INTRODUCTION

Extensibility is one of many claims of improved performance being made for distributed processing systems. Extensibility means system properties, such as “incremental growth and configuration flexibility,” “highly adaptable to changes in workload,” “incremental replacement and/or upgrading of components (both hardware and software)” and “easily expanded in both capacity and function.” Although the need for extensibility is strongly felt, research results for it appear to be still far from satisfactory.

A distributed data base system is one example of distributed processing. Although its usefulness is well recognized, the research in this area is also still very much in its infancy. MINIMET, a transaction-oriented homogeneous distributed data base system, has contributed towards the understanding of the distributed data base system. The MINIMET research put more emphasis on the lower level of system control so that any data model can be built on it.

The data base system described in this paper is a relational data base, one of whose characteristics, among others, is capacity extensibility.

Extensibility, in this paper, means the following three things:

(a) Physical extensibility
(b) Logical extensibility and data independency
(c) Minimal performance degradation in data base growth.

Physical extensibility means that the system can physically be extended to meet a workload increase. Logical extensibility and data independency mean that the system can logically be extended and that application programs are not affected by system extension. The third characteristic, minimal performance degradation, means that degradation in performance, response times in particular, in the growth of data base is kept minimal. Because of these characteristics, the system will be called an Extensible Distributed Data base System (EDDS).

For physical extensibility, it is apparent that the less hardware for interconnections is provided the more extensible the system is. Obviously, a system consisting of independent subsystems is then most extensible. However, this architecture provides no advantages in coordinating various kinds of information. Distributed systems must consist of autonomous, but not totally independent, cooperating subsystems. The next most extensible architecture would be a common bus system, as shown in Figure 1, in that a number of subsystems are connected by an intersubsystem bus. Any more complex architecture would impose a greater difficulty in expanding the system. Particularly, a shared memory causes a great difficulty.

In the above common bus architecture, the bus capacity is most important in determining system performance. As the workload of the system and the number of subsystems increase, the bus capacity must also be increased. One method to increase bus capacity is to add additional buses. Another is to replace the bus with a faster one. The first method is architecturally more complex than the second, but it has the advantage that no replacements are necessary to increase the capacity. The second method requires bus replacement, but it is architecturally simpler. Both approaches are possible. Further details will not be gone into, it will simply be assumed that a common bus architecture is the basis of the present system.

Another important aspect of extensibility is performance degradation in data base growth. There are two major areas which cause performance degradation; input/output channels and the system itself. In order to avoid performance degradation due to the limitation in input/output channel capacity, it is assumed that the system is connected by n input/output channels to the user systems as shown in Figure 2. This is one of the reasons that a centralized control should be avoided. If there is a centralized control, then any and all data and control information must go through it and the central control may become a major bottleneck.

Like the intersubsystem bus, it is also necessary to try to minimize input/output channel replacement cost. For this purpose, additions rather than replacements are desirable.
when more capacity is needed. Both approaches are feasible and the architecture shown in Figure 2 is assumed to be the basis of discussions.

Performance degradation in the system itself is mainly caused by speed limitation of the intersubsystem bus, which is necessary to transfer data and control information between subsystems to assure proper system operation. The bus capacity is thus an important system parameter. Performance degradation is analyzed in a later section.

As indicated earlier, the system described in this paper is a relational data base. The query language for the system is relational-calculus oriented and has the following three basic statements.

Retrieval
Get R | condition
Those tuples that meet the "condition" are retrieved and placed into relation or view "R". The condition is any relational calculus involving relations and views, which are specified only by name.

Relation creation
Create R | condition
Relation "R" is created under the "condition".

Relation deletion
Purge R
Relation "R" is deleted.

Logical extensibility and data independency are now described in more detail.

LOGICAL EXTENSIBILITY AND DATA INDEPENDENCY

Logical extensibility and data independency mean that neither internal nor external logic is affected by system extension. The approach described in this paper, whereby to obtain this property, is one of the extremes of modular approach. In a general modular approach, a number of modules are prepared and put together to form a particular system. Extensibility is obtained simply by adding more modules. Depending on the size of the modules, the cost of interconnecting them and flexibility of resulting systems vary. The present system requires the following:

(1) The system must be able to work even when it consists of only one subsystem. This implies that the subsystem must be able to perform all the functions required for the data base processing.

