USING THE ES-KIT AS A TESTBED FOR LOAD-BALANCING EXPERIMENTS

V. Karamcheti and B. W. Wah

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Approved for Public Release. Distribution Unlimited.
Using the ES-Kit as a testbed for Load-balancing Experiments

Karamcheti, V and Wah, B.W.

A brief discussion about the ES-Kit runtime system is in order here to explain the terminology in the rest of the report. The ES-Kit software platform provides an environment for the execution of applications written as a collection of interacting C++ objects. The future construct is used to introduce concurrency in the applications. All application classes are derived from two base classes local_base and remote_base with the "(...)" operator of the latter being overloaded to provide for remote-method calls. An invoke message is despatched to the node on which the desired object is located. If the invoking object requires a reply, the invoked method replies with a response message. In addition to these, the environment also supports system and construct messages (used for the creation of new objects). When an object is instantiated on a node, the object-code of the class is loaded (if not already present) followed by the initialization of the data area of the object according to the arguments of the constructor. The execution of the application objects is controlled by the kernel on each node. The kernel picks up a message from the post.office structure (FIFO) and...
then switches to the context of the object by allocating a stack for the object (if none exists) and invoking the appropriate method. Execution returns to the kernel context whenever the object either blocks or completes processing all invoke messages for its methods. Arbitrary migration support in such a system is quite difficult to provide and hence the report looks at a scheme that handles the migration of a subset of object-states.
Using the ES-Kit as a testbed for Load-balancing Experiments

Vijay Karamcheti and Benjamin W. Wah

This report looks at the additions and modifications that need to be made to the ES-Kit software platform for it to be used as a testbed for conducting load-balancing experiments. The experimental testbed should provide

- support for collection of information from various sources in the system (user-supplied, kernel-statistics, program-extracted, etc.) and the use of these for the derivation of load measures at the different nodes comprising the system.

- facilities for reducing this imbalance in the form of intelligent placement of newly created work, and the migration of existing work units from a heavily loaded node to one less loaded. The argument in favor of requiring the system to support migration in addition to intelligent placement of objects is that the former compensates for the uncertainty in the decisions regarding the latter.

A majority of the modifications discussed in the report are needed for migration support.

A brief discussion about the ES-Kit runtime system is in order here to explain the terminology in the rest of the report. The ES-Kit software platform provides an environment for the execution of applications written as a collection of interacting C++ objects. The future construct is used to introduce concurrency in the applications. All application classes are derived from two base classes local_base and remote_base with the "→ ()" operator of the latter being overloaded to provide for remote-method calls. An invoke message is despatched to the node on which the desired object is located. If the invoking object requires a reply, the invoked method replies with a response message. In addition to these, the environment also supports system and construct messages (used for the creation of new objects). When an object is instantiated on a node, the object-code of the class is loaded (if not already present) followed by the initialization of the data area of the object according to the arguments of the constructor. The execution of the application objects is controlled by the kernel on each node. The kernel picks up a message from the post_office structure (FIFO) and then switches to the context of the object by allocating a stack for the object (if none exists) and invoking the appropriate method. Execution returns to the kernel context whenever the object either blocks or completes processing all invoke messages for its methods. Arbitrary migration support in such a system is quite difficult to provide and hence the report looks at a scheme that handles the migration of a subset of object-states.

In order to motivate the changes that need to be made, the first half of the report deals with the issues of load-balancing of object-oriented programs without bringing in details about how
a particular execution feature is implemented. Section 1 addresses the collection of information which can be used to make load-balancing decisions. The computation of load estimates at the nodes on the basis of this information is looked at in Section 2. Section 3 contains a discussion about placement and migration of objects. Sections 4 and 5 provide a listing of the changes required in the ES-Kit software platform.

1 Collection of Information

1.1 Sources of Information

Information for determining the load at a particular node in the system can be obtained from various sources. A listing of the sources and the particulars of typical information that can be obtained from them is presented below:

System Level

- system configuration.
- communication channel capacities: speed, bandwidth, traffic, etc.

