

Enabling Flexibility in the Legion Run-Time Library *

Charles L. Viles, Michael J. Lewis, Adam J. Ferrari
Anh Nguyen-Tuong, Andrew S. Grimshaw
Department of Computer Science
University of Virginia
Charlottesville, VA, U.S.A.
<http://legion.virginia.edu>

Abstract *This paper describes the design and implementation of the Legion run-time library (LRTL), focusing specifically on facilities that enable extensibility and configurability. These facilities include management of heterogeneous communication, an event-based mechanism for inter-component communication, and automated memory management. The paper provides several examples that illustrate the inherent flexibility of the LRTL implementation.*

Keywords: configurable protocol stack, events, implicit parameters

1 Introduction

The widespread deployment of gigabit networks will effectively shrink the distance between computing resources and will enable wide area distributed-object computing systems that will consist of many heterogeneous, distributed, unreliable resources. Legion [1, 2] will be one such system. Without significant software support, users will not be able to manage the complexity of this environment. Meta-systems software [3]—software that resides “above” physical resources and operating systems and “below” users and applications programs—is needed. Legion meta-systems software will include a run-time system, Legion-aware compilers that target this

run-time system, and programming languages that present applications programmers with a high level abstraction of the system. Thus, Legion will allow users to write programs in several different languages, and will transparently create, schedule, and utilize distributed objects to execute the programs.

Legion’s users will require a wide range of services in many different dimensions, including security, performance, monetary cost, and functionality. No single policy or static set of policies will satisfy every user, so users must be allowed to implement their own solutions and to determine their own trade-offs as much as possible.

Legion supports this philosophy in several different ways. For example, Legion provides the mechanisms for system-level services such as naming, binding, and migration, but does not mandate these services’ policies or implementations. Legion requires a certain object-mandatory functional interface to all Legion objects; the implementation of the interface is left up to the object. Legion also specifies the minimum functional interface to a set of core system object types, the implementation of which can and will vary. Furthermore, Legion delegates much of what is usually considered system-level responsibility to classes, which are special Legion objects. For instance, classes are responsible for creating and locating their instances, and for selecting appropriate security and object placement policies.

The Legion run-time library (LRTL)—the subject of this paper—is the cornerstone of our

*This work was partially supported under DARPA(Navy) contract N66001-96-C-8527, DOE grant DE-FD02-96ER25290, and DOE contract Sandia LD-9391.

Legion meta-systems software. Legion object implementations are linked with LRTL, which provides the basic mechanisms to allow Legion objects to communicate with one another using Legion-compliant mechanisms. LRTL is intended to be used both by Legion-targeting compilers and by user-level code; thus, when we refer to LRTL’s “users,” we mean both compiler writers and applications programmers. In building LRTL, we were driven by several sub-goals and constraints. First of all, we wanted to abstract much of the complexity that is inherent to heterogeneous distributed computing. For example, we wanted to alleviate the need for LRTL’s users to deal directly with the varying data formats on different machine architectures. More importantly, in accordance with the overall Legion philosophy that one size does not fit all, we wanted LRTL to become a useful software tool with which users could build different policies and algorithms along many different dimensions, without having to build an entirely different library. Thus, we built LRTL itself to be extensible and configurable.

This paper describes how LRTL supports this flexibility, and provides several examples of how the flexibility can be exploited to implement different programming styles and useful distributed systems components. Section 2 begins with a brief system description, Section 3 explains the major mechanisms through which LRTL enables flexibility, and Section 4 illustrates several examples of this flexibility. We conclude with related work and a brief summary. In the interest of brevity, we have omitted many details, especially in our description of LRTL itself. For a more complete exposition of LRTL, please see [4].

2 System description

Before discussing the the flexibility and extensibility features of LRTL, we must put it in context with the overall Legion system. In this section, we examine the general features of Legion, including its basic object model, the core objects that implement primitive system mech-

anisms, and the typical usage of the system.

Legion is an object-based system—all entities of interest to the system are objects. Legion objects are logically independent address space disjoint collections of data and associated methods. Objects can contain one or more associated threads of control, and communicate via asynchronous method invocations. Objects are identified by unique names called *Legion object identifiers (LOIDs)*. LOIDs are location independent, and must be mapped to low-level, ephemeral *object addresses* when inter-object communication takes place. Legion objects are persistent and may be in one of two states: *active* or *inert*. Active objects contain one or more threads of control and are ready to service method calls. Inert objects exist as passive object state representations on persistent storage. Legion transitions objects between active and inert states to use resources efficiently, to support object mobility, and to enable some approaches to failure resilience.