(2) No additional programming work, either internal or external, should be required in order to add additional subsystems. This implies that every subsystem must be logically identical.

The EDDS tries to obtain internal logical extensibility in the following way.

(a) Any internal logic is exactly copied and placed in each subsystem. Therefore, no program moves are required. Note that although user programs are allowed to be cataloged and executed at a user request, they are here considered to be data.

(b) Data are separated into two categories: relations and internal data. Relations are made up of information directly created and accessed by users. Internal data are the information necessary for assuring proper system operation. These data are created and maintained by the system. Relations keep spreading over the subsystems and only one copy of each is maintained at any time. A single relation can be spread over several subsystems. On the other hand internal data are duplicated and placed in each subsystem. Some examples of internal data are relation sizes, locations and structural characteristics. This information is distributed among subsystems under a loose control. Here, loose control means that the control of mutual exclusions is neglected and, thus, there might be some discrepancy at some time between multiple copies of internal data distributed among subsystems. The correctness of any decisions based on this information is, as a result, probabilistic.

The above discussions focus on the internal logic of the data base system. The external logic’s logical extensibility is also very important, which is commonly known as “data independency.” Particularly, we are here concerned with the number of subsystems. Data independency requires that accesses to data base be made independently of the internal structure, including the number of subsystems. In order to obtain this data independency, the following basic approach is used.

(a) Each request from a user system is broadcast to every
subsystem. Note that, at this point, it is not yet known where the necessary relations reside and which subsystem is going to take care of the request. The request itself does not contain any such information. The request is written in the query language mentioned before.

(b) Let every subsystem start to try the processing of the request, but make all the subsystems except the optimal one drop out during the course of processing. Eventually, the request is taken care of by the optimal subsystem. The results are returned to the user system directly from it.

A key characteristic of this approach is expressed by the term “eventually” in the above sentence. No central control exists and no one tells any subsystem when to quit. Each subsystem is autonomous. This approach can be realized by the following algorithm.

**Algorithm**

1. A user system issues a request. The request is accompanied with the time when it was created. This created time, together with the user system identification, will be used as a unique identification of the request.
2. The request is broadcast to every subsystem in the database system.
3. The request is placed in the waiting queue of each subsystem.
4. When the turn of the request comes, its subsystem tests whether the processing of the request has already been started by another subsystem. If so, the subsystem simply discards the request; otherwise it analyzes the request and, using some criterion, it determines whether it should accept the request. If it has decided not to do so, then the request is simply discarded. Otherwise, the processing is started. The above criterion will be discussed later.
5. A subsystem now starts processing the request and sends a signal to other subsystems to notify it. If the request involves a creation of a relation, an OPEN to a dummy relation is made prior to any operations necessary for the request.
6. It is assumed that any relation must be OPENed before any accesses are allowed. When an access to a relation is made with an OPEN procedure, a check is made to determine whether an access to it has been made for the same request from another subsystem. If so, the subsystem terminates the processing of this request and simply discards it. Otherwise, it continues the processing. This checking mechanism will also be discussed later.
7. When the results are produced, they are simply returned directly from the subsystem which accepted the request to the user system that issued the request and no other subsystems and/or user systems will be involved.

Next, the acceptance criterion mentioned in (4), notifying mechanism in (5), checking mechanism in (6) and an OPEN to a dummy relation in (4) are discussed.

**Acceptance criterion**

Each subsystem determines whether it accepts a request or not according to some acceptance criterion.

This acceptance criterion should be aimed at obtaining a balanced load assignment with a minimal overhead. It must assure that at least one subsystem accepts a request. A simplest algorithm is to let each subsystem accept every request and attempt to try its processing. This algorithm is simple and uses no information about the request. We call this algorithm “all selection algorithm” for later references and assume that the queue discipline is first-come, first-served.

The algorithm can be improved by utilizing information about the request. The following information is useful:

(a) Which relations are to be referenced.
(b) How often they are to be referenced.
(c) Which of them are in the subsystem concerned.