Node Level

- node characteristics:
  - CPU architecture
  - memory capacity, cache size(s)
  - communication-port characteristics, etc.
- node statistics:
  - number of processes, open file-descriptors
  - amount of CPU/memory utilization, amount of secondary-memory accesses, etc.

Application Level

- application characteristics:
  - I/O-bound vs. compute-bound
  - number of constituent classes, class-hierarchy, object-code characteristics (size), etc.
- application statistics:
  - number of objects (also compared with number of objects of other applications: Background Loading)
  - number of remote-method requests, length of the invoke/response queue (of messages): proximity between object-pairs
  - locations of managing objects (librarian, etc.), object-code location(s) for a particular class, etc.

Class Level
• class characteristics:
  - number of data objects and friend objects
  - size (of an instance of a class)
  - behavior of simple (well-defined) methods, etc.

• class statistics:
  - number of instances, average lifetime of an instance
  - average intra-remote-call execution times and memory requests of methods, constructor overhead (in terms of memory requirements)
  - average number of times a particular method is invoked (class-level popularity factor), etc.

Instance Level

• instance statistics:
  - number of invoke/response messages for a particular instance, queue lengths (can be helpful in deciding whether or not intra-object concurrency is called for)
  - number of objects created, memory asked for
  - estimates of residual execution times (before the next remote-method invocation) for active instances, average intra-remote-call execution times
  - list (number) of objects which access a particular method (instance-level popularity factor), etc.

1.2 Methods of Collection

This subsection describes how each of the information entities described above are obtainable from the system. The implementation details in the case of the ES-Kit can be found in Sections 4 & 5.

System Level

• the system configuration, usually supplied beforehand, is quasi-static (no changes happen within short periods). any changes are broadcast to all nodes of the system.
• the channel capacities are also known apriori. The traffic is a runtime factor that is measured by monitoring equipment which forms part of the node hardware.

Node Level

• node characteristics are either provided apriori or are obtained using the monitoring hardware.
• all the node statistics are maintained by the operating system and extracting them is a matter of making the appropriate kernel calls. For example, the number of page-faults quantifies the secondary-memory accesses.

Application Level
- the compiler supplies most of the application characteristics. A proper heuristic is required to classify a program as either I/O- or compute-bound.

- the application statistics are easily obtained by invoking probe-methods which are defined for classes encapsulating the tables that keep track of the required parameters. This assumes that the kernel calls that create and manage applications also manage the appropriate tables (this needs to be ensured as part of the modifications). The characterization of the background load (as seen by a particular application) is done using techniques similar to those developed by Pankaj Mehra.

Class Level

- the number of statically-defined data and friend objects are obtained by compile-time analysis, as is the size of an instance of the class. Determining the number of dynamically-created data objects will be discussed below. The execution-time estimates of well-defined methods (whose behavior can be completely predicted) are obtained by compile-time analysis of the program.

- the number of class instances is obtained by a probe-method into the class encapsulating the instance table. All other class statistics are averages (estimates) of quantities defined at the instance level and are computed as running averages of the same. The frequency of update of the running average is a parameter for study and will typically be smaller than the update frequency for each of the contributors of the average.

Instance Level

- the number of received/processed/outstanding messages are all obtained by probe-methods, as also is the number of dynamic object creations and the statistics of memory usage. The tables are updated as and when the corresponding requests are made. The list of objects that access a particular method is maintained by the instance itself (actually, by the base class of the instance) by extracting information about the invoking object from the invoke message.

Among the listed statistics it is the most difficult to obtain residual-time estimates of method executions. Depending on whether a particular instance is active (currently in control of the CPU), blocked (in the midst of a method execution), or waiting (may have an invoke message but has not started processing it yet), the estimation is done differently:

- for the active and blocked instances the estimates are exported by the instance on the execution of the annotated method code at the boundaries of every intra-remote-call block (the section of code between two remote-calls). The exported values can either be fixed or depend on the run-time environment in which the method is executing. The issue of what functions to express these values with is rather involved and will be left for a future discussion.