All Legion objects are described by an interface specified using an Interface Description Language (IDL). A Legion object’s interface describes the methods that it supports and the types of their parameters and return values. In addition to a user defined interface, all Legion objects support a set of object-mandatory methods that help implement basic Legion mechanisms such as object persistence, migration, and security. Methods on Legion objects are executed using a macro dataflow model [5]. This model requires that any method invocation sent to an object include (in addition to its parameters) a description of where the results produced by the method should be forwarded. For example, instead of being returned to the caller as in an RPC model, the result of a method invocation might be forwarded directly to some other object as a parameter to one of its methods.

Legion objects are instances of classes, which are themselves Legion objects. In addition to supporting object-mandatory methods, *class objects* support a set of class-mandatory methods that implement instance creation, location, migration, and deletion. Thus, classes provide

basic object management services, but they also act as policy makers for their instances. For example, classes have final authority in placing their instances, and can thus implement various performance and security policies.

Legion defines the interfaces to several special object types that implement traditionally system-level functionality. The activation and deactivation of object instances on physical hardware is managed by *host objects*. Each CPU resource in a Legion system is managed by at least one host object. The persistent storage on which inert Legion objects reside is managed by *vaults*; vaults manage inert Legion objects in much the same way that host objects manage active Legion objects. Finally, *binding agents* resolve LOIDs to object addresses.

Legion provides a variety of programming interfaces on several different levels. Some programmers will use Legion by writing programs in high level parallel languages such as Mentat [6]. The programs will then be transformed automatically by Legion-targeting compilers into Legion object implementations. Other programmers will use Legion by specifying an object interface in an IDL, using an IDL compiler to generate client and server stubs, and then providing the method implementations in a high level sequential language; this process corresponds to the most common method of building CORBA objects. Still other programmers will write PVM [7] code and link it with a Legion PVM library, which will rely on Legion objects for its implementation. Finally, another set of users will require low level programming details, possibly for the implementation of system level objects such as hosts or vaults; these users will be able to program directly to the LRTL API.

3 Mechanisms for flexibility

This section describes several components of LRTL that help make it configurable and extensible. The components enable this flexibility in one of two different ways, (1) by abstracting some type of complexity that is inher-

ent to building software for distributed-object computing, or (2) by providing mechanism designed specifically to allow LRTL to be extended or configured easily. Legion buffers and automatic reference counting (Sections 3.1 and 3.3) fall in the former category; events, program graphs, and implicit parameters (Sections 3.2, 3.4, and 3.5) fall in the latter.

3.1 Legion buffers

An *lbuffer* is the fundamental data container in LRTL. An lbuffer exports operations to read and write data from and to physical storage. Instances of different kinds of lbuffers export the same interface and perform the same basic function, but can have different characteristics from one another. For example, one kind of lbuffer may copy the data it contains into heap-allocated memory, another may simply maintain pointers to the data, and a third may read and write its data from and to a file. Further, lbuffers may also choose to compress or encrypt data, or both. To define its characteristics, each lbuffer contains a *storage*, a *packer*, an *encryptor*, and a *compressor*.¹ Eight bytes of *meta-data* also accompany lbuffers; the meta-data indicates the format in which the data is stored, and the algorithms, if any, that were used to encrypt and compress the data.

The storage associated with an lbuffer determines where and how the data is contained. One type of storage provided in LRTL is *scattered storage*. Scattered storage maintains data as a linked list of pointers to data chunks. Scattered storage can be configured to copy the data into chunks that it allocates, or to maintain pointers to data “owned” by other parts of LRTL. Another type of storage is *persistent storage*, which stores data in a file. Obviously, scattered storage and persistent storage will have very different performance characteristics.