Suppose that request $R_i$ references relations $D_{i1}, D_{i2}, \ldots, D_{im}$ with frequency $F_{i1}, F_{i2}, \ldots, F_{im}$, each. Then, the total number of references is

$$\sum_{j=1}^{m} F_{ij}.$$  

Assume that a fraction $W_{ij}$ of relation $D_{ij}$ resides in the subsystem concerned. Then, the probability that an access falls into the subsystem concerned is

$$W = \frac{\sum_{j=1}^{m} W_{ij} F_{ij}}{\sum_{j=1}^{m} F_{ij}}.$$  

A natural acceptance criterion is then:

accept if $W \geq \alpha$  

for some threshold value $\alpha$. If we take $\alpha=0$, it becomes the all selection algorithm. If we take $\alpha=1$, many requests would be missed and there would be a very high probability that a request would not be served by any subsystem at all. The maximum value that insures the acceptance of a request by at least one subsystem is

$$\frac{1}{n},$$  

where $n$ is the number of subsystems. We call the above improved criterion “sufficient selection algorithm” for later references.

**Notifying mechanism**

The request acceptance notification by a subsystem is not essential for proper system operation, but is employed to reduce overhead. The mechanism is a very simple one. Each
request is labeled with a unique identification, as mentioned before. Whenever a subsystem starts processing, it broadcasts a signal to notify every other subsystem of this. No control is placed on the work of this mechanism. Assurance of the proper operation of this mechanism requires a system-wide or intersubsystem mutual exclusion of signal setting, resetting and checking. Without a shared memory, this implementation of intersubsystem mutual exclusions is a rather time-consuming process. Rather than trying to make the mechanism complete, instead it is made probabilistic. In other words, no intersubsystem mutual exclusions are employed, although intrasubsystem mutual exclusions are guaranteed. This simplification causes cases where, due to the delay in signal transferring, a subsystem checks the signal before it arrives. Therefore, a subsystem may proceed in spite of the fact that another subsystem is already processing. This causes some loss in processing power due to duplicated work, but the mechanism itself does not need a difficult control and the architecture can remain simple. Note that a logic can be built so that there are no opposite cases; namely the case that the signal is set while no subsystems actually have started. This guarantees a necessary request acceptance condition wherein a request is always accepted by at least one subsystem.

Checking algorithm

The checking algorithm is the last gate that a request must go through. If a request is passed through this check, then it will be processed through the completion. The checking algorithm is implemented in the OPEN procedure for a relation. Each OPEN procedure has a list of processed requests augmented with their subsystem's names. Whenever an OPEN is requested, it checks whether or not another OPEN for the same request from a different subsystem has already been made. If so, the OPEN procedure returns control indicating this and the subsystem then simply discards the whole request. It is assumed that any relation is always OPENed by its owner subsystem. This assures the mutual exclusion of OPEN procedure and makes the checking mechanism much easier. Note that it is assumed a relation is owned by a single subsystem although it may be spread over several subsystems.

OPEN to a dummy relation

An OPEN to a dummy relation is issued to assure proper relation creation. For relation creations, the following problems must be considered:

(a) How to control a relation creation so that one and only one is properly created.
(b) How to distribute relations in a balanced way.

The first problem is specific to the EDDS, in which a number of subsystems essentially try to do the same operation in regard to the same relation, including a relation creation, at the same time. Thus, when several subsystems try to create a relation, operations must be controlled so that one and only one subsystem will do the job. For this control, an access is defined and attached to a dummy relation to each request that involves a relation creation. It is assumed that this access to the dummy relation is made at the top of processing the request. In this way, through using the above checking algorithm, it will be assured that one and only one subsystem completes the operation and that no extra operations are made. A small problem is how to determine the name of the dummy relation. The naming mechanism should have the characteristics:

(a) Dummy relations are distributed to each subsystem so that the system can operate irrespective of the number of subsystems.
(b) Accesses to the dummy relations are evenly distributed among subsystems.

In order to accomplish these characteristics, the following method is adopted.

(a) A dummy relation is assigned to each subsystem, whose names are subsystem numbers 1 through n.
(b) Each request is accompanied with its created time, t. The name of the dummy relation for this request is computed as (modulo n of t)+1.

In this way, it should be obvious that the required characteristics are achieved, although the total number of subsystems, n, must be known to each subsystem.

The second problem, how to distribute relations in a balanced way, is another important problem. A simple and perhaps natural way to obtain this is to let the first available subsystem accept the request and create the required relation. This method sets up a relation to be assigned to a most likely lightest-loaded subsystem. Another simple and natural way is to assign the relation to the subsystem that has the largest amount of empty space. Control of this method is also easy. However, the former load-oriented approach is taken, since it is easier when it is combined with data search operations.

