Effective Load Balancing in a Distributed Object-Support Operating System

Peter Dickman
University of Cambridge Computer Laboratory
Pembroke Street, Cambridge CB2 3QG UK

Abstract

An important mechanism in any distributed object-based system is a means of managing load imbalances. This extended abstract introduces the Bellerophon load balancer. The approach described provides dynamic system-level load balancing in a topology independent fashion that is appropriate for avoiding gross imbalances of load. A number of novel techniques are combined with a heuristic to form a hybrid local garbage detector and load balancer. Multiple load measures are regularly compared, by each node with a dynamically varying subset of the others; and clumps of objects, selected to avoid breaking cycles, are transferred to improve the load balancing.

1 Background

One possible approach to adaptive load balancing, in object-based systems, is to provide a migration mechanism and rely on distributed applications to perform their own load balancing. There are two major difficulties with this: that the application load balancers may be inadequate or omitted entirely; and that in a system executing a number of applications in parallel there will be a lack of coherence and poor overall balancing. This position paper introduces a distributed approach for automatically avoiding gross imbalances in a large distributed object-based system, with the flexibility of allowing application programmers to improve the load balancing if they so desire.

Although a complete system with all of the features described below has not been implemented, partial prototypes have been constructed [4]. As yet it has not been possible to fully evaluate the main heuristic described in §6 and further work is needed.

1.1 Object model

The object model assumed throughout this paper is deliberately simple, to ensure that the strategies outlined are as widely applicable as possible. An object encapsulates a (dynamically varying, possibly empty) set of references, and, potentially, other purely internal state. Each object is assumed to include a set of invocable methods. Messages are produced at objects and passed to another object, to which the originator must hold a reference. Messages trigger method invocation; methods being code fragments which manipulate the reference-set and internal state of their host object and which may generate additional messages. Objects with similar behaviour, (i.e. the same embedded methods) are viewed as belonging to the same class and only one instance of the methods for any class is needed on any one processor.

A procedure call, rather than message-passing, model can be similarly presented. It is important to note, if this is done, that the chain of stack frames associated with a given thread, will not itself be referenced other than from within that chain, and that all of the internal pointers go in one direction. These chains will therefore not introduce additional cycles of references into the system (cycles are critical to the operation of the clump detector described in §6).

1.2 Processor model

Heterogeneity amongst the processors which comprise a distributed system is, potentially, a significant issue. Two distinguishable forms of heterogeneity exist: architecture heterogeneity (different instruction sets etc); and resource heterogeneity (varying amounts of memory or dissimilar processor speeds). Resource heterogeneity is addressed in §3.2.

The simplest solution to the problems introduced by architectural heterogeneity is to use an intermediate language for the methods, which can then either be interpreted or compiled to native code when needed. The problems of object transfer then are, essentially trivially, solved by means of structure descriptor languages associated with the networks, with translation between local and network forms being handled during marshalling. It is assumed here that any object may be, in principle, migrated to any processor, but that both forms of heterogeneity of processors should be a factor in decisions about which objects to move where.

1.3 Network model

It is assumed that a reliable object migration mechanism is available (failure to migrate an object may occur, but this must be detectable). It is also assumed that the underlying message delivery systems are efficient. In other words, it is assumed that system load imbalances are due to the distribution of objects and the pattern of method invocations between objects, rather than problems with the underlying routing mechanism for messages. This is certainly valid if it is implicitly assumed that the underlying network is completely connected or that network load varies...
over time, but not between parts of the network at the same time. A single Ethernet would meet these requirements.

Note that it is not assumed that a broadcast mechanism is available. Many published mechanisms for load balancing rely on broadcasting of either load information [7] or bidding requests (in which other nodes are invited to bid for the excess load at some machine [12]). The mechanism as described here is built on top of a point-to-point message delivery subsystem and the potential overheads of broadcasts are avoided.