A *packer* determines the data format conversion operations, if any, that are performed on the contained data when it is written to

¹For brevity, we omit descriptions of encryptors and compressors in this paper.

and read from the lbuffer. A packer is the primary mechanism in LRTL for dealing with the fact that machines with different architectures, which store data in different formats (i.e. big vs. little endian, 32-bit vs. 64-bit words, etc.), need to communicate with one another. LRTL currently supports three different equivalence classes of architectures, “Alpha,” “Sparc,” and “x86.” For efficiency reasons, Legion assumes a “receiver makes right” [8] data conversion policy; the sender of a message (i.e. the original creator of an lbuffer) packs the message in its own native format, and the receiver is responsible for converting the data to the format appropriate for the architecture on which it resides.

Legion lbuffers enable the concept of *packable* C++ object classes in LRTL. A class is packable if it exports “pack()” and “unpack()” functions. Both pack() and unpack() take a single reference parameter that names an lbuffer. The pack() function writes the object’s state into the lbuffer parameter in such a way that the unpack() function of an object of the same class can read it out. LRTL’s C++ object classes are made packable for two reasons, (1) so that they can be passed between heterogeneous architectures within an lbuffer, and (2) so that they can be written to a persistent lbuffer to save the state of an object before it is deactivated or migrated.

3.2 Events

One of the fundamental issues in the implementation of a significant piece of software is the definition of how logical components of the software communicate with one another. This is especially important when the software will be extended by its users. The many approaches that exist range from the careful definition of interfaces and interactions, to more ad-hoc approaches that require intimate knowledge of many aspects of the code. LRTL employs a well-understood technology, events, and prescribes how to use it to facilitate flexible and extensible inter-component communication.

The prescription is straightforward. When

component X wishes to communicate with component Y, X announces an *levent*, which contains user-defined data and a tag that denotes an *event kind*. Each event kind has one or more associated *event handlers* (C++ functions) which may be called whenever an event of that kind is announced. Handlers for a particular event kind are given a priority that determines the order in which they are called. Any handler of a particular priority can prevent the execution of handlers with lower priority.

Component X communicates with component Y using levents in four steps: (1) Y registers a handler for the particular event kind that X will announce. This ensures that when X announces the event, Y will receive it. (2) X creates an event using one of the provided event kinds as a template. If data needs to be sent along, then the data is attached to the event as well. (3) X announces the event to an *event manager*, an entity that enqueues events and ensures that its handlers are executed in priority order and only if preceding handlers have not prevented further execution. (4) The event manager dequeues and processes the event by calling its handlers.

Outside of application specific data manipulation, each of these actions requires only one or two lines of C++ code. The entire process is depicted in Figure 1.

The event mechanism provides flexibility in a number of ways. Handlers can be added, modified, and removed, event kinds can be added, and handler priorities can be set and reset to effect the order in which they are processed. One of the primary uses of events in LRTL is a configurable protocol stack, which we describe in Section 4.1.

3.3 Reference counting

LRTL implements a mechanism for automatic reference counting and safe dynamic memory management. The mechanism is intended for heap allocated C++ objects. It keeps track of references to each object that is “shared” by different parts of LRTL, and automatically