- the residual-time estimates for waiting objects is done by using the moving averages of actual intra-remote-call execution times of either the same instance executing the same method, or of all the instances that executed the same method, or of all possible method executions by an instance of the same class. The actual execution times are measured as the difference of the timestamps of the return event from one remote-call and the send event for the next remote-call. The actual execution time(s) can also be used for modifying the function used in the annotated code for estimating the same.
2 Using the Information

The main issues addressed are the computation of load estimates and their exchange among the nodes of the system.

2.1 Amount of Information

Since obviously it is unreasonable for all the information (in its raw form) to be used to make decisions at all levels in the hierarchy, decisions need to be made about what exactly acts as an input for load estimates at each level. An additional constraint is the amount of overhead allowable for the purposes of information collection.

The approach that shall be taken in the rest of the section is that the information gathered at a lower level in the hierarchy is abstracted into a smaller number of indices such that there is a compression in the quantity of information without losing out on most of the contents.

At the instance level most of the information will be used in raw form as inputs for the computation of load averages. The instance level exports the averages of all these events to the next higher level. A weighted average is computed based on the importance attached to the various sources of information. For example, the average execution time for any method of a particular class can be weighted by the number of objects that reference the methods in the class.

The load factor due to contributions from the objects of a particular class should reflect both the external view (number of outstanding messages for the objects of the class) and the internal view (estimated execution time of a method of the class). A similar condition must be satisfied in case of the contribution of the classes of a particular application to the load average.

The load-balancing strategy influences the choice of the load factor at a node to a large extent. For example, if the load-balancing strategy is trying to favor the earlier completion of a particular application, the load averages must reflect the background loading characteristics and be able to adapt to it.

The use of internal information about the objects of an application (estimates of residual-execution times and the characteristics of the objects) in computations of the load-estimate is advantageous in the following ways:

- Firstly, the accuracy of such information is more than that obtained by estimations using the external view alone. Thus, better information is available regarding the objects' behavior than got by looking at some kind of averages of past events. This is because in the kind of applications being targeted, the constituent objects have a rather unpredictable behavior pattern. This (more accurate estimates) is a tremendous advantage when we are dealing with objects which execute for sufficiently varying periods. On the basis of what the currently executing object has experienced about its behavior, it is possible to manage the other tasks in the system to maximum benefit (this assumes that the migration of objects pre-empts the execution of the current object).

- Further, since most of the information is exported from within the program itself, and the derivation of the functions that compute these values has been done predominantly at compile-time, such a system would lend itself to real-time operation. This is as opposed to observing
the behavior of an executing method from outside and trying to derive the function characterizing the behavior, or estimating how much time the next execution would last.

2.2 Computing Load Estimates

At the time of writing this report not much work has been done on deciding which functions to use as load estimators. The load at a node should be some combination (appropriately weighted) of the information that is obtained from both the *internal* and *external* views, i.e. it should take into account the behavior statistics of the objects on the node as well as the node statistics (number of processes, memory utilization, number of disk accesses), and some quantification of the background loading characteristics.

As an example, consider the computation of a load estimate with contributions only from the *internal view* (behavior) of objects executing on the node under the following assumptions:

- all objects executing on the node belong to the same application and are the only entities executing on the node (an assumption is made regarding the lack of daemons or any such background loading).
- the node is assumed to have ample resources of all types except the CPU (this assumption ensures that we do not need to worry about resource constraints regarding memory or disk accesses, etc.)
- the scheduling strategy among the resident active objects is FIFO with a new object getting control of the CPU only after the previous occupant blocks, either due to waiting for an external result or due to the lack of any new work.

then the load at a node can be described as

$$\text{Expected waiting time for a new arrival (active object) before it can get CPU control.}$$

and can be computed as the sum of the estimated intra-remote-call execution times of the active objects in the scheduling queue.

2.3 Exchanging Load Estimates

Every node in the system needs to be aware of the loads on a subset of the other nodes in the system (in its neighborhood). This is to facilitate imbalance-correcting decisions, like deciding which node to create a new object on, and to which node to migrate an existing object so as to improve its chances of executing at an earlier time.

The frequency of load-estimate exchanges is dependent on the overhead involved in each exchange (due to the kernel using CPU cycles and due to the increased message traffic). However, the frequency should not be small enough so as to cause out-of-date load-estimates to be present at the neighboring nodes.