Given a non-homogeneous network architecture, the local network loading would become a parameter to the load comparison and clump determination phases. Whether such a parameter can be measured in a usable fashion, without also requiring that the network topology be known to the load balancer, is moot. This particular issue is not addressed further in this paper.

2 Introduction

Traditionally there have been two views of load balancing (as against load placement [9]): given a single large application, how best to distribute parts of the application to minimise overall execution time; and, given a multitude of small independent tasks, how best to utilise the resources in a distributed system so as to ensure that processors are idle for as little time as possible, when work remains queued elsewhere. Object-based load balancing fall into neither of the these categories. There are likely to be a number of (almost) independent applications executing in the system at any given time, potentially including several instances of the same application. Furthermore, the objects — which form the units of migration — will not be independent: an application instance will consist of a large number of inter-related objects some of which may be shared with other instances of the same application, perhaps even with other applications. In addition, there is a significant likelihood that some applications in such systems cannot be confined to a single processor; either because of scale, inherent distribution or the desire for fault-tolerance. It follows that the communications overheads of an application cannot be treated as independent of the distribution of objects.

Finally, any usable implementation will, of course, have to make some allowance for the cost of the migration of parts of the computation and for the monitoring and evaluation involved in the load balancing itself.

A load balancer essentially performs the following tasks (this enumeration is similar to, but differs from, that given in [16]):

- Load is measured locally at each node
- Comparisons of the load at different nodes are made
- Ways of rectifying imbalances are sought, involving the transfer of groups of objects from node to node
- Particular objects for transfer are selected
- The transfers are performed

Many studies have been made of some parts of this list. In particular, a number of proposals for appropriate load measurements have been published (e.g. [14]) and object migration has been implemented in various systems (e.g. [8, 6]). Several strategies for load comparison have also been developed. The work on load balancing in the Bellerophon project has concentrated on the topic of particular interest for object-based systems, namely mechanisms for selecting the objects to be transferred, and on the issues that arise in large systems.

The remainder of this paper briefly describes three aspects of the strategy for load balancing, at a given node, that are somewhat unusual:

- The use of multiple load measurements and non-linear scaling to accommodate heterogeneity, with comparison being performed at intervals which are random variations of an essentially regular pattern;
- The restriction of load comparison to a subset of the other nodes, called the buddy set, which is dynamically varying, and selected in a non-uniform random fashion with refusal;
- The determination of clumps of objects as candidates for transfers, based on: information on local cycles provided by the garbage collector; embedded counters which monitor activity; and the presence or absence of special references.

It should be noted that only the last is really specific to object systems.

3 Multiple load measures

A variety of measures of load can be envisaged in a distributed object-based system, as there are several scarce resources and a number of ways of measuring these resources. Examples of scarce resources include memory, space in object tables, and CPU cycles. Possible measurements for, say, memory include paging rates and the ratio of virtual to physical memory.

There are, however, two primary categories of measurement: those which are space-related and those which are time-related. Trivial examples of these include, respectively, the number of objects in a node and the number of messages received or generated per second on that node. Given two such measurements a two-dimensional space is produced within which the current load value can be identified.

A third category of measure, related to network utilisation, would also be of value in spatially non-homogeneous networks. That is, if the network load may vary, not just in time but also in space, some consideration of this may be of value. Packet-switched based mesh networks, for example, may exhibit high loads on some interconnection paths at the same time as low load on others. Whether there is a means of using this information independently of detailed knowledge of the network topology is unclear as yet. This
point is not considered further in this paper; hence the assumption of spatial homogeneity in the network model adopted.