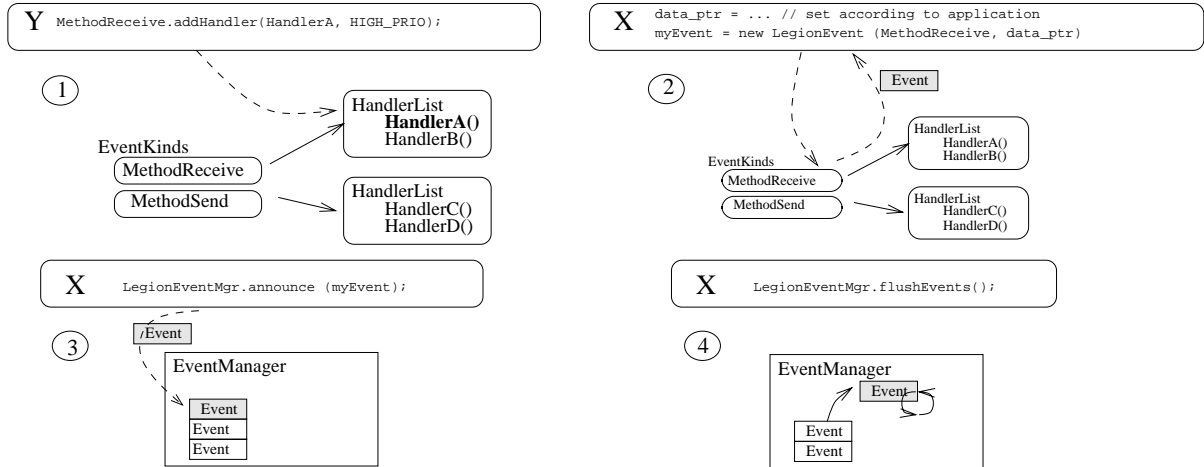


Figure 1: Communication between two components using events: (1) handler registration, (2) event creation, (3) event announcement by the event originator, and (4) handler execution by the event manager.

deletes the object when all meaningful references to it have disappeared. The mechanism is based on *references*, which take the place of C++ pointers, and *reference counting objects*. Each reference counting object is derived from an LRTL-provided base class that enables it to maintain a non-negative integer that indicates the number of references that “point to” that object. When a new reference is made to point to an object, the reference count within that object is incremented automatically. When an reference gets overwritten with another value, or when a local variable reference falls out of scope, the reference count in the object to which the reference points is automatically decremented. When the reference count falls to zero, the object is deleted automatically. All of this happens without any intervention by the programmer or user of references. With a couple of minor exceptions, using the reference counting mechanism is exactly like using pointers to dynamically allocated objects. The main thing to remember is not to delete the object that the reference points to, since that is done automatically.

The decision to include an automatic reference counting mechanism in LRTL was moti-

vated by two observations: (1) memory copies are expensive and often hinder the performance of message passing code, and (2) keeping track of shared pointers and deciding which parts of the code are responsible for deleting which chunks of heap-allocated memory is extremely error prone and difficult to document effectively. The automatic mechanism combines the better performance that comes from avoiding memory copies with the safety and the correctness that comes from not having to worry about managing dynamically allocated memory. Obviously, the automatic reference counting mechanism introduces some overhead over simple pointer copies, but it is cheaper than garbage collection, and we believe the benefits outweigh the costs.

3.4 Program graphs

At a high level, computation is expressed in the Legion system using *program graphs*. A program graph is a data-flow graph whose nodes represent method invocations on Legion objects, and whose arcs represent data dependencies between the method invocations. This computation model is exactly the one described in [5], so we omit a detailed descrip-

tion. Figure 2 shows a simple user program and the resultant data dependencies expressed as a program graph.

The implementation of program graphs in Legion enables two important properties of the Legion system, (1) support for concurrency and parallel processing, and (2) support for graphs as first class objects. The latter aspect is important because some applications may require the specification of a computation in one object and the initiation of that computation in a different object. One conceptually simple way to accomplish this is to support program graphs as first class objects.

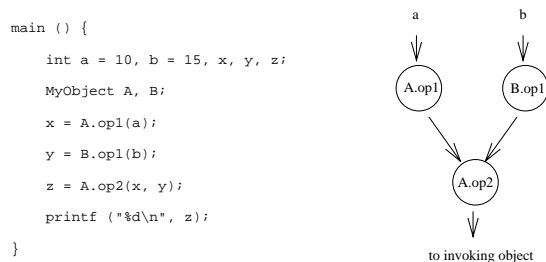


Figure 2: Example user code and the resulting program graph.

3.5 Implicit parameters

In a wide area distributed-object system, it is often desirable and necessary to append meta information to high level entities such as method invocation requests and return results. LRTL uses *implicit parameter lists* to provide this capability. For example, programmers use *implicit parameters*, which accompany all message transmissions, to specify that they want all messages to be encrypted before transmission, or to send context information that indicates how error conditions should be handled and who should be notified when errors occur.

An implicit parameter is a $\langle tag, type, value \rangle$ triple. The tag is a string that names the item, the type is an integer whose value corresponds to one of several well-known parameter types, and the value is a packable entity that contains whatever data is associated with the parame-

ter. Together, the tag and type determine how the value field of the implicit parameter should be interpreted.

LRTL provides two default implicit parameter lists, one for messages and one for method requests. This separation is supported because some activities take place at a message level granularity, while others occur at a method level granularity. Implicit parameters can be used in a variety of ways for a variety of purposes. In Sections 4.2 and 4.3 we explore some of these uses.

