it retries the send. Otherwise it gives up sending this packet.

Broker-Broker Protocol

1. Upon receiving a service request from a client, a broker forms a bid request (**bidreq**) and sends this to all other brokers in the system.

2. Upon receiving a bid request, the receiving broker determines if it will make a bid or not to this request. If not, it does nothing. Otherwise, it forms a bid reply (**bidrep1**) and sends this reply back to the requesting broker.

[1] Options exist which allow a broker to locally broadcast bid requests using special broadcasting methods available on its network. These broadcasts might not use the VOP socket.

3. The requesting broker gathers all bids received in TO_BRQ microseconds. After this time, it evaluates these bids and determines if it will accept a subset of them.

4. If the requesting broker cannot find an acceptable subset of bids, it does one of two things. If the request has not been attempted RT_BRQ times, it resends the bid request and begins the bidding process again. Otherwise it sends a service reply to the client telling it no bids were accepted.

5. If the requesting broker accepts a subset of bids, it sends the results back to the client as described above. It also sends bid acceptance packets (**bidacc**) to all brokers that submitted bids. A field in this packet tells whether the bid is being accepted or rejected.

6. If the bidding broker does not receive a bid acceptance packet within a timeout period which is determined by the bidding algorithm being used, it removes this bid from its schedule.

7. If the bidding broker does receive a bid acceptance packet within time, it either confirms this bid or removes this bid from its scheduling depending on whether the bid is being accepted or rejected.

2.3 Implementation of the Brokers

All packets are received on a broker's well-known socket as a generic packet type. Every packet is the same size and contains a common field which tells the packet type. After a packet's type is checked, it is sent to a function which processes this type of packet. These functions may send the packet to another subfunction based on the bidding algorithm being used. See the implementation notes in the section "A System Supporting Multiple Bidding Algorithms", for details on how the hierarchy of packet types and functions is achieved in the broker system. In this section, we will describe how the broker keeps track of a session throughout the processing of a service request. We will explain the data structures used for tracking a session and organizing the various packets received and created by a broker.

Each broker loops continuously through three checks. First it checks to see if any incoming packets are available on its well-known socket, and is so, it receives and processes one of these packets. Next, it determines if any packets are waiting on its write queue; it so, it sends the first packet on the queue to its destination. Finally, the broker checks its timeout queue. This queue contains packets which have been sent and which require some processing when the timeout is reached. For example, a bid request is placed on the timeout queue after being sent. When its timeout is reached, those bids received in response to this request are evaluated. Other packets on the timeout queue include service replies and service bids. If they are not acknowledged by the time their timeout is reached, they may need to be requeued for writing.

The write queue and timeout queue use the following data structures for storing packets:

```
/* Structures used for packet queues */
struct qnode{
    struct pack *pkt;        /* Packet to be sent, just sent or received */
    int broadcast;           /* Broadcast packet or not */
    struct sockaddr_in dest; /* Address to send to, just sent to, or */
                             /* received from */
    struct timeval timeout;  /* Timeout for timeout queue */
    struct session *ss;      /* Session which this packet is for */
    struct qnode *next;      /* Linked list pointers */
    struct qnode *prev;
    };

struct queue{
    struct qnode *first;
    struct qnode *last;
    };
```

The **pkt** field points to the packet to be sent in the write queue or the packet just sent in the timeout queue. The **broadcast** field tells if this packet is to be sent to or was just sent to a single address, in which case the **dest** field gives that address. Otherwise, the packet is or was sent to all available brokers in the system, as in the case of a bid request. The **timeout** field is used only in the timeout queue for telling when this packet is to be timed out and processed.

The **ss** field in the **qnode** points to the **session** to which this packet belongs. A **session** is created whenever a service request is received from a client, and it exists in the broker while the bidding for this session is in process. A list of the following session structures is maintained in each broker:

```
/* Used by client side of broker to track a bidding session */
struct session{
    struct servreq *request;        /* The original service request */
    struct sockaddr_in client_addr; /* The client address */
    struct queue bidllist;          /* Bid received so far */
    int repidnt;                    /* Identifier unique to this session */
    int wait_repls;                 /* Number of acks to expect before */
                                    /* removing session */
    struct session *next;           /* Linked list pointers */
    struct session *prev;
    };
```

The **request** field contains the original service request, and the **clientaddr** field holds the Internet address of the client making this request. After a bid request is made, the returned bids are stored in the **bidlist** field, which has the same structure as the write and timeout queues described above. The **repidnt** is a unique integer which the broker assigns to each session. It is included in the bid request and is used to associate returned bids with the proper session. Finally, there is the **wait_repls** field. This is used to keep track of how many service reply and service bid packets have been sent back to the client after a subset of bids is accepted. As each of these packets is acknowledged (or times out and is not resent), the **wait_repls** counter is decremented. When all of these packets have been acknowledged or timed out, the broker knows that this bidding session is finished and that the **session** structure can be deallocated.

