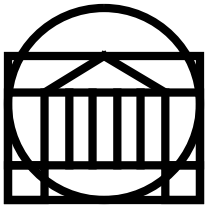
Support for Object Placement in Wide Area Heterogeneous Distributed Systems

John F. Karpovich
University of Virginia

University of Virginia Department of Computer Science
Technical Report CS-96-03. Also available at
ftp://ftp.cs.virginia.edu/pub/techreports/CS-96-03.ps.Z

January 16, 1996



DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF VIRGINIA
THORNTON HALL
CHARLOTTESVILLE, VIRGINIA 22903-2442
(804) 982-2200 FAX: (804) 982-2214

Abstract

One of the open challenges in distributed computing systems is determining how to place tasks onto processors when they are needed (in the Legion project being developed at UVA the basic computational units are modelled as objects, so the problem is one of object placement). The placement decision is crucial because it determines the run-time behavior of an object, including performance, cost and whether it can run at all. Many approaches have been developed to address this problem in a distributed system environment, but it is our claim that these efforts do not take the proper approach for supporting the needs of the large wide area heterogeneous virtual computer systems we envision will exist in the future. In particular, the systems developed to date are inadequate because they 1) focus on solutions for a narrow set of application types, environments, or user objectives, and 2) often inadequately support the full complexity and features of large distributed systems.

We propose to better support the placement process in distributed systems by employing a new approach. Our approach is different from previous ones in that we propose to design a *framework* for supporting a wide range of different placement problems, user objectives and placement algorithms, rather than building a system that supports a single placement technique. The goal of the framework is to provide programmers with the basic mechanisms to support each facet of the placement process which will enable them to implement the placement policies and techniques that meet *their* needs. On the other hand, individuals will be competing for limited resources owned by different people or organizations. Therefore, the framework must also contain mechanisms to enforce the policies of resource owners and to resolve conflicts between users.

The research effort proposed here will focus on developing the mechanisms needed to support flexible distributed object placement. To identify the main components of the placement process and the key issues that must be resolved, we will first develop a general model of the placement process. Using this model, we will next develop and implement a framework to support object placement within the Legion system. Developing such a framework will lend insight into our placement model and will also provide a proof of concept for our approach. Finally, we will demonstrate the usefulness of our framework approach by mapping a range of placement algorithms to the Legion framework and evaluating the performance of several algorithms versus that provided by the system today.

1 Introduction

The development of very high speed network technologies has opened the door to new possibilities in distributed and remote computing. High bandwidth communication capabilities enable distributed sites to more quickly and efficiently exchange information. As a consequence, it is possible to run many new types of applications in a distributed environment without incurring the unacceptable performance penalties that were previously unavoidable[1]. Existing applications can also receive a performance boost, providing faster end-user turn around times or enabling larger or more complex problems to be studied.

The potential benefits of exploiting the resources of a distributed environment are enormous. Benefits to the user community include increased resource sharing among sites, easier and more effective collaboration among colleagues, more efficient utilization of computational resources, and better performance for applications. Unfortunately, there are significant obstacles that must be overcome before distributed computing will be able to realize much of this potential. Large scale distributed systems are extremely complex due to the number and wide variety of components that comprise them. In the current development environment, distributed application programmers must explicitly manage much of this complexity which poses a significant barrier to the creation of new distributed applications and limits the extent to which distributed computing is exploited today (recent experience at Supercomputing 95 highlighted the difficulties of managing distributed environments).

The Legion project [12,16] at UVA is an effort to lower the barriers to exploiting distributed computing technology. Legion is designed to provide a software framework that manages and hides much of the complexity of the underlying system, allowing developers and users to focus more on their problem rather than on managing system resources. The goal is to provide an environment that is flexible enough to support a wide range of application types, including high performance applications, is easy to use and program, and can support a very large number of distributed resources efficiently.

The Legion model follows an object oriented approach, i.e. objects are the basic units in the Legion system. All Legion objects are instances of a Legion class, have a name that is universal within the system, and may persist over long periods of time. Each object has an interface, determined by its class, that defines the member functions to which it will respond. Users accomplish tasks by invoking the appropriate member functions on objects, which in turn perform their task and issue an appropriate response.

One of the open challenges in the Legion project is determining how to place objects onto processors when they are needed, so that they can perform their tasks. Object placement is an important concern because it can greatly influence an object’s run-time behavior, e.g. its performance. In fact, an improper placement decision can cause an object to not be able to perform its tasks at all, for example because no implementation exists for the target platform or because some key resource is not locally accessible. Such improper placement decisions are called *infeasible.* Obviously, a placement decision must be feasible in order to be useful. However, a feasible placement decision does not guarantee that the decision is a good one from the user’s perspective. Every user has his own goals and priorities when running a task, e.g. minimizing execution time or cost, or striking some balance between the two. A "good" placement decision, therefore, is a very personal choice and may differ from user to user and from task to task.

Determining good object placements in a large distributed, heterogeneous environment can be very difficult because of the complexity of the underlying system and because object behavior can be influenced by many different factors. Such factors include system status (number, type, and load of components), hardware capabilities (processor, network, I/O, memory, etc.), interactions between

1. Of course, not all applications will be efficient in a distributed environment, as communication latency cannot be reduced beyond a certain limit. Therefore, a wide-area distributed system is likely to be unsuitable for latency intolerant applications.

objects, object-specific characteristics (size, location of persistent state, estimated function performance, etc.) and many others. The factors that are relevant for a given placement decision are determined by the properties of objects involved and the objectives of the user. These factors are often expressed mathematically as an *objective function* which models the expected user satisfaction of each possible placement mapping. Other factors, such as security concerns, fault tolerance objectives, implementation availability and special resource requirements, may place hard restrictions on where an object can feasibly be placed. These factors are usually expressed as constraints on the placement decision, perhaps using a constraint language.

Given an objective function and placement constraints, an optimal placement decision can be made by finding the feasible placement mapping that maximizes the objective function. Unfortunately, it has been proven that in the general case finding such an optimal placement mapping is prohibitively expensive. In light of this, many past research efforts have focused on finding either near-optimal solutions or optimal solutions to very constrained cases. These efforts have developed a wide variety of algorithms and systems for generating good task placement decisions for certain placement problem types or user goals.

However, it is our claim that the approaches developed to date do not adequately support the object placement process for a large wide-area distributed environment. First, no single approach can generate good placement decisions for all of the diverse problem types and user objectives likely to be found in such a system. In fact, many of the approaches work well only for a very select group of applications and a single objective function. Second, many of the approaches were not designed to handle the complexity that such a system involves. Specifically, many approaches are deficient in at least one of the following areas: 1) scalability, 2) support/exploitation of heterogeneous resources, 3) ability to cope with uncertain or outdated information, and 4) consideration of uncertain and irregular communication costs. Finally, most previous approaches do not handle the problems introduced by having a persistent and shared object space, i.e. problems associated with managing an object's persistent representation and resolving placement conflicts for shared objects.

The goal of the research proposed here is to develop a new approach to adequately support the placement process in a large scale wide-area distributed system. Our approach is different from previous ones in that we will design a *framework* for supporting a wide range of different placement problems, user objectives and placement algorithms, rather than simply designing a new algorithm. In order to meet our goals the new framework at a minimum must exhibit the following properties:

- ***Support for definition of new placement algorithms*** - To exploit the placement approaches already developed and to allow for the evolution of new approaches, the framework must be flexible and robust enough to support the incorporation of new algorithms.
- ***Support for user selection of placement approach*** - Users must be allowed to employ the placement approach that best suits their needs in each placement situation;
- ***Support for resolving placement conflicts*** - In a system that supports shared objects and resources, it is almost guaranteed that some conflicts will arise over their use. The framework must define the behavior of the system during such conflicts and provide mechanisms for resolving them.
- ***Ease of use*** - The framework must not be prohibitively difficult to use, otherwise users will avoid exploiting it and will either suffer with poorer service than necessary or will abandon using Legion entirely. The framework should also support specification of default placement mechanisms so that users who do not require sophisticated placement decisions are not unnecessarily burdened with choosing placement algorithms and learning how to manipulate the framework;
- ***Low overhead*** - The framework must not impose unnecessary performance penalties for users who do not require sophisticated placement decisions - this follows the Legion design rule of "pay only for what you need";
- ***Integration with other Legion services*** - The framework must support and cooperate with other Legion services, such as security or fault tolerance mechanisms.
- ***Scalability*** - Mechanisms must be designed to allow the system to gracefully grow to at least

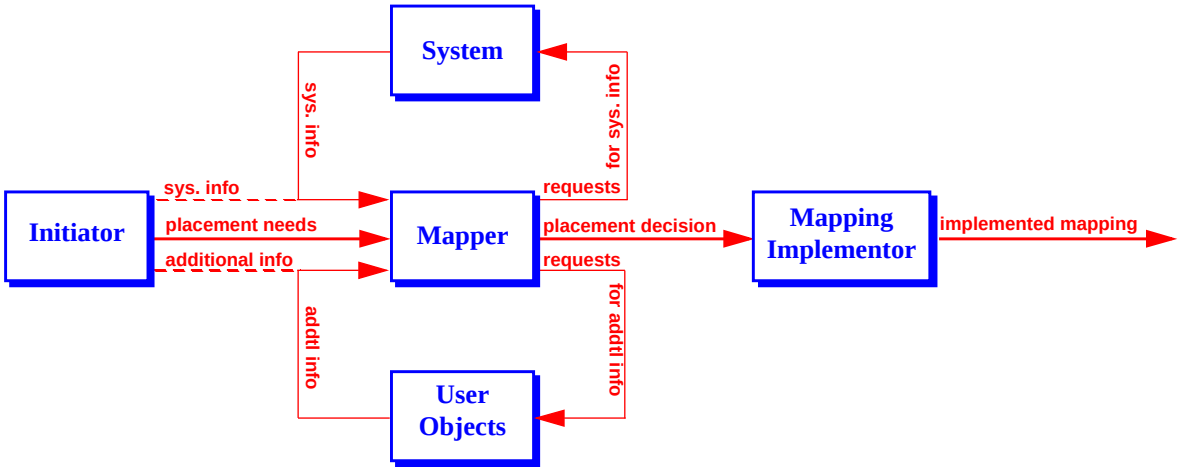


Figure 1 - Placement Problem Decomposition

millions of resources and objects.