The following issues need to be resolved for any load-balancing strategy:
• the amount of information to be exchanged: options vary from this being just be a single quantity that indicates the load at the node, to a set of values which give the utilizations of various node resources.

• the exchange-neighborhood.

• the frequency of the information transfer.

• triggering of transfer: a periodic transfer of information puts a bound on how much out-of-date the load information can be. Transferring load estimates only when the load at a node changes from the previously exported value by some thresholded quantity reduces the overall system traffic resulting from the exchanges.

• mechanism of information transfer: piggybacking the new load information to invoke/response messages in the system saves bandwidth but increases the message-processing time.

3 Decreasing Load-Imbalance

There are three broad strategies that can be adopted to alleviate the problem of an imbalance in the load.

3.1 Intelligent Placement

Placement refers to the association of a subtask with a group of processing resources (cluster, node, etc.). Placement can be talked of at two levels.

Application Level

New applications submitted at any of the front-ends to the computing system are allocated to the appropriate subsystem to be executed. This means that the objects of the application execute (their methods) on the nodes forming the subsystem.

This is of relevance in both heterogenous and homogeneous distributed systems. In the former, the intelligent placement scheme would try to match the resource requirements of the application objects with the capabilities of the different subsystems (for example, an image-processing application would be the responsibility of a subsystem that has digital signal processing hardware), while in the latter case, the placement would try to match communication patterns between the constituent objects.

Placement at the application level is not being addressed currently. To start with, only single applications running on a dedicated system will be considered.

Instance Level

Whenever a request for creation of an object is encountered at run-time (even when the program is being loaded), place the object on a node which minimizes the maximum time any method of the object would have to wait before executing after being invoked or after receiving a reply.
The decision about which node to place the object is dependent on the load estimates at the nodes and in addition will be influenced by the following information supplied either by the object or computed separately at the node:

- the number of new objects that the object is going to create. In the cases where this information cannot be provided by the object, it is estimated on the basis of the observed behavior of other instances of the class.
- the location of the invoking object.
- the location of the object code for the friend classes (this becomes important only if the object code for friend classes is not loaded whenever the code for a class is loaded), etc.

3.2 Migration of Entities

Migration refers to the moving of the site of execution of an application or an object from its current site (node) to a less-loaded one. Migration need not always be from heavily-loaded sites to less-loaded sites: objects can also be moved to heavily-loaded sites if it is expected that the move will eventually result in the sites having more balanced loading.

Migration can also be talked about at two levels.

Application Level

On observing severe imbalance in load among the subsystems in a computing system, a decision is made to move an entire application across subsystems. Such a move would involve the system providing support for at least one of the following options:

- checkpoining, which means that the states of the objects (local variables, variables on stack, etc) should be recorded from time to time. When a decision has been made to migrate an application, the application is restarted at the chosen site from the last checkpoint. This also assumes support in the underlying system for creating objects whose state is as defined at the checkpoint.
- tagging all the objects of the selected application and migrating the application on a per-object basis. An object is migrated only after its state becomes well-defined (more below) when one does not need to bother with copying activation records across sites. However, this may not always be possible (since in most applications, the main object always remains active).

Migration of applications requires a large amount of support from the underlying system which is quite difficult to provide in case of a heterogenous system. In the initial stages of the research, only object-migration (described below) will be looked at.

Instance Level

This deals with the migration of class instances across the nodes of a subsystem. The objective in mind is usually a lower completion time for the application. As mentioned earlier, the discussion here would restrict itself to migration in a limited sense. An object (instance) is migrated only when its state is well defined. This obviates the necessity of carrying along execution contexts when the object starts up on another node. In the ES-Kit system an object passes through the following states after creation:
• **Active**: the object is executing a method.
• **Blocked**: the object is in the middle of a method but does not currently have CPU control as it is waiting for a result from a remote-method invocation.
• **Waiting**: the object is waiting for an invoke message for one of its methods. This would also describe the state of the object when the invoke message has arrived at the node but the object has not obtained CPU control.

Of the states described above it is possible to migrate the object only when it is in the *Waiting* state. It is only in this state that the object is completely defined by the following:

- the local data of the object.
- the set of messages meant for the methods of the object.