CORRECTNESS AND OPTIMALITY

This section proves the correctness of the algorithm presented in the previous section and shows that the algorithm is statistically optimal for system balancing or load sharing.

Correctness

Algorithm correctness will be proved by showing:

(a) At least one subsystem accepts a request,
(b) No more than one subsystem accepts a request,
(c) No deadlocks occur
(d) System is alive
Methodology

The methodology used here is a slight extension of Bochmann and Gecsei's method. It is extended by adding remote read and write operations, denoted by READ (a:=v) and WRITE (v:=a). A READ (a:=v) specifies an access to a remote variable v and assigns the value of v to local variable a. Some arbitrary time delay exists between the initiation of the action and an actual reference to variable v. Control, however, stays at the READ command until the action is completed. A WRITE (v:=a) causes a remote variable v to be changed to the value of local variable a. A WRITE action does not wait for the completion of the action and some arbitrary delay exists between initiation and an actual change in v.

Model

For simplicity, it is assumed that the acceptance criterion is the all selection policy and limits the number of subsystems to two. Also, the number of relations used by a request is limited to two. As will be seen, an extension to a general case is very straightforward. A description of each subsystem can be given, referring to the algorithm described in the previous section.

(a) State transition diagram for subsystem k.
(see Figure 3)
(b) Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>C[i][j]</td>
<td>h if the ith relation for the jth request is OPENed by the hth subsystem.</td>
</tr>
<tr>
<td>S[k][j]</td>
<td>1 if the jth request is already being processed by some subsystem</td>
</tr>
<tr>
<td>0 otherwise</td>
<td></td>
</tr>
</tbody>
</table>

(c) Actions

<table>
<thead>
<tr>
<th>Transition</th>
<th>Enabling predicate</th>
<th>Action</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival</td>
<td>at least one request is waiting</td>
<td>;</td>
<td>take the next request, say jth, from the waiting queue</td>
</tr>
<tr>
<td>Notify</td>
<td>S[k][j]=0</td>
<td>;</td>
<td>notify the start of processing</td>
</tr>
<tr>
<td>Processed</td>
<td>S[k][j]=1</td>
<td>S[k][j]:=1; WRITE (S[i][j]:=1) for i=1,2,...,n, i≠k; ;</td>
<td>discard the request</td>
</tr>
<tr>
<td>OPEN OK</td>
<td>C[k][j]=null</td>
<td>C[k][j]:=k;</td>
<td>the first OPEN of the jth request is for the kth subsystem</td>
</tr>
<tr>
<td>OPEN</td>
<td>C[k][j]=null</td>
<td>C[k][j]:=k;</td>
<td>another subsystem has already OPENed relation h</td>
</tr>
<tr>
<td>OK</td>
<td>C[k][j]=null</td>
<td>C[k][j]:=null;</td>
<td>the first OPEN for the ith relation</td>
</tr>
<tr>
<td>Good</td>
<td>C[k][j]=null</td>
<td>C[k][j]:=null;</td>
<td>another subsystem has already OPENed relation i</td>
</tr>
<tr>
<td>OPEN Failure</td>
<td>C[k][j]=null</td>
<td>C[k][j]:=null;</td>
<td>after the completion start a new cycle</td>
</tr>
</tbody>
</table>

Verification

The enabling predicates for OPEN OK and OPEN Good involve more than a single request. Because of this, deadlocks can occur between conflicting requests. Since these deadlocks can be avoided by a usual technique, it is assumed they will not be a problem. Then, it is only necessary to analyze the case for the single same request.

Let us denote by (t1,t2) the global system state, where t1 and t2 are two subsystem states. Possible transitions of the global system can be drawn as shown in Figure 4. Note that, since the subsystems are logically identical, the order of executions of OPEN's for the relations for the same request is the same in each subsystem. From Figure 4, we can prove the four properties of the system.

(a) At least one subsystem accepts a request.
This is verified because once the system makes a transition "Arrival," it always comes to a state involving a subsystem state (5).
(b) No more than one subsystem accepts a request.
This is verified because no (5,5) states are allowed and because once a state (5,x) is reached, then state
(y,5) can never be reached for the same request, and vice versa.
(c) No deadlocks occur
   Since no states are blocked, no deadlocks occur.
(d) System is alive
   This can be shown because, after the system reaches state (5,x) or (y,5) it always returns to the original state (1,1).