Our agenda is to first develop a general model of the object/task placement process. Such a model will determine the key issues that must be addressed by the subsequent framework design and will also serve as the foundation for future research and discussion of placement issues. Next, we will design and implement a placement framework within the Legion system[1]. Legion provides a convenient platform for applying our approach to a real distributed environment. Finally, we will test how successfully the framework met our goals. At the conclusion of this effort we will have a detailed model of the placement process, a proof of concept for our placement framework approach and the useful artifact of a flexible placement mechanism for the Legion system.

Section 2 presents a model we have developed to describe the object placement process. Creating such a model has helped us to decompose the problem and to identify the key issues that need to be addressed by our framework. These key issues are described as part of the discussion on the model. Section 3 discusses the research agenda for this effort and section 4 presents related work on task placement/scheduling.

2 Placement Process Model

2.1 Overview

To design a framework to support object placement it is important to understand how the placement process works. Figure 1 shows a high level model for the general placement decision process. When a need arises to make a placement decision, some program or object will detect this need and begin the process. This entity, referred to in the model as the *initiator*, selects a placement mapping generator, or *mapper*, to create the placement decision and sends it a request describing the placement problem. Once the mapper receives the request, it is responsible for gathering whatever additional information is necessary for making its decision. At a minimum, the mapper must determine which objects are involved in the problem, the placement constraints for these objects, and the target resources that are available at the current time. More sophisticated mappers may also require additional information about the problem, the underlying system, or the objects involved. The mapper gathers all of

1. Even though our framework design will be implemented in Legion, there is no reason to believe that our basic approach and framework concepts will not apply to other distributed environments. In fact, it is one of our goals to keep the design as general as possible.

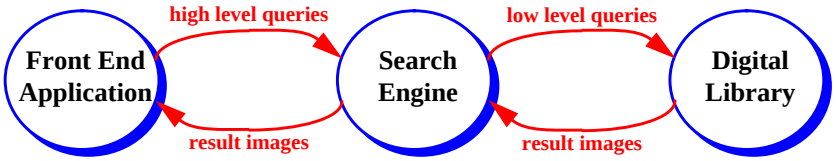


Figure 2 - Example: Digital Image Library Search Application

this information either from the initial placement request, via requests to other objects, or from its own internal state. Using this information a mapper determines a proper placement strategy for the problem. The placement strategy is then used by an *implementor* which attempts to realize the placement decision by activating the objects on the designated processors. If the implementor is successful, the needed objects will be active and ready to respond to user requests. The process is completed by doing any necessary bookkeeping and responding to the initiator that everything is now ready.

The following example illustrates how the process may work in a simple but realistic scenario. A user wishes to search a large digital image library to find images with certain characteristics. The user has at his disposal an application designed for just such a purpose that provides a nice graphical user interface and a rich query language for searching images based on their characteristics. Behind the scenes, the application has the structure shown in Figure 2. The application front end issues high level queries to a separate object which contains the actual search engine functionality. The search engine object then translates the high level query into a series of low level queries that the digital library supports. These low level queries are then sent to the digital library to retrieve candidate images, which are analyzed by the search engine and matched to the criteria of the higher level query. These results are passed back to the front-end application for display, etc.

The developers of the image searching application software wanted to sell lots of copies of their product, so they were very concerned about its run-time performance. They analyzed the application components and determined that the key performance bottlenecks were 1) the computation time involved in extracting and matching image characteristics within the search engine and 2) the time to transfer the data between the components. Therefore the application's performance is highly dependent on the placement of the search engine and library objects. They then developed a model to estimate the completion time of a query based on the following factors: network bandwidth and latency between each component, MIPs rating and current load of the search engine processor, and certain query parameters. This model then became the core of a placement routine that gathered the appropriate data and searched for a good coordinated placement for the search engine and library objects.

Figure 3 shows how the placement process works in the example application. The application front end acts as the initiator by sending the necessary query parameters to the placement routine (mapper) before each query. The placement routine gathers the necessary input for its placement model as follows: the objects involved are known and fixed within the model; the available processors and their current loads are determined from system daemons; and the processor MIPs ratings and network capabilities are found in a database containing information about system resources. The placement routine employs a heuristic search algorithm to find a good pair of processors for the objects and returns these values to the application front end. The front end then implements the placement decision itself and begins the query.

2.2 Placement Model Issues

The model presented in Figure 1 gives a high level picture of the basic components of the placement process and provides a foundation for building an object placement framework. However,

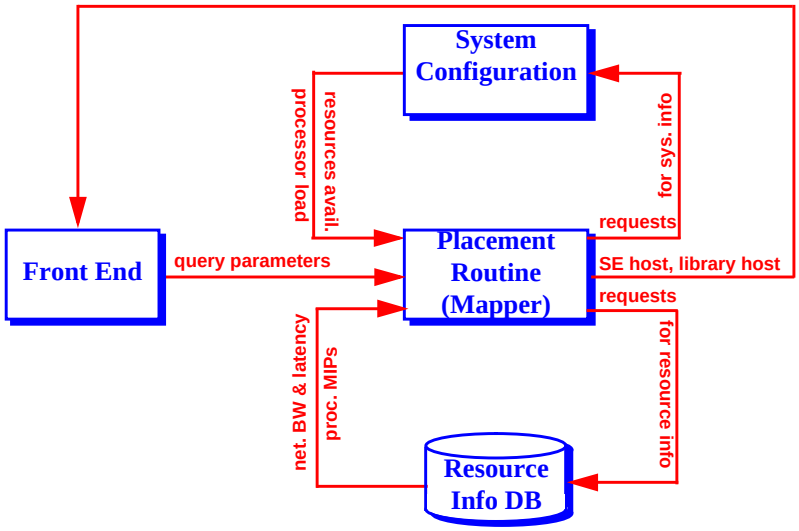


Figure 3 - Placement Process for Digital Image Library Search Example

there are many issues that must be resolved in order for the framework to be effective. These issues involve the functionality of the components, the mechanisms for maintaining and describing placement related information and the protocols required to manage the placement process. The sections that follow discuss each component in turn and the major framework issues associated with them.

One issue that spans several different design areas is that there must be a common mechanism for identifying the resources in the system and describing their important attributes. Within the placement process, this information is needed in at least three areas: 1) describing the system configuration and current resources available, 2) specifying placement constraints, and 3) describing placement decisions. The mechanism must be flexible enough to allow the addition of new resource types and new resource properties as technology and mapper needs evolve in the future. The mechanisms to describe system resources and their properties should also be flexible enough to describe user objects and their properties. In this way, developers and users would only have to learn one mechanism for both tasks.

2.2.1 Placement Initiator

Placement decisions can be initiated in two ways. First, when an object needs to interact with other objects, it can manually start and manage the placement process for these objects itself. Both user and Legion system objects will employ this method whenever the object developer or user feels that the placement decision is important enough to warrant the effort. However, it does not seem reasonable to force all Legion users to explicitly manage the placement process. After all, one of the main goals of Legion is to provide an environment where a user need not be *unnecessarily* concerned with the details of locating and managing objects with which they wish to interact. Therefore, a second object placement mechanism is required to automatically place and activate objects when necessary[1].

Automatic Object Activation

When an object or program invokes a member function on another object, the call should be completed without further effort required by the user (assuming the called object exists and that the call is valid). If the called object is not active, someone must automatically activate the object, and as part of

1. Such a mechanism may not always produce a good placement decision in all cases, but the hope is that is can do a good job most of the time, thus completely freeing most users from the burden of object placement and activation.

this process, some default placement mechanism must be invoked to determine where to activate the object. One possible solution is to define a system-wide default placement mapper that is invoked whenever automatic activation is initiated. The advantage of this approach is that it is simple - since there is only one default mapper, the automatic placement mechanism only needs to understand one mapper interface.

However, the single default mapper approach ignores the fact that all users and objects are different and have different placement needs. Employing a single mapper for all automatic placement decisions will force object users to either accept unnecessarily poor placement decisions or to manually manage object placement more often than they should have to. A better scheme is to allow developers or users to define default mappers at the class or even object level to provide more pin-point placement decisions during automatic placement. Hopefully, in most cases, the developer will do a good job designing a default mapper for the class and most users will never need to be involved in the placement process at all.