4 Examples of use

In this section, we provide several examples of how the mechanisms described in Section 3 have been or can be used to expand or configure LRTL.

4.1 The Legion protocol stack

Legion maintains a configurable protocol stack to handle location independent method requests by transforming them into machine and process specific messages, and to perform the reverse operation on the receiving side. Figure 3 depicts a high level view of the stack. Since one of Legion’s design goals is to provide flexibility and extensibility within a working implementation, we implemented the configurable protocol stack using the event mechanism described in Section 3.2. Figure 4 shows the shape of this implementation.

As an example, consider how an invoked (receiving) object passes messages up the communication protocol stack. When the *data delivery* layer of the receiver (the layer of the protocol stack that abstracts various low level communications APIs such as BSD sockets) receives a raw message off the wire, it creates a *message-received* event and attaches the raw message as the data part of the event. The *message layer*, the layer of the protocol stack that interprets raw messages to construct *Legion message* data structures, registers a handler with the message-received event kind. When the message layer handler is invoked, it extracts the raw message data from

the event, constructs a Legion message data structure, and attaches this Legion message as the new data associated with the event. Higher layers of the protocol stack (i.e. those with lower priority handlers for the message received event kind) will then be invoked in turn, and will manipulate the Legion message contained in the event.

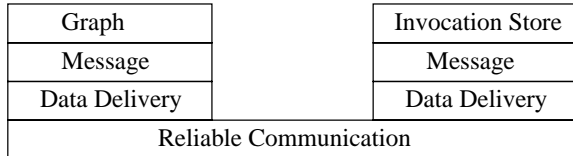


Figure 3: LRTL’s default configurable protocol stack.

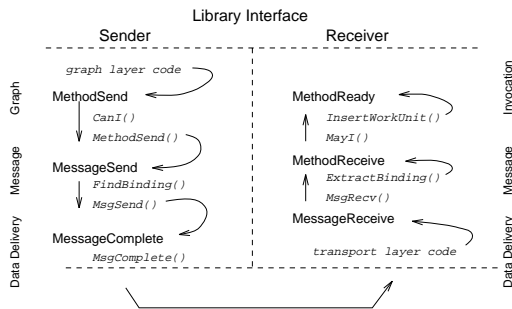


Figure 4: Instantiation of LRTL’s default protocol stack using events.

4.2 Encryption and decryption

Encryption and decryption are being implemented in LRTL using a combination of the mechanisms described in Section 3. An implicit encryption parameter is given a value indicating the level and kind of encryption that was performed on the message. The set of values that the encryption parameter can take includes NoEncryption, RSASignatures, and RSAEncryptionDESX. An encryption event handler executes as the last handler before the message is delivered to the data delivery layer for transport to the destination object. On the receive side, a decryption handler is invoked on a message-received event.

4.3 Message and method logging

One useful feature of a wide area distributed object system is method- or message-level logging. Logging is important for a variety of reasons, including debugging, performance tuning, and replaying computation.

Suppose we have an object A that wants to know how many messages are sent on its behalf as part of a computation that it has initiated. We assume that there is already a message logger object in existence, or that A creates the message logger before the computation of interest begins (Figure 5). Object A sets up a logging-on implicit parameter, and specifies which message logger object should receive the notification. This specification is contained in the value part of the implicit parameter either as the LOID of the object, or as a self contained program graph that will be executed by the receiving object when the message arrives. In either case, the receiver must register an event handler that looks specifically for a logging-on parameter, and if present, takes the appropriate action. Of course, both sender and receiver must agree on how to name the implicit parameter and how to interpret the attached value. Each message sent between objects (solid lines in Figure 5) carries an implicit parameter identifying a message logger to notify once the message is received. Dotted lines from each object to the message logger represent this notification. If B performs method calls on A’s behalf, then the implicit parameters that arrived with A’s method invocation should be propagated accordingly. LRTL’s implementation ensures that the correct implicit parameter lists are propagated. So if A calls B which calls C, then B attaches the implicit parameters it received from A, not its own default parameters.

Another possible use of implicit parameters is for visualization. For example, a graphical display might show the locations and communication patterns of a set of objects. Figure 5 illustrates one method for doing so. The message logger uses the message traces to drive a graphical display showing object locations and interactions. The locational informa-

tion needed to drive this geographical display might be specified by the communicating objects themselves or by some other entity that has knowledge of LOID to object address mappings.