The above verification is limited to a special case, but an extension to a general case should be obvious and straightforward.

**Optimality**

It can be seen that the algorithm described in the previous section contains so called automatic load sharing. Several methods for automatic load sharing have been proposed. LeLann's two methods, for instance, are:

(a) Diffusion technique
(b) Circulating vector technique.

The above two methods and many others exchange information about the load of each subsystem. Based on this information, each subsystem determines whether it accepts a new request. The present method, under the all selection discipline, does not need any information exchange and can be proved to be optimal for load sharing in a statistical sense. The present scheme is free from any consistency problems caused by communication delays, because no information exchanges are required. Note that any scheme that exchanges information may have consistency problems, due to communication delays.

**Model and proof**

The present method, under the all selection discipline, is statistically equivalent to a single queue with multiple servers, because each request is placed in every queue. Any scheme that exchanges information corresponds to a system of separate-queues with some jockeying. Whatever jockeying is used, they cannot be better than a single queue as far as load sharing is concerned. This can be proved very easily. Some simple analysis for jockeying for a variety of strategies is made by Koenigsburg and a detailed analysis of two queues is made by Maekawa.

**PERFORMANCE ANALYSIS**

For performance analysis, the data base is simply considered to be a collection of pairs (attribute, value). They are put together to form records, domains and relations. In this section, the only concern is with their statistical characteristics. There is no concern over whether they are actually relations, domains or records.

The present data base system consists of a number of subsystems, as shown in Figure 2. Each subsystem has a
processor, main memory and secondary memory, as shown in Figure 5. In the main memory, there are two kinds of buffers; one is for data handled directly by its own processor and the other for accesses from other subsystems. The former buffer contains a working set of data. It is assumed that both main memory and secondary memory are divided into pages of equal size and that a paging scheme is employed. A model of the whole system is shown in Figure 6.

(1) There are $m$ users and $n$ subsystems connected by $n$ input/output channels

(2) Each user issues a request with rate $\lambda$.

(3) A request is first examined by its subsystem. It is discarded unless the subsystem decides to accept it. It is assumed that the initial examination time is governed by random variable $A$ and the probability of acceptance is denoted by $a$. Probability $a$ is dependent on the system state.

(4) The processing of a request is a cycle of an access to data, usually a tuple of a relation, and its processing. When data are not in the main memory, they are brought into the main memory from a secondary memory. Time to access data in the secondary memory is assumed governed by random variable $T$. When data are obtained in the main memory, processing the data is started. Its service time is assumed governed by random variable $S$. On the completion of the service, request processing completes with probability $q$ and repeats with probability $(1-q)$. Thus, the number of cycles per request is geometrically distributed with mean $1/q$.

(5) The missing page rate is denoted by $p$. When a page is not already in a main memory, the required page is brought in from its own subsystem’s secondary memory or from another subsystem. It is assumed that the probability that a page is brought in from another subsystem is $(1-r)$. If a page is in its own subsystem’s secondary memory, it takes only time $T$ to bring the page in. If it is not, however, a request for data acquisition and the obtained data must go through the intersubsystem bus, whose times are governed by random variables $U$ and $V$, respectively.

The major performance problems in this system are

(a) How much performance degradation is imposed in each subsystem due to this architecture.

(b) How good is load balancing.

(c) How much overhead due to this architecture is imposed.

The performance degradation is, to some extent, controlled by the acceptance criterion. The performance degradation of the system, excluding I/O channels, is mainly due to the following overhead.

(a) An unsuccessful attempt for access to a relation per subsystem for each request may be necessary.

(b) When a page is in another subsystem’s secondary memory, it must be brought in through the intersubsystem bus.

Overhead (a) is negligibly small, compared to the total processing necessary for a request. Thus, overhead (b) is the major cause for performance degradation. Instead of analyzing the whole system shown in Figure 6, the performance of each subsystem may be approximated by the model shown in Figure 7. The rationale behind this approximation is obvious. The number of subsystems affects probability $r$.
as well as transfer times \( U \) and \( V \). As a simple comparison, make a mean-value analysis.