Initiator Tasks

Each placement initiator must perform three tasks, regardless of whether the initiation is done manually or via the automatic activation mechanism. The first task an initiator must perform is to select a mapper to make the placement decision. Selecting the proper mapper is important because each mapper will use a different technique and apply different criteria in determining a "good" placement for the problem. The mapper should have the same objectives as the user, e.g. reducing completion time, reducing cost, etc. The mapper itself will require time and resources to make its decision, so its performance and cost must be weighed against its anticipated benefits. When an object is manually managing the object placement process, it is up to that object to find the appropriate mapper. However, for automatic activation, there must be a mechanism to determine which mapper to employ. Class objects are a natural place to store object and class default mapper information. If this approach is used then finding the default mapper is a straightforward process - simply ask the object's class object.

The second task of an initiator is to send the selected mapper a properly formatted placement request to begin the mapping process. The question is what is the format of this request? Each mapper will implement a different algorithm, require different information, and operate on different types of problems. Therefore, it is possible that each mapper will require a different description of the problem[1]. For example, a sophisticated mapper may require a complete annotated precedence graph for the problem, while another mapper may only require the name of each object involved. For objects manually managing the placement process, it is their responsibility to format the placement request properly for the mapper used. However, there must be a way for the automatic activation mechanism to determine the proper format for a mapper request. Two possibilities are 1) develop a mapper input description language which can be used to determine the proper format for mapper input or 2) develop a standard interface that all default mappers must adhere to - additional information about the problem can be gathered explicitly by the mapper after it has been started.

The third task of the initiator is to specify problem-specific placement constraints to the mapper[2]. There are several approaches that can be taken to support this. The first is to not support it at all. If a user wants to restrict the placement of a problem, he can create/derive a new mapper that follows the proper constraints. However, this does not seem like a very user-friendly approach. Two other approaches are 1) to pass constraints to the mapper as part of the placement request or 2) develop a protocol for the mapper to request the constraints from the initiator after it receives the placement request. In either case, an important issue is how to describe problem constraints - should it be an issue solved individually by each

1. A single mapper may even understand multiple different problem description formats to support different users or legacy codes.
2. The constraints discussed here are specific to a single placement request and are in addition to object and class constraints (section 2.2.4) which define restrictions on where individual objects must be placed during all requests.

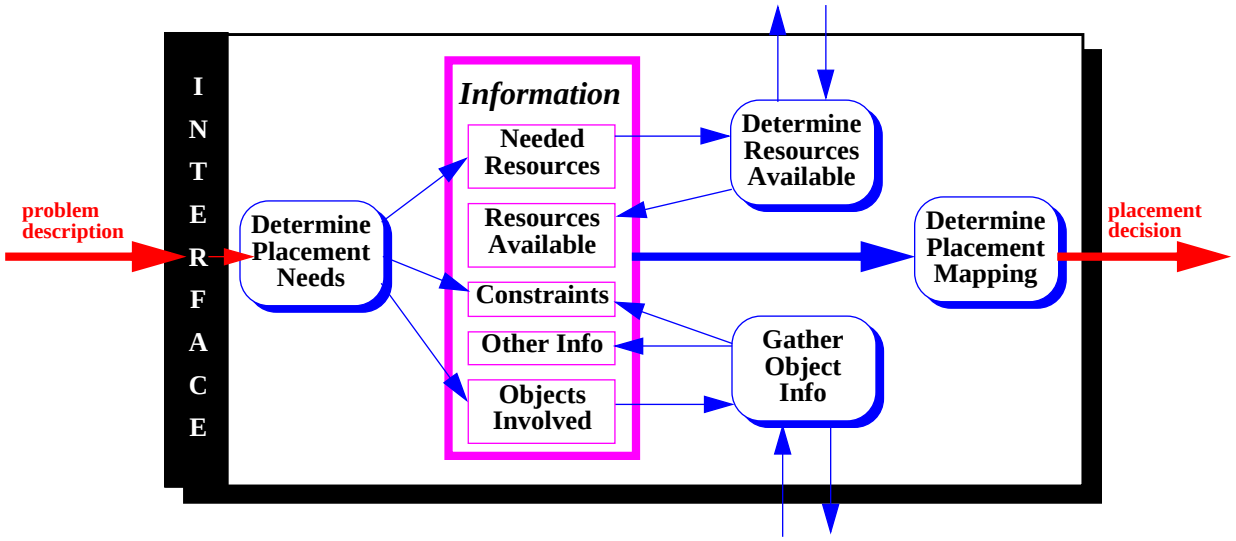


Figure 4 - Typical Mapper Functionality

mapper or should there be a common constraint language for all mappers.

2.2.2 Placement Mapper

The placement mapper is at the heart of the object placement process and therefore its requirements drive the design of the placement framework. It is important to understand the main tasks that mappers will typically perform, even though the internal structure of each individual mapper is outside of the control of the framework (the programmer who develops the mapper has complete control over its internal structure). Figure 1 shows the main tasks that mappers typically perform and the types of information they require.

- ***Determining Placement Problem Needs*** - The mapper must first extract the details of the placement problem from the problem description provided by the initiator - e.g., what objects are involved, what constraints are imposed on their placement, and what resources are needed to fulfill the request (see section 2.2.1 for discussion of problem description format).
- ***Determining Resources Available*** - In order to make a placement decision the mapper must determine which target resources are available. As part of this process, the mapper must decide on a strategy for how far and where to search for candidate resources. This is an important decision because the potential scale of a Legion system makes it impractical to search for all resources available. Section 2.2.3 discusses gathering information on available resources in more detail.
- ***Gathering Object Information*** - Mappers will also need information about the objects involved in the placement decision, including object properties, placement constraints, current location, etc. Mappers will collect this information from various sources such as databases, class objects, etc. See section 2.2.4 for more detail.
- ***Determining Placement Decision*** - The ultimate goal of the mapper is to use the information it has gathered to make a good placement decision for the problem. The algorithm used for this can be of any form - e.g. the mapper may break the problem into pieces, solving each piece in a different manner, possibly even sub-contracting another mapper to solve some pieces of the problem. Once a placement is determined, the mapper must format the placement decision in such a way that it can be understood by an implementor. Section 2.2.5 discusses the format of placement decisions in more detail.

2.2.3 System Supported Information

One of the key components in making any placement decision is gathering all the necessary information that will drive the decision process. Certain types of information will be needed by many or all of the different placement mappers that will exist in Legion. To facilitate the task of developing placement mappers, the system should maintain and provide access to commonly needed placement data whenever practical. The challenge is to determine which types of information are worth maintaining, what their precise semantics are, how they will be maintained efficiently, and how they will be accessed by mappers (and other objects as well). Some types of information that may fit into this category are:

- ***Current Resources Available*** - This information will be used by every mapper in order to make proper placement decisions. The design of such a mechanism must consider several attributes of this type of data, including the scale of the system, the high variability of resource availability, and the fact that placement decisions can be greatly affected if this information is unavailable (may want to consider replication or fault-tolerance mechanisms). One possible design is to create a number of distributed daemons each responsible for gathering data about a portion of the overall system (this approach has been used many times in distributed systems). Information about larger portions of the system can be collected by retrieving data from the appropriate local daemons.
- ***Host Attributes*** - This includes both (relatively) static information (e.g. processor type, OS type/ version, attached devices, memory size, CPU performance metrics, security policies, and reservation policies) and dynamic information (CPU load, memory available, run queue length, temporary and swap space available). The Legion design includes the notion of host objects, which are responsible for loading and starting object implementations on local hosts once a placement decision has been made. These host objects are a natural place to maintain information about host characteristics.
- ***Networking Attributes*** - The capabilities of the underlying network can be a key factor in determining the placement of distributed objects. Network performance properties, e.g. latency, bandwidth and current network load, affect inter-object communication properties and therefore affect the performance of tasks that involve multiple distributed objects. For applications which need network performance guarantees, such as video or soft real-time applications, it is necessary to know what quality of service guarantees are supported between hosts. Modelling network capabilities for a large system is a very difficult challenge and needs to be addressed in distributed system research. However, an in depth analysis of providing network information in a distributed system is beyond the scope of our work.

2.2.4 User Level Information

The key ingredient to the success of any placement mapping mechanism is accurate and detailed information. All mapping techniques require some information about each of the objects involved in a placement problem. At a minimum the mapper must know about placement constraints for each object (see below), but may also require additional information that is specific to each object or its class. Some examples are the number of records or total size of a data object, or the average FLOPs required per data item for a filtering object. Because every object and class is different, the mechanism for describing object/class specific data must be very flexible to allow specification of new types of information as new classes develop. One possible approach is to model the data as a 2-tuple, consisting of a name string and a value string[1]. This mechanism is very flexible and extensible as long as name conflicts are avoided.

Since object-specific placement information is often needed while the object is inactive, it cannot be placed within the object itself. A natural place to store this information is within an object's class object, which already is responsible for maintaining other information about the object. The class object can also store class wide information that applies to all objects of that class. The definition of the class base class can be modified to add storage, update, and retrieval functionality for both class and object

1. This is the same approach as that taken for shell variables in Unix.

specific data. All Legion classes are derived from the class base class, so this functionality will be automatically inherited by all Legion classes.

Object Placement Constraints

In order to determine the feasibility of a placement mapping, one must know the placement constraints for each object in the mapping. Constraints are not hints - they place hard restrictions on where objects can be placed. Object placement constraints arise due to several factors, including 1) target resource properties required by the object, 2) implementation availability, 3) security concerns, and 4) fault-tolerance concerns.