Thus, any migration of the object should also transfer the above. In more detail, instance-migration involves the following:

1. **Initiation of Migration**
   This deals with the issues of what causes migration and when does it take place. Some of the options are detailed below:
   - Whenever the node computes its new estimate of load it determines the difference between the new estimate and the load average across all nodes, and decides to migrate if \( \Delta(\text{load-estimates}) > \delta \) (in the *Sender Initiated* case) for an arbitrarily chosen \( \delta \). A corresponding *Receiver Initiated* scheme would migrate when \( \Delta(\text{load-estimates}) < \delta \). The load average is computed as the weighted average of all the nodes in the neighborhood. An implicit assumption is that the load average is computed using the current loads on the neighboring nodes (in practice, estimated using the most recent information).
   - The frequency of migration-decision points is quite independent of the times when load-estimates are updated. For example, a timer interrupt could initiate the decision-making process. Again the decision is made based on the local estimate's difference from the global average. There is slightly more flexibility in this case vis-a-vis the recency of the load-estimates, particularly if the frequency of migration is significantly smaller than that of the exchange of load-estimates.

2. **Selection of Node**
   Once a decision to balance the load has been taken, one needs to decide the node to migrate to (in case of *Sender Initiated* schemes) or the node to get work from (*Receiver Initiated* schemes). The choice of this node is done using any of a variety of methods from just picking the least loaded node in the neighborhood to selecting the node that has the best bid (*Contract-Bidding Protocol*) and depends on the following:
   - matching of resource requirements of the object being migrated and the resource capacities on the node.
   - whether or not the node already has the object-code for the class (else, more overhead will result due to the necessity of loading the same from disk), etc.

One node must not get selected by more than one receiver or more than one sender. This leads to an inconsistency in the load-information of the selected node, as seen by the participating nodes, and could lead to undesirable migrations. This is also referred to as the *stability* property of the load-balancing algorithm: the system should not keep migrating objects within very small intervals of time.
3. Selection of Object
The object that is selected to be migrated should be the one that benefits most from the migration. The benefits could be either in getting control of the CPU earlier, or in not blocking for extended periods of time when making remote-method accesses. It is not clear how the latter would contribute to decrease overall completion time but can be tried out as an alternate object-selection heuristic. In case the load at the node were estimated as in the example of Section 2.2, the object to be migrated can be selected to be the last element of the scheduling queue.

4. Number of Objects to be Migrated
This is decided based on the difference between the load-estimate at the node and the load-average across the nodes in the neighborhood. On the basis of how much load has been transferred (if we again consider the simple example, the decrease in load at the current node would be the estimated residual-execution time of the object that is being migrated). Care has to be taken in computing the new load-estimates (after migration) since it is possible for the same node to be selected by more than one of the initiating nodes.

5. Mechanism of Migration
Having selected the object to migrate and the node to migrate it to, the following protocol needs to be observed before the object can start executing methods on the new node. The protocol is described from the *Sender Initiated* sense. The *Receiver Initiated* protocol can be similarly constructed.

(a) tag the selected object:
This is to prevent it from executing any methods that might have been invoked due to messages taken off the FIFO queue. There are two options for where the messages themselves can be stored:

- a lower-priority FIFO queue in the node mailbox structure. This option can create a problem if all messages in the higher-priority queues have been processed (a decision for migration should be made only when this cannot happen).
- in a separate structure along with the object that has been marked for migration. This would enable easy access to the messages as and when the object gets instantiated at its new location.

(b) send migrate request to selected node:
The request contains all the information necessary to create the object on the selected node. The initiating node receives the particulars of the constructed object (a *handle*) in reply. In case a different load-balancing strategy was being used where the node selection is done by mutual consent, the reply message would act as either the acceptance or rejection of the migration request.

(c) transfer data of the object:
The state of the object is represented partially by the values of its data variables. These have to be packed into messages and sent off to the new location of the object. Care must be taken to remove all local references from the data variables (this means that structures pointed to by the data variables must be expanded and packed into messages). These messages have to be appropriately unpacked at the new location of the object. Such local references as cannot be resolved should not be present in a migratable object (this includes local file descriptors, etc).