The session list and timeout queues are only used on the client serving side of the broker. A **session** tracks information needed by the requesting broker about the bidding process for each request it is servicing. On the bidding broker side, the write queue is still used to queue packets which must be sent (with the ss field **NULL**). However, it keeps a different structure called an **sjob** to keep track of each bid request to which it has made a bid. The **sjob** structures below are kept in a scheduler queue so that the broker can keep track of all the bids which it has made:

```
/* Scheduler structure which tracks each bid made */
struct sjob{
    int status;         /* TEMP or CONFIRMED */
    int sidnt;          /* Identifier unique to this job */
    long start;         /* Scheduled start time of this job */
    long elapse;        /* Scheduled elpase time of this job */
    long timeout;       /* Time this job will be removed if not confirmed */
    struct sjob *next; /* Linked list pointer */
    };
```

As each bid is made, an **sjob** is created and queued on the scheduler with its **status** being TEMP. If this bid is rejected or the **timeout** is reached without hearing back from the requesting broker, then the **sjob** is removed from the scheduler queue. However, if the bid is accepted, then the status of this **sjob** is changed to CONFIRMED. The **sidnt** field functions like the repidnt on the client side; it is used to associate accepts/rejects with the sjob to which they refer. The **start** and **elapse** fields are used to keep track of when the computation is scheduled to run. By using these fields, the broker knows when a computation is supposed to be finished and thus it can remove the sjob from its scheduler queue.

2.4 Conversion and Implementation of the Broker on Multiple Platforms

With the large number of workstations in the marketplace today, there is a great need to allow the broker to function on as many as practical. The original implementation used a Sun specific method for broadcasting bids, and a method was needed to allow a generic broadcast. The mechanism for this broadcast would be based on TCP/IP sockets to allow for compatibility between machines. The machines chosen for this initial porting operations were the NeXT workstation running the Mach operating system. Since Mach was based on Unix, no significant rewrite was necessary. Part of the port was to maintain the ability to use local communication capabilities, such as NIT on the Suns, if required. The ability is currently implemented using the C preprocessor directive #ifdef. This directive allows the code specific to a platform to be inserted into the program before compilation. If no capabilities are present on a platform, a default mechanism is used, currently UDP point-to-point communications. The current code has been tested on the Sun-3 workstations and the NeXT workstation and requires no modifications to work on either platform. It has been ported to HP workstations using HP-UX, but requires additional libraries to be linked on them but no additional changes are needed to the code.

The current method for porting the system to a new platform is to make a tar file of the directory containing the broker system. Once the tar file is made, FTP is used to transfer the system to a new machine. The file is then untarred, any binary files are removed, and the executables are remade using the Makefile in the appropriate source directory. Many problems that occur during the port are due to differences in the structure of the host operating system. By using the previously mentioned #ifdef and #ifndef, any specific need of the host can be met. The other problems that occur frequently are compiler warnings, since not all C compilers are the same.

Most changes needed to remove the warnings don't need the use of #ifdef since they will work with all machines.

Adding New Capabilities for Existing Options

The only option that uses the #ifdef construct is the local broadcast. The #ifdef construct can be used for any future option that will vary by using the format used by local broadcast. When a new machine characteristic needs to be added to a preexisting #ifdef, place the new code in an #ifdef following the #else. If no #else exists, add the code before the #endif and place an #else before the second #ifdef. Note that the number of #endif should equal the number of #ifdef and the number of #else should be the same if it existed before the addition. The default characteristic should follow the last #else:

```
ex: Add new code for machine type abc

existing code:
        #ifdef sum
             nitwrite(nitDev,mesg,strlen(mesg));
        #else
             bcstsnd(... );
        #endif

add line: ethbcast(mesg,strlen(mesg));

new code:
        #ifdef sun
            nitwrite(nitDev,mesg,strlen(mesg));
        #else
        #ifdef abc
            ethbcast(mesg,strlen(mesg));
        #else
            bcstsnd(... );
        #endif
```