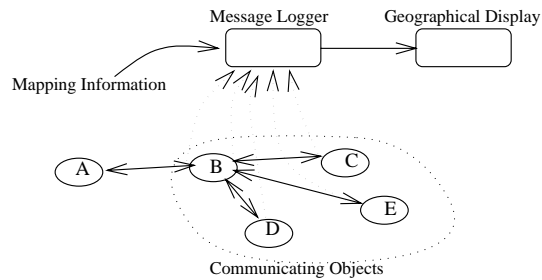


Figure 5: Using implicit parameters to provide logging and visualization information.

4.4 Active messages

The active messages programming model [9] is a message passing scheme that is intended to integrate communication and computation in order to increase the compute/communicate overlap, thereby masking the latency of message passing and increasing performance. The basic idea behind active messages is simple. Messages are prepended with the address of a handler routine that is automatically invoked upon receipt of the message. Active messages are not buffered and explicitly received, as is common with standard message passing interfaces. Instead, the receiving process invokes the handler routine specified for the message immediately upon message arrival. The handler may execute as a new thread of control, or may interrupt the running computation. The job of the active message handler is to incorporate the received message into the on-going computation.

In the current Legion implementation, when a method invocation is received by an object, it is normally inserted into a database of ready method requests (the *invocation store*), and a *method-ready* event is announced. Thus, it is entirely possible that the invocation request

may be buffered for some time before it is handled.

A Legion version of active messages could be constructed by making Legion methods serve as message handlers, and by replacing the method-ready event handler with one that creates a new thread to service incoming methods instead of buffering them in an invocation store. This method-ready event handler would need to be registered with the method-ready event kind at object startup.

In some ways, the model supported here is more general than the traditional active messages model. For example, if a method (i.e. a handler) required two messages from different sources for activation, this requirement would be enforced in the *invocation matcher*,² the level just below the invocation store. Programs might be entirely composed of standard single-token active messages, providing a programming model as flexible as the original. On the other hand, programs might also include multi-token active messages, for a more general programming model that might best be called “active methods.”

4.5 Path expressions

The various method invocation semantics covered thus far have offered a “one size fits all” concurrency control mechanism. A more general approach to customizing the concurrency control requirements of operations on an object can be designed based on path expressions [10]. Path expressions permit the programmer to specify (1) sequencing constraints among operations; (2) selection (mutual exclusion) between operations; and (3) allowable concurrency between operations. These concurrency control primitives let programmers maintain the sequential consistency of their programs and at the same time indicate potential concurrency to a run-time environment.

Path expression based method sequencing could be implemented for Legion objects, again

²In Legion, the parameters to a method may be coming as messages from several different objects. The invocation matcher collects these messages and assembles them into a complete invocation request.

by utilizing the inherent configurability of LRTL’s protocol stack. As with active messages, supporting different method invocation semantics requires replacing the method-ready event handler. In this case, the method ready handler must examine the function numbers of available operations and determine if they may be safely fired given the ordering constraints specified by the program’s path expressions. If a method can be fired safely, a new thread is created and allowed to run, starting at the entry point for the given member function (as in the active messages case). On the other hand, if the ordering constraints of a newly arrived method are not satisfied, the method must be buffered (e.g. in an LRTL-provided invocation store) and later extracted and fired when safe. This need to defer the firing of methods requires that code be executed whenever methods complete execution. One possible way to satisfy this requirement is to use the *method-done* event kind, and announce events of this kind when methods complete execution. A handler for method-done events can then be used to re-evaluate buffered methods with respect to the path expression ordering constraints whenever a running operation completes.

The result of this configuration of LRTL would be a run-time environment that could be used to support path expression style method invocation semantics. This run-time system might be used explicitly by a programmer, or might be the target of a compiler that accepted a Path-Pascal like implementation language for Legion methods.

5 Related work

The majority of the techniques used by LRTL have been successfully applied in other systems. For example, events are well understood and have been used for a variety of purposes, from graphical user interfaces such as X Windows, to the construction of operating system kernels such as SPIN [11]. The versatility of an event-based abstraction resides in its ability to decouple communication between various

components of a system both temporally and spatially. This feature is essential to LRTL, which is required to support a variety user requirements. A diverse set of dynamically configurable user and system components must interoperate— the event paradigm serves as the “glue” that binds these components together.