In order to perform properly, each object must be placed on a host that contains all of the necessary hardware and software required to support it. For instance, the host must have enough memory, swap space and temporary storage to properly execute the object's implementation. The host must also have all of the necessary system software available to support the object. Some objects may require that special hardware be available to the target host, such as a printer, display device, camera, etc. In addition, the object must have an implementation available for the type of processor and operating system found on the host.

Security concerns may also place some restrictions on where an object is executed. When an object is executed remotely, the machine on which it runs may ultimately be under someone else's control. For objects that require tight security, this poses a risk because the root user (to use Unix terminology) can look into the object, e.g. by looking at memory or by forcing it to produce a core file. So, objects may only trust hosts at certain locations or of certain types.

One of the design goals for the Legion placement mechanism is to allow users to employ different placement mappers in different situations. This means that the placement decision for an object may be made by potentially many different mappers over the course of its existence. Also, the same mapper may operate on different objects and classes of objects over time. Therefore it is crucial that there be a common method of describing constraints within Legion. Otherwise mappers will be very constrained in the types of objects with which they can work and vice versa[1].

2.2.5 Placement Decision Implementation

In order to discuss the role of the placement implementor, it is important to understand how objects will be activated in Legion. Legion is purposely designed to provide a high degree of autonomy to each member organization. Ultimately, each organization has final authority over its resources - when they are available to the system, which objects are permitted to execute on them, etc. To enable local control of resources, Legion supports the notion of local jurisdictions for each organization. Each jurisdiction encompasses a set of resources, i.e. hosts, and is managed by a local jurisdiction magistrate (JM) object which enforces the policies of the organization for that group of resources. All requests to place objects on resources within the jurisdiction, must go through the JM, which will decide if the activation request will be honored or not. The magistrate may reject the request for any reason, e.g. the resource is now unavailable, is too highly loaded, is reserved by someone else, or the security or other policies of the jurisdiction do not permit execution of the particular object or class of object.

On the other hand, Legion also supports the sovereignty of each object. Each object may have constraints on where it can be placed (section 2.2.4) for reasons ranging from the hardware and software requirements to security issues. These constraints are enforced by restricting access to the object persistent representation (OPR) of the object which is needed to execute any persistent object. At any time the OPR of an object is stored and managed by some JM which the object trusts. It is the job of the JM to enforce the placement policies of the object by not transferring its OPR to another JM that the

1. Note that a common constraint description mechanism does not imply that all mappers will work with all objects - there are other issues, such as the types of information available about the object, etc.

object does not trust.

It is the job of a placement implementor to translate the placement decision of a mapper into an activation plan and then to carry out the plan by issuing the proper requests to the appropriate JM's. There are several interesting design issues that must be resolved for the implementation process:

- ***Mapper/Implementor Interface*** - The biggest challenge is defining how to describe the placement decisions. A simple solution is for the mapper to produce a set of object ID/host ID pairs specifying which objects are to be activated and on which hosts. Unfortunately, this simple solution is not very flexible. A better design would allow the mapper to specify a number of equivalent resources for an object or a series of alternative solutions in case the first mapping cannot be implemented. The mapper should also be able to describe any reservations which it has negotiated, so that the implementor can use them (or release them if for some reason the placement fails). Finally, to help resolve conflicts over shared objects (see exceptions section, below) the mapper may wish to specify the priority of the placement of each object.
- ***Design Jurisdiction Manager Interface*** - A thorough and precise interface must be defined to handle all aspects of activating objects, including placement failures, object migration and deactivation, querying of jurisdiction placement, security, and reservation policies, OPR transfer and storage, etc.
- ***Placement Implementation Exception Handling*** - There are several reasons why the implementation of a placement decision may fail, e.g. the object's placement constraints are violated, the destination host rejects the request or becomes unavailable, some key object fails, or a placement conflict arises for a shared object. As alluded to above, part of the answer to placement implementation failures may be to try another alternative mapping supplied by the mapper. However, when a problem cannot be resolved the failure semantics of the implementor must be defined.
- ***Placement Implementation Bookkeeping*** - The final part to implementing a placement decision is making sure that any bookkeeping required by the system is completed. For example, a newly activated/migrated object's location and other placement information (e.g. priority) must be recorded (probably in its class object) and appropriate objects must be notified (e.g. the initiator or some object that will monitor the activated object's progress).

3 Research Agenda

The research proposed in this document has two primary goals. The first goal is to analyze the distributed system placement process in detail to determine the key components involved and key issues that need to be resolved. Documenting the placement process and developing a model to describe it is not only a necessary prerequisite for the remainder of the research proposed here, but also provides a foundation for future research and discussion of placement issues within the research community. As such, the placement process model and description will be one of the key contributions of this research.

The second goal is to design a framework to support the placement process within the Legion system. Such a framework will serve several purposes. The design and implementation of an actual framework will provide additional insight into the placement problem, which will in turn drive new refinements to our model of the placement process. Furthermore, the Legion design will provide a specific instance of a placement framework as a proof of concept for our approach. This is the crux of the research agenda - to prove that our framework approach can support the placement needs of a large diverse distributed system. The Legion design and implementation will provide the platform upon which to evaluate how well our framework approach is able to meet the goals identified in section 1. Finally, the implemented framework will be a useful artifact and become the actual Legion placement mechanism (assuming our design proves to be acceptable).

The rest of this section describes the research agenda in greater detail. Section 3.1 describes the development of the placement process model. Section 3.2 describes our approach for developing the Legion object placement framework. An initial "strawman" design for the Legion framework is presented

in Appendices A and B (Appendix A describes the design while Appendix B presents class interface definitions). Finally, section 3.3 details how we will evaluate our Legion framework design.

3.1 Development of a Placement Process Description and Model

The analysis and decomposition of the distributed system placement problem has already been partially completed as part of the research for this proposal. The placement process model and component descriptions presented in Section 2 detail a portion of the work accomplished to date in this area. The description of the placement process will be further refined as feedback is provided by experience gained in designing the prototype framework and by the continued development of other parts of the Legion system - e.g. the security and fault-tolerance mechanisms.

3.2 Design and Implementation of Object Placement Framework

Because designing and implementing an object placement framework is breaking new ground, it is difficult to assess the effort and risks involved in doing so, especially since the underlying software infrastructure (Legion) is not yet complete. It is important to realize this element of risk and to plan the research agenda accordingly. Development of the placement framework will follow the spiral model [5] of software engineering, which is a risk-driven approach to the development and enhancement of software. What this means is that we will start with an initial framework design (presented in Appendices A and B) and iteratively enhance pieces of the design that were found to be deficient during evaluation of an earlier phase. Each phase will begin with the specification of a framework feature to be added or enhanced. Some examples include upgrading the placement constraint specification mechanism to be more expressive, adding support to gather and store information about network properties, or enhancing the placement conflict resolution mechanism. After alternative designs have been identified, the benefits and risks associated with each alternative will be evaluated. Only if a design alternative is perceived to be beneficial enough to warrant the risks involved will the feature be implemented. In this manner, the best implementation can be developed within a reasonable amount of time. Some of the risks anticipated in this effort include overambitious design (implementation effort must be in line with the time frame given for the project and not preclude finishing the rest of the research agenda) and dependence on unfinished or volatile functionality under someone else's control.

The design of the Legion placement framework will focus on resolving the key issues identified in section 2 and will be driven by the design goals defined in section 1. At a minimum the Legion framework must include functionality in two basic areas: 1) mechanisms to describe, store, and acquire information about resource availability and resource and object attributes; and 2) mechanisms to effect a placement decision. These two mechanisms represent the core functionality necessary for supporting user developed placement mappers - (1) provides the necessary inputs and (2) provides the vehicle to carry out mapping decisions. Using our spiral development model, the design will proceed in phases, each phase improving upon the previous phase. Before accepting each successive phase, the suitability of the new design will be verified by translating a representative set of placement algorithms using the proposed new design. This process will hopefully expose design flaws before excess time is wasted on implementation. Appendix C describes several application domains that not only are representative of the types of applications we expect to run in the Legion environment, but also are diverse in their placement needs. Placement algorithms suitable for these domains will form the basis for the design verification suite.

3.3 Analysis of Framework Design

The final part to the research effort proposed here is to analyze the Legion framework we develop. The first part of our analysis is to justify our design by showing that it is in fact sufficient to support object placement for diverse placement mappers. We will use two approaches for justifying the sufficiency of our design. The first approach will be written arguments about the power of the

mechanisms we have designed - these will be developed both during the design process and afterwards as a post-design phase. The second approach will be to do a detailed paper design of a diverse and representative group of placement algorithms. This is basically the same process we will use to evaluate intermediate designs, except for the level of detail and number of algorithms employed. Results of the paper design process will be documented, including any exposed shortcomings of our framework, ease of design, etc. To further analyze the framework we will implement several of the simple placement algorithms using the framework implementation.

The second part of our analysis will focus on whether our approach to supporting user definition of placement algorithms adds value to the object placement process. To do this we will compare the performance achieved by several applications/objects using simple user defined placement mappers tailored to their needs versus random placement and perhaps another general placement technique. Through this, we hope to demonstrate the potential that our approach holds, even employing relatively simple placement algorithms.

3.4 Summary of Research Artifacts

The research proposed in this document will generate several lasting (and hopefully useful) artifacts.