2.5 Controlled Resource Allocation and the Use of Configuration Files

Originally a simple text file of hostnames was used to enable the brokers to broadcast to each other. With the need to allow for local broadcast, a new file was needed to allow the brokers to distinguish between the hosts that could be reached by local broadcast and those that could not. This fact, together with the need to allow for flexibility and expansion, defined the requirements for a configuration file. This file would define options for the broker, the machines with which to communicate, and possibly other specifications like services to offer. Instead of simply reading names and values from a file, LEX and YACC were used to develop and implement specifications for the configuration. The current specifications allow for three types of block structures: options for the broker, local machines (those reachable by local broadcast), and remote machines. In addition, comments are allowed anywhere in the file when begun with a '#'.

The intention of the option structure is to allow the broker to be configured by simply setting a flag when the option is encountered. This flag will be used by other routines to change their operations and should only be set by code in the YACC code. Each of the options is entered

following the delimiter OPTION. Currently the only option in use is a flag to specify whether to use local broadcast. The local machines and remote machines sections are the same except for their use. It is important to note that if local broadcast is set, none of the machines in the local machines section will be reachable using UDP. Following the delimiter for each section, LOCAL and REMOTE, a list of machines can be entered. The entry for each machine can either be a valid name or an IP address. Comments can be used in the configuration file to delimit different networks or machine types in the sections. The machines will be placed into two arrays, one for local machines and one for remote machines. The arrays contain structures which contain the information retrieved from /etc/hosts by the use of standard network procedures, gethostbyname and gethostbyaddr. The use of LEX and YACC allows for the future expansion of the configuration. Possibilities for expansion include the exclusion of machines and nets and the services that the local machine is to offer. All new capabilities will require changes in both the LEX and YACC specifications. The changes to LEX would be minor but new entries must be developed to be placed in the YACC specifications.

BNF Specification for Configuration File

```
<config> ::= <options> <local> <remote>
           | <options> <local>
           | <options> <remote>
           | <options>
           | <local> <remote>
           | <local>
           | <remote>

<options> ::= OPTION <option1>

<local> ::= LOCAL <machines>

<remote> ::= REMOTE <machines>

<option1> ::= <option>
            | <option1> <option>

<option> ::= LCLBCAST
           | lclbcast

<machines> ::= <machine>
             | <machines> <machine>

<machine> ::= <address>
            | <name>

<address> ::= <NUM> . <NUM> . <NUM> . <NUM>

<NUM> ::= <digit>
        | <NUM> <digit>
```

```
<name> ::= <IDENT>
         | <IDENT> . <IDENT>
         | <IDENT> . <IDENT> . <IDENT>
         | <IDENT> . <IDENT> . <IDENT> . <IDENT>

<IDENT> ::= <letter> <ident1>

<ident1> ::= <letter>
           | <ident1> <letter>
           | <digit>
           | <ident1> <digit>
           | <identchar>
           | <ident1> <identchar>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o
           | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D
           | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S
           | T | U | V | W | X | Y | Z

<identchar> ::= - | _
```

2.6 Implementing New Features in LEX and YACC

The use of LEX and YACC was picked because the complexity of the configuration file would increase as more functionality in the broker is needed. The current LEX specification allows for comments beginning with '#', the delimiters OPTIONS (option section), LOCAL (local machines), and REMOTE (remote machines), numbers composed of the digits 0–9, Internet machine names, and '*' for use when specifying the networks like "128.*.*.*". All but '*' are currently used. The current YACC code provides the grammar needed for the Options, Local, and Remote sections. Only one option, lclbcast (local broadcast), is defined and the machines are defined by a common grammar.

The LEX specifications defines the basic token types for use by the YACC grammar. The YACC code specifies the sections and order in which they appear in the configuration file. Each of the sections do not have to appear in the file, but must always appear in the same order. The option section is started when the delimiter, OPTION, appears. The next item in the file must be a valid option with no order necessary for the options. The local section is started by the delimiter, LOCAL, and must be followed by machine names or addresses. The remote section is started by the delimiter, REMOTE, and like the local section, must be followed by machine names or addresses. The same code is used to process the machine names and addresses, but a flag tells whether it is local or remote. The valid names and addresses are then placed into the appropriate array of machine structures.