(A) Single-subsystem case

In this case, \( r=1 \). Then, the mean service time for a request is

\[
E[\text{service time}] = E[\text{cycle time}]E[\text{number of cycles}]+E[A]
\]

\[
= (pE[T]+E[S])/q+E[A] \quad (2)
\]

(B) Multi-subsystem case

\[
E[\text{service time}]_m = E[\text{cycle time}]E[\text{number of cycles}]+E[A]
\]

\[
= (pE[T]+p(1-r)(E[U]+E[V])+E[S])/q+E[A] \quad (3)
\]

Since \( E[A] \) is small, compared to the total processing time, ignore \( E[A] \) and then take the ratio of the two for comparison purposes.

\[
d = \frac{E[\text{service time}]_m}{E[\text{service time}]_1}
\]

\[
= \frac{p+E[S]/E[T]}{p+p(1-r)(E[U]+E[V])/E[T]+E[S]/E[T]}
\]

Table I shows the values of \( d \) for various values of

\[
h = (1-r)(E[U]+E[V])/E[T]. \quad (4)
\]

It is assumed that \( E[U] \), is one-tenth of \( E[V] \), page size is 2 kilo-byte, \( E[T] \) is 30 ms and \( 1-r \) is 0.3. Then \( h \) and \( c \) are related by

\[
h = (1-r)(E[U]+E[V])/E[T]
\]

\[
= 0.022/c. \quad (5)
\]

Under normal operation conditions, the missing page rate, \( p \), should be less than 1/100. If it is assumed that \( h=0.1 \), then performance degradation is 2 to 6 percent. For \( h=0.1 \), 0.22 mega-byte/second per subsystem is needed. Thus, if a 1 mega-byte/sec bus is provided, then a system, made up of 4 subsystems, should be able to work with 2 to 6 percent performance degradation. When there is a fairly fast secondary memory and \( E[S]/E[T]=1/20 \) is justified, then, for \( h=0.7 \), or \( c=1 \) mega-byte/sec, performance degradation is only 10 percent with 30 subsystems. At any rate, it is observed that the smaller the missing page rate, and the faster the secondary memory, the less performance degradation is incurred and the larger a system can be constructed. Thus, a faster buffer memory between main and secondary memories would greatly improve system performance. A magnetic bubble or CCD memory would be a great help here.

Next, the acceptance criterion effect is analyzed. The two policies discussed before are compared.

(a) All selection policy
(b) Sufficient selection policy

Again, only a mean-value analysis is made. The assumptions are as follows:

(1) The subsystem reference frequency is described by a truncated geometric distribution with parameter \( 1-w \). Namely, a request \( R_i \) refers a subsystem \( S_1j \) with probability

\[
f(j) = \frac{w(1-w)^{j-1}}{1-(1-w)^r}, \quad 0<w<1. \quad (5)
\]

(2) Each subsystem is identified with the number \( k, 1 \leq k \leq n \). The \( k \)th subsystem is denoted by \( S_k \). The correspondence of \( S_1j \) to \( S_k \) is assumed to be random. Namely, \( S_k \) corresponds to \( S_0 \) with equal probability \( 1/n \).

(3) The request \( R_i \)'s processing time by subsystem \( S_0 \) is denoted by

\[
P_{m,r}. \quad (6)
\]

Since the system is symmetric and, if a page transfer time between subsystems is assumed independent of