- ***High Level General Model of Placement Process*** - this model will provide a basis for discussion of placement issues and will hopefully be extended and refined in future efforts.
- ***Legion Object Placement Framework*** - the specification of the framework will define the Legion placement mechanism and can be used as both a reference and as the basis for testing the compliance of Legion system placement mechanisms. The framework design will also include a justification of the design, in particular that the mechanisms provided are sufficient to support a wide range of possible placement mappers.
- ***Implementation of Legion Placement Framework*** - the framework implementation developed as part of this research will be incorporated into the implementation of the full Legion system.
- ***Evaluation of Framework Design*** - we will provide the results of our evaluation of the placement framework. In particular, the evaluation will include 1) a brief performance analysis of several simple mappers versus random or round-robin placement; and 2) analysis of the framework's ability to support a diverse range of mappers.

4 Related Work

The problem of placing tasks on shared resources is certainly not new. The object placement problem is really a special case of the general task scheduling problem, which has been studied at great length in the computer science, systems and operational research literature. Research efforts have spanned a wide range of issues and viewpoints. Some of the different efforts have included development of mathematical models for scheduling problems [8], proofs of the complexity of optimal placement determination, modelling of computer system behavior, and the development of a large number and variety of scheduling and placement algorithms and systems. One of the principle results of scheduling theory research is that optimally solving general forms of the scheduling problem has been proven to be NP-hard. In practice this means that optimally solving most scheduling problems (or in our case placement problems) is intractable.

Point Solutions

Since the complexity of the scheduling problem was proven, much of the research in resource scheduling has been devoted to finding either 1) constrained situations for which determining the optimal schedule is tractable or 2) methods to generate near-optimal or at least good scheduling decisions. The search for tractable optimal scheduling cases has had limited success - more often than not these efforts

have produced either solutions with limited applicability or proofs of more NP-hard situations. The efforts aimed at finding good approximate schedules have had significantly more success. These efforts can roughly be grouped into two non-mutually exclusive approaches. The first approach has focused on producing better searching algorithms that can more efficiently and accurately search the solution space. Examples of such techniques include heuristically directed search engines, simulated annealing, and genetic algorithms. The second approach has focused on improving the accuracy of the objective functions or the precision of their inputs. Research has included identifying indicators that more accurately estimate actual system and program performance and constructing more sophisticated models of programs, computer resources and their interactions.

All of these efforts have produced literally hundreds of different scheduling algorithms and systems. In order to fine tune scheduling decisions, many of these algorithms are highly specialized for specific program types or resource configurations. For example, algorithms have been tailored to exploit knowledge of certain common communication patterns (e.g. 1D, 2D, ring, or tree), or to exploit the specific properties of popular supercomputer architectures (e.g. their model of parallelism, interconnection topology, I/O properties, etc.). Each of these algorithms performs well for certain classes of scheduling problems and user goals, but no single algorithm has emerged that can satisfy the needs of all users. I like to think of each algorithm as being well suited for a particular point in the space of all scheduling situations. Hence I call these algorithms and systems *point solutions*.

While none of the myriad of point solutions presented in the literature solves the general task placement problem, they are related to our work in the sense that each of these solutions can be transformed into a placement mapper in our model[1]. Therefore, existing algorithms and new research efforts and trends (e.g., the recent trend towards exploiting heterogeneous supercomputer resources) suggest guidelines for the design of our placement framework.

Load Sharing and Load Balancing Systems

Many algorithms and systems have been developed to support load sharing[2] among a set of distributed hosts. These efforts share a common goal with Legion - to better utilize the power of a distributed computer system by supporting remote sharing of resources. The load sharing systems developed to date vary widely in a number of dimensions, e.g. transparency of remote execution, scalability, support for heterogeneous resources, placement objective function(s), and assumptions about system configuration just to name a few. The following section describes the significant current or recent research efforts aimed at load sharing in distributed systems. Each of these efforts is aimed at a particular category of placement problems and therefore amount to point solutions. However, studying them is important to understand where current research is headed and how the effort proposed here is truly unique.

The Utopia system [22] developed at the University of Toronto was designed with many of the same objectives that we have identified for our proposed framework. Utopia is designed to support remote execution of tasks transparently, to be scalable to at least 1000s of hosts, to support and exploit host heterogeneity of various kinds[3], and to support a wide range of application types, including interactive and parallel applications. However, while the objectives are similar, the philosophy for how to achieve them is very different. Utopia was designed to support a particular placement policy deemed to be good enough for most placement requests. Because of this, the design of Utopia is tailored to

---

1. Unfortunately, many algorithms will likely be inappropriate or will need to be modified to work in a dynamic distributed environment.
2. Load balancing systems are a subset of load sharing systems. In load balancing systems, such as LoadLeveler [14], NQS [15] or its successor PBS [13], the objective function is designed to balance work across all available processors. Load sharing systems support other objective functions as well, e.g. minimizing completion time of an application.
3. Configurational, architectural and operating system heterogeneity.

specifically support the chosen placement algorithm. For example, the types of information available about hosts and other resources are built into the system design and fixed. The same is true of the types of placement constraints that can be specified as well as many other pieces of the system. Because of the closed nature of the Utopia design employing different placement strategies can be difficult depending on the placement algorithm.

Two other contemporary load sharing projects deserve mention. The NOW (Network Of Workstations) project at Berkeley [1] is focusing on harnessing the power of relatively cheaper workstations for applications that have traditionally been the purview of expensive supercomputers. The ultimate goal is to provide both better application performance while significantly reducing the cost of system hardware components. Their approach is to better exploit all of the resources of a cluster of workstations to attack three of the main performance bottlenecks for applications by: 1) expanding the memory available to a single application without requiring costly swaps to disk by using available RAM on other workstations (i.e. network RAM); 2) improving I/O performance by caching files across all workstations; and 3) supporting parallel computation across all workstation nodes. Another effort connected with the NOW project is exploring the effects of running parallel programs in a non-dedicated rather then a dedicated environment [3] and developing methods for better supporting parallel program scheduling in a non-dedicated environment [9].

The Condor project [6,17] at the University of Wisconsin is a somewhat earlier attempt (1990-1991) at supporting load sharing in a workstation environment. Condor is designed as a distributed batch system, i.e. jobs are submitted to the Condor system which determines a "fair" placement for each job. One of the interesting policies chosen by the Condor developers is to ensure that machine owners are impacted as little as possible by Condor scheduled jobs. To implement this policy, Condor automatically checkpoints jobs and migrates them to another machine as soon as it detects that the machine’s owner has returned. Overall, the basic Condor system is a simple and effective load sharing mechanism. However, it is fairly rudimentary in its functionality and has several drawbacks for the large scale diverse environment we wish to support, e.g. it supports only one simple placement and migration policy, requires a centralized "machine manager" (scalability problem), contains no support for co-scheduling, etc. A recent extension to Condor, the Condor Application Resource Management Interface (CARMI) [18] was developed to address several of these drawbacks with respect to resource management. CARMI provides resource management services to allow applications to explicitly acquire new resources (hosts), create new processes on acquired resources, and be notified when resources are lost or reclaimed by workstation owners. The mechanisms for describing resource attributes and defining resource classes deserve special attention. Resource attributes are exported by each resource using what appears to be a very similar mechanism to the name-value tuple mechanism discussed in our strawman design (Appendix A). Resource classes are defined using logical expressions which can reference attributes exported about the resources - this is very similar to one of the design possibilities we identified - the use of a PROLOG-like language to describe placement constraints (Appendix A).

Distributed Heterogeneous Supercomputing (DHS)

Perhaps the hottest research topic in the area of scheduling/placement in distributed systems is developing methods to best exploit heterogeneous supercomputing resources. The advent of gigabit networks makes cooperative computation using multiple supercomputers more feasible while the high cost of supercomputers provides plenty of economic incentive for maximizing their productivity. Freund and Conwel introduced the concept of *superconcurrency* [10,20], which is an approach for maximizing the performance of a given suite of heterogeneous computers by placing tasks on the best-matching machine[1]. By minimizing the computation time for each task in the system, superconcurrency strives to improve both system throughput and application performance. Freund, et al, are now applying the

1. Matching tasks to machines is achieved through code profiling and analytical benchmarking.

superconcurrency approach to the SmartNet project at the Naval Research and Development (NRaD) facility, but unfortunately there are no publications available yet.

The Superconcurrency/SmartNet work does not account for some important application characteristics, such as I/O properties. The work by Ghafoor and Yang on the Distributed Heterogeneous Supercomputing Management System (DHSMS) [11] at Purdue University addresses some of these issues and provides a systematic methodology for code profiling and analytical benchmarking. They propose a framework for hierarchically classifying supercomputer characteristics that forms the basis for machine categorization, code profiling and analytical benchmarking. One advantage of their classification scheme is that it is general and flexible - each site can define its own classification scheme based on the properties of its machines.

Other Related Work

The Zoom project [2] has focused on developing a model to describe heterogeneous applications. The model is hierarchical in that it supports three levels of detail. The first level describes only the overall structure of the application; the second level elaborates the first by adding the available implementations for each program component and by further describing the communication patterns and characteristics of the application; finally the third level adds details about communication granularity, data conversion requirements and legal pairings of component implementations. The goal of this effort is to provide an abstract representation for heterogeneous programs that can be used as a foundation for developing heterogeneous program development and scheduling tools. To this end, a further effort [21] has explored coupling the Zoom representation with the HeNCE graphical language and design tool to develop heterogeneous programs on top of PVM. A current focus of the Zoom project is exploring how to use the information provided by the Zoom model to develop scheduling support for heterogeneous applications. Such a research effort will provide valuable guidance in our own effort to develop a placement framework. We currently have a working relationship with Rich Wolski and hope to work closely with him. If possible we would like his scheduling work to be implemented on top of our Legion framework.