The modification of the definition of the configuration file requires knowledge of the C language, LEX, and YACC. The most complicated modification will be the addition of a new section to the

specifications. This will require additions to both the LEX and YACC specifications, along with the code to use the new features in the other procedures of the broker. In the LEX file, scanner.l, all components of the new section, the section delimiter, and entry definitions need to be specified. In the YACC code, parser.y, new token types have to be established for all of the new entries in the LEX specification. Also the new section has to be defined and the start of the section needs to be entered into the starting point for YACC,the config list.

A simpler addition is the modification of the code to add a new option. There are four files which have to be modified to accomplish this task. The first, options.h, needs to be modified to have the option name as an extern char and also needs to be included in any file using the option. The second, decls.c, needs to be modified to include the new declaration for the new option declared as a char. The next, scanner.l, needs to be modified to accept the option name as upper or lower case and to return the option token value. The last file, parser,y, requires two changes. The first is to add a new token for the option using %token and the second is to add a new entry in the options type similar to the lclbcast entry. The new entry will recognize the option and set the corresponding variable.

3 A System Supporting Multiple Bidding Algorithms

3.1 The Need for Multiple Bidding Algorithms

One of the primary intentions in designing the broker system described here was to support multiple scheduling models and bidding algorithms. By a scheduling model, we refer to the requirements of the compute-intensive task in breaking this computation into subtasks. Can the computation be divided into varying sized pieces or must each workstation be given an equally sized subtask? Do the subtasks have to run concurrently or can they be processed independently? The scheduling model is dictated by the type of computation which is to be processed.

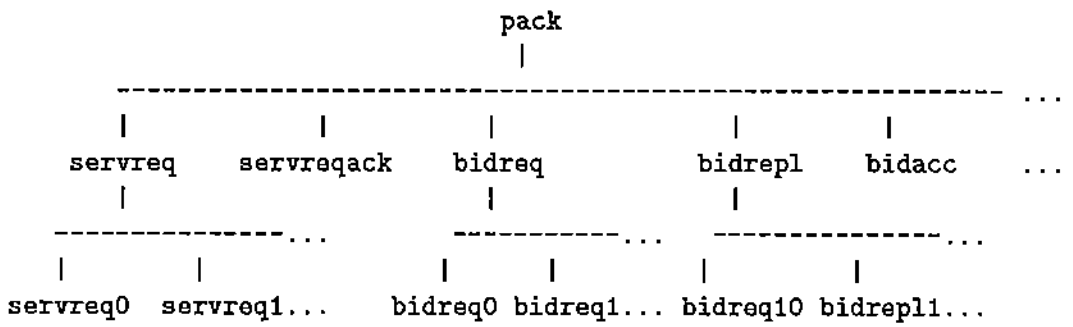
The bidding algorithm must be able to meet the requirements of the scheduling model. However, there may be many such bidding algorithms to support any one scheduling model. The bidding algorithm determines not only when a bidding broker will make a bid and what information will be included in this bid, but it also specifies how these bids will be evaluated by the requesting broker and a "best" subset chosen. Will a bidding broker make a bid only if it has no outstanding bids for other computations, or will it offer CPU time it has free between other obligations? What criteria will be used by the requesting broker to determine the best subset of bids to accept: that which minimizes the completion time of this computation; or that which best utilizes the resources of the network, such as total CPU cycles?

It is evident why an experimental broker system should support multiple scheduling models and bidding algorithms. The need to support multiple scheduling models is evidenced by the many different types of compute-intensive tasks for which distributed processing is desirable. For example, some computations, such as the solution of large numerical problems using iterative methods, require intermediate results to be shared among the servers performing the computations. Such computations require "co-scheduling" or "gang scheduling" in which the task is broken into subtasks which are scheduled to execute concurrently on a set of workstations. Other computations, on the other hand, can be broken into subtasks which can be scheduled independently of each other on a set of workstations. There is also a need to support multiple bidding algorithms, even for the same scheduling model. This allows us to compare empirically the performance of various bidding algorithms.

3.2 Implementation Notes

In the broker system described here, the client, at the time it makes a service request, specifies which bidding algorithm to use. The bidding algorithm by definition supports a certain scheduling model. One of the primary design goals was to allow new bidding algorithms to be added to this broker as they are needed. This is achieved in two major ways; new packet types are defined which hold information specific for this algorithm, and new procedures are added to process and fill in these packets.