The design of LRTL was influenced by the x-Kernel project [12], which demonstrated the importance of unifying inter-layer communication within a single framework. While the emphasis was on building network protocols, the lessons are very much applicable to our goal of supporting diverse functionality in LRTL. The x-Kernel does not use an event-based abstraction for inter-layer communication though recent extensions [13] have added an event-based model for composing components within a layer. The result is a two-tiered framework wherein communication between components is no longer unified.

While the SPIN and x-Kernel projects seek to provide flexibility and extensibility at the kernel level, LRTL operates entirely in user space. This design decision reflects the Legion philosophy that we cannot mandate changes to underlying host operating systems. Except for low-level components in the transport layer, LRTL does not contain operating system specific code, and hence is portable to many operating systems and hardware architectures.

Message logging is an important aspect of distributed and parallel systems [14, 15]. While this feature has typically been hard coded into system interfaces (e.g. as in the PVM message logger facility [16]), Legion program graphs and implicit parameters provide a flexible and generic way of specifying information propagation between objects for the purpose of message logging, program visualization, and other meta-applications.

6 Summary

LRTL, in conjunction with the Legion core system objects and class system, provides a flexible and extensible foundation for constructing distributed meta-systems based on objects.

A prototype implementation of the core LRTL mechanisms as described in this paper has been built and is currently being used as the basis for experimental meta-systems construction and application programming. Recent Legion systems have been run across multiple institutions spanning the continental United States, and prototype applications have included domains such as multimedia, biochemistry, electrical engineering.

Future work on LRTL will provide advanced configurations to support alternate method invocation semantics, additional security features, and message logging as described in Section 4. While the core mechanisms and basic LRTL configuration have been implemented, advanced add-on features such as these are still under active development.

References

- [1] A. S. Grimshaw and Wm. A. Wulf. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, 1997.
- [2] M. J. Lewis and A. S. Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, pages 551–561, Syracuse, NY, 1996.
- [3] A. S. Grimshaw, J. B. Weissman, E. A. West, and E. Loyot. Metasystems: An Approach Combining Parallel Processing And Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 21(3):257–270, 1994.
- [4] A. J. Ferrari, M. J. Lewis, C. L. Viles, A. Nguyen-Tuong, and A. S. Grimshaw. Implementation of the Legion Library. Technical Report CS-96-16, Department of Computer Science, University of Virginia, 1996.
- [5] A. S. Grimshaw, J. B. Weissman, and W. T. Strayer. Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing. *ACM Transactions on Computer Systems*, 14(2), 1996.
- [6] A. S. Grimshaw. Easy-to-use Object-Oriented Parallel Programming with Mentat. *IEEE Computer*, pages 39–51, May 1993.
- [7] G. A. Geist, A. Beguelin, J. Dongarra, W. Jian, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing. *The MIT Press*, 1994.
- [8] H. Zhou and G. A. Geist. Receiver Makes Right Data Conversion in PVM. In *Proceedings of the 14th International Conference on Computers and Communications*, pages 458–464, Phoenix, AZ, March 1995.
- [9] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the International Symposium on Computer Architecture*, pages 256–266, 1992.
- [10] R. H. Campbell and A. N. Habermann. The Specification of Process Synchronization by Path Expressions. *Lecture Notes in Computer Science*, 16:89–102, 1973.
- [11] P. Pardyak and B. N. Bershad. Dynamic Binding for an Extensible System. In *Second USENIX Symposium on Operating System Design and Implementation*, 1996.
- [12] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, January 1991.
- [13] N. T. Bhatti and R. D. Schlichting. A System For Constructing Configurable High-Level Protocols. In *SIGCOMM '95 Symposium on Communications Architectures and Protocols*, August 1995.
- [14] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. *International Journal of Supercomputing Applications* to appear.
- [15] B. Topol, J. T. Stasko, and V. S. Sunderam. Monitoring and Visualization in Cluster Environments. Technical Report GIT-CC-96-10, College of Computing, Georgia Institute of Technology, March 1996.
- [16] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM (Parallel Virtual Machine): A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.