Related Work Conclusion

Unfortunately, none of the systems to date adequately supports all of the objectives of the Legion placement process. In particular most systems fall short in at least one of the following areas: scalability, support for heterogeneous resources, or user customizability of the placement mapping process. In addition, the current systems view tasks as the basic placement/scheduling unit, whereas Legion views objects as the basic unit. Therefore, these systems do not address object-related issues such as retrieving object persistent state or resolving placement conflicts for shared objects. In the final analysis, there are no scheduling/placement research efforts that approach the problem in the same manner as the work proposed here. There is a certain amount of synergy between the research discussed in this section and our proposed work in that the techniques employed by these systems provide the guidelines for what is needed in a general purpose object placement framework.

5 References

[1] Thomas E. Anderson, David E. Culler, David A. Patterson, et al, "A Case for NOW (Network of Workstations)", Principles of Distributed Computing, August 1994, also http://now.cs.berkeley.edu/Case/case.html.

[2] Cosimo Anglano, Jennifer Schopf, Rich Wolski, Francine Berman, "Zoom: A Hierarchical Representation for Heterogeneous Applications", University of California San Diego Technical report CS95-451, October 1995.

[3] Remzi H. Arpaci, Andrea C. Dusseau, Lok T. Liu, "The Effects of a Non-Dedicated Environment on Parallel Applications", http://http.cs.berkeley.edu/~dusseau/Papers/papers.html, 1993.

[4] Larry Bergman, Hans-Werner Braun, et al, "CASA Gigabit Testbed: 1993 Annual Report", Caltech Concur-

rent Supercomputing Facilities Technical Report CCSF-33, May 1993.

[5] Barry Boehm, "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, Vol. 21, pp. 61-72, May 1988.

[6] Allan Bricker, Michael Litzkow, Miron Livny, "Condor Technical Summary", University of Wisconsin Department of Computer Sciences Technical Report 1069, January 1992.

[7] W.F. Clocksin, C. S. Mellish, *Programming in Prolog*, 2nd Edition, Springer-Verlag, Berlin, 1984.

[8] Edward G. Coffman, Jr., Editor, *Computer and Job/Shop Scheduling Theory*, John Wiley and Sons, New York, New York, 1976.

[9] Andrea Dusseau, Remzi H. Arpaci, David E. Culler, "Re-examining Scheduling and Communication in Parallel Programs", University of California at Berkeley Computer Science Technical Report UCB/CSD-95-881, 1995.

[10] Richard F. Freund, D. S. Conwel, "Superconcurrency: A Form of Distributed Heterogeneous Supercomputing", *Supercomputing Rev.*, Vol. 3, No. 10, October 1990, pp. 47-50.

[11] Arif Ghafoor and Jaehyung Yang, "A Distributed Heterogeneous Supercomputing Management System", *IEEE Computer*, Vol. 26, No. 6, June 1993, pp. 78-86.

[12] Andrew S. Grimshaw, William A. Wulf, Jim C. French, Alfred C. Weaver, Paul F. Reynolds, Jr., "Legion: The Next Logical Step Towards a Nationwide Virtual Computer", UVA CS Technical Report CS-94-21, June 1994.

[13] Robert L. Henderson and Dave Tweten, "Portable Batch System - Requirements Specification Revision 1.5", NASA Ames Research Center, April 1995.

[14] IBM Corporation, *IBM LoadLeveler: User's Guide*, 1993.

[15] Brent A. Kingsbury, "The Network Queuing System", Sterling Software, Palo Alto, CA, September 1994.

[16] Mike Lewis, Andrew S. Grimshaw, "The Core Legion Object Model", UVA CS Technical Report CS-95-35, August 1995.

[17] Mike Litzkow, Miron Livny, "Experience with the Condor Distributed Batch System", IEEE Workshop on Experimental Distributed Systems, Huntsville, AL, October 1990.

[18] Jim Pruyne and Miron Livny, "Parallel Processing on Dynamic Resources with CARMI", Workshop on Jon Scheduling Strategies for Parallel Processing, International Parallel Processing Symposium (IPPS), April 1995.

[19] Bjarne Stroustrup, *The C++ Programming Language, 2nd Edition*, Addison-Wesley, Reading, Mass., 1994.

[20] Mu-Cheng Wang, Shin-Dug Kim, Mark A. Nichols, Richard F. Freund, Howard Jay Seigel, Wayne G. Nation, "Augmenting the Optimal Selection Theory for Superconcurrency", *Proceedings Workshop on Heterogeneous Processing*, 1992, pp. 13-22.

[21] Richard Wolski, Cosimo Anglano, Jennifer Schopf, Francine Berman, "Developing Heterogeneous Applications Using Zoom and HeNCE", *Proceedings of the Workshop on Heterogeneous Computing Scientific Computing*, Santa Barbara, CA, April 1995.

[22] Songian Zhou, Xiaohu Zheng, Jingwen Wang, Pierre Delisle, "Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems", *Software Practice and Experience*, Vol. 23, No. 12, December 1993, pp. 1305-1336.

Syntax:

```
DATA_ITEM => ID(VALUE_LIST).
VALUE => INTEGER | FLOAT | STRING | [VALUE_LIST]
VALUE_LIST => VALUE, VALUE_LIST | VALUE
```

Examples:

```
size(100).
current_load(1.2).
supported_platforms(['sparc2/SunOS5.x', 'sparc2/SunOS4.x', 'mips/IRIX5.x']).
platform_implementation('sparc2/SunOS5.x', 'implementation1_ID').
platform_implementation('mips/IRIX5.x', 'implementation2_ID').
```

Figure 5 - Object Data Description Syntax and Examples

Appendix A - Initial Framework Design

The following sections describe an initial "strawman" design for the object placement framework. It is intended to provide a concrete starting point for the design process while also demonstrating the feasibility of the placement framework proposed in this document. This strawman design *is definitely not intended to be the final design*. Subsequent refinement phases will lead to a richer and more capable framework.

A.1 Object Data

Data about objects is important for making placement decisions and therefore, there must exist mechanisms to describe, store, and access information about each object in Legion. The first question that must be answered is how such data will be represented. For the strawman design, this data will be represented as a list of named data items. The syntax chosen for object data description is similar to that used in PROLOG [7] for describing facts, i.e. name(value_list). There are two reasons for choosing such a representation. First, it is simple and expressive enough to describe a wide range of data types, including lists and structured data. Second, one of the options under consideration for describing placement constraints in future designs is to use PROLOG-like rules statements. So using a PROLOG-like data representation may yield benefits later. Figure 2 shows the syntax for data items and some examples.

Object data will be stored in two places - the object itself and in the object's class object. While the object is active, it will be the authoritative location to access its data. However, when an object is inactive, some of its data must still be accessible (e.g. to make placement decisions). Therefore, this type of information will be stored in the object's class object. Access to such data (retrieval, update removal) will be accomplished via member functions in the Legion class and Legion object base classes.

A.2 System Provided Data

The Legion system will initially provide information only about system hosts. To accomplish this, a "host" base class will be created which among other tasks, will maintain certain types of host-related information. Since instances of the host class will be Legion objects, this information will be represented and stored just like object information for any other Legion object (see above). Host information maintained will include: processor type, operating system type and version, memory size, temporary disk space available, swap space available, current load, frequency of update for current load, 5 minute load average, 15 minute load average, current queue length, MIPs rating, FLOPs rating, SPECMarks rating, special devices attached, etc.

A.3 Selecting Object Implementation

In order to realize a placement decision, the implementor must be able to determine whether an executable exists for the specified host type and if so where it is (mappers may also use this information to avoid making infeasible decisions). In the initial design, this information will be stored in the object's class object just like any other data. The data format will be a series of data items named "platform_implementation", each containing a tuple of string values - 1) the name of the platform (some standard for platform names will need to be determined), and 2) the name of the implementation for the specified platform. The last two lines in Figure 2 show examples of object implementation entries. If no entry is made for a particular platform, that implies that the platform is not supported.

A.4 Placement Decision Description

Once a mapper has determined what it thinks is a good placement mapping it must describe its decision so that it can be implemented. For the strawman design, placement decisions will be represented as a set of individual object placement descriptions, each of which will contain the ID of the object and an ordered list of host IDs. The first host in the list defines the primary choice of the mapper for placing the specified object; the remaining hosts in the list provide an ordered sequence of alternative selections in case the placement on the primary host fails for some reason.

A.5 Object Placement Constraints

To keep the initial design simple, only rudimentary support will be given for defining constraints for object placement. Object constraints will consist of either 1) a list of good hosts and/or magistrates OR 2) a list of prohibited hosts and/or magistrates. This information will be stored with the rest of an object's data in its class object. The following defines the format and meaning of each type of constraint.

- "Positive" constraints - the combination of hosts defined by "good_hosts" and "good_magistrates" are the ONLY hosts that are suitable for the object.
  - good_hosts([HOST_ID1, HOST_ID2, ...]). - any host in list is acceptable.
  - good_magistrates([MAG_ID1, MAG_ID2, ...]). - any host watched over by the given magistrates are acceptable.
- "Negative" constraints - the combination of host defined by "bad_hosts" and "bad_magistrates" are the ONLY hosts that are NOT suitable for the object.
  - bad_hosts([HOST_ID1, HOST_ID2, ...]). - all hosts in list are unacceptable.
  - bad_magistrates([MAG_ID1, MAG_ID2, ...]). - no host watched over by the given magistrates is acceptable.