### 3.1 Load measurement

It is common practice to use some form of composite or combined load measure, as in, for example, [2]. Consider a composite load measure, constructed by combining the current space and time loads with some function $C$. A strict order on these composite loads could then be used for load comparison. Let $t$ represent time and $s$ represent space; write $<_t$ and $<_s$ for the strict single-measure orders and $<$ for the total order on the composite. It is sensible to require that:

\[ t_t(s_t) \wedge t_s(s_t) \Rightarrow (t_s) < (s_t) < (t_t) \]

Now, any given processor will have associated with it some nominal memory capacity and processing capability. It is usually also possible to trade time for space by means of virtual memory mechanisms and so forth. This trade-off will permit larger versions of a system to be executed, but the performance will depend crucially on the pattern of message passing. The facilities at a given processor could, therefore, be modelled as a curve through the same 2-D space; indexed by a particular scalable collection of objects and pattern of messages.

Example: Consider a simple example, a processor in which $max(t) = 5$ and any space usage beyond 10 causes a loss of available $t$; that is the amount of real processing which is possible cannot exceed 5 units per time interval. Assume that the composite measure satisfies $C(1,24) < C(3,15) < C(4,15) < C(1,25)$. Assume also that four distinct systems are offered to the processor, each of which can be divided into the above four parts; that is, over an interval each requires some amount of processor time and space. Let the $(1,24)$ & $(3,15)$ systems be both such that they would induce the processor to have a time-space trade-off given by $50s + t^4 = 1125$ [(for 10 $< s$, $t < 5$), whilst for the other two this is $50s + 26t = 120$]. The processor can then execute either of the $(1,25)$ or $(3,15)$ systems without being over-loaded, however this is not true of the other two. Unfortunately, the composite measure suggests that the system should prefer the $(1,24)$ system.

An obvious solution is to require that the function $C$ reflect the processor trade-offs, but this will not work, unfortunately. Four new systems, much as above but with the alternate time-space trade-off, could also be presented; any $C$ which correctly selects in one case will then incorrectly select in the other. Nor is it adequate to select the function $C$ based on knowledge of the current load levels alone. The original four systems executing on different processors (with different paging algorithm, say) might exhibit the opposite time-space trade-offs.

This problem can be generalised to any case in which at least two processor trade-off curves cross, and applies to any reasonable composite function $C$. The use of strict orders is also not fundamental to the problem formulation.

The conclusion is that load measures should not be combined into a single value. Thus each processor should assess its load as a value in a multi-dimensional space, and pass a vector of values when comparing loads. It would then follow that, for example, one processor could indicate a preference for space-consuming but low activity objects, or vice versa.

### 3.2 Load comparison

There remains a minor difficulty with the comparison of loads if there is resource heterogeneity. The absolute value could be passed, in which case the recipient has no indication of whether the originator is over or underloaded in that measure or by how much. Alternatively a difference between that absolute measure and some nominal limit could be passed; but then there is no indication of the actual system size. Two systems could each by overfull by the same amount, whilst in one this represents running at twice the nominal maximum, the other being only a few percent above the maximum.

Using relative measures (as in [15]) may provide a useful compromise, but then forces the free or excess load to be distributed proportionately. Whilst it is a reasonable approximation to suggest that changes in the space measure will occur in proportion to the current value (which assumes that objects are homogeneous in their child creation and lifetime characteristics) this does not account for the possibility of rapid introduction of many new objects into a node. Similarly, assuming that activity levels remain constant again implies object homogeneity and ignores the possibility that processor intensive activities may occur. Such inflationary hot-spot behaviour arises when an application is initiated on a processor and a better load balance can be achieved if this is explicitly considered.

One possible approach is to assume that space and time loads will change in two ways, one proportionate to the current load levels, the other which is either a constant for a given processor or a function of the nominal limits of the processor (rather than its current load). Combining these results in a non-proportionate distribution of the current overload or free capacity. To perform load balancing in such a system requires that (at least) two parameters, rather than one, are passed for each load measure when performing comparisons. This idea, of explicitly considering future changes in load balance, follows directly from [3].