<table>
<thead>
<tr>
<th>( h ) (mega-byte/sec)</th>
<th>( p=1/3 )</th>
<th>( p=1/10 )</th>
<th>( p=1/100 )</th>
<th>( p=1/1000 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.002</td>
<td>0.998</td>
<td>0.998</td>
<td>0.999</td>
<td>0.999</td>
</tr>
<tr>
<td>0.02</td>
<td>0.981</td>
<td>0.981</td>
<td>0.987</td>
<td>0.987</td>
</tr>
<tr>
<td>0.04</td>
<td>0.966</td>
<td>0.966</td>
<td>0.974</td>
<td>0.963</td>
</tr>
<tr>
<td>0.08</td>
<td>0.939</td>
<td>0.939</td>
<td>0.949</td>
<td>0.929</td>
</tr>
<tr>
<td>0.1</td>
<td>0.21</td>
<td>0.910</td>
<td>0.913</td>
<td>0.938</td>
</tr>
<tr>
<td>0.15</td>
<td>0.15</td>
<td>0.871</td>
<td>0.871</td>
<td>0.909</td>
</tr>
<tr>
<td>0.2</td>
<td>0.11</td>
<td>0.835</td>
<td>0.835</td>
<td>0.882</td>
</tr>
<tr>
<td>0.3</td>
<td>0.073</td>
<td>0.772</td>
<td>0.772</td>
<td>0.833</td>
</tr>
<tr>
<td>0.5</td>
<td>0.044</td>
<td>0.670</td>
<td>0.670</td>
<td>0.750</td>
</tr>
<tr>
<td>0.7</td>
<td>0.031</td>
<td>0.592</td>
<td>0.592</td>
<td>0.682</td>
</tr>
<tr>
<td>1.0</td>
<td>0.022</td>
<td>0.504</td>
<td>0.504</td>
<td>0.600</td>
</tr>
<tr>
<td>1.5</td>
<td>0.015</td>
<td>0.404</td>
<td>0.404</td>
<td>0.500</td>
</tr>
<tr>
<td>2.0</td>
<td>0.011</td>
<td>0.337</td>
<td>0.337</td>
<td>0.429</td>
</tr>
<tr>
<td>3.0</td>
<td>0.0073</td>
<td>0.253</td>
<td>0.253</td>
<td>0.333</td>
</tr>
<tr>
<td>5.0</td>
<td>0.0044</td>
<td>0.169</td>
<td>0.169</td>
<td>0.231</td>
</tr>
</tbody>
</table>
the subsystems chosen, Eq. (3) can be used as $P_{w,j}$ where $r$ is to be replaced with $f(j)$ of Eq. (3).

$$P_{w,j} = p[E[T] + p(1 - f(j))(E[U] + E[V]) + E[S]]/q + E[A]$$

The following observations are made.

(1) Under the all selection policy, each request is equally selected by a subsystem. Thus, the average service time would be

$$\frac{\sum_{j=1}^{n} P_{w,j}}{n}$$

For a system made up of a single subsystem, service time is $P_{1,j}$, which is described by Eq. (2). The system load would be very well balanced, because each request is assigned to the first subsystem that becomes empty. The assignment is a reversed FIFO.

(2) Under the sufficient selection policy, only $k$ subsystems become candidates for selections, where $1 \leq j \leq k$ and $k$ is determined by the policy. Variable $k$ itself is a random variable. The average service time would be

$$\frac{\sum_{j=1}^{k} P_{w,j}}{k}$$

Under this policy, the load would also be well balanced but there may be some processing time loss, because there are some chances that empty subsystems cannot take unprocessed requests because they are assigned to some other busy subsystems. However this loss is ignored for simplicity and the necessary intersubsystem bus capacity is calculated for $k=1, 2, 4$ and $6$, whose results are shown in Table III. It is assumed that the system consists of six subsystems. In Table II, the ratio

$$e = \frac{\sum_{j=1}^{k} (1 - f(j))}{k}$$

is shown. The above ratio $e$ is related to variable $h$ defined by Eq. (4) as

$$h = e(E[U] + E[V])/E[T]$$

From Table III it can be observed that, for $p=0.01$ and $E[S]/E[T]=1/200$, the system can work with 10 percent degradation with $h=0.15$ or $c=0.247$ for $j=2$ and $w=0.9$. Thus with an intersubsystem bus whose capacity is 1 mega-byte/sec, the system of four can be constructed. With a bus of 4 mega-byte/sec, a system of 16 can be built. If $E[S]/E[T]=1/20$ can be assumed, $h=0.900$ or $c=0.0529$ is sufficient. This allows 19 subsystems for a 2 mega-byte/sec bus. It should be noted that the above analysis is very simple. A more elaborate and precise analysis is clearly necessary and is now under way. Its results will be reported elsewhere. Relating to this, there are better subsystem selection policies. One of them is, for instance, to use the following criterion:

(a) Compute $w$, using Eq. (1)
(b) If $w$ is fairly high for a subsystem, say $w>0.7$, then any subsystem whose value of $w$ is not greater than 0.7 drops out.
(c) The smaller the value of $w$ is, the more subsystems can be assigned.
(d) When $w=1/n$, assign all the subsystems. Analysis of this policy would require a more elaborate analysis and is not included in this paper.