A.6 Mapper Interface and Placement Problem Description

Each mapper can define its own interface if its designer wishes. However, to support automatic placement there needs to be a standard interface that all default mappers use. To encourage non-default mappers to also support a common interface, the standard interface provides a bit more functionality than is strictly needed for default placement. The standard mapper interface is described below:

placement_mapping create_mapping(object_description_list, problem_data_list)

- placement_mapping has the format described in section A.4.
- object_description_list has the form [object_desc1, object_desc2, ...].
- each object_desc has the following format:
  - object_desc(object_ID, [data_item1, data_item2,...]).
    each data_item uses the standard format of object data items (see section A.1).
- problem_data_list allows the caller to pass in additional, problem-wide information to the mapper - the format is the same as the data_item list within object_desc.

A.7 Automatic Activation Support

In order to enable the system to perform automatic object activation, the system must be able to select the appropriate default mapper, issue a properly formatted request to the selected mapper, and receive the placement decision so that the system can implement it. To support default mapper selection, each Legion object will add a data item (default_mapper(MAPPER_ID)) to its object information database stored in its class object (if no default mapper is specified, then a system-wide default mapper will be invoked). To enable the system to properly invoke the default mapper, all default mappers will be required to support the standard interface described above (section A.6). The automatic activation mechanism is responsible for formatting the required arguments and invoking the default mapper. The object_description_list will consist of a single object description while the problem_data_list will include a single data item - the ID and current address of the caller whose invocation started the automatic placement process. The latter allows the mapper to determine the location of the caller and to possibly retrieve additional about it and its intended use of the to-be-placed object.

Appendix B - Strawman Class Interfaces

For the strawman design introduced in section A, the following sections describe the basic interface features for the classes involved in the object placement within Legion. The interfaces are by no means complete, but provide some insight into what is needed and how the classes will be used. The interfaces are described using C++-like function declaration syntax [19].

B.1 Legion Object Base Class

Accessing Object Info

```
data_item_list   get_all_data_items();
data_item        get_data_item(data_item_name);
data_item_list   get_data_item_list(data_item_name_list);
boolean          update_data_item(data_item or data_item_list);
boolean          remove_data_item(data_item_name or name_list);
```

B.2 Legion Class Base Class

Accessing Object Info

These are the mechanisms for accessing data about individual objects of a particular class. This information is stored in the object's class object so that it can be accessed while the object is inactive.

```
data_item_list   get_all_object_data(object_ID);
data_item        get_object_data_item(object_ID, data_item_name);
data_item_list   get_object_data_item_list(object_ID, data_item_name_list);
boolean          host_OK(object_ID, host_ID); // is host suitable for executing this object?
boolean          update_object_data_items(object_ID, data_item_list);
boolean          remove_object_data_items(object_ID, data_item_name_list);
```

Accessing Class-Wide Info

Some data about objects will be stored on a class-wide basis because it is common to all or most class instances. If the same data item is defined at both the class and object level, than the object level data overrides the class level information.

```
data_item_list   get_all_class_data(object_ID);
data_item        get_class_data_item(object_ID, data_item_name);
data_item_list   get_class_data_item_list(object_ID, data_item_name_list);
boolean          update_class_data_items(object_ID, data_item_list);
boolean          remove_class_data_items(object_ID, data_item_name_list);
```

Locating and/or Activating Objects

```
Legion_addr      get_object_address(object_ID); // get address if object is active
Legion_addr      activate_object(object_ID, host_ID); // activate object on specified host.
boolean          deactivate_object(object_ID, Legion_addr); // deactivate object.
```

B.3 Legion Host Base Class

Accessing Host Info

Host objects are full-blown Legion objects and will therefore derive the basic object information manipulation functions. In addition, hosts will have:

```
boolean          object_OK(object_ID); // does host allow object to execute locally?
```

Activating/Deactivating Objects

```
Legion_addr     execute(implementation_ID);
boolean         deactivate(Legion_addr);
```

B.4 Legion Magistrate Base Class

Finding Available Hosts

These functions are used to gather information about available hosts in the system.

```
host_list       get_all_avail_hosts();     // return all hosts currently available within jurisdiction.
host_list       get_avail_hosts(host_ID_list); // return hosts available out of given list.
host_list       get_hosts_accepting_object(object_ID); // return hosts that accept given object.
```

Activating Object/Migrating Object Persistent Representation

```
boolean         store_OPR(object_ID, OPR);
OPR             get_OPR(object_ID);
boolean         move_OPR(object_ID, magistrate_ID);
Legion_addr     activate(object_ID, host_ID);
boolean         deactivate(object_ID, Legion_addr);
```

Appendix C - Placement Scenarios

To determine the needs for the Legion placement framework one must study the different scenarios under which it will likely be used. After looking at a number of possible scenarios I have grouped them into two categories: single object scenarios and multiple object scenarios. Roughly speaking, single object placement is the placement of one object at a time independent of the placement of other objects, while multiple object placement is the *coordinated* placement of more than one object at once. There is a grey area when only one object is placed, but its placement process relies on information about the current location of other objects. I have chosen to include this case under single object placement, though an argument can be made to the contrary.

This single/multiple object placement categorization is useful for a number of reasons. First, the single object placement problem is easier to analyze and easier to understand and therefore makes for a good starting point on the subject. However, single object placement can be viewed as a special case of multiple object placement and therefore any observations about single object placement will provide insight into multiple object placement as well (i.e. commonalities and differences between them will emerge - both of which are important). Second, it is likely that single object placement will be the dominant placement method in Legion and therefore deserves special attention. Whenever possible, the mechanisms for single object placement should take advantage of any knowledge of their constrained nature. Third, single and multiple object placement will potentially be used in very different situations. Multiple object placement will only be employed when the user or class developer feels that for some reason a group of objects must be placed in a cooperative manner. In contrast, many users who are less sensitive to an object's run-time behavior, will simply want the objects they use to be placed anywhere that will enable them to perform their functions. For these users, placing objects independently is adequate.

The following sections describe a number of placement scenarios in more detail. The single object placement scenarios have been lumped into two general categories, one for placement during automatic object activation and one for "coordinated single object placement" - the case when the caller wants to guide the placement of an object with which it interacts. The multiple object placement scenarios, on the other hand, are more complex and cover a wide range of application types, each with different placement approaches or special requirements. To capture this diversity, the multiple object placement section discusses a number of different application domains and their particular requirements and tolerances. The seven application types presented in the multi-object section were chosen because they represent a wide range of important existing or likely future parallel and distributed applications that are likely to run in the Legion environment[1]. In particular, these were chosen because they span a wide range of placement approaches and needs.

C.1 Single Object Placement

C.1.1 Automatic Default Object Placement

One of the goals of Legion is to provide an environment where a user need not be *unnecessarily* concerned with the details of locating and managing objects with which they wish to interact. When a user object invokes a member function on another object, the call should be completed without further effort required by the user (assuming that the call is valid and the called object exists). This is relatively easy to do if the object is actively running - all that is needed is to find the address of the object's associated process and to invoke the called member function[2]. However, due to the large number of

1. Of course, one must be aware of the inevitable emergence of new applications domains, especially as support software for distributed computing, like Legion, matures and becomes more wide spread.
2. Actually, even if the object is active, it may be desirable to explore the possibility of migrating the object to provide better performance.

objects anticipated in a large scale Legion system, it will not be possible or desirable to keep all objects active at once. Therefore, in order to maintain transparency to users, a mechanism must be built into Legion that will automatically activate objects when necessary. As part of this activation process, an object placement decision is required.

From an application developers viewpoint, exploiting this automatic mechanism is very desirable. We believe that for most object invocations the automatic placement decision will provide an adequate decision to satisfy the caller. We base this on the belief that many invocations will fall into one or more of the following categories: 1) The automatic mechanism does a reasonably good job of placing the object for the given invocation; 2) Behavior of the invoked member function is not very sensitive to placement decisions, either because the member function does little work or because the object is highly constrained to run on fairly homogeneous resources; or 3) The caller is not very sensitive to object run-time variations and will not mind a poorer placement decision, at least up to some threshold.

For these reasons, we envision that automatic activation and placement will be one of the most used mechanisms in the Legion system. As such, special care must be taken to ensure that the mechanisms developed are:

- *Easy to use* - that is, easy for developers to implement default object placement routines;
- *Flexible* - developers must be able to employ the appropriate placement techniques for their objects, otherwise placement decisions will suffer unnecessarily (we must recognize that every object/class is potentially different and should be treated that way);
- *Low overhead* - the basic mechanism should not impose a significant amount of overhead. If there is a large overhead cost, some applications, e.g. those interacting with using many objects, may experience unacceptably poor performance due solely to the mechanism.

A final note about automatic object activation. Though it is desirable for the object activation mechanism to be automatic, it is also desirable that the placement decisions made by default mappers be as accurate as possible. Otherwise, users affected by poor decisions will be forced to enter into the activation process in order to improve the placement of called objects. Often, all that is needed to improve a mapper's accuracy is some additional information that the caller has readily available. For example, knowing the intended future use of the object or the location of the caller may improve the mapper's decision. Therefore, the automatic activation mechanism must provide a way for the caller to provide this necessary information to the default placement mapper so it can make more informed decisions without the caller having to take direct control (of course the default mapper must be designed to exploit this added information, else the point is moot).