Example: A simple technique, useful when the performance heterogeneity of system components is not extreme, is to assume that if the nominal capacity is $L$ and current load is $V$ then the future load will be $\alpha V + \beta$, with $\alpha$ and $\beta$ fixed across all processors. This leads to the following result: if another node has limit $L'$ and load $V'$, the net transfer to that node should be such as to leave behind: $[\alpha L'(V + V') + \beta (L - L')] / (\alpha L + L')$. Selecting a good strategy depends on factors which, in the absence of data from real systems and in the light of the variability of applications, cannot be predicted in advance. It follows that an adaptive system is required (such as, for example, a learning automata). This then compares changes in actual load against predicted values and, for example, dynami-
functionally adjusts the parameters \( \lambda \) and \( \gamma \) in a prediction function \( \lambda V + \gamma \); say, where \( V \) was the current value of the measure. This approach, whilst requiring that additional values be passed during load comparison, offers some hope that the resulting decisions will more accurately reflect the requirements of the system. Obviously, estimating two parameters from a single measure is somewhat awkward. It is assumed that rapid changes are due to the initiation of new activity, whilst gradual changes reflect shifts in the extant objects. Thus \( \lambda \) is taken to change over rather longer periods than \( \gamma \).

4 Demand-initiated load balancing

There are various strategies that can be used for determining when the load balancer is to execute. It may be continually active, execute at fixed intervals or at an interval determined as part of the preceding execution, or it may be triggered by changes in load. Clearly the load estimation function used above must depend on the the initiation strategy, since the required 'interval' for the prediction is influenced.

The clump detector design renders it infeasible to continually adjust the load balance, so this possibility is not considered further. Using a measure-triggered load balancer has significant advantages in ensuring that load balancing occurs when likely to be needed, but is also somewhat more likely to lead to the problems with harmonics in the loading. A sine-wave load, over the bulk of the system, may trigger load balancing near the peaks, leading (potentially) to incorrect predictions about the likely future load and some risk of somewhat inappropriate balances being sought. However, if measure-triggered balancing is not used it is essential that the intervals between balances be sufficiently short that great inequities cannot develop in the interim.

The Bellerophon approach is to execute the load balancer at slightly-randomised, but essentially fixed, intervals and to only proceed with the load comparison stage and, if necessary, the clump detection and exchange of load, if this appears worthwhile. That is, load balancing occurs at some subset of the approximately regular intervals. Note that the intervals are not absolutely regular, to reduce the probability of phase locking between the balancer and an application. Also, the measures are, where appropriate, not instantaneous values, since these can be misleading. Instead rolling or weighted averages are maintained over the interval between balances.

One consequence of this is that the prediction mechanism can be based on fixed intervals, with proportional scaling applied to correct for the random fluctuations. If triggering had been measure-based then the two parameters above would effectively be reduced to one, representing the ratio between the two causes of change in the loading. The preferred mechanism has two real parameters.

4.1 Thresholding the measures

Since the purpose of the load balancer is to avoid gross imbalances, rather than to achieve a near-optimal distribution, a thresholding approach is used

[5]. If the local load does not exceed certain bounds then no attempt to compare loads is taken. Similarly, when comparing loads, if the differences encountered are small then no attempt to compare against a load balancer threshold is made. The purpose of the load balancer is to avoid gross imbalances, rather than to achieve a near-optimal distribution. A thresholding approach is used.

Considerable experimentation is still required to determine how these thresholds should be set. For the moment, they are (respectively): a fixed proportion of the nominal limit; and a non-negligible but small fraction of the nominal capacity of the two machines comparing load.

Alternative approaches, such as gradient-based techniques have been considered [10]; however it was felt that the dynamic variation of the buddy sets (see below) should initially be investigated separately from the global gradients that are generated in more complex load balancers. Later work will, it is hoped, evaluate the interaction of these mechanisms.

5 Non-uniform random buddy sets with refusal

Buddy-Sets are a useful technique in load-balancing a large system. In such systems, rather than each node comparing its loading with all others, or with some mean load which must, therefore, be calculated and maintained, a small subset of the nodes are designated as buddies of a given node. Comparisons of load, and the transfer of objects, then only occur between buddy nodes. It is, of course, desirable that the buddy sets form a connected cover over the graph of nodes and that the diameter of this cover be low (i.e. starting from any node, the others can be reached in very few steps, where a step involves moving to a buddy of the current node).