(d) create forwarding information:
This ensures that any messages that arrive for the object at its original location are now forwarded onto its new location. The forwarding information cannot be created earlier in the protocol since then there is a possibility of messages being processed by the object even before it retrieves its original state. The above also assumes that messages are received in the same order in which they were transmitted. In case this cannot be guaranteed another handshake is required between the two nodes (which says that the object has retrieved its old state and is now ready to process messages).

NOTE: An additional issue with forwarding information is what happens on repeated migrations. It can be shown that if an object has been migrated more than once, then on every migration the forwarding information needs to be updated on all the nodes in the path (to ensure correctness). This requires a broadcast after every migration and one after the object gets deleted (so that the forwarding entries can be freed). Optimizations can be made such as returning the current location of every migratable object whenever they are accessed (along with the reply messages). (This is somewhat similar to informing close friends about the change in address after moving houses).

(e) move the temporarily stored messages to the new location:
This assumes that the correctness of the program is not affected by the order in which different requests to a method of the object are processed. It is the programmer’s responsibility to take care of this.

(f) delete the instance on the current node.

3.3 Intelligent Scheduling

This is being considered only at the instance level.

Another option (which has not been explored to any significant depth) to reduce the imbalance among the loads is to schedule the execution of the invoke messages (the methods invoked by the messages) in a strategy more selective than FIFO. Such a strategy would prioritize executing those messages which invoke the methods of 'popular' objects, or of objects whose methods take comparatively less time to execute (shortest job first). It is not immediately clear as to what kind of heuristic needs to be adopted so as to reach the ultimate goal of lower completion time. However, this is another feature of the system which can be used to manipulate the load on a node.

4 ES-Kit Modifications: Information Collection

The ES-Kit software platform already provides for information collection with the help of the following structures:

- a list of system-wide static properties of the form NODE TYPE, MEMORY CAPACITIES, etc in the structure sys.properties.

- a description of the system configuration, and a list of the per-node properties (can be further refined to contain per-application properties) in an instance of the class sys.configuration.

This class is a traditional C++ class and all its methods are executed in a fashion similar
to the execution of procedure-calls. Probe methods are provided that can extract the value of a property for a particular node or an application (which may be spread across several nodes) given the name or the type (a number) of the property. Each node has a copy of this structure which it generates independent of the other nodes using the "config.file" file. Changes to these (basic) properties at the system and host levels are broadcast using the ISSD (the Interactive Shell Support Daemon).

- each node has an instance of the class **app_table** which keeps track of the various applications executing on the node via **app_entry** structures. Each **app_entry** structure consists of a single instance of class **class_table** which contains information about the different classes present on the node. The **inst_table** entry for each class contains an entry for each of the instances of the class executing on the node. Each **instance_entry** in turn points to the actual instance on the node. Each entry also contains a queue of requests (processed by the kernel) to the instance, statistics about which may be obtained by putting in the appropriate probe methods (ES-Kit does not supply this feature presently).

- the class **mem.mgt.h** provides two methods **get_resources(int)** and **get_resources(int, appl)** which returns the current state of resource type (int) for either the current application or any specified application.

- the method **statistics(int *)** of the structure **msggen_list** returns the number of cached messages and the space occupied by them. New **invoke/response** messages are allocated from the cached list of messages and returned to the list after use. This saves on the overhead ensued when the memory allocator is invoked for allocating the space required for a message.

- the method **get_free_list_size()** of the class **fifo** gives the free-list size of the list from which messages are allocated. The class **post-off** has a method **statistics** which returns the number of messages queued in each of the priority queues as well as the net space occupied by them.

Thus, the modifications that need to be made in order to collect the information identified in Section 1 fall mainly into two categories:

**Additional Data Structures**

These include space for storing all the information about each instance as well as the class-level estimates for the method execution times (since application-level measures are already provided). There are two options for the location of the data structures:

1. along with the **instance_entry** structures. These are expanded to contain information about the execution estimates of the method currently being executed by the object. Two variables are also required to store the pre- and post-remote-call estimates for each instance. To keep the modifications general, a hash structure will be associated with each instance entry that keeps track of all the instance level properties that are of interest and their values.