C.1.2 Single Object - Caller Guided Placement

Though automatic object activation and placement is certainly desirable, there will remain times when the performance of a default scheduler will not be adequate for all invocations. The reason for this is that for some classes it is likely that even the developers of classes will not realize all of the potential uses for their class objects. It is unrealistic to assume that in such a case the developer will have the ability or even the desire to provide a default mapper that produces excellent decisions under all circumstances. In these cases there must be a mechanism to override the default mapper when it cannot adequately do the job.

To make the point more concretely, take the example of a simple Unix-like (i.e. byte oriented, randomly accessible) file object. Such file objects can be used in many different ways and can represent very different size files. The placement decision for such an object will depend on the size of the file, the location of the calling object, the number, pattern, type (read, write) and volume of requests from the caller, and potentially the anticipated request patterns from other objects. It may be very difficult to construct a mapper that deals well with certain uses, e.g. very irregular access patterns. In such a case it seems more practical to expect the caller to worry about such unusual uses if he deems it necessary,

rather than expect the file class developer to think of and handle all possible access patterns.

C.2 Multiple Object Cooperative Placement

C.2.1 Data Parallel Applications

The data parallel (DP) application domain is very important because it is widely used in the parallel computing community, especially in the scientific computing community. The reason for the DP model's popularity is that it is easy to understand and implement and it fits the needs of many problem domains, e.g. low-level image processing, climate modelling, and many codes that employ mathematical solvers using large matrix operations.

In general, DP applications (DPAs) tend to work on large data sets, which are divided among some number of workers. Data distribution usually falls into one of two scenarios: 1) the entire data set is split up at the beginning of the computation by assigning a piece to each worker; or 2) data is distributed as needed in a receiver initiated fashion, i.e. whenever a worker is ready to process more work, it acquires a new piece of the data set to work on. On the macro level, operations are employed on the data set as a whole, but these operations are carried out in parallel, each worker operating on its piece of data. Often, the amount of work per data element to perform an operation is very regular and predictable. Between operations, workers are often required to synchronize and to exchange partial results of the previous operation before work may continue on the next operation (barrier synchronizations).

The properties of DPAs pose many challenges and opportunities for scheduling them. What follows is a list of some of the relevant DP characteristics and their effect on placement.

- *Large data sets* - the anticipated performance of data access operations on the input and output data object(s) is often a critical factor in determining the overall placement for DP application objects. Data object performance is effected by such factors as network performance, disk performance, processor performance and load, data access patterns and data volume.
- *Predictable amount of work per operation* - this can be exploited by mappers to derive very accurate estimates of worker execution times and in conjunction with good estimates of worker communications costs, can be used to accurately derive the computation granularity of the workers.
- *Interoperation synchronization points* - for those DP applications that must synchronize in between certain operations, this poses is difficult challenge to mappers. The problem is that each worker must wait at the barrier until all other workers have finished the operation. Therefore, the performance of the application is extremely sensitive to imbalances among the workers and the placement decision must be just right to avoid unnecessary delays.
- *Inter-worker communication* - For DPAs whose workers exchange data in between operations, the placement decision must be sensitive to the effects that communication will have on the performance of the overall application.

Due to the popularity of the DP model and its employment in very large problem domains, a lot of research has gone into optimizing the performance of DPAs, including algorithm and language design, specialized scheduling techniques and even hardware design. The literature contains a large number of specialized DP placement and scheduling algorithms or systems and many are tuned for a specific problem domain, inter-worker communication topology, hardware configuration, etc. The applicability of each technique to a specific DP problem varies widely, as does the information each requires and the quality of the placement decision. The important point is that even within the domain of DPAs, there will need to be many placement algorithms to best support different applications or platforms.

C.2.2 User Interactive Applications

The defining characteristic of user interactive applications, as the name implies, is that a user

actively interacts with and/or steers what the application does using some input and output device(s). Many applications fall into this category, including many of the applications we run everyday, for example word processing or text editing, interactive "talk" sessions, world wide web browsing, interactive terminal sessions, games, and many others.

From a placement viewpoint, interactive applications in general have certain common requirements. Placement of objects responsible for input from or output to users must be constrained to processors that can access the required I/O devices. Since users are physically waiting for responses to their input, performance of interactive applications is very important and has special requirements. Many studies have researched the response time limits that human users find acceptable for various I/O devices and application types. The studies have generally found that users are most happy with consistent response times that do not exceed certain thresholds. They also found that since users work on human time scales - on the order of 10’s, 100’s or 1000’s of milliseconds depending on the situation - that there is often little perceived benefit to users when response times are much shorter than required. Therefore, the key goal for placing interactive applications is not necessarily the minimization of total run time, but to provide consistently adequate performance throughout the duration of the application. One way to accomplish this goal is to negotiate for dedicated resources or guaranteed resource quality of service commitments.

C.2.3 Video Applications

Due to recent and expected future advances in network technologies, applications requiring high resolution video such as teleconferencing and video playback from remote repositories will soon be a reality. We anticipate that the number of such applications will increase dramatically in the near future when advanced network technologies reach the main stream. However, providing smooth, jitter free video display is very demanding on currently available technology and requires that the involved resources be very well managed. To guarantee a level of quality for the video transmission, usually some guarantees on performance must be negotiated from the network, the transmitting and receiving machines, the I/O devices producing or displaying the video and/or the storage system on which the video resides.

These requirements make placement of the various components of such an application a very challenging and demanding job. In particular, the placement mechanism will need to have access to information about network capabilities and load, and the capabilities of the display and storage devices. Placement mappers will also need mechanisms to negotiate for dedicated service or at least a guarantee of a minimum level of service from various devices.

C.2.4 Multi-Disciplinary Applications

A recent trend, especially with scientific modelling applications, has been to merge or couple previously separate applications into larger more comprehensive applications. For example, as part of the CASA gigabit network testbed project [4], researchers have coupled together a global atmospheric model and a global water model to form a more comprehensive global climate modelling application. The practice of fine tuning these types of applications is often referred to as *multi-disciplinary optimization* because they are formed from applications in different disciplines and the goal is to optimize the performance of the resultant mixed or multi-disciplinary application. From a placement standpoint, these applications are interesting because 1) each component is often a large, computationally intensive program in its own right, 2) each component is often well suited to a particular type of parallelism and consequently its implementation is often already highly optimized for one or more computer architectures, particularly supercomputers, and 3) the components often have a well known communication pattern among themselves. Due to their computationally intensive nature, large problem sizes, and the fact that each component often has a strong affinity to particular architectures, component placement is crucial to the overall performance of these applications. Because of these attributes, MDAs

are often the focus of scheduling research in the distributed heterogeneous supercomputing community.

C.2.5 Parallel Simulations

Parallel simulations are employed in many fields, from weather forecasting to high energy physics or to economic or social forecasting applications. These codes are usually very large and long running, hence the reason for running them in parallel, and are often very complex. Computation times for different subtasks are often driven by the data or events occurring within the simulation, so predicting computation times *a priori* can be difficult or impossible. Also, since data values are continually changing within the simulation, the amount of computation done by a particular subtask may vary over time. Inter-task communication patterns can also vary from simple or complex, but for many subtasks communication is constrained to a relatively small group of other subtasks (this communication group may change during the simulation).

A key feature of simulations is that the individual subtasks are all modelling the state of one large process and the events that effect it over time. As such, the progress of each subtask through simulated time must be kept even (or nearly so), so that an event generated by one subtask is received by another effected subtask before it proceeds beyond the event time in simulated time (the requirement that subtasks proceed at the same rate is similar to data parallel applications where all subtasks must finish one phase at the same time before moving to the next). The combination of needing nearly even progress across subtasks (at least within groups) and data or event driven computation times may encourage the use of dynamic subtask migration to maintain the balance of subtask progress and minimize overall simulation execution time. To adequately support these types of applications, the placement framework will likely have to supply object migration functionality.

C.2.6 Soft Real-Time Applications

Soft-real time applications are deadline driven and the primary goal is not necessarily to achieve peak performance, but rather to achieve steady, reliable performance for each component within some specified bounds. Video applications can be thought of as a special case of soft real-time applications, where deadlines are established for the receipt and display of each video frame. User interactive applications also share many qualities with soft-real time applications, though the real-time goals of such applications may be even more "soft" than in true soft real-time applications. Like video and user interactive applications, placing soft real-time program objects requires in depth knowledge of resources attributes, including the processor and network, and may require service guarantees from various devices.

C.2.7 General Task Parallel Applications

Task parallel applications, as the name implies, achieve high performance by executing different subtasks of the overall program concurrently. This form of parallelism is the most general form - there are no restrictions placed on the interactions between subtasks other those the programmer chooses to enforce. Because of this the program graphs for task parallel applications can vary from simple and regular graphs to highly complex and irregular graphs. All of the other parallel application domains discussed in this section are special cases of the general task parallel model. However, there are many other important parallel applications which do not fit nicely into another classification, so general task parallel applications are important in their own right.

The approach to making good placement decisions for unrestricted task parallel applications is often somewhat different from the approach taken when the problem can be constrained in some way. For one, optimally solving the general case is NP-hard and therefore intractable in practice, so a heuristic or approximate approach must be used. Also, there is no special knowledge that can be brought to bear to help guide the search. One characteristic that is common to many approaches is the requirement of a task graph or precedence graph, possibly annotated with additional information, as input to the placement mapping process. Our placement framework will have to be able to handle such applications.