Buddy sets can be allocated in advance, but in a system in which the executing applications change dynamically, or more especially in which nodes enter and leave the system over time, this may not be appropriate. Simulation and analytical studies led to the following features in the buddy set system in the Bellerophon project:

- Buddies are randomly selected, locally at each node. This generates a low-diameter connected cover for the network as a whole, with extremely high probability. Furthermore, random slow variation of the buddy sets has advantages: in avoiding oscillations in object distribution; and in adapting the buddy set (see below) to match the current distribution of objects.

To avoid problems with clump dispersion, part of the buddy set selection is biased towards selecting nodes to which the given node holds many references. However, were all buddies to be selected in this way, there would be a danger of generating a disconnected cover; therefore, only part of the selection is made with this bias, the rest is made in a uniform manner.

Finally, selection of a new buddy involves an exchange of messages with that node. This provides some confidence that the buddy is currently functioning. To avoid problems caused by one node being selected by excessively many others (which has a very low probability of occurring) a node may refuse any such selection. Refusal has significant advantages in
simplifying the system model. A print engine, for example, can refuse to participate in load balancing whilst still accepting objects for printing. Thus the buddy set generator need not distinguish references to nodes which will accept transferred load from those which will not. However each node, instead, has some limit on the maximum number of other nodes which may have it selected as a buddy at any given time.

During load balancing, each node uses both the buddies it selected, and those nodes which selected it, as partners. Load is only ever transferred away from a node, since the mechanism for selecting objects to transfer is costly. Were it possible to 'suck' as well as 'blow', the clump detector would be invoked at all partners of a load balancing node, with considerable cost to the system as a whole.

The use of randomising mechanisms bears some similarities to the randomised distribution of load information in [1]; but is not nearly as dynamic (since the random buddy sets change only slowly).

6 Clump detection

The most important requirement of object-based load-balancing is that appropriate clumps of objects be found; these being moved from one node to another in order to balance the load. In doing so problems must not be caused for the garbage collector nor should the number of object-to-object messages which must cross the network be vastly increased. That is, locality of reference must not be sacrificed in order to notionally improve the load balancing, nor must local cycles be unnecessarily converted into distributed cycles.

Most algorithms for cluster detection are intended for statistical analysis of complex datasets. Indeed, throughout the history of these algorithms the aim has been to find relatively cheap mechanisms for determining high-quality patterns in data. The problem faced here is that a relatively large and complex graph structure is provided from which appropriate candidate clumps must be generated as efficiently as possible. The load balancer must provide an overall advantage and will certainly not do so if extensive computation is involved in selecting the objects to move. Indeed, considerable effort is often expended to ensure that a load balancer is fast and efficient [13]. However, a poor choice of clump will lead to increased network traffic and may also lead to local cycles of references becoming distributed cycles, which may increase the cost of subsequent garbage collections.

The approach taken was, therefore, to use structural information from the garbage collector as an input to a heuristic. This generates a small number of clumps from the peripheral regions of the graph, which match reasonably closely the optimal requirements established during load comparison. The best of these is then migrated away to the relevant node.

6.1 Integration with a garbage detector

In fact generating the information required by the clump detector involves activities sufficiently close to those of a garbage detector that there was little point in not combining the two functions. Given also that the structural information so generated rapidly ages there is little point in retaining it between rebalances. The combined mechanism builds descriptors for cyclic structures that are encountered during a mark phase (these being combined together if two cycles intersect). In addition, all objects are initially viewed as belonging to single-element clumps; these are merged whenever objects lie in the same cyclic structure.