To understand how new packet types are defined, one must understand the subtyping of packets in the broker system. All packet types are the same size. They are subtyped in a hierarchical manner.



When packets are first received on the broker's socket or are being sent from this socket, they are typed as the generic packet type:

```
/* Generic Packet */
struct pack{
   octet1 bidalg;        /* Bidding algorithm number         */
   octet1 version;       /* Used to coordinate communication */
   octet1 pactype;       /* Packet type                      */
   char info[PACSIZE-3];
   }
```

Any information beyond the first three fields is stored in the **info** section and unavailable at this level.

When further processing of the packet is required, the **pactype** field is checked to find this packet's type. It is then sent to another procedure which processes this type of packet. The receiving procedure types its formal parameter as the specific packet type. For example, bid requests are sent to a procedure called **make_bid** which types this packet parameter as a **struct bidreq**:

```
/*Bid Request */
struct bidreq{
   octet1 bidalg;          /* Bidding algorithm number          */
   octet1 version;         /* Used to coordinate communication */
   octet1 pactype;         /* Packet type                       */
   octet4 rephost;         /* Host of requesting broker         */
   octet4 repport;         /* Port of requesting broker         */
   octet4 repidnt;         /* Used by requesting broker to      */
                           /* match replies to request          */
   char info[PACSIZE-15];
   }
```

In this way, the additional fields of the bid request can be retrieved only in procedures which need to access them. Upper level procedures can send and receive all packets without having to know their packet types.

Further subtyping of service requests, bid requests, and bid replies is done for each bidding algorithm. These packet types are the ones which must be defined when a new bidding algorithm is added to the broker system. When bidding algorithm specific information must be retrieved, the **bidalg** field is checked and the packet is sent to the appropriate algorithm specific procedure. The procedure then retypes this packet parameter as the bidding algorithm specific packet type. For example, algorithm 0 is defined on the existing broker system [Ata 91] for a description of this algorithm). The procedure **make_bid** calls a procedure **make_bid0** which receives the bid request as the packet type **bidreq0**:

```
struct bidreq0{
   octet1 bidalg;          /* Bidding algorithm number           */
   octet1 version;         /* Used to coordinate communication   */
   octet1 pactype;         /* Packet type                        */
   octet4 rephost;         /* Host of requesting broker          */
   octet4 repport;         /* Port of requesting broker          */
   octet4 repidnt;         /* Used by requesting broker to       */
                           /* match replies to request           */
   octet4 exectime;        /* Computation total execution time   */
   octet4 deadline;        /* Deadline for completing computation */
   char info[PACSIZE-23];
   }
```

Besides additional packet types, each added algorithm must include algorithm specific procedures to process and fill in these packets. The design goal was to make these procedures as simple as possible to implement. Therefore, these procedures are required only to process and fill in those fields of those packets specific to this algorithm. All other functions – filling in general packet fields, packet creation/deletion, communication, etc., – are handled by upper level procedures in the existing broker.

The additional procedures which are needed for each new bidding algorithm include a procedure to read the bidding algorithm specific fields of the service request and fill in the appropriate fields of the bid request, a procedure to interpret the bid request and formulate a bid reply, and a procedure which evaluates the bid replies and chooses an optimal subset of bids to accept. Each type of procedure is called from an upper level procedure which handles the general processing of the

packet type. For example, **make_bid** expects to call a bidding algorithm specific function of the following format (where n in the procedure name and packet types refer to the bidding algorithm number):

```
int make_bidn(reqpkt, replpkt, strt, elapse, timeout)
    struct bidreqn *reqpkt;
    struct bidrepln *replpkt;
    long *strt, *elapse, *timeout;
```

This procedure evaluates the bidding algorithm specific fields of the bid request and fills in the algorithm specific fields of the bid reply. It also returns, based on the bid it is making, the time the computation will start, the amount of time it will take on this workstation, and when the bid will timeout if it is not accepted or rejected by then. However all other tasks which need to be done when making a bid, such as addressing the reply to the appropriate broker and scheduling the bid on the broker's internal scheduler, are handled by the upper level **make_bid** procedure.