<table>
<thead>
<tr>
<th>$w$</th>
<th>$j=1$</th>
<th>$j=2$</th>
<th>$j=4$</th>
<th>$j=6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.9$</td>
<td>0.1</td>
<td>0.505</td>
<td>0.750</td>
<td>0.833</td>
</tr>
<tr>
<td>$0.7$</td>
<td>0.3</td>
<td>0.545</td>
<td>0.752</td>
<td>0.833</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$h$</th>
<th>$j=1$</th>
<th>$j=2$</th>
<th>$j=4$</th>
<th>$j=6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.002$</td>
<td>3.665</td>
<td>18.5</td>
<td>27.5</td>
<td>30.6</td>
</tr>
<tr>
<td>$0.02$</td>
<td>0.367</td>
<td>1.85</td>
<td>2.75</td>
<td>3.06</td>
</tr>
<tr>
<td>$0.04$</td>
<td>0.183</td>
<td>0.925</td>
<td>1.38</td>
<td>1.53</td>
</tr>
<tr>
<td>$0.06$</td>
<td>0.122</td>
<td>0.617</td>
<td>0.917</td>
<td>1.02</td>
</tr>
<tr>
<td>$0.08$</td>
<td>0.0916</td>
<td>0.463</td>
<td>0.688</td>
<td>0.764</td>
</tr>
<tr>
<td>$0.1$</td>
<td>0.0733</td>
<td>0.370</td>
<td>0.520</td>
<td>0.611</td>
</tr>
<tr>
<td>$0.15$</td>
<td>0.0489</td>
<td>0.247</td>
<td>0.367</td>
<td>0.407</td>
</tr>
<tr>
<td>$0.2$</td>
<td>0.0367</td>
<td>0.185</td>
<td>0.275</td>
<td>0.306</td>
</tr>
<tr>
<td>$0.3$</td>
<td>0.0244</td>
<td>0.123</td>
<td>0.183</td>
<td>0.204</td>
</tr>
<tr>
<td>$0.5$</td>
<td>0.0147</td>
<td>0.0740</td>
<td>0.100</td>
<td>0.122</td>
</tr>
<tr>
<td>$0.7$</td>
<td>0.0105</td>
<td>0.0298</td>
<td>0.07086</td>
<td>0.0873</td>
</tr>
<tr>
<td>$1.0$</td>
<td>0.00733</td>
<td>0.0370</td>
<td>0.0550</td>
<td>0.0611</td>
</tr>
<tr>
<td>$1.5$</td>
<td>0.00489</td>
<td>0.0427</td>
<td>0.0367</td>
<td>0.0407</td>
</tr>
<tr>
<td>$2.0$</td>
<td>0.00367</td>
<td>0.0175</td>
<td>0.0275</td>
<td>0.0306</td>
</tr>
<tr>
<td>$3.0$</td>
<td>0.00244</td>
<td>0.0123</td>
<td>0.0183</td>
<td>0.0204</td>
</tr>
<tr>
<td>$5.0$</td>
<td>0.00147</td>
<td>0.00740</td>
<td>0.0110</td>
<td>0.0122</td>
</tr>
</tbody>
</table>
CONCLUDING REMARKS

Extensibility is one of the important system properties of future systems. Among many extensibility aspects, the present work is concerned with extensibility in capacity. Extensibility, in this sense, is first defined and then a scheme to obtain it is proposed and described. The scheme is shown as an algorithm and then its correctness is verified. Extensibility contains automatic load sharing. Scheme optimality is also shown. Performance analysis shows that this scheme can be used to build a fairly large size extensible data base system without much performance degradation.

Extensibility in capacity means three things; physical extensibility, logical extensibility and minimal performance degradation. The key property of the scheme for logical extensibility is that each request is broadcast to each subsystem and is then tried by every subsystem. Eventually, however, one and only one subsystem completes the request. Each subsystem is autonomous and it itself determines when to drop out. In doing this, no exchanges of information about the load of each system are required. For this reason, no consistency problems occur. No system wide mutual exclusions are necessary in this respect, which makes the logic considerably simpler.

The system performance is analyzed using a simple mean-value analysis. It is observed, from this analysis, that a fairly large size data base system can be built, based on the proposed scheme. Necessary capacity of the intersubsystem bus can remain realistic. Any means of reducing missing page ratio would greatly help improve the performance of the system. Here, magnetic bubble or CCD memory can be a great help.

REFERENCES


ACKNOWLEDGMENTS

The authors would like to acknowledge Prof. Eric Manning of the University of Waterloo for his helpful comments and our colleagues, particularly Mr. Shigeru Oyanagi at Toshiba Research and Development Center, for their valuable discussions and various supports.