6.2 Activity monitoring

References and objects have attached small efficient counters which indicate the number of messages passed (these are non-atomically incremented on message passing). Where a tight binding is indicated (by a high message count in a reference compared with the total received at the referenced object) the two containing clumps are combined into a single one. Thus at the end of the garbage detection both descriptors of cyclic structures and cycle descriptors of tightly connected objects have been constructed. These will further have been labelled with counts of the number of pointers to remote objects they contain, in particular a separate count is maintained for each potential recipient of the clump and an additional one for all others. An estimate of the increased network message traffic which would follow the migration of this clump to one of the partners could then be calculated.

6.3 Colocators and contralocators

For efficiency reasons (and others not relevant to this paper) the Bellerophon architecture supports colocators and contralocators. These are references which have additional properties. Two mutually colocated objects (ie objects which contain colocators to each other) which are on the same machine will remain together even if migrated; and two mutually contralocated objects on different machines will never be migrated onto the same machine.

Clearly colocators must lead to the merging of clumps and combining of cycle descriptors. Note that this is done even if just a single colocator is used, as there is no guarantee that some such structuring may not prove valuable. Contralocators are noted as they are used to eliminate possible candidate clumps from further consideration. Contralocators referencing another object in the same machine are, in addition, ignored when considering cycles. That is an apparent cycle which contains a contralocator is deliberately not treated as a cycle.

The colocators and contralocators are the principle mechanism through which the application programmer can direct the work of the load balancer.

6.4 Constructing the candidate clumps

Rather than dividing the whole local object graph into components, the bulk of it is fairly rapidly discarded and only a limited set of objects is used further. This has the advantage of allowing the relatively small amount of CPU power that is available to be focussed on determining good candidates. The initial passes to generate cycle descriptors and clumps of objects are made, as described above, with the object/reference graph being traversed. Co- and con-
Candidate clumps are then constructed by associating each of these initial seeds with values indicating the level of internal activity (messages passed between members of the clump) and the external activity (messages passed into or out of the clump), using the values of the counters again. The activity values can be viewed as a form of binding energy with highly cohesive, relatively separate, object clumps being the usual candidates for transfer. The cohesiveness of a clump (the ratio of the internal and external activity) and its size are the primary determinants of a clumps usefulness for load balancing. In addition the weight of the references (if any) that indicate the potential recipient and the absence of contralocators are also factors.

Assuming that there are several potential recipients, each with indicated preferences, the clumps can be compared with these requirements in turn. A limited number of close matches are noted, as are a selection of those clumps which contain references to remote objects but no relevant contralocators. If no close match is found then these selected clumps are joined together to generate better candidates. (If all clumps are too large then the process is repeated with no cycle merging due to the counter values. Clumps then do not enclose cycles and cycles are permitted to split.)

It should be noted that several candidates are finally selected, unless the preferred clump contains no contralocators. This is necessary because the target of a contralocator is itself an object which may just have been migrated to the proposed recipient node. Fall-back strategies for balancing the load are therefore required in case the first attempt proves impossible to complete.

### 6.5 Datastructures

Two styles of datastructure have been considered in this work. The conceptually simpler initial approach involved descriptors; for simplicity only cycles and cycle merging are described here. Each object would contain a pointer to a descriptor, as (potentially) would each descriptor. When a cycle was detected, the objects in the cycle would be scanned to find any descriptors associated with them. If no object indicated a descriptor then one would be generated. Otherwise the descriptor pointers would be followed to locate a descriptor which contained no descriptor pointer. All objects in the cycle that contained no pointer would be set to indicate this one, as would all other descriptors reachable from the objects in the cycle either directly or indirectly. Clearly if the graph of objects and references is complex, with many interconnected cyclic structures, this approach is unwieldy.

The second and preferred approach consisted of each object containing a single pointer (and a few bits). When constructing cycles these are linked to form cyclic lists (initially they are all self-references and so form trivial cycles). On merging two such cycles a crossing piece consisting of two pointers is inserted into both lists. This generates a complex structure in a fashion which protects from disconnection should a structure be merged with itself. It therefore allows the merging to be done correctly with constant cost. The only major drawback is the space requirement of the crossing pieces.

