Implementation of the Legion Library

Adam J. Ferrari, Mike Lewis, Charles L. Viles
and
Anh Nguyen-Tuong, Andrew S. Grimshaw

The Legion Research Group[1]
Technical Report CS-96-16
Department of Computer Science
University of Virginia
DRAFT: 11/26/96

Abstract

Legion is a multi-year effort to design and build scalable meta-computer software. Legion provides the software "glue" for seamless integration of distributed, heterogeneous hardware and software components. Much of this "glue" is provided in Legion's run-time library. In this document we provide a detailed description of the implementation of the Legion run-time library. The library is designed and implemented as a layered, configurable software protocol stack. This design, coupled with an event-based mechanism for inter-layer communication, enables a considerable flexibility and extensibility. In this document, we first present the library interface and show how the library can be used "as is", without internal modification. We then show how to modify or add functionality to the library. Finally, we provide qualitative descriptions of how the library could be extended to encompass programming styles and methods besides the object-oriented, macro data-flow style that is Legion's initial target.

1. Introduction

The Legion project at the University of Virginia (*http://legion.virginia.edu/*) attempts to facilitate the integration of disparate, distributed, and heterogeneous system components into a single virtual meta-computer. The Legion project does not specifically prescribe how system components should be implemented - only the functionality of these components and through this functionality, some of the interactions between them. However, our goals are much larger than the simple specification of component functionality: some issues can only be examined with a working system. From our perspective, the importance of a working implementation cannot be overstated. In this paper we describe the initial implementation of the Legion Run-time Library (hereafter, the "Library").

This paper is specific in its thrust in the following ways. We focus narrowly on the design and implementation of the Library. We do not attempt to describe the Legion project in its entirety

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

(see [4]) nor do we describe a complete implementation of a system that will achieve all of Legion's objectives. The purpose of this paper is to describe our initial implementation experience with the Library and to document its state as of mid-November 1996. *The implementation will continue to evolve*.

This report targets those who wish to use the Library to build their own services/applications or those who intend to modify or augment the Library in some fashion. Accordingly, this document is more a manual than a paper. After a short introduction into the Legion project and its goals, we introduce several key concepts of the Library and its implementation in Section 2. These concepts are necessary to understand how the library works and thus to unlock the flexibility it provides. In Section 3, we explain in detail how to use the Library as is, i.e without internal modification. Section 4 describes how to modify or extend the Library's capabilities. To illustrate the Library's flexibility, we describe how it might be modified to accommodate simple message passing, active messages, and multiple threads. The Appendix gives a complete translation of a C++ based Legion object that uses the Library to accept method invocations and to return results. The Appendix also includes detailed interfaces for several of the important C++ classes in our implementation of the Library.

On-line documentation about the implementation (including code) is available at *http://legion.virginia.edu/*.

1.1 The Legion Vision

Legion is an ambitious project. We hope to revolutionize high-performance distributed computing by providing unified access to distributed, heterogeneous computational resources where such access has previously been impossible or extremely difficult. Our overall vision is shaped by the following ten objectives:

- Scalability
- High Performance
- Ease of use
- Extensibility
- Persistent object space
- Security
- Multi-language support
- Exploitation of resource hetero-geneity
- Site Autonomy
- Fault-tolerance.

The Legion Library implementation is a key component for achieving *extensibility*, and is vital in a number of other areas, especially *security*, *ease of use*, and *fault-tolerance.* More detail on the Legion vision and objectives can be found in [4].

2. Basics

This section describes some of the most fundamental aspects of the Library. The modules, classes, and concepts described in this section pervade the library's implementation, and must therefore be

familiar to you before we describe the rest of the library.

2.1 LegionBuffers and packability

A *Legion buffer*, represented by the C++ object class `LegionBuffer`, is ***the*** fundamental data container in the Legion Library. A Legionl buffer exports operations to read and write data from and to a logical buffer. Instances of different kinds of `LegionBuffers` export the same interface and perform the same basic function, but can have different characteristics from one another. For example, one `LegionBuffer` 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, `LegionBuffers` may also choose to compress or encrypt data, or both. To define its characteristics, each `LegionBuffer` contains a `LegionStorage`, a `LegionPacker`, a `LegionEncryptor`, a `LegionCompressor`, and `MetaData`.

2.1.1 LegionStorage

A `LegionStorage` determines how the data is stored. One type of `LegionStorage` provided in the Library is `LegionStorageScat`, which stores its data as a linked list of pointers to chunks that contain the data. A `LegionStorageScat` can be configured to copy the data into chunks that it allocates, or to maintain pointers to data "owned" by other parts of the library. Another type of `LegionStorage` is a `LegionStoragePersistent`, which stores data in a file. Different `LegionStorages` will have unique characteristics in terms of performance and even operation. For instance, selecting a `LegionStorageScat` that does not copy its data may be efficient, but extra care must be taken not to delete data "out from under" the buffer, leaving dangling pointers.

Although a `LegionStorage` exports functions to directly access a buffer's data (Section 8.4.2), these functions typically should not be called directly by the user of a `LegionBuffer`. Instead, the user should call the functions in the interface provided by the `LegionPacker` portion of a `LegionBuffer`.

2.1.2 LegionPacker

A `LegionPacker` determines the data format conversion operations, if any, that are performed on the contained data when it is written to and read from the buffer. `LegionPacker` is the primary mechanism in the Library 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. The Library supports three different equivalence classes of architectures, which we call `Alpha`, `Sparc`, and `x86`. For efficiency reasons, Legion assumes a "receiver makes right" [11] data conversion policy, where the sender of a message (i.e. the original creator of a `LegionBuffer`) packs the message in its own native format, and the receiver of the message is responsible for converting the data to the format appropriate for the architecture on which it resides. Thus, the Library provides six different types of packers of the form `LegionPackerX2Y`, where X and Y are two different members of {`Sparc`, `Alpha`, `x86`}. None of the six packers does any conversion when writing to the buffer, but each converts data in the appropriate way when reading from the buffer. `LegionPackerX2Y` is appropriate to use when the data is stored in format X, but the machine currently holding the buffer uses format Y. When the data is already stored in the appropriate format for the architecture on which it is being read, or when data conversion is done outside a `LegionBuffer`, `LegionPackerDefault`

can be used to ensure that no data conversion will take place on reads from the buffer.

The interface provided by a LegionPacker consists of functions of the form

```
put_zzz(zzz *source, int how_many)
```

and

```
get_zzz(zzz *source, int how_many)
```

where `zzz` names a basic C++ data type, i.e. `char`, `short`, `ushort`, `int`, `long`, `ulong`, `float`, or `double`. Thus, a `LegionPacker` exports `put_char()`, `get_char()`, `put_short()`, `get_short()`, etc. "`Source`" points to an array of "`how_many`" instances of type `zzz`. The `put_zzz()` function copies this data into the buffer after first performing the appropriate data conversion operation if it is necessary for the type of `LegionPacker` that is instantiated. The `get_zzz()` function fills the complementary role, first converting the data and then copying it into `source`.

The code example in Figure 1 illustrates the use of the `LegionPacker` interface of a `LegionBuffer`. It also uses the `seek()` function—which is actually part of the `LegionStorage` interface (Section 8.4.2)—to "rewind" the logical position within the buffer back to the beginning.

2.1.3 LegionEncryptor

A `LegionEncryptor` determines the encryption and decryption algorithms, if any, that are applied to the data. The Library currently provides only `LegionEncryptorDefault`, which defines empty encryption and decryption operations.

2.1.4 LegionCompressor

A `LegionCompressor` determines the compression and decompression algorithms, if any, that are applied to the data. The Library currently provides only `LegionCompressorDefault`, which defines empty compression and decompression operations.

2.1.5 MetaData

Eight bytes of "meta-data", three of which are currently used, are associated and carried with each `LegionBuffer`. The meta-data indicates the format in which the data is stored, and the algorithms, if any, that were used to encrypt and/or compress the data. The meta-data fields and values that are supported by the Library are defined in the Appendix.

2.1.6 Packability

`LegionBuffers` enable the concept of "packable" classes in the Library. A class is packable if it is derived from the abstract base class `LegionPackable` (not to be confused with `LegionPacker`), and therefore exports the functions

```
pack(LegionBuffer &lb)
```

and

```
unpack(LegionBuffer &lb)
```

Both `pack()` and `unpack()` take a single reference parameter that names a `LegionBuffer`. The `pack()` function of a packable class writes its state into the `LegionBuffer` in such a way that the `unpack()` function of the same class can read it out. The state is typically written to and

read from the buffer using the `LegionPacker` part of a `LegionBuffer` interface, which encapsulates the data format conversion operations.

Suppose class `Alpha` (Figure 2 - top) is packable and exports the C++ == operator. If `Alpha` is implemented correctly, then the code in Figure 2 - bottom should print “OK”.

Classes are made packable for two reasons, (1) so they can be passed between heterogeneous architectures within a `LegionBuffer`, and (2) so they can be written to a `LegionBuffer` as part of a “save state” operation (Section 3.6). Since these two operations are so fundamental and common to the Library, many parts of the Library only operate on packable classes. For instance, many of the templated data structures are packable, and require the ability to call the `pack()` member function of the contained data. Further, on a Legion object method invocation, each function parameter is passed within a `LegionBuffer`, so the easiest and “best” way to allow an object to be a parameter of a function is to make its class packable.

Making a class packable, i.e. implementing the `pack()` and `unpack()` functions for a class, is generally quite easy. The `LegionBuffer` exports storage operations for the primitive C++ types (Section 2.1.2). For complex types, when a class X contains an instance of another packable

```
// Use the default constructor to declare a new empty
// LegionBuffer, which will be configured to contain the
// default storage, packer, encryptor, and compressor.
LegionBuffer buffer;

// Declare and initialize data to write into the buffer.
char *in_string = "Hello World";
int in_int_array[5] = {100, 101, 102, 103, 104};

buffer.put_char(in_string, 11);      // Insert the string.
buffer.put_int(in_int_array, 5);     // Insert the integers.
buffer.put_char(&in_string[6], 1);   // Insert a single char.
buffer.put_int(&in_int_array[3], 1);// Insert a single int.

// "Rewind" the buffer back to the beginning so we can read
// out the data we just wrote in.
buffer.seek(BEGINNING, 0);

// Declare data structures to read the buffer data into.
char out_string[12];
int out_int_array[5];
char out_char;
int out_int;

// Data must be read out in the same order it was put in,
// but not necessarily the same way.
buffer.get_char(out_string, 6);      // Read 1st 6 chars.
for (j = 6; j < 11; j++)                // Read the next 5,
   buffer.get_char(&out_string[j], 1); // one at a time.
out_string[11] = '\0'
buffer.get_int(out_int_array, 5);    // Read the integers.
buffer.get_char(&out_char, 1);       // Read the single char.
buffer.get_int(&out_int, 1);         // Read the single int.
```

Figure 1. Using the LegionPacker interface.

```
// A generic, packable class
class Alpha : public LegionPackable {
   private:
      // private data
   public:
      // constructors and member functions

      int operator==(Alpha &other_alpha);
      pack(LegionBuffer &lb);
      unpack(LegionBuffer &lb);
};
```

```
Alpha *A, B;
LegionBuffer buffer;

A = new Alpha(/* appropriate initial values */);
A->pack(buffer);
buffer.seek(BEGINNING, 0);
B.unpack(buffer);

// Make sure we unpacked into B exactly what we packed in A
if (*A==B)
   printf("OK\n");
else
   printf("Bad news\n");
```

Figure 2. Declaration and use of a packable class.

class Y, X's `pack()` function can simply contain a call to Y's `pack()` function. Thus, if Y is packable, X does not need to know the data types that Y contains in order to pack Y as part of X's state. Consider the simple example of a templated array class depicted in Figure 3. Notice that the `Array` class can be made packable even though it doesn't know the type of the elements it contains. `Array` only requires that the contained elements are themselves packable.

```
template<class T>
class Array : public LegionPackable {
   private:
      T *array_data;
      int num_elements;
   public:
      // "unpack" construct
      Array(LegionBuffer &lb) {
        unpack (lb);
      }

      // Other constructors, functions, and destructor go here

      // Pack Array up. Assumes the elements are themselves packable
      int pack(LegionBuffer &lb) {
         lb.put_int(&num_elements, 1);
         for (int j=0; j<num_elements; j++)
            array_data[j].pack(lb);
      }

      // Unpack the array in the same order they were packed.
      int unpack(LegionBuffer &lb) {
         if (array_data)
            delete array_data;

         lb.get_int(&num_elements, 1);
         array_data = new T[num_elements];

         for (int j=0; j<num_elements; j++)
            array_data[j].unpack(lb);
      }
};
```

Figure 3. A packable template class whose data members are themselves packable.

`LegionBuffer` is itself a packable class. Thus, one `LegionBuffer` can be contained (packed) in another. This is shown at the top of Figure 4. If data is packed as a LegionBuffer, it should be unpacked as one. Thus the data that was packed in Figure 4 (top) ***cannot*** be unpacked correctly as using the code in Figure 4 (middle). This code will compile and run, but it will not have the desired effect of unpacking the 10 characters—"`HelloWorld`"—that were packed into the buffer. This is because `LegionBuffers` prepend "user data" with meta-data. Therefore, `hello_world_buf` contains meta-data at the beginning of the buffer, and between "`Hello`" and "`World`". The correct way to unpack the data is given in Figure 4 (bottom).

```
LegionBuffer hello_buf;
char *hello = "Hello";
hello_buf.put_char(hello, 5);

LegionBuffer world_buf;
char *world = "World";
world_buf.put_char(world, 5);

LegionBuffer hello_world_buf;
hello_buf.pack(hello_world_buf);
world_buf.pack(hello_world_buf);
```

```
char hello_world[11];
hello_world[10] = '\0';

hello_world_buf.seek(BEGINNING, 0); // "Rewind" the buffer.

// Try (unsuccessfully) to unpack all 10 characters at once.
hello_world_buf.get_char(hello_world, 10);
```

```
// Declare two separate LegionBuffers for unpacking the two
// buffers that were packed into hello_world_buf.
LegionBuffer out1, out2;

hello_world_buf.seek(BEGINNING, 0); // "Rewind" the buffer.
out1.unpack(hello_world_buf); // Unpack hello_buf into out1.
out2.unpack(hello_world_buf); // Unpack world_buf into out2.

char hello_world[11];
hello_world[10] = '\0';

out1.get_char(hello_world, 5);      // Unpack "Hello".
out2.get_char(&hello_world[5], 5);  // Unpack "World".

// This line will print "HelloWorld".
printf(hello_world);
```

Figure 4. Packing a Legion Buffer into another Legion Buffer (top). An incorrect (middle) and correct(bottom) method for unpacking are given as well.

`LegionBuffers` pack and unpack their bytes raw (without data format conversion) so that each buffer maintains its own meta-data. This means, for example, that a `LegionBuffer` created on an `x86` architecture can be contained in a `LegionBuffer` whose other data is in `Alpha` format. When the contained buffer is unpacked, a `LegionBuffer` with appropriate data conversion operations will be instantiated; when the bytes are read out, the data will wind up in the correct format for the architecture of the machine on which the data resides.

2.2 Reference Counting and Memory Management

The template class `UVaL_Reference`, in concert with the class `UVaL_ReferenceCountingObject`, is the Library's 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 the library, and automatically deletes the object when all meaningful references to it have disappeared. Each reference counting object—i.e. each instance of a class derived from `UVaL_ReferenceCountingObject`—maintains an integer that indicates the number of `UVaL_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 a `UVaL_Reference` gets overwritten with another value, or when a local variable `UVaL_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 automatically deleted. All of this happens without any intervention by the programmer or user of `UVaL_References`.

The decision to include an automatic reference counting mechanism in the Library was motivated 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. Hopefully the automatic mechanism will combine the better performance that comes from avoiding memory copies with the safety and 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. We believe the benefits outweigh these costs.

To be a “casual user” of `UVaL_References`, you need only remember one simple rule of thumb and two simple exceptions:

UVaL_Reference Rule of Thumb:

> *Read “`UVaL_Reference<X> t`” as “`X *t`” and then treat the variable `t` exactly as if it were in fact a C++ pointer to class `X`.*

Just about every operator that is legal on a pointer to a C++ object has been overloaded to work correctly for `UVaL_References`. The example in Figure 5 shows that `UVaL_References` can be used just as C++ object pointers would be. The implementation of the `MyRCO` class in Figure 5 is unimportant beyond the fact that it is derived from `UVaL_ReferenceCountingObject` and implements the functions that are used to illustrate the point.

Exceptions to the Rule of Thumb:

1. Never delete the memory that a `UVaL_Reference` “points to.” The memory will be deleted when all references to the memory have been overwritten or go out of scope.

2. Do not use a `UVaL_Reference` alone as a boolean. “`if (t)...`” will not compile, but “`if (t != NULL)...`” and “`if (!t)...`” will work as expected.

`UVaL_References` can also refer to non-heap-allocated memory, i.e. global and local variables. To insure that these objects are not automatically explicitly deleted by the mechanism, the

```
// Definition and implementation of class MyRCO, which is
// a reference counting object by virtue of
// being derived from class UVaL_ReferenceCountingObject.
class MyRCO : public UVaL_ReferenceCountingObject {
   private:
      int contained_val;
   public:
      MyRCO() {contained_val = 0;}
      MyRCO(int val) {contained_val = val;}
      int set_value(int val) { return (contained_val = val);}
      int get_value() {return (contained_val);}
      int operator==(MyRCO &other_rco) {
         return (contained_val == other_rco.contained_val);}
      int operator!=(MyRCO &other_rco) {
         return (contained_val != other_rco.contained_val);}
      ~MyRCO() {printf("Destructor called\n");}
};

// Create three new reference counting objects, pointed to by
// variables a, b, and c.  Notice that type (MyRCO *) is
// automatically cast correctly to type UVaL_Reference<MyRCO>.
UVaL_Reference<MyRCO> a = new MyRCO(1);
UVaL_Reference<MyRCO> b = new MyRCO(2);
UVaL_Reference<MyRCO> c = new MyRCO(3);

// Show that * and -> work just like pointers.
a->set_value((*a).get_value()); // no change to the object

c = a;
// The object to which c originally pointed now has no more
// references to it. Therefore, that object's destructor will
// be called automatically.  The object to which a points,
// now has two references to it, a and c.

a = b;
// The object to which a pointed is not automatically deleted
// because c still points to it.

// All of the print statements below will be executed.

// Comparing objects is still different from comparing pointers.
if (*a == *b) printf("a and b refer to objects whose values are ==.\n");
if (*a != *c) printf("a and c refer to objects whose values are !=.\n");
if (a == b) printf("a and b point to the same object.\n");
if (a != c) printf("a and c do not point to the same object.\n");

// Make a and c point to objects that contain the same value.
c->set_value(a->get_value());

if (*a == *c) printf("Now a and c point to objects whose values are ==\n");
if (a != c) printf("a and c still do not point to the same object.\n");
```

Figure 5. Example declaration of a reference counting object and its use with UVaL_References.

programmer should call the function `makeNonHeapReference()` on the reference.

2.3 Legion Object Identifiers (LOIDs)

Naming in Legion occurs at two levels. All objects are named by a *Legion Object Identifier* (LOID), but Legion will use standard protocols and the communication facilities of host operating systems to support communication between Legion objects. Since LOID's have meaning only at the Legion level, Legion provides a mechanism by which LOID's can be bound to "low-level" names—object addresses—that have meaning to the underlying protocols and communication facilities. A *Binding* is an <LOID, object address> pair. Binding <X, Y> indicates that the object named by LOID X currently resides at the physical address indicated by object address Y. The Library provides classes for LOID's, object addresses, and bindings. A majority of library users will deal with naming only at the level of LOID's, leaving object addresses and bindings to the low-level library code. Therefore, we describe only the `LegionLOID` class in detail in this section.

2.3.1 LegionLOID

In the Library, a LOID is represented by the class `LegionLOID`. A LOID contains a *type* field, followed by a variable number of variable size fields. The first field indicates the Legion *domain* in which the LOID was created, the second field is the *class identifier*, the third field is the *instance number*, the forth is a *public key.* The type, domain, and class identifier fields must be present. An empty instance number indicates that the LOID names a Legion class object, and empty public keys are allowed for less security conscious objects. A LOID may be appended with any number of other fields, each of whose size and meaning are dependent on the LOID type.

`LegionLOID` is intended to be the base class for derived LOID classes that implement different types of LOIDs. We imagine that the classes that implement each LOID type will enforce different properties having to do with the structure, content, and meaning of the various fields of the LOID. Therefore, `LegionLOID` is not intended to be instantiated directly when the internal fields of the LOID are being used and interpreted; instead the appropriate derived class should be instantiated. So far, we have not implemented any derived classes that enforce special properties about the fields of the LOID. Until we do, the `LegionGeneralPurposeLOID` class can manipulate all the fields of the LOID without enforcing any structure or special meaning to the fields.

To show how to build a general purpose LOID, we give the code for a function called `make_loid()` (Figure 6). As parameters the function takes a string that represents the class identifier of the object, and an integer that represents the instance number of the object. The function builds a LOID that names an object in the domain `UVaL_LegionDomain_Virginia` (a library-defined constant), with class identifier and instance number assigned to the values passed as parameters, and with an empty public key. The function returns a `UVaL_Reference` to a `LegionLOID`.

Assuming that function `make_loid()` exists, the example in Figure 7 shows some of the functionality of LOID's.

```
LegionLOID *
make_loid(char *classname, int instance_number) {
   int i;
   short *fld_sz;  // Will carry the field sizes
   char **fld_val; // Will carry the field values

   // Create new structures to fill in an pass to the constructor.
   fld_val = new (char *)[4];
   fld_sz = new short[4];

   // Field 0 : Domain
   fld_sz[LEGION_DOMAIN_FIELD] = strlen(UVaL_LegionDomain_Virginia) + 1;
   fld_val[LEGION_DOMAIN_FIELD] = UVaL_LegionDomain_Virginia;

   // Field 1 : Class ID
   if (classname != NULL)
      fld_sz[CLASS_ID_FIELD] = strlen(classname) + 1;
   else
      fld_sz[CLASS_ID_FIELD] = 0;
   fld_val[CLASS_ID_FIELD] = (char *) classname;

   // Field 2 : Instance Number
   if (instance_number == 0) {
      fld_val[INSTANCE_NUM_FIELD] = NULL;
      fld_sz[INSTANCE_NUM_FIELD] = 0;
   }
   else {
      char temp_inum[20];
      sprintf(temp_inum,"%d",instance_number);
      fld_val[INSTANCE_NUM_FIELD] = temp_inum;
      fld_sz[INSTANCE_NUM_FIELD] = strlen(temp_inum);
   }

   // Create a new LOID by passing the fld_sz and fld_val
   // arrays to the constructor. This illustrates the use
   // of the most general constructor---but others do exist.
   LegionLOID *rval = (LegionLOID *)
      new LegionGeneralPurposeLOID(1,4,fld_sz,fld_val);

   // Fld_val and fld_sz can be deleted since the
   // constructor copies the data in.
   delete [] fld_val;
   delete [] fld_sz;

   // Return the new LegionLOID.
   return(rval);
}
```

Figure 6. Creating a LegionLOID.

2.4 Events

The Library is implemented as a configurable protocol stack. A layer of the stack communicates with other layers through an event mechanism. The basic idea is straightforward. If layer A

```
LegionLOID *a, *b, *c, d;
// Use the function defined above to create two new LOID's
a = make_loid("Class Name", 0);
b = make_loid("Class Name", 1);

// The copy constructor is overloaded.
c = new LegionLOID(*a);

// == and != operators are overloaded.
if ((*a == *c) && (*a != *b))
   printf("== and != are overloaded\n");

// A special EmptyLOID type is defined.  This is useful for sending
// empty LOID's as parameters and return values.
if (d.is_empty())
   printf("LOID's are empty if they have not been initialized\n");

// LegionLOID's are packable.
LegionBuffer buffer;
b->pack(buffer);
buffer.seek(BEGINNING, 0);
d.unpack(buffer);
f (*b == d)
   printf("LOID's are packable.\n");

// is_class() indicates whether or not an  LOID refers to a
// Legion class object.
if (a->is_class())
   printf("If the class identifier of an LOID is empty\n",
          "   then the LOID refers to a Legion class.\n");
if (!b->is_class())
   printf("If not, then the LOID doesn't refer to a Legion
class.\n");

// Finally, LOID's can be printed neatly, showing only
// the class id and instance number fields.
fprintf(stderr,"LOID b: ");
b->show();
fprintf(stderr,"\n");
```

Figure 7. The functionality of Legion LOIDs.

wishes to communicate with layer B, then A announces a `LegionEvent`. Each `LegionEvent` has a tag that denotes a `LegionEventKind`, the kind of event it is. Each `LegionEventKind` has one or more associated event handlers which may be called whenever an event of that kind is announced. Handlers for a particular `LegionEventKind` are given a priority that determines the order in which the handlers are called when an event of that kind is announced. A `LegionEvent` can carry arbitrary data and this is the method by which data is passed and transformed from layer to layer. So if layer B has registered a handler for the kind of event that layer A has announced, then layer B will get that event. We describe the configuration and flexibility of Legion Events in much greater detail in Section 4.

2.5 Legion Message Database

Each Legion object services requests for member function invocations and returns the results of these invocations. The parameters to these functions as well as the requests themselves arrive in Legion Messages. A complete method invocation may involve composing a number of Legion Messages. Since these messages may arrive from different locations and at different times, partially complete requests are kept in part of the *Legion Message Database* called the *Invocation Matcher*. Once a complete method invocation is composed, it becomes a *Legion Work Unit* and is promoted to the *Invocation Store*. The invocation store is essentially a list of method invocation requests that are ready to be invoked.

Each object has a server loop that continuously checks the invocation store for ready work units, extracts them if they are available, and performs the requested invocation. A partial interface to the C++ class `LegionInvocationStore` is given in Figure 8.

2.6 Legion Program Graphs

A *program graph* represents a set of method invocations on Legion objects and the data dependencies between invocations and objects. The program graph is a data-flow graph whose nodes represent method invocations and whose arcs represent data dependencies between the method invocations. This computation model is exactly the one described in [3], so we omit a detailed description.

Suppose objects A and B each export methods `op1()` and `op2()`. Figure 9 shows a simple user program and the resultant data dependencies. It is clear from the code that the parameters to both `A.op1()` and `B.op1()` are available locally. We call these *constant* parameters. On the other hand, the parameters to `A.op2()` are not available locally because they are the results of method invocations that are executed elsewhere. These are *invocation* parameters

```
// This class implements the "datatbase" for ready
// work units and stores the results from method
// invocations on other Legion objects
class LegionInvocationStore {
public:
  // Accept function invocations for the given fcn
    int enable_ function (int function_number);

    // Check to see if there are any ready work units
    int any_ready();
    int any_ready_for_func(int function_number);

    // Remove the next work unit from the store
    UVaL_Reference<LegionWorkUnit> next_matched();
    UVaL_Reference<LegionWorkUnit> next_matched_for_func(int function_number);
}
```

Figure 8. Selected elements of the LegionInvocationStore interface.

```
// The "user" code
main () {
    int a = 10, b = 15, x, y,
    z;
    MyObject A, B;
    x = A.op1(a);
    y = B.op1(b);
    z = A.op2(x, y);
    printf ("%d\n", z);
}
```

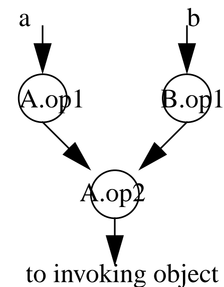


Figure 9. Example user code and the program graph derived from thte data dependencies.

3. Using the Library

This section describes how to use the Library "as is", without internal modification. We begin by describing the class `LegionLibraryState`, which encapsulates start-up and initialization routines and provides a public interface to an object's Legion related state. We then explain how a method invocation is implemented in the Library. Finally, we describe the Library interface from the invoker and invokee perspectives.

3.1 Initialization and Library State

The `LegionLibraryState` C++ object class provides the interface to important parts of an object's Legion related state information. This state include the object's own LOID, the LOID of the object's class, and so on. `LegionLibraryState` also provides implementations of key object control mechanisms such as object creation, activation, deactivation and deletion. Finally, the `LegionLibraryState` interface provides the `ClassOf()` operation, which encapsulates the mechanism by which the class object LOID of a given object can be obtained based on the object's LOID. We will now examine each of these general features of the `LegionLibraryState` class in more detail.

The primary function of the `LegionLibraryState` object class is to encapsulate the internal state of the Legion library and to provide programmers with the public interface to this state. A single global object of the class `LegionLibraryState` named `Legion` is included as part of the Library. Before using any of the Library features (for example, invoking a method on a Legion object), the Library state should be initialized using the `init()` function.

```
Legion.init();
```

This function initializes several different parts of the Library, including the LOID of Legion Class, initializing the Legion program graph layer (Section 3.4), and initializing the Legion invocation matcher and invocation store (Section 2.5). After this call, the object can enable or disable functions in the invocation store, set its own LOID if it needs to, and perform any user initialization that must be performed before any methods are serviced (or any messages are sent or received by the object). After calling `Legion.init()`, the object cannot yet invoke or service

methods. These services require a second initialization phase which is encapsulated in the `AcceptMethods()` member function of the `LegionLibraryState` class.

```
Legion.AcceptMethods();
```

After making this call, the object can both accept and invoke methods. Consequently, basic object control mechanisms such as object creation and `ClassOf()` (which rely on the ability to invoke methods on remote objects) are also enabled. The reason for a two-phase initialization for the Library is simply to de-couple method service configuration from the enabling of method arrival events.

To summarize, the Library initialization should proceed as follows:

1. Call `Legion.init()`. This initializes various data structures in the library.

2. Enable acceptable methods (i.e. function numbers) in the objects invocation buffering mechanism (the Legion invocation store). This is accomplished by calling `LegionInvocationStore.enable_function(int)` for each method the object will be exporting.

3. If methods will be serviced by an event handler, add this event handler for `MethodReady` events. This event handler should contain code to extract work units from the invocation store and call the appropriate method implementations based on incoming function numbers.

4. Call `Legion.AcceptMethods()`. This call initializes the Legion message passing system and notifies the object's creator (i.e. its class) that the object is up and ready to accept method requests.

5. Start the server loop. The details of this depend upon how work units are being extracted from the invocation store. Two options are described in Section 3.5.1.

Once the Legion library state is fully initialized, the object can use `Legion` object to determine its own LOID, the LOID of its vault, the LOID of LegionClass, and so on. For example:

```
UVaL_Reference<LegionLOID> MyLOID, VaultLOID;
MyLOID = Legion.GetMyLOID();
VaultLOID = Legion.GetVaultLOID();
```

In addition to these basic state accessor functions, the `LegionLibraryState` class also provides an interface to a number of key system services such as object creation, activation, deactiva-

tion, and deletion. An example is given in Figure 10.

```
test_object_control(char *class_id)
{
   UVaL_Reference<LegionLOID> testObj1, testObj2;

   // Create an object of the specified class
   testObj1 = Legion.CreateObject(class_id);

   // Create an object of the same class as testObj1
   testObj2 = Legion.CreateObject(testObj1);

   // Test object deactivation and activation
   Legion.DeactivateObject(testObj1);
   Legion.ActivateObject(testObj1);

   // Test object deletion
   Legion.DestroyObject(testObj2);
}
```

Figure 10. Example use of object Legion of class LegionLibraryState.

An object can also use the LegionLibraryState interface to report to its class when it plans to delete itself without having been requested to do so by the class. For example, an object before exiting could execute:

```
fprintf(stderr,"Problem - this object must exit\n");
Legion.DeleteSelf();
exit(1);
```

Beyond object control services, the `LegionLibraryState` class encapsulates the important Legion `ClassOf()` operation, which can be used to determine the LOID of a class object based on the LOID of one of its instances, or based on just a class identifier (that part of an LOID that indicates an object's class). For example:

```
UVaL_Reference<LegionLOID> foo, classOfFoo;

//...set foo to some LOID of interest (not shown)...

classOfFoo = Legion.ClassOf(foo);
```

Note that unlike simple state accessors such as `GetMyLOID()`, object control methods such as `CreateObject()` and `ClassOf()` all result in Legion method invocations, and thus the cost of these member functions are non-trivial.

3.2 Legion messages

Legion objects communicate with one another via method invocation and return values. To invoke methods and return results, Legion objects send messages to one another in a standard Legion message format. A Legion message can carry (1) part (or all) of a method invocation, or (2) the function return value that resulted from an invocation, or (3) a return value for an *out* or *in/out* parameter. Every Legion message contains, in order, source and destination LOID's, a func-

tion number, the number of parameters to expect, a computation tag, a list of parameters, a continuation list, and an environment.

Source LOID: The source LOID names the sender of the message.

Destination LOID: The destination LOID names the object to which the message is being sent.

Function number: The function number field should contain an integer that is packed in the data format of the architecture of the machine from which it was sent. If the message is intended to implement a method invocation, the packed integer should match some function number in the public interface of the object to which the message is being sent. If the message is the "filled-in" value of an out or in/out parameter or a normal return value, then the packed integer should be the constant `LEGION_RETURN_FUNCTION_NUMBER`.

Computation tag: A single method invocation can be split up into several Legion messages which can come from different sources (see Section 3.4). A *Computation Tag* is a long integer, packed in the message sender's data format, that uniquely identifies a computation, or method function invocation. The computation tag should be assigned by the invoker. Messages that carry function return values and filled in `out` and `in/out` parameters should contain the same computation tag as the messages that carried the invocation that generated the results. The invoker matches on the computation tag when awaiting results.

Although a computation tag is simply a long integer, the Library provides a C++ class called `LegionComputationTag` that encapsulates the integer and exports appropriate functions on computation tags. The Library also provides a class called `LegionComputationTagGenerator` that can be used to generate random computation tags. Typical use of these two classes is shown in the example shown in Figure 11.

Parameters to expect: The "parameters to expect" field should contain an integer packed in the sender's data format, that indicates the total number of parameters that are being passed in the function invocation of which this message is part. If the message is part of a return value, this

```
// Declare a new computation tag generator.
LegionComputationTagGenerator gen;

// Declare variables to point to computation tags.
UVaL_Reference<LegionComputationTag> t1;
UVaL_Reference<LegionComputationTag> t2;

// Get the next two tags from the generator.
t1 = gen.next_tag();
t2 = gen.next_tag();

// Print the values of the tags.
printf("Tag 1 is: %d\n", t1->get_value());
printf("Tag 2 is: %d\n", t2->get_value());

// Sample output
// Tag 1 is: 2023717593
// Tag 2 is: 1683023
```

Figure 11. Use of Legion computation tags.

field is ignored.

Parameter list: If the message is part of an invocation, the parameter list contains the values of the parameters contained in the message. The parameter list is packed as an integer that indicates how many parameters are present, followed by the parameters themselves. Each parameter contains an integer that indicates the number of the parameter followed by a `LegionBuffer` that contains the value of the parameter. Return values are passed back in a parameter list as well. The C++ object classes `LegionParameter` and `LegionParameterList` implement parameters. The code in Figure 12 builds a parameter list that contains two parameters, an integer and a 14-element character string.

```
// Declare the variables to be packed into a parameter list.
int int_parameter;
char string_parameter[14];

// Initialize the variables appropriately.
int_parameter = 7;
sprintf(string_parameter,"Hello, World.");

// Create a LegionBuffer to hold the integer parameter.
UVaL_Reference<LegionBuffer> lb1;
lb1 = new LegionBuffer();
lb1->put_int(&int_parameter, 1);

// Create a LegionBuffer to hold the string parameter.
UVaL_Reference<LegionBuffer> lb2;
lb2 = new LegionBuffer();
lb2->put_char(string_parameter, 14);

// Create parameters out of the buffers.
UVaL_Reference<LegionParameter> param1;
param1 = new LegionParameter(1, lb1);
UVaL_Reference<LegionParameter> param2;
param2 = new LegionParameter(2, lb2);

// Create a new parameter list.
UVaL_Reference<LegionParameterList> plist;
plist = new LegionParameterList();

// Finally, insert the parameters into the parameter list.
plist->insert(param1);
plist->insert(param2);
```

Figure 12. Example use of LegionParameterList and LegionParameter.

Continuation list: A *Continuation List* describes where results should get forwarded. A continuation contains a computation tag and result number, which together identify a return value. It also contains the LOID, function number, and parameter number to which the result should be sent. The continuation list describes where all the results of a particular computation should be forwarded. The motivation for continuation lists arises from the macro data-flow programming model that is the initial Legion target. In this paradigm, results from method invocations are sent directly to other invocations that use these results as parameters. These data dependencies are

determined through analysis of the program code. See [3] for more information. LegionProgram Graphs are the representation of these data dependencies and are described in Section 3.4. For further information, refer to the on-line documentation.

Environments: An environment is a list of environment items, each of which is a <*tag*, *type*, *value*> triple. The *tag* is a string that names the item. The *type* is an integer whose value corresponds to one of the well-known environment types, and which determines how the value field should be interpreted. For further information, refer to the on-line documentation.

The `LegionMessage` class implements Legion messages in the Library. Its most useful constructor takes parameters that correspond to all of the constituent parts described above. Thus, an instance of `LegionMessage` can be created as shown in Figure 13.

```
// Declare variables for the LegionMessage's constituent
// parts.
UVaL_Reference<LegionLOID> source_LOID;
UVaL_Reference<LegionLOID> destination_LOID;
int function_number;
int parameters_to_expect;
UVaL_Reference<LegionComputationTag> computation_tag;
UVaL_Reference<LegionParameterList> parameter_list;
UVaL_Reference<LegionContinuationList> continuation_list;
UVaL_Reference<LegionEnvironment> environment;

// Initialize the constituent parts appropriately (not shown).

// Create a new LegionMessage from the constituent parts.
LegionMessage *msg;
msg = new LegionMessage(source_LOID, destination_LOID,
             function_number, parameters_to_expect, computation_tag,
             parameter_list, continuation_list, environment);
```

Figure 13. Example creation of a Legion message.

Although `LegionMessage` provides a mechanism for implementing method invocation in the library, `LegionProgramGraph` (described in tSection 3.4) provides a higher level abstraction that is simpler to use.

3.3 Overview of Method Invocation

Suppose Legion object A (the invoker) invokes a member function on Legion object B (the invokee). This method invocation proceeds through the invoker and invokee as shown in Figure 14. The invoker builds a Legion message containing salient information about the member function invocation. Typically, the Legion program graph interface builds this message. The Legion message is then passed to the *Legion Message Layer*, which binds the LOID of the recipient to a particular address. The binding process is a key aspect of Legion and is described in considerable detail elsewhere ([7]). The outcome of the binding process is a <LOID-ObjectAddress> tuple called a “binding.” The binding represents the logical name and current physical address of the referenced object. The message and its binding are then passed to the *Data Delivery Layer*. The data delivery layer linearizes the message for transport over the wire, uses the object address to create a physical connection to the referenced object, and sends the message.

On the receiving side, the data delivery layer of the destination object unpacks the data back into a new instance of `LegionMessage` and passes the message up to the message layer. The message layer then inserts this message into the *Legion Message Database* (Section 2.5). Conceptually, the Legion message database is divided into two parts. The "bottom" part, called the *Legion Invocation Matcher*, manages the list of partially complete method invocation requests for the Legion object[1]. A method invocation request is partially complete if one or more of it's parameters are missing. The "top" part of the database, the *Legion Invocation Store*, maintains two separate lists. The first list contains complete method invocation requests i.e. requests with a complete parameter set and that have passed all security checks. The second list contains return values the object has seen as a result of it's own method invocations on other Legion objects. So, a Legion message is inserted into the Legion invocation matcher. As soon as all parameters are present, the message becomes a *Legion Work Unit* and is bumped up to the invocation store.

Within each Legion object is a server loop that periodically checks the Legion invocation store for ready work units. A Legion work unit is similar in composition to a Legion message, but by definition contains all information needed to perform a method invocation and forward the results to the proper place. When a work unit is ready, the object removes it from the invocation store and ascertains which it method is being called. The object then calls this method with the supplied parameters, packs up the results (if any) into a Legion message, marks the message as containing a result, and inserts the message into the Legion message layer. The Legion message transport mechanism then takes over.

When the return result reaches its destination, it is handled like any other Legion message until it reaches the invocation store. The invocation store examines the contents of the work unit, realizes that it is a return result and not a method request, and inserts it into the separate list for return values. These values are then available to the original invoking object through the program graph interface.

1. A method invocation request can be partially complete because the parameters to the invocation may be coming from objects dispersed throughout the running Legion system.

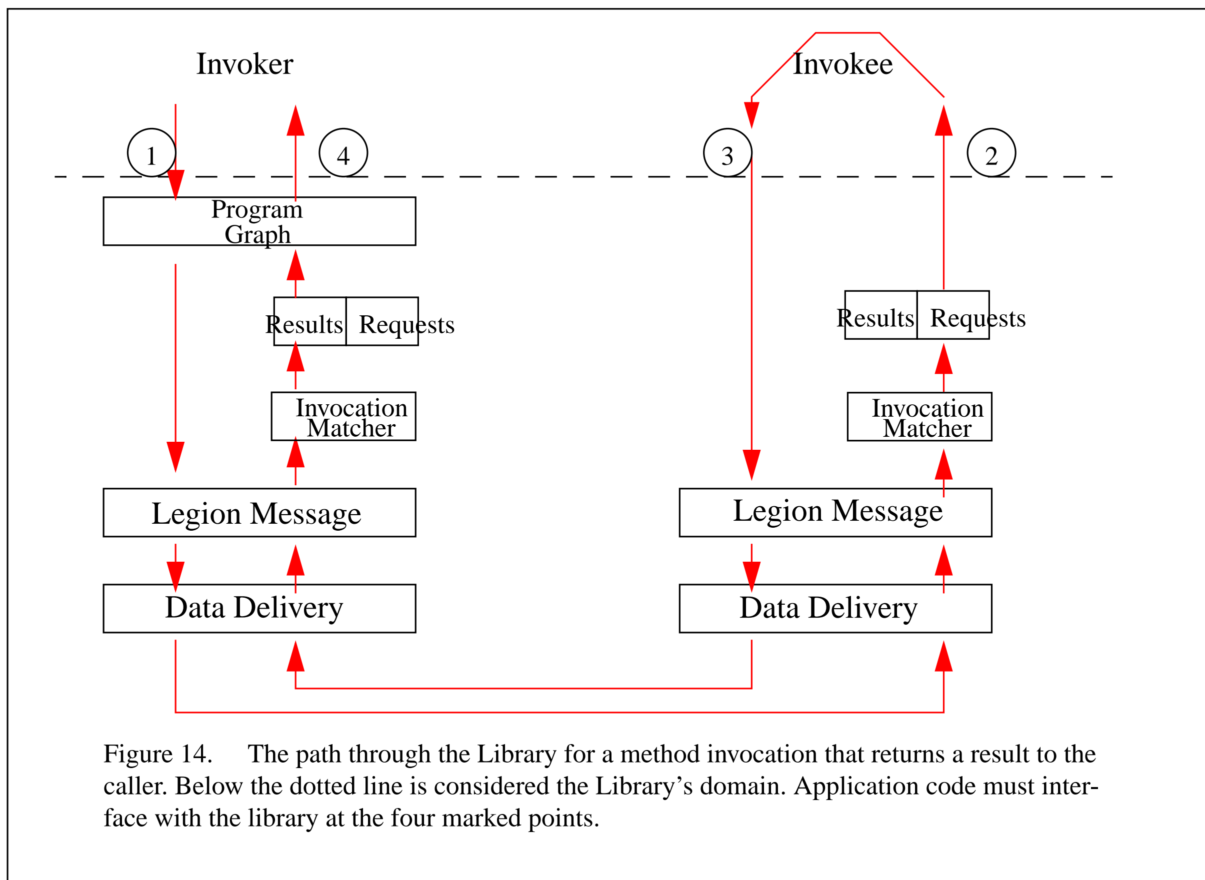


Figure 14. The path through the Library for a method invocation that returns a result to the caller. Below the dotted line is considered the Library's domain. Application code must interface with the library at the four marked points.

An application program must interface with the Library at four different points (Figure 14).

1. Making an invocation request
2. Removing an invocation request for execution
3. Returning the results of an invocation
4. Getting a return result.

We handle 1 and 4 together in Section 3.4, and 2 and 3 in Section 3.5.

3.4 The Invoker: Invocation Requests and Return Results

The most straightforward mechanism by which to make an invocation request is to build a program graph (Section 2.6) using the interface provided by the C++ object class `LegionProgramGraph`. It is also possible to interface directly with the Legion Message Layer if so desired, though we do not document this method here.

Salient parts of the `LegionProgramGraph` interface are given in Figure 15. A fuller description of the interface constituents appears in the Appendix.

```
class LegionProgramGraph {
public:
    // these methods are for making invocation requests
    UVaL_Reference<LegionInvocation>
        add_invocation(UVaL_Reference<LegionInvocation> inv);
    ParameterStatus
        add_constant_parameter(UVaL_Reference<LegionInvocation> target,
                               UVaL_Reference<LegionParameter> parameter,
                               int param_number);
    void add_result_dependency(UVaL_Reference<LegionInvocation> inv,
                               int param_number);
    int execute(LegionInvocation *inv);

    // these methods are for managing return values
    UVaL_Reference<LegionBuffer>
        get_value(UVaL_Reference<LegionInvocation> inv, int param_number);
    int release_value (UVaL_Reference<LegionInvocation> inv, int param_number);
    int release_all_values();
}
```

Figure 15. Some elements of the `LegionProgramGraph` interface

Now we can show the necessary library calls to implement the example given in Figure 16.

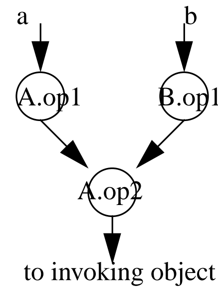
Start-up. The call to `Legion.init()` initializes various data structures in the Library. `Legion.AcceptMethods()` is called because the invoking object may itself be accepting member function requests from other objects.

Object creation. The calls to `Legion.CreateObject()` create the two objects of interest and return LOIDs to these objects. Given these LOIDs, local handles for the objects are created.

Member function invocation. For each method, we use the object's local handle to create an invocation.[1] We can then add the invocation to the program graph using `add_invocation()`. Every added invocation becomes a node in the program graph. To create arcs, parameters must first be packaged into instances of `LegionParameter` (see Figure 12). Once packaged, they are added to the graph using `add_constant_parameter()`. Internal arcs in the graph must be handled differently, because they represent values that are not locally available—they have not been computed yet. Internal arcs are added using `add_invocation_parameter()`. Once a program graph is constructed, the `execute()` member function must be called. Calling `execute()` causes every node in the program graph to be packed up as a Legion Message and shipped to the appropriate object for execution. Results from this remote execution then become available and are automatically sent to the objects that require them. In the example, the return

1. A Legion invocation identifies a *particular invocation* on one of an object's member functions. An invocation contains a computation tag that identifies it within Legion for the duration of the invocation's existence. An invocation is obtained through a *Legion Core Handle*. Core handles export functions that allow programmers to ask for invocations and to obtain a description of the corresponding object's interface. A handle can be thought of as a local representation of an object and as a generator of invocations for that object.

```
// The "user" code
main () {
    int a = 10, b = 15, x, y,
    z;
    MyObject A, B;
    x = A.op1(a);
    y = B.op1(b);
    z = A.op2(x, y);
    printf ("%d\n", z);
}
```



```
// The corresponding calls to the library to implement the "user" code
main() {
    UVaL_Reference<LegionInvocation> inv1, inv2, inv3;
    UVaL_Reference<LegionBuffer> buffer;
    UVaL_Reference<LegionParameter> parm;
    int a = 10, b = 15;

    // Start-up
    Legion.init();
    Legion.AcceptMethods();

    // Object creation
    UVaL_Reference<LegionLOID> A_name, B_name;
    A_name = Legion.CreateObject(MY_OBJECT_CLASS_ID);
    B_name = Legion.CreateObject(MY_OBJECT_CLASS_ID);

    // Member function invocation
    LegionProgramGraph G(Legion.getMyLOID());
    LegionCoreHandle A_handle(A_name), B_handle(B_name);
    inv1 = A_handle.invoke(OP1_FUNC_NUM, 1, 1);
    G.add_invocation(inv1);
    parm = make_parameter (a, 1);
    G.add_constant_parameter (inv1, parm, 1);

    inv2 = B_handle.invoke(OP1_FUNC_NUM, 1, 1);
    G.add_invocation(inv1);
    parm = make_parameter (15. 1);
    G.add_constant_parameter (inv1, parm, 1);

    // Return value retrieval
    inv3 = A_handle.invoke(OP2_FUNC_NUM, 2, 1);
    G.add_invocation(inv1);
    G.add_invocation_parameter (inv1, inv3, 1, 1);
    G.add_invocation_parameter (inv2, inv3, 1, 2);
    G.execute(inv3);

    buffer = G.get_value(inv3, UVAL_METHOD_RETURN_VALUE);
    int z;
    buffer.get_int(&z, 1);
    printf ("%d\n", z);
}
```

Figure 16. Sample user code (top-left), the corresponding program graph (top-right), and the library calls needed to implement it (bottom). In this case `make_parameter()` takes an integer, wraps it up in a `LegionBuffer`, then wraps the buffer in a `LegionParameter`.

values from `A.op1()` and `B.op1()` are forwarded directly to `A` so they can become the parameters to `A.op2()`.

Return results. Getting return values that are results of method invocation requests is straightforward. The `LegionProgramGraph` class has a method called `get_value()`, which takes the parameter number of the result value as one of its arguments. If the result is unavailable, then `get_value()` blocks. The constant `UVaL_METHOD_RETURN_VALUE` can be passed to `get_value()` to obtain the return value of the function call. The constant also serves as the parameter number (position of the parameter in the function signature) for in/out parameters. By default, the return values of all method invocations are returned to the invoker. For in/out parameters, `add_result_dependency()` (not shown) must be used to explicitly ask for the parameter to be returned. `Add_result_dependency()` must be called *before* `execute()` is called on the program graph that contains the associated method invocation. Otherwise the parameter will not be returned and a call to `get_value()` for that parameter will block indefinitely.

3.5 The Invokee: Invocation Execution and Result Return

The Library announces a *Method Ready* event each time a ready method invocation request is inserted into the invocation store. Each request is maintained as a `LegionWorkUnit`, a class similar in structure to `LegionMessage`, but with additional semantics, namely that all parameters for the particular method are present. The general algorithm for getting a `LegionWorkUnit` out of the database and invoking the requested method is as follows:

1. Remove the work unit from the invocation store

2. Construct and perform the requested method invocation.

3.5.1 Removing the Legion Work Unit

User code can remove work units from the invocation store in at least two different ways, each of which require a server “loop” that continuously checks the invocation store for ready work units. Both mechanisms require a server ‘loop’ which continuously checks the invocation store for ready work units. One mechanism is to supply and register an event handler for `MethodReady` events. The code for the handler and the server loop then look like that in Figure 17 -top. The other mechanism, illustrated in Figure 17 - bottom does not require the user to supply an event handler. Instead, the user checks the invocation store each time through the server loop.

3.5.2 Constructing the Method Invocation

Once a work unit is removed from the invocation store, it needs to be unpacked to a form suitable for method invocation. There must be specific code to do this for each public method in the invoked object. Figure 18 contains an example invocation construction. The sequence is functionally the same as server stubs in RPC. First ascertain the requested method, then remove each parameter from the work unit based upon the particular requested method. Each parameter is returned as a LegionBuffer, so these need to be unpacked to get the actual parameters for the method invocation. Once this is done, then the method can be called like any C++ member function.

For methods that have return results, these values must be sent to the list of objects defined in the

```
int MethodReadyHandler (LegionEvent *event) {
   UVaL_Reference<LegionWorkUnit> wu;
   if (LegionInvocationStore->any_ready()) {
      wu = LegionInvocationStore->next_matched();
      invoke_method(wu);
   }
}
void server_loop() {
   LegionEventMgr.serverLoop();
}
```

```
void server_loop() {
   while (TRUE) {
      LegionEventMgr.flushEvents();  // causes all events to be handled
      while (LIS->any_ready()) {
         wu = LIS->next_matched();
         invoke_method(wu);
      }
      LegionEventMgr.blockForEventAvailable(); // blocks on any event
   }
}
```

Figure 17. Two mechanisms to get a ready work unit out of the invocation store. One method (top) is to supply an event handler that is called whenever a MethodReady event is announced. The accompanying server loop is quite short. The other method (bottom) is to have a longer server loop that checks the invocation store whenever any event occurs. In this case, no event handler is needed.

`LegionContinuationList` part of the work unit from which the method invocation was constructed. The most straightforward way to do this is as follows. For each return result, allocate a new `LegionBuffer` and insert the return value into the buffer. Then call `Legion.return()` with the buffer, continuation list, and number of the return value as arguments. This sequence is illustrated at the bottom of Figure 18.

A complete example of a C++ class, it's translation into the appropriate Library calls, and some sample method invocations are given in the Appendix.

3.6 Legion Object Persistent Representations

Legion objects are endowed with a persistent representation in which they can store volatile state in the event that they must be deactivated during the operation of the system [7]. Objects might also use their persistent representation to store data structures that are too large to contain in volatile storage (e.g. a "file" object need not keep its entire state, including the contents of the file, in memory). The persistent representation of an object is referred to as a *Legion Object Persistent Representation*, or OPR. In this section we examine the Library interface to manipulating object persistent representations.

The most basic interface to a Legion OPR is provided by the `LegionOPR` class. Instances of the class `LegionOPR` are constructed based on a *Legion OPR Address*, a description or pointer to a `LegionOPR`. This construction is taken care of internally by the library implementation. At the

```
// Each object might have a functio like this to figure out
// which member function to call.
invoke_method (UVaL_Reference<LegionWorkUnit> wu) {
    switch (wu->get_function_number()) {
        case SAMPLE_OP_FUNCTION_NUMBER:
            sample_op_wrapper(wu);
            break;
        // cases for other methods go here
    }
}

// assume this method has two parameters, an int and a float
void sample_op_wrapper(UVaL_Reference<LegionWorkUnit> wu) {
    float float_parm;
    int int_parm, return_value;
    UVaL_Reference<LegionBuffer> buffer;

    // unpack the parameters
    buffer = wu->get_parameter(1);
    lb->get_int(&int_parm, 1);
    buffer = wu->get_parameter(2);
    lb->get_int(&float_parm, 2);

    // invoke the method
    return_value = sample_op (int_parm, float_parm);

    // Return results to whomever has asked for them
    buffer = new LegionBuffer();
    buffer->put_int (&return_value, 1);
    Legion.return(METHOD_RETURN_VALUE, wu->get_continuation_list(), buffer);
}
```

Figure 18. An example method construction, invocation, and return once a work unit has been removed from the invocation store. Other code structures are possible.

time of activation, objects are passed a `LegionOprAddress` by the responsible Legion Host object so that they can locate and access their persistent representation. When the Library is initialized, the OPR Address is automatically converted into a `LegionOPR` instance using `getLegionOPR()`. The programmer can then access the `LegionOPR` instance for a Legion object using the `GetOPR()` method of the `LegionLibraryState` object class (Section 3.1). For example:

```
UVaL_Reference<LegionOPR> myOPR;
myOPR = Legion.GetOPR();
```

The interface to the OPR provides two key functions that provide access to the two basic forms of an OPR: *linearized* and *inflated*. For the purposes of object migration, the persistent representation of an object can be gathered into a linearized form, suitable for transport. The Legion application programmer will typically have no use for the linearized form of the OPR, which can be accessed via the `getLinearized()` method on the `LegionOPR` class. The more important form of the OPR for the applications programmer is the directly manipulatable form, the inflated form. The inflated form of a `LegionOPR` is encapsulated by the `LegionPersistent-`

BufferDir class. As its name implies, the LegionPersistentBufferDir is a directory of Persistent LegionBuffer objects. The inflated form of the OPR is accessed via the getInflated() method on the LegionOPR class. For example:

```
UVaL_Reference<LegionPersistentBufferDir> myState;
myState = myOPR.getInflated()
```

The LegionPersistentBufferDir class implements an association set that maps null terminated character strings to objects of the LegionBuffer class and subdirectory objects of the LegionPersistentBufferDir class. Objects of this class can be thought of as directories in a file system that contain string named files (persistent LegionBuffers) and subdirectories (LegionPersistentBufferDirs), although the implementation of these objects need not be based on a file system. Some elements of the interface to the LegionPersistentBufferDir class are given in Figure 19. It contains methods to determine the number of contained buffers and subdirectories, to determine if a given string maps to a contained buffer or subdirectory, to access, add to, or delete from the contained buffers and subdirectories by name, and to iterate over the contained buffers and subdirectories.

```
// Get the OPR representation
UVaL_Reference<LegionPersistentBufferDir> myState;
myState = myOPR.getInflated()

// Check the contents of a directory
if (myState.NumSubdirs() == 0) return -1;
if (! myState.IsContainedSubdir("State")) return -2;

// Access a subdirectory
UVaL_Reference<LegionPersistentBufferDir> subDir;
subDir = myState.GetSubdir("State");

// Access a contained buffer
UVaL_Reference<LegionBuffer> myData;
myData = subDir.GetBuffer("My Data");
```

Figure 19. Sample invocations on an object of class LegionPersistentBufferDir.

The Legion buffers contained in LegionPersistentBufferDir objects are persistent. That is, these buffers are based on storage that is contained in the object's persistent representation and will thus persist after the object is deactivated. Thus, in manipulating these buffers, the object is directly manipulating its persistent state. Of course, this does not imply any changes to the LegionBuffer interface. Data structures that were rendered packable for the purposes of transport in Legion messages are equally packable into the Legion buffers obtained as part of the object's OPR. The implementation of an object's SaveState() method is typically a sequence of pack() operations on the data structures that make up the object's state, many of which already needed to be packable (or made up of packable constituents) for the sake of method service and invocation.

During the operation of a Legion object's SaveState() method, or for the purposes of taking checkpoints, the programmer may need to capture the state of the Library. This functionality is

provided through the `saveState()` method on the `LegionLibraryState` class. To save the state of the Legion library, the programmer simply writes:

```
Legion.saveState();
```

To recover the state from the persistent representation, a complementary `restoreState()` operation is provided, e.g.:

```
Legion.restoreState();
```

4. Modifying the Library

One of the major design objectives of Legion is to provide an extensible system - it must be easy for future implementors to insert modules into the Library. To enable this extensibility the Library provides

1. a layered design and implementation, and

2. a standard mechanism for inter-layer communication.

The layered design of the Library is depicted in Figure 20. The “client” side (left) is the “invoker”, the code that is requesting a method invocation on some Legion object. The “server” side (right) is the “invokee”, the Legion Object upon which the method invocation has been made. While it is convenient to think of the library in terms of clients and servers, it is artificial in that the full library functionality is provided to both parties. In many cases an object’s role changes as execution progresses. Sometimes clients are servers and vice versa.

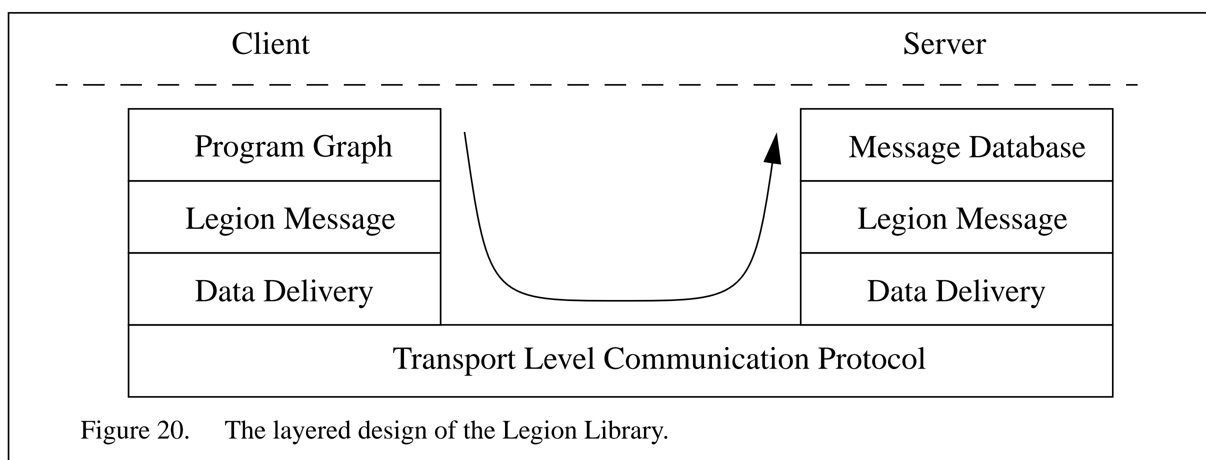


Figure 20. The layered design of the Legion Library.

4.1 Implementing the Legion Configurable Protocol Stack

As Figure 20 illustrates, the Legion protocol stack supports a variety of functions. One of our goals in designing the protocol stack was to allow modules to be added and configured easily, an approach similar to that used in the *x*-Kernel [6]. The problem with the traditional approach to building protocol stacks is that each layer in the stack explicitly calls the layer below or above.

This static coupling makes it difficult to dynamically configure the stack.

To provide a dynamically configurable stack, we have chosen a well understood technology [1], events, and have applied it to enable flexibility and extensibility. Four main classes implement events: `LegionEvent`, `LegionEventKind`, `LegionEventHandler` and `Legion-EventManager`.

When an event has occurred, we "announce" an event to an *Event Manager.* The event manager, an instance of class `LegionEventManager`, notifies interested parties, i.e. *Legion Event Handlers*, of the event. An event manager may notify an event handler using one of two policies: the handlers can be notified immediately or at a later time.

To be notified of an event, event handlers register themselves with a *Legion Event Kind*, an event template that contains an unique tag and a default list of handlers. Since there may be more than one event handler per event kind, we associate a priority with the handler at the time of registration. In our scheme, a handler with a lower priority number is executed before a handler with a higher priority number. Not all event handlers associated with a LegionEvent are guaranteed to be executed because an event handler is allowed to prevent the execution of subsequent handlers.

The class `LegionEventKind` serves as a template for instances of class `LegionEvent` (Figure 21). When an event is created, it obtains a list of `LegionEventHandlers` from its corresponding `LegionEventKind`. This allows users to modify the behavior of the protocol stack without having to change existing modules. To enable inter-layer communication, Legion events may also carry arbitrary data. This data can be update, modified, and transformed in essentially arbitrary ways by the event handlers that process each event.

A Legion event handler takes as its sole argument a reference to the `LegionEvent` that it is servicing. Thus, each `LegionEventHandler` associated with a particular `LegionEvent` may inspect and modify the data carried by the `LegionEvent`. In general, this is how information is shared between various `LegionEventHandlers`.

4.2 Interfaces

4.2.1 LegionEventKind, LegionEvent and LegionEventHandler

Users may add `LegionEventHandlers` to a `LegionEventKind`. When a `LegionEvent` is created, it will obtain its unique event identifier and a list of `LegionEventHandlers` from its corresponding `LegionEventKind`. `LegionEventHandlers` are ordered and the handlers with lower numbered priorities have higher precedence. An example of a event handler is given in Figure 22.

A Legion event maintains a logical pointer to the currently executing `LegionEventHandler`. This allows the event to suspend the execution of its event handlers and to resume that execution at a later time.

There are no restrictions on the code implementing a `LegionEventHandler`. In particular, a an event handler may

1. Inspect and modify the data field of the incoming event

```
class LegionEventKind {
public:
    // Construct a new event kind and give it a unique identifer
    LegionEventKind(int kind);

    // Add and delete handlers
    // Note that handlers are added in priority order
    int addHandler(LegionEventHandler, LegionEventHandlerPriority);
    int deleteHandler(LegionEventHandler);
};

class LegionEvent : public UVaL_Reference {
public:
    // Construct an event using a LegionEventKind as a template
    LegionEvent(LegionEventKind&);
    LegionEvent(LegionEventKind&, void * data);

    // Adding and deleting handlers
    int addHandler(LegionEventHandler, LegionEventHandlerPriority);
    int deleteHandler(LegionEventHandler);

    // Setting and getting the data associated with the LegionEvent
    void* getData();
    void setData(void*)

    // Invoking event handlers
    LegionEventHandlerStatus callNextHandler(UVaL_Reference<LegionEvent> ev);
    void callRemainingHandlers(UVaL_Reference<LegionEvent> ev);
};
```

Figure 21. Some elements of the LegionEventKind and LegionEventInterface

```
// Signature of a LegionEventHandler
typedef LegionEventHandlerStatus
          (*LegionEventHandler) (UVaL_Reference<LegionEvent>)

// Example of a valid LegionEventHandler
LegionEventHandlerStatus myHandler(UVaL_Reference<LegionEvent> myEvent) {
    // arbitrary code
}
```

Figure 22. LegionEventHandler

2. Inspect and modify the list of handlers contained in the incoming event

3. Create and announce new events

4. Prevent the next handlers from being executed

5. Save the current event

```
class LegionEventManager {
public:
    // There are two ways to announce an event
    // (1) LegionEventAnnounceLater - Defer execution of the handlers
    // (2) LegionEventAnnounceNow - Immediately invoke the handlers
    announce(UVaL_Reference<LegionEvent>,
             LegionEventQueingDiscipline queueEvent  = LegionEventAnnounceLater);

    // flush all events from the queue and execute the handlers
    unsigned flushEvents();

    // blocking call that returns only when there are events in the queue
    unsigned blockForEventAvailable();

    // the server loop repeatedly calls blockForEventAvailable and flush Events
    unsigned serverLoop();
};
```

Figure 23. Some Elements of the LegionEventManager Interface

4.2.2 LegionEventManager

When users wish to notify the system that something of interest has occurred, they must announce a `LegionEvent` to a `LegionEventManager` (Figure 23). The LegionEventManager is responsible for deciding when to execute the handlers associated with an event. In our current implementation, there are two ways of announcing events to an event manager. Depending upon the chosen method, the event manager will either invoke the handlers immediately, or will defer the execution of the event handlers and store the LegionEvent in an internal queue.

The `flushEvents()` method is used to execute all pending events. The `blockForEventAvailable()` method is used to block the thread of control until there are some pending events available. Finally, the `serverLoop()` method repeatedly calls `blockForEventAvailable()` followed by `flushEvents()`.

4.3 Default Protocol Stack

The list of default `LegionEventKinds` and their associated `LegionEventHandlers` is shown in Table 1. These implement the protocol layers shown in Figure 20.

Table 1: Default LegionEventKind and LegionEventhandlers

LegionEventKind	LegionEventHandler	Description of LegionEventHandler
Sending Object		
LegionEvent_MethodSend	LegionDefault_Can_I	Determines whether to allow the outgoing method invocation [10].
	LegionDefaultMethodSendHandler	Generates a *LegionEvent_MessageSend* for each method invocation

Table 1: Default LegionEventKind and LegionEventhandlers

LegionEventKind	LegionEventHandler	Description of LegionEventHandler
LegionEvent_MessageSend	msg_layer_MsgSnd_handler	Binds the destination LOID into an Object Address
	data_delivery_MsgSnd_handler	Sends LegionMessage over the wire
LegionEvent_MessageComplete	data_delivery_MsgComplete_handler	Indicates that the message has been successfully sent.
LegionEvent_MessageError	data_delivery_MsgError_handler	Indicates that the data delivery layer was unable to send the message
	msg_layer_MsgError_handler	Indicates that the message was not sent successfully
Receiving Object		
LegionEvent_MessageReceive	data_delivery_MsgRcv_handler	Extract message from the transport layer.
	msg_layer_MsgRcv_handler	Unpack the data into a `LegionMessage` and cache the sender's object address.
	LegionDefaultMessageHandler	Inserts the message into the Invocation Matcher. If this LegionMessage completes a partial method invocation, then generate a *LegionEvent_MethodReceive* event.
LegionEvent_MethodReceive	LegionDefault_May_I	Determine whether to allow the incoming method invocation [10].
	LegionDefaultWorkUnitHandler	Stores incoming method invocation in the Invocation Store. Generates a *LegionEvent_MethodReady* event.
LegionEvent_MethodReady	LegionMethodDispatcherMonitors_MethodReadyHandler	Enforce monitor semantics on incoming method invocation
	UVaL_ObjectMandatory_LegionMethodInvoke	Invoke the actual function
LegionEvent_MethodDone	LegionMethodDispatcherMonitors_MethodDoneHandler	Indicates that a method has been complete. Generate a *LegionEvent_MethodReady* event if there are pending methods.

On the sending side, the program graph layer generates a `LegionEvent_MethodSendEvent`. The security handler `LegionEvent_Can_I` may disallow the remote method invocation [10]. If it doesn't, then a following handler generates a `LegionEvent_MessageSend` event for each method invocation. Once the message has been successfully sent, the data delivery layer generates a `LegionEvent_MessageComplete` event.

On the receiving side, the data delivery layer will generate a LegionEvent_MessageReceive once it has successfully assembled a complete message. The LegionDefaultMessageHandler is the last handler for the event LegionEvent_MessageReceive and generates a LegionEvent_MethodReceive once the invocation matcher has assembled a complete method invocation. The first handler for LegionEvent_MethodReceive is a security handler and implements access control on this object [10]. If the security handler grants access, the method invocation is deposited into the LegionInvocationStore, and we generate a LegionEvent_MethodReady event.

4.4 Adding new functionality to the Legion protocol stack

To add functionality to the existing stack, users may either define a new LegionEventKind or may register their own handlers with one of the predefined event kinds. The latter option is the simpler method for adding functionality.

Defining a new event kind consists of creating an instance of the class LegionEventKind with a unique identifier. For example:

```
LegionEventKind LegionEvent_Foobar (UniqueIdentifier);
```

Once the event kind has been defined, creating and announcing a LegionEvent is shown below:

```
LegionEvent myEvent(LegionEvent_Foobar);
LegionDefaultEventMgr.announce(myEvent);
```

Adding a handler to an existing event kind is best illustrated through an example. In this example we show how a user can add a security layer to encrypt outgoing messages and decrypt incoming messages. We first define the handlers encryptionHandler(), decryptionHandler() and register them with the appropriate LegionEventKind (Figure 24).

For encryption, we would like the encryption handler to be the last handler called when sending a message. For decryption, we would like the decryption handler to be the first handler called when a message is received. There ordering constraints are realized by registering encryptionHandler() with a high numbered priority and decryptionHandler() with a low numbered priority. The new protocol stack is now shown in Figure 25.

5. Diversity and Extensibility

The Library can be used to support a diverse array of programming models and styles, or to extend existing support for basic programming models. The flexible Legion event mechanism is the key Library feature that enables diversity and extensibility. By adding to or replacing the event handlers that comprise the Legion protocol stack, a programmer can achieve many different library configurations that implement different method invocation semantics, security protocols, communication mechanisms, object instantiation environments, and other important elements of the desired programming model. In this section, we examine a number of possible Legion protocol stack configurations that implement some well known programming models, including active messages, path-expressions, and basic message passing. These examples show how the Library can be used to support programming models beyond the basic remote procedure call and macro-dataflow/program graph based styles implemented by the default library configuration.

```
// Declaration of the encryption handler
LegionEventHandlerStatus encryptionHandler(UVaL_Reference<LegionEvent> ev) {
      // extract the LegionMessage from the data field of the event,
      // encrypt the message, and
      // allow the next handler to be called.
      return TRUE;
}

// Declaration of the decryption handler
LegionEventHandlerStatus decryptionHandler(UVaL_Reference<LegionEvent> ev) {
      // extract the LegionMessage from the data field of the event,
      // decrypt the message, and
      // allow the next handler to be called.
      return TRUE;
}

//
// Register the handlers with the appropriate LegionEventKind
//

// The encryptionHandler should be the last handler before the
// message is sent by the data delivery layer
LegionEvent_MessageSend.addHandler(encryptionHandler, encryptionPriority);

// The decryptionHandler should be the first handler called
// after the message is delivered by the data deliver layer
LegionEvent_MessageRecv.addHandler(decryptionHandler, decryptionPriority);
```

Figure 24. Adding encryption and decryption capabilities to the protocol stack

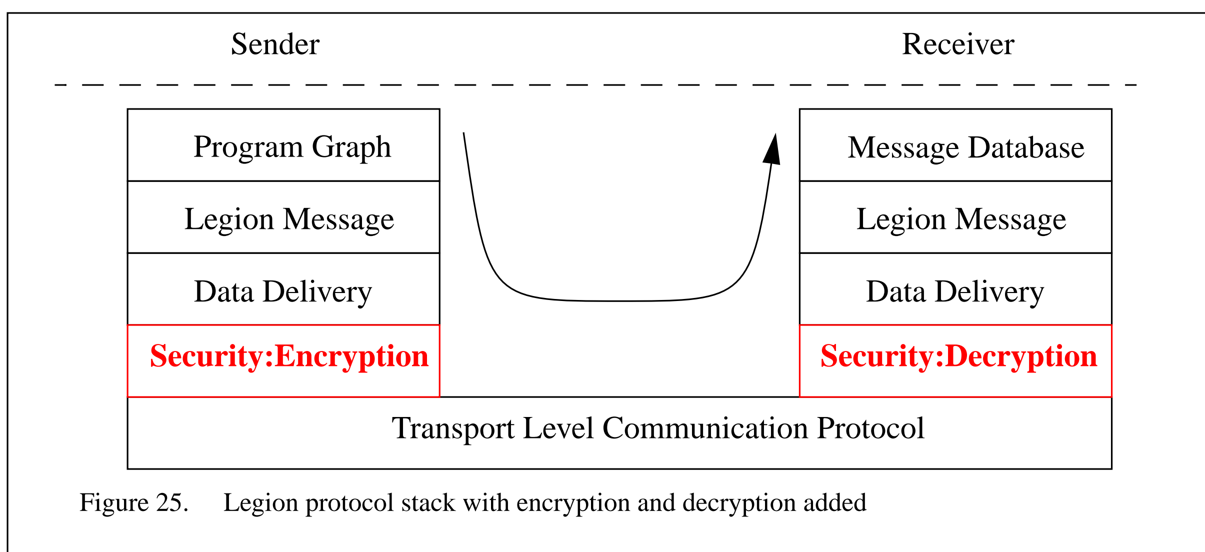


Figure 25. Legion protocol stack with encryption and decryption added

5.1 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.

A Legion version of active messages could be constructed by making Legion methods serve as message handlers, and by replacing the Legion "method ready" event handler with one that creates a new thread to service incoming methods instead of buffering them in an invocation store. Pseudo-code for such a method invocation handler is given in Figure 26.

This "method ready" event handler would need to be registered with the "method ready" event kind. The code to do this might look like:

```
LegionEvent_MethodReady.addHandler(ActiveMessageMethodHandler,1.0);
```

This line of code would need to be executed before any methods arrived at the object. This can be achieved by placing this line of code before any calls to `Legion.AcceptMethods()`.

The effect of this new "method ready" event handler is to provide an active messages style programming model. 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 by the Legion invocation matcher. Programs might be entirely composed of standard single-token active messages, providing a programming model as flexible as the original [9]. 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".

5.2 Path Expressions

The various method invocation semantics covered thus far have offered a "one size fits all" con-

```
int ActiveMessageMethodHandler(UVaL_Reference<LegionEvent> ev) {
    // Extract the work unit from the event
    LegionMethodEventStructure *mes;
    mes = (LegionMethodEventStructure *)ev->getData();
    UVaL_Reference<LegionWorkUnit> wu = mes->work_unit;

    // Spawn a thread with the appropriate start-up function
    // based on the function number associated with the method
    switch (wu->get_function_number()) {
        case METHOD1_FUNCTION_NUMBER:
            pthread_create(&thr_id, &thr_attrib, method1, wu);
    } // Similar cases for other methods...
}
```

Figure 26. An example method handler for implementing active messages.

currency control mechanism. For example, the supported remote procedure call model allows exactly one method to be serviced at a time by a given object. The active messages approach, on the other hand, allows any number of operations of all types to be active at the same time in the same object. A more general approach to customizing the concurrency control requirements of operations on an object can be designed based on path expressions [2]. 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 by utilizing the inherent configurability of the Library's protocol stack. As with active messages, supporting a different method invocation semantic requires replacing the Legion "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 safely fired, 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 a library-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 `LegionEvent_MethodDone` event kind, and announce events of this kind when methods complete execution. A handler for this event kind can then be used to re-evaluate buffered methods with respect to the path expression ordering constraints whenever a running operation completes.

To examine the implementation of the scheme in more detail, we assume a path expression runtime support class, `PathExpressionManager`, that exports methods to specify the ordering, selection, and sequencing constraints of operations (i.e. Legion method function numbers). This class would also support methods to determine if a given method is safe to fire, and to determine which (if any) methods are ready to be fired upon the completion of a running operation. The first modification we must make to the Library configuration is to add a new "method ready" event handler that might look like that of Figure 27.

This method handler would need to be registered with the Legion method ready event kind, as in the case of the active messages handler. The other requirement of our path expression solution is that code be executed upon method completion in order to re-evaluate the safety of firing buffered methods. To accomplish this, we use "method done" events that must be announced whenever a method is finished running. A handler (Figure 28) must be registered with the `LegionEvent_MethodDone` event kind that tries to fire any runnable buffered methods.

```
LegionEvent_MethodDone.addHandler(PathExprMethodDoneHandler, 0.0);
```

Finally, an event of this type would need to be announced upon the completion of each method by the object. The data for the event would need to be set to reflect the function number of the completed method.

```
UVaL_Reference<LegionEvent> done = new LegionEvent(MethodDone);
```

```
done.setData((void *)my_function_number);
LegionEventManagerDefault.announce(done);
```

The result of this configuration of the Library 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.

5.3 Message Passing

Thus far, the programming models we have examined have been variations of an object-based

```
int PathExprMethodHandler(UVaL_Reference<LegionEvent> ev){
    // Extract the work unit from the event
    LegionMethodEventStructure *mes;
    mes = (LegionMethodEventStructure *)ev->getData();
    UVaL_Reference<LegionWorkUnit> wu = mes->work_unit;

    int function_number = wu->get_function_number();
    if(PathExpressionManager.canFire(function_number)) {
        // We can safely fire this method now
        PathExpressionManager.runningOperation(function_number);
        switch(function_number) {
            case METHOD1_FUNCTION_NUMBER:
                pthread_create(&thr_id, &thr_attrib, method1, wu);
        } // Similar cases for other methods...
    } else {
        // Buffer this method until ordering constraints are met
        LegionInvocationStoreDefault->insert(wu);
    }
}
```

Figure 27. An example method handler for implementing path expressions.

```
int PathExprMethodDoneHandler(UVaL_Reference<LegionEvent> ev) {
    int done_function_num = (int)ev.getData();
    PathExpressionManager.completedOperation(done_function_num);

    while (PathExpressionManager.anyReady()) {
        int function_num = PathExpressionManager.nextReady();
        UVaL_Reference<LegionWorkUnit> wu;
        wu = LegionInvocationStoreDefault->next_matched_for_func(function_num);
        PathExpressionManager.runningOperation(function_num);
        switch(function_num) {
            case METHOD1_FUNCTION_NUMBER:
                pthread_create(&thr_id, &thr_attrib, method1, wu);
        } // Similar cases for other methods...
    }
}
```

Figure 28. One possible handler for MethodDone events in an implementation of path expressions.

method-invocation-oriented model. This is natural given the object oriented nature of the Legion system. However, Legion can support alternative programming models such as message passing or distributed shared memory. In this section, we examine the ways in which the Legion library can be configured to support a message passing model. We describe how Legion can be uses as run-time support to implement a message-passing interface such as MPI[5] or PVM[8]. These systems allow asynchronous send and receive operations - messages are buffered until explicitly requested at the receiving process, and send operations are permitted to return before the destination process has received the message.

One possible implementation of such a message passing system would be to construct a message passing Legion base class which exports a single `messageDeliver()` method. The parameters to this method could be an integer message tag and an un-interpreted string of bytes containing the message. The operation of the `send()` library function would simply involve invoking the `messageDeliver()` Legion method on the intended destination LOID. The implementation of the `messageDeliver()` Legion method would simply accept and buffer the received message in an internal message queue. The `receive()` library function would then consist of a loop that could check the message queue for the desired message tag, dequeue and return it if available, or block for a `messageDeliver()` invocation if not.

Although the above solution is very simple to implement using the available library support mechanisms, it has the potential drawback of incurring the cost of the mechanism associated with method invocation (e.g. token matching, additional event handlers, etc.), while only requiring the very simple support needed for message passing. An alternative implementation strategy is to insert a handler lower in the Legion library protocol stack. The natural place for such a message passing handler would be at the Legion “message receive” event layer. Here, a handler could be inserted to capture messages with certain desired function numbers (i.e. a special “message passing” function number), and enqueue the message contents on a message queue for use by the message passing library. Messages with other function numbers (for example, those associated with object mandatory methods[7]) would be allowed to continue up the protocol stack and through the normal method invocation mechanism. In this scheme, the `send()` library operation would construct a `LegionMessage` object containing the message contents and reflecting the appropriate agreed upon function number. This Legion message would be added to a `LegionMessageEventStructure`, which would be placed into a new `LegionEvent` of the kind `LegionEvent_MessageSend`. This event would be announced and the message would be sent. The `receive()` operation would simply loop, examining the message queue utilized by the message receive handler described above, and blocking for available events. Thus, in this scheme, the overhead of the standard Legion method invocation mechanism is avoided for low-level message passing traffic.

Although this scheme could improve performance, it has serious security ramifications. If messages are caught before the token matching process, the automatic `MayI()` method will not be invoked and the object’s security may be compromised. While this may be acceptable for certain performance-critical, security-optional applications, the first message passing implementation described would be more suitable for balanced security/performance applications.

6. References

1. B. Bershad *et al*. Extensibility, Safety, and Performance in the *SPIN* Operating System. In *15th Symposium on Operating System Principles*, 1994.

2. R. H. Campbell and A. N. Habermann. The Specification of Process Synchronization by Path Expressions. *Lecture Notes in Computer Science*, No. 16, Springer Verlag, pages 89-102, 1973.

3. Andrew S. Grimshaw, Jon Weissman, W. Timothy Strayer. Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing. *ACM Transactions on Computer Systems*, 14(2), 1996.

4. Andrew S. Grimshaw, William A. Wulf. Legion - A View from 50,000 Feet. In P*roceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, Los Alamitos, California, August 1996.

5. W. Gropp, E. Lusk, and A. Skjellum. Using MPI: Portable Parallel Programming with the Message Passing Interface,MIT Press, 1994.

6. N. C. Hutchinson and L. L. Peterson. The *x*-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64-76, 1991.

7. Mike Lewis, Andrew S. Grimshaw. The Core Legion Object Model. In P*roceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society Press, Los Alamitos, California, August 1996.

8. V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Journal of Concurrency: Practice and Experience*, 2(4):315-339, December 1990.

9. Thornsten von Eicken, David E. Culler, Seth C. Goldstein, and Klaus E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In P*roceedings of the International Symposium on Computer Architecture*, pages 256-266, May 1992.

10. William A. Wulf, Chenxi Wang, Darrell Kienzle. A New Model of Security for Distributed Systems. University of Virginia, Department of Computer Science Technical Report CS-95-34, August 1995.

11. H. Zhou and A. Geist. Receiver Makes Right Data Conversion in PVM. In *Proceedings of 14th International Conference on Computers and Communications*, Phoenix, AZ, pp. 458-464, March 1995.

7. Appendix A - Example Translations

The most up-to-date version of these translations are also available at http:/legion.virginia.edu/.

7.1 Example 1 - The Simple Class

The first example is a nonsense class in that it does no meaningful computation. However, it exercises the full functionality of the library and also illustrates how our compiler will generate translations. We give the C++ class, its translation, and a sample program that illustrates how we invoke methods.

To write the translation, we define a wrapper class called `Legion_Simple`. This wrapper contains a protected data member that is an object of the C++ class being translated. We call this the wrapped object. For each member function in the wrapped object we define a corresponding member function in `Legion_Simple`. The job of these member functions is to take a work unit, build the corresponding call to the wrapped object's member function, execute the call, and package up any return values (much like a server stub in RPC implementations).

`Legion_Simple` also contains additional member functions to figure out which wrapper member function should be called (`invoke_method()`), to indicate to the invocation store which member functions will be accepted (`enable_functions()`), and to perform the server loop (`accept_member_functions()`).

7.1.1 Simple.h & Simple.c

```
%-----------------Simple.h------------------------%
// A very simple class definition
#ifndef _H_Simple_
#define _H_Simple_

#include <stdio.h>

class Simple
{
   int data;
 public:
   Simple();
   int op1(int foo);
   int op2(int &foo, int &bar);
};

#endif
%-----------------Simple.c------------------------%
// A very simple class definition
#ifndef _C_Simple_
#define _C_Simple_

#include <stdio.h>
#include "Simple.h"

Simple::Simple() {
   data = 0;
}

Simple::op1(int foo) {
```

```
   data = foo;
}

int
Simple::op2(int &foo, int &bar) {
   foo = data*data;
   bar = data+data;
   return foo+bar;
}
#endif
```

7.1.2 Simple.trans.h & Simple.trans.c

```
%--------------Simple.trans.h----------------------------%
// Legion 'wrapper' class definition for the Simple class
#ifndef _H_Simple_trans_
#include <stdio.h>
#include "legion/Legion.h"
#include "Simple.h"


#define SIMPLE_OBJECT_CLASS_ID   "Simple"

#define SIMPLE_OP1_FUNCTION_NUMBER 1
#define SIMPLE_OP2_FUNCTION_NUMBER 2

// class Legion_Simple
class Legion_Simple
{
 private:
   // the object being wrapped
   Simple *object;

   // For generating methodDone events
   virtual void generate_MethodDoneEvent();

 public:
   Legion_Simple();
   ~Legion_Simple();

   // wrapper member functions for each member function in 'object'
   void Legion_op1(UVaL_Reference<LegionWorkUnit> wu);
   void Legion_op2(UVaL_Reference<LegionWorkUnit> wu);

   // auxiliary member functions needed
   virtual void enable_functions(LegionInvocationStore *);
   virtual int  invoke_method(UVaL_Reference<LegionWorkUnit> wu);
   virtual void accept_member_functions();
};
#endif

%--------------Simple.trans.c----------------------------%
// The 'wrapper' class definition
#ifndef _C_Simple_trans_

#include <stdio.h>
#include <unistd.h>
#include "legion/Legion.h"
```

```
#include "Simple.trans.h"

// This is the wrapped object
static Legion_Simple *wrapper;

// This is the event handler for invoking a method
static int LegionMethodInvoke(UVaL_Reference<LegionEvent>);

// -----------------------------------------------------------------------
// Generates a MethodDone event. Should be called right after the call
// to the wrapped object's member function.
void
Legion_Simple::
generate_MethodDoneEvent()
{
   UVaL_Reference<LegionEvent> methodDoneEvent;

   methodDoneEvent = new LegionEvent(LegionEvent_MethodDone, (void *) NULL);
   LegionEventManagerDefault.announce(methodDoneEvent, LegionEventAnnounceNow);
}

// -----------------------------------------------------------------------
// Allocates the wrapped object and enables the corresponding
// functions in the invocation store.
Legion_Simple::
Legion_Simple()
{
   object = new Simple();
   wrapper = this;
   enable_functions(LegionInvocationStoreLL_Default);
}

// -----------------------------------------------------------------------
// De-allocates the wrapped object
Legion_Simple::
~Legion_Simple()
{
   delete object;
   wrapper = NULL;
}

// -----------------------------------------------------------------------
// wrapper member function for wrapped.op1()
void
Legion_Simple::
Legion_op1(UVaL_Reference<LegionWorkUnit> wu)
{
   // get the parameter from the work unit
   int parm1;
   UVaL_Reference<LegionBuffer> lb;
   lb = wu->get_parameter(1);
   lb->get_int(&parm1, 1);

   // Invoke the wrapped member function
   int result = object->op1(parm1);
   generate_MethodDoneEvent();

   // Return the result.
   UVaL_Reference<LegionBuffer> return_lb;
```

```
   return_lb = new LegionBuffer();
   return_lb->put_int(&result, 1);
   Legion_return(UVaL_METHOD_RETURN_VALUE, *(wu->get_continuation_list()), return_lb);
}

// -------------------------------------------------------------------------
// wrapper member function for wrapped.op2()
void
Legion_Simple::
Legion_op2(UVaL_Reference<LegionWorkUnit> wu)
{
    // get the parameters from the work unit and unpack them
   int parm1, parm2;
   UVaL_Reference<LegionBuffer> lb;
   lb = wu->get_parameter(1);
   lb->get_int(&parm1, 1);
   lb = wu->get_parameter(2);
   lb->get_int(&parm2, 1);

    // invoke the requested function
   int result = object->op2(parm1, parm2);
   generate_MethodDoneEvent();

   // Return all results. In this case, there are three of them
   // (return value and two in/out parameters).
   UVaL_Reference<LegionBuffer> return_lb;
   return_lb = new LegionBuffer();
   return_lb->put_int(&result, 1);
   Legion_return(UVaL_METHOD_RETURN_VALUE, *(wu->get_continuation_list()), return_lb);
   return_lb = new LegionBuffer();
   return_lb->put_int(&parm1, 1);
   Legion_return(1, *(wu->get_continuation_list()), return_lb);
   return_lb = new LegionBuffer();
   return_lb->put_int(&parm2, 1);
   Legion_return(2, *(wu->get_continuation_list()), return_lb);
}

// -------------------------------------------------------------------------
// Invokes the appropriate method based on the function number in
// the supplied work unit.
int
Legion_Simple::
invoke_method(UVaL_Reference<LegionWorkUnit> wu)
{
   switch (wu->get_function_number()) {
     case SIMPLE_OP1_FUNCTION_NUMBER:
       Legion_op1(wu);
       break;
     case SIMPLE_OP2_FUNCTION_NUMBER:
       Legion_op2(wu);
       break;
     default:
       fprintf(stderr,"Legion_Simple::invoke_method()\n");
       fprintf(stderr,"This object does not export function number %d\n",
          wu->get_function_number());
       exit(0);
       break;
   }
}
```

```
// -----------------------------------------------------------------------
// This is the server loop.
// EventMgr.serverLoop() continuously flushes events and then
// blocks waiting for events to become available.
void
Legion_Simple::
accept_member_functions()
{
  LegionEventManagerDefault.serverLoop();
}

// -----------------------------------------------------------------------
// Enable the wrapped object's functions. The LIS must be explicitly
// told which functions to accept. There will eventually be some
// object mandatory functions in here too.
void
Legion_Simple::enable_functions(LegionInvocationStore *LIS)
{
   // Enable the function numbers that I can handle...
   LegionInvocationStoreLL_Default->enable_function(
     SIMPLE_OP1_FUNCTION_NUMBER, DEFAULT_PRIORITY);
   LegionInvocationStoreLL_Default->enable_function(
     SIMPLE_OP2_FUNCTION_NUMBER, DEFAULT_PRIORITY);

   // Register my event handler...
   LegionEvent_MethodReady.addHandler(LegionMethodInvoke, 1.0);
}

// -----------------------------------------------------------------------
// This is the event handler that get called on a MethodReady event.
// A MethodReady event is generated every time a ready invocation is
// inserted into the invocation store.
static int
LegionMethodInvoke(UVaL_Reference<LegionEvent> event)
{
   UVaL_Reference<LegionBuffer> lb;
   int parameter;

   if (LegionInvocationStoreLL_Default->any_ready()) {
      UVaL_Reference<LegionWorkUnit> wu;
      wu  = LegionInvocationStoreLL_Default->next_matched();
      wrapper->invoke_method(wu);
   }
   return 0;
}

// -----------------------------------------------------------------------
int
main (int argc, char **argv)
{
   Legion.init();
   wrapper = new Legion_Simple();
   Legion.AcceptMethods();
   wrapper->accept_member_functions();
}
#endif
```

7.1.3 ex1_Simple.c

```
%--------------ex1_Simple.c---------------%
#include <stdio.h>

#include “Simple.trans.h”
#include “legion/Legion.h”

// Make a LegionParameter from the supplied arguments
UVaL_Reference<LegionParameter>
make_int_parameter(int parm_value, int parm_number)
{
   UVaL_Reference<LegionBuffer> lb;
   UVaL_Reference<LegionParameter> parm;

   lb = (LegionBuffer *) new LegionBuffer();
   lb->put_int(&parm_value, 1);
   parm = (LegionParameter *) new LegionParameter(parm_number, lb);

   return parm;
}


int
main(int argc, char **argv)
{
   // Variables for the ‘user’ code
   int a = 10, b = 15;
   int x, y, z;

   // Initialize legion state
   // All of the below to get a random instance number
   int my_instance_number;
   struct timeval tv;
   gettimeofday(&tv,NULL);
   srand(tv.tv_sec ^ tv.tv_usec);
   my_instance_number = rand() ^ tv.tv_sec ^ tv.tv_usec;

   // Initialize Legion Library
   Legion.init();

   // Manufacture my own loid because I’m a command line object
    Legion.SetMyLOID(make_loid(UVaL_CLASS_ID_COMMANDLINE, my_instance_number));

   // Tell my creator I’m ready to go
   Legion.AcceptMethods();

   // Create an empty program graph
    LegionProgramGraph G(Legion.GetMyLOID());

    // Create a couple of ‘Simple’ objects
    UVaL_Reference<LegionLOID> A_name, B_name;
   A_name = Legion.CreateObject(SIMPLE_OBJECT_CLASS_ID);
   B_name = Legion.CreateObject(SIMPLE_OBJECT_CLASS_ID);

   // Get handles for each object
   LegionCoreHandle A_handle(A_name), B_handle(B_name);

   // First call: x = A.op1(a);
```

```
    // invoke()'s signature is invoke(function_num, num_parms, num_results);
    UVaL_Reference<LegionInvocation> inv1;
    inv1 = A_handle.invoke(SIMPLE_OP1_FUNCTION_NUMBER, 1, 1);
    G.add_invocation(inv1);
    UVaL_Reference<LegionParameter> parm1;
    parm1 = make_int_parameter(a, 1);
    G.add_constant_parameter(inv1, parm1, 1);

    // Second call: y = B.op1(b);
    UVaL_Reference<LegionInvocation> inv2;
    inv2 = B_handle.invoke(SIMPLE_OP1_FUNCTION_NUMBER, 1, 1);
    G.add_invocation(inv2);
    UVaL_Reference<LegionParameter> parm2;
    parm2 = make_int_parameter(b, 1);
    G.add_constant_parameter(inv2, parm2, 1);

    // Third call: z = A.op1(x, y);
    // Both parameters are values yet to be computed,
    // so they must be invocation parameters.
    UVaL_Reference<LegionInvocation> inv3;
    inv3 = A_handle.invoke(SIMPLE_OP2_FUNCTION_NUMBER, 2, 3);
    G.add_invocation(inv3);
    G.add_invocation_parameter(inv3, inv1, 1, UVaL_METHOD_RETURN_VALUE);
    G.add_invocation_parameter(inv3, inv2, 2, UVaL_METHOD_RETURN_VALUE);

    // We specifically ask for the in/out parameters. Don't
    // get them otherwise.
    G.add_result_dependency(inv3, 1);
    G.add_result_dependency(inv3, 2);

    // printf ("%d\n", z);
    // We need 'z', so we must execute
    G.execute();

    // and wait for the return.
    UVaL_Reference<LegionBuffer> lb;
    lb = G.get_value (inv3, UVaL_METHOD_RETURN_VALUE);
    lb->get_int(&z, 1);
    printf ("z is%d\n", z);

    // Since we asked for them, we can get the other values too.
    G.release_all_values();
    lb = G.get_value (inv3, 1);
    lb->get_int(&x, 1);
    printf ("x is %d\n", x);
    lb = G.get_value (inv3, 2);
    lb->get_int(&y, 1);
    printf ("y is %d\n", y);

    Legion.DestroyObject(A_name);
    Legion.DestroyObject(B_name);
}
```

8. Appendix B - Interfaces

This section provides selected parts of the interface from selected objects that make up the Library implementation.

8.1 LegionProgramGraph

UVaL_Reference<LegionInvocation> add_invocation ()

Parameters: UVaL_Reference<LegionInvocation>

Add an invocation to the program graph. Returns the invocation if successful, NULL if not.

ParameterStatus add_constant_parameter()

Parameters: UVaL_Reference<LegionInvocation> target
UVaL_Reference<LegionParameter> parameter
int parameter_number

Adds the given parameter to the specified invocation as the ‘parameter_number’th parameter. The parameter is a LegionParameter, which means that it contains an already computed value. The other possible is that it is an invocation parameter. This means that the parameter itself is a LegionInvocation, thus representing a computation that has yet to be performed.

ParameterStatus add_invocation_parameter()

Parameters: UVaL_Reference<LegionInvocation> source
UVaL_Reference<LegionInvocation> target
int source_parameter_number
int target_parameter_number

Adds the given source parameter as a parameter to the given target parameter. This call creates what is called a *Legion Continuation*, and adds the continuation to the continuation list for the source parameter. When an invocation request is eventually executed, the invokee must know where to send the results of the execution. Each continuation identifies a destination LOID to which a result should be sent.

void add_result_dependency()

Parameters: UVaL_Reference<LegionInvocation> source
int parameter_number

The current implementation always sends the return value of a method request back to the invoker. If the invoker wishes to receive other result values that e.g. might correspond to in/out parameters to the method invocation, then those values must be explicitly asked for using add_result_dependency. This call then creates a LegionContinuation corresponding to the requested parameter.

UVaL_Reference<LegionBuffer> get_value()

Parameters: UVaL_Reference<LegionInvocation> inv
int parameter_number

Gets the specified return value for the given LegionInvocation out of the program graph. This is essentially a call through the program graph to the underlying message database. If the result is not available, get_value blocks until it is. A LegionBuffer is returned.

int execute()

Parameters: None

Takes the program graph rooted at the provided invocation and fire it off to be executed. Does not block.

8.2 LegionInvocationStore

int enable_function ()

Parameters: int func_num
int priority

Enables the supplied function so that the LIS will accept method requests for it. Method requests for disabled functions are not accepted. High values for `priority` mean high priority.

int any_ready()

Parameters: None

Checks to see if any method requests are ready.

int any_ready_for_func()

Parameters: int func_num;

Checks to see if any method requests for the given function are ready.

UVaL_Reference<LegionWorkUnit> next_matched()

Parameters: None

Returns the next work unit. The priority scheme is obeyed.

UVaL_Reference<LegionWorkUnit> next_matched_for_func()

Parameters: int func_num;

Returns the next work unit with the given function number.

int set_priority ()

Parameters: int func_num
int priority

Sets the priority for the given function number.

int insert()

Parameters: UVaL_Reference<LegionWorkUnit> new_work_unit

Inserts the provided work unit into the invocation store. This work unit may be a method request or it may be a result from a previous method invocation.

UVaL_Reference<LegionWorkUnit> get_return_value()

Parameters: UVaL_Reference<LegionComputationTag> tag
int parameter_number

Returns the work unit with the given tag and parameter number. Typically, this function is called by a higher layer (e.g. LegionProgramGraph) which will unwrap the returned work unit to get the parameter inside.

int release_return_value()

Parameters: UVaL_Reference<LegionComputationTag> tag
int parameter_number

Deletes the work unit that matches the supplied tag/parameter_number pair.

int release_all_return_values()

Parameters: None

Deletes all return values from the invocation store.

8.3 LegionWorkUnit

`int get_function_number()`

`Parameters: None`

Returns the target function number for the work unit.

`UVaL_Reference<LegionContinuationList> get_continuation_list()`

`Parameters: None`

Return the continuation list for the results of this work unit.

`UVaL_Reference<LegionBuffer> get_parameter()`

`Parameters: int parameter_number`

Returns the given parameter as a LegionBuffer. NULL if the parameter is not in the work unit.

8.4 LegionBuffer

8.4.1 Constructors

`LegionBuffer()`

`Parameters: None`

Creates an empty buffer with a `LegionStorageScat` storage, and default implementations of the packer, compressor, and encryptor function sets.

`LegionBuffer()`

`Parameters: UVaL_Reference<LegionStorage>`

Wraps a default buffer around a given storage object. It uses default implementations for the packer, encryptor, and compressor.

`LegionBuffer()`

`Parameters: LegionMetaData metadata`

Creates a new empty `LegionBuffer` with a `LegionStorageScat` default storage implementation. It instantiates packer, encryptor and compressors based on the metadata

`LegionBuffer()`

```
Parameters: UVaL_Reference<LegionStorage>
            LegionMetaData metadata
```

The full featured constructor wraps a specified storage in a buffer appropriate for the given metadata, selecting