Linkage to these bidding algorithms specific procedures is achieved through an array of pointers to functions. For example, the following array is declared for bid making functions:

```
int (*makebidarr[MAXBIDALGS])();
```

This array is initialized in the procedure **init_bidalgs** so that a pointer to the appropriate function for a given bidding algorithm is indexed by that algorithm's number. For example, the following initialization is made in **init_bidalgs**:

```
makebidarr[0] = make_bid0;
```

Then, in the **make_bid** function, the **bidalg** field of the request packet is checked. This field is used to index into the function array and call the appropriate function for making a bid according to this algorithm.

References

[Ata91] Atallah, M. et al, Models and Algorithms for Co-Scheduling Compute-Intensive Tasks on a Network of Workstations, *Proc. 11th Intl. Conference on Distributed Computing Systems*, IEEE Computer Society Press, 1991 (in press).

[Bar81] Bartlett, J.F., A non-stop kernel, *Proc. 8th Symp. on O.S. Principles*, 1981, pp. 22–29.

[Buf77] Buffa, E.S., Modern production management 5, ed., John Wiley & Sons, New York, 1977.

[Bla90] D. L. Black, Scheduling Support for Concurrency and Parallelism in the Mach Operating System, *IEEE Computer*, Vol. 23, No. 5, pp. 35-43, 1990.

[CoD73] Coffman, E.G. and Denning, P.J., Operating systems theory, 1973, Prentice-Hall, Englewood Cliffs, N.J.

[CoM67] Conway, R.W., Maxwell, W.L. and Miller, L.W., Theory of scheduling, 1967, Addison-Wesley, Reading, Mass.

[Con77] Gonzalez, M.J., Deterministic Processor Scheduling, *ACM Computing Surveys*, Vol. 9, No. 3, Sept. 1977, pp. 173–204.

[Hag8677] Hagmann, R., Processor Server: Sharing Processing Power in a Workstation Environment, *Proc. 6th Intl. Conference on Distributed Computing Systems*, IEEE Computer Society Press, pp. 260-267, 1986.

[Jon79] Jones, A.K., et al, Star OS, A multiprocessor operating systems for the support of task forces, *Proc. 7th Symp. on O. S. Principles*, 1979. pp. 117–127.

[Laz81] Lazowska, E. D., et al, The Architecture of the Eden System, *Proc. 8th ACM Symp. on Operating Systems Principles*, 1981, pp 148-159.

[Mar90] Marinescu, D. C., et al, Distributed Supercomputing, *Proc. Workshop Future Trends in Distributed Computing*, IEEE Computer Society Press, 1990, pp. 381-387.

[MuL87] Mutka, M. W., Litvny M., Scheduling Remote Processing Capacity in a Workstation-Processor Bank Network, *Proc. 7th Intl. Conference on Distributed Computing Systems*, IEEE Computer Society Press, 1987, pp. 2-9.

[OuS80] Ousterhout, J., Scelza, D. and Sindhu, P., An experiment in distributed operating system structure, *CACM*, Vol. 23, No. 2, 1980, pp. 92–105.

[Pop81] Popek, G, et al., LOCUS: A network transparent, high reliability distributed system, *Proc. 8th Symp. on O.S. Principles*, 1981, pp. 169–177.

[PoM83] Powell, M.L. and Miller, B.P., Process Migration in DEMOS/MP, *Proc. 9th Symp. on O. S. Principles, O.S. Review*, Vol. 17, No. 5, 1983, pp. 110–119.

[Smi80] Smith, R.G., The contract net protocol: High-level communication and control in a distributed problem solver, *IEEE Trans. on Comp.*, Vol. C-29, No. 12, 1980, pp. 11-4–1113.

[Tay89] Taylor, D., The promise of cooperative computing, *HP Design and Automation*, Nov. 1989, pp. 20–21.

[TsL83] Tsay, D.P. and Liu, M.T., MIKE: A network operating system for the distributed double-loop computer network, *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 2, March 1983, pp. 143–154.

[VaD76] Vairavan, K. and DeMillo, R.A., On the computational complexity of a generalized scheduling problem, *IEEE Transactions on Computers*, Vol. C-25, No. 11, 1976., pp. 1067–1073.

[WiV80] Wittie, L.D. and Van Tilborg, A.M., MICROS: A distributed operating system for MICRONET, A reconfigurable network computer, *IEEE Transactions on Computers*, Vol. C-29, No. 12, 1980, pp. 1133–1144.