Having constructed this latter datastructure it is not difficult to traverse a connected component of the merged lists. An Euler-cycle generating algorithm, based on randomly constructing a path and then backtracking and extending where possible, can be used to efficiently walk the structure in \( O(E) \) time (where \( E \) is the number of edges in the structure, that is 1 per object and 2 per crossing piece). In other words, at most three times the number of references associated with the objects in the component). This algorithm is guaranteed to work correctly, since the structure generated is a directed graph in which the in-degree and out-degree at each vertex are equal.

### 6.6 Architecture heterogeneity

Real object-based distributed systems often include a variety of different machine architectures. One simple solution to the problems this induces at the language level is to provide an interpretable intermediate code and virtual machine that can be supported over all of the disparate hardware architectures. To improve performance it may also be possible to compile specific methods to local machine code, either when they are repeatedly used, or on receipt of the object or class.

These issues also have significance for the load balancer. If transfer of objects triggers recompilation of methods it is clearly desirable to limit migration, as far as is possible, to minimise this effect. There are several conceivable ways in which this may be reflected in the load balancer. In [11], for example, it is suggested that a different gradient surface may be maintained for each processor type.

Approaches which consider the presence or absence of a particular class on a particular remote machine, as part of the load balancing, may be superficially attractive. Unless there is an extremely simple mechanism for doing this, which does not adversely affect the performance of the balancer, such approaches will prove unwieldy. No such mechanisms have been considered in this work. The Bellerophon balancer supports just one mechanism specifically aimed at the issue of heterogeneity, which is not to suggest that others may not also prove valuable.

Each machine architecture is uniquely identified and when buddy sets are generated these tokens are exchanged. It follows that each machine can easily determine which of its buddies have the same architecture. If a load imbalance can be corrected by transferring load to either or both of a pair, one with matching architecture, one with a different one, then load will be preferentially transferred to the similar architecture. It is important, however, that this not be a strict division, otherwise very different loading levels could occur on the different architectures in a single system.

The approach taken is to order the potential recipients of load in a biased fashion. Clearly, given the use of multiple measures, producing a total order is
not very helpful. Instead the preferred recipients are randomly ordered, and then pairs are exchanged. The probability that a pair of entries are switched is determined by combining the associated imbalances (as proportions of the nominal local capabilities) and biasing in the event that only one of the pair has matching architecture.

This probabilistic bubble-sort like approach leads to a list of recipients against which the clamp descriptors can now be compared, as described above. Since a candidate clamp will be associated with the first good match, this provides a bias in favour of the recipients placed early in the list.

7 Summary

The heuristic is fairly complex and, although the initial stages are based on the mark and sweep phases of a garbage collector, several additional passes through part of the data are required. However, after the initial passes, which merely involve generating several straightforward datastructures, and a pass through the candidate clumps generated at that stage, only the best candidate clumps are considered further. Since there are relatively few such clumps the total cost of the combined local garbage detector and load balancer is of the same order as that of the garbage detector alone, albeit noticeably higher.

An approach to load balancing has been investigated and some issues and novel features have been presented. Since object migration is supported, and the load measures used by the system can be made available to the application, it is perfectly possible for an application-level load balancer to be produced which works in conjunction with this. Furthermore, the application programmer can use the colocators and contralocators mentioned above to directly influence the system load balancer.

The progress made so far strongly indicates that a distributed object-support operating system can indeed provide low-grade load balancing facilities in a scalable fashion, without excessive cost, and can thus avoid gross imbalances occurring in the system.

References

[1] A. Barak & A. Shiloh A Distributed Load-Balancing Policy for a Multi-Computer Software Practice and Experience Vol. 15 #3 pp 901–903

This work was partially funded by a UK Science and Engineering Research Council studentship.