A similar approach is taken with the class level properties (number of instances, number of times a particular method has been invoked, execution time estimates for each of the class' methods, etc.) by providing the appropriate data structures in the **inst_table** class. The **app_entry** structure is augmented to provide the information which cannot be encoded in the currently available "property-value" format.
The class `klib.pso` will be augmented to keep track of how many times the librarian was queried (which gives the number of instances of the class that were created throughout the system).

Structures also are provided for storing the load estimates from the neighboring nodes.

2. by extending the `sys.configuration` class to provide information about the classes and the instances (the application level information is currently put in using the `special_properties` structure). This is done by creating a hierarchy of properties similar to the `app.table` structure. This has the advantage that since all information is contained in one class, it shall be less expensive (in terms of indirect memory references) for the load-estimating functions to access them. In fact, the load-estimating functions will be provided as methods of the `sys.configuration` class. However, the indirect references will still be present when the tables are being updated on every event which takes place at the instance or class level.

Addition of Probe Methods

Methods to access the data structures discussed above need to be supplied. In addition to methods that read from and write into the structures methods are also required that compute load-estimates at each level in the hierarchy. The load estimates are as discussed in the earlier sections. In the absence of exact knowledge about the nature of these functions, and to keep the modifications sufficiently general, some hook-routines will be provided for computing the load at each level. The specific routines will either be defined externally or obtained by redefining the virtual methods of a base class. Hook-routines need to be provided for computing the instance-level averages, the class-level behavior, and the load-contribution due to a particular application. The internal load on a node is computed as some function of the contributions from its applications. Methods also need to be provided for storing the load-estimates of the neighboring nodes and for exporting own estimate within the neighborhood. Methods are also required with the `post_off` structure so as to identify messages in the queue as being for a particular instance. This will be done by providing a method in the `mail_system` class.

Since the computation of the node level estimate is being done in a periodic fashion, some of the above routines need to be labelled as exception handlers which will be executed on a timer interrupt.

The exact classes which need to be modified depend on which of the two schemes (above) is adopted. Both schemes satisfy the important requirement that all collection of information and computation of load estimates should be performed in a non-blocking fashion (i.e. with the aid of traditional C++ objects).

5 ES-Kit Modifications: Migration Support

In order to support the various steps of migration outlined in Section 3.2, the ES-Kit software platform will be used in the following fashion. Only instance level migration support is being discussed.

Initiation of Migration
Probe methods are required to indicate when migration is necessary. These may be part of the exception handler depending on which approach is adopted. These methods should be provided along with the class that contains all the load information of the neighborhood nodes.

Selection of Node and Object

The options include

- specially selecting the node by considering the loads and the object characteristics (methods for these are required in addition to the above)
- implicitly selecting it by using one of the forms of .user.new. This requires providing the get.next.node() routine(s) of the sys.configuration class.

Selection of the object can be done using the information contained in the app.table structure (this has been derived using the instance and class level load contributions).

Number of Objects to be Migrated

Whether or not more objects need to be migrated needs to be decided by a method after the completion of every migration sequence. The method should take into account both the local migration and the migrations that might have gone on in other parts of the system at the same time (this is done by utilizing the broadcast messages containing the forwarding information for all the objects that have been recently migrated).

Mechanism of Migration

1. tag the selected object
   There is need for a structure where one can store the invoke messages that the kernel has looked at while the migration sequence is still incomplete. These will be stored in the req.list structure of the instance.entry struct (along with other messages meant for the instance). However, a flag indicating that the instance has been tagged for migration needs to be added. This shall be added along with the base.flags enum in the local.base class definition.
   The main.loop and deq.req methods of the node.kernel class need to be modified to ensure that none of the application messages intended for the object get executed.

2. send migrate request to selected node
   In order to have migration request messages processed quickly they need to be marked as being system level messages. A construct message will be used for the purpose of carrying the object particulars. Methods for changing priorities of construct messages and for executing a method on receipt of a system message will be added in the msg.stuff and p.fun classes respectively.

3. transfer data of object
   On receipt of the response message (also marked as system level), the object data is transferred. Since local references could be present which need to be bundled into the transfer message, two virtual methods need to be provided in the remote.base class. These methods, pack(void *, char *), and unpack(void *, char *), are responsible for bundling and unbundling the data areas respectively. The first argument contains the destination or the source pointer for the bundled data while the second argument contains the data format string in a syntax similar to the printf and scanf functions. The
application class definitions need to specify the data formats of the class in these functions. Bundling takes place by copying all structures (immediate and referenced) to the message structure pointed by the destination pointer. The copy_blocks instances are used to transfer the data over to the new location of the object. Some issues that need to be carefully implemented are the following:

- A kernel call to the pack() method would cause an invoke message to be created for the instance. If the priority of the message is left unchanged, then the method will never get executed since the instance has been tagged as “in_migration” and all messages get queued to the req_list structure. Furthermore, there will be a large delay before the message is picked up by the kernel (FIFO processing). To avoid both these problems, the message that invokes the pack() method has to be labelled as a system message. This causes it to be picked up earlier (higher priority) as well as not block (if the code in the main_loop of class node_kernel is appropriately modified). The call shall result in a data message (also tagged as system) being sent to the new location.

Similar arguments hold in case of the unpack() method. However, there is no necessity for the object to be blocked to take care of the undesirable event of some message executing before the object has retrieved its state. This is because it is assumed that transmission-order is preserved at the reception end. Hence, the unpack() message will be processed before any other message meant for the instance. The unpack() method is invoked when the kernel receives the data sent from the old location. The system message tag ensures that the state of the object is consistent. An option to this scheme is that the message containing the data is sent directly as a system-level invoke message to the newly created object. Unbundling occurs as routine method execution.

The behavior of the kernel when it receives system level messages will be incorporated by modifying the appropriate methods of node_kernel and p_fun classes.

- Adopting the method above of using pack and unpack methods also ensures that consistent allocation and freeing of structures is present. All structures allocated by methods of the instance before migration are freed by the pack() routine. Similarly, all structures allocated by the unpack() routine are freed by the other methods of the class after the migration is complete.

- The problem of deciding where to start copying the user data from is resolved by properly offsetting the pointer to the class by the sizeof of the kernel-supplied bases. This is a problem only when inheritance of migratable classes is allowed (otherwise, the pointer to the first user variable will be used in conjunction with the user supplied format-string). In the event that inheritance is allowed, the pack() and unpack() routines of the base classes need to be called before calling the methods for the migrated instance.

NOTE: This assumes that the data areas of base classes are allocated in front of (and contiguous to) the data area of the class being considered. All C++ compilers satisfy this requirement.

4. creating forwarding information

A new class that defines an array of forwarding structures needs to be defined. This would associate the old handle of the migrated objects with the new handle. The methods allow grabbing a free struct, releasing structures, and updating the forwarding information in a structure. There are also methods that change the destination handle field on an
Whenever an object that has already been migrated at least once gets migrated again, this requires the broadcast of the new location of the object so that all nodes in the path can update their forwarding structures. This implies that, firstly an object should be flagged as being migrated multiple times, and secondly that the system should support broadcasts (which it does). The kernel code acts upon the broadcast messages appropriately by augmenting the p_fun class.

A way of optimizing the number of hops that each message intended for a migrated object goes through is to send the new handle of the object through the response message. If the invoking object had blocked on the method, this information is used to update the local variable which contains the handle. This requires a modification in the arguments that are passed to the overloaded "—► ()" operator. The pointer to the local variable also needs to be an argument so as to facilitate writing the new address into it. The same approach can be used even in the cases where the invocation results in the creation of a future. The information regarding the variable that stores the handle of the invoked object has to be provided to the instance of the future_base class that is created to receive the response. When the future is finally accessed the variable can be updated. Care needs to be taken to ensure that the future is not remote as otherwise the pointer to the variable will not be valid.

5. transferring stored messages

This is simply accomplished using the methods of the forwarding structure which change the handle on the address.

6. deleting the instance

This is done using the method Delete of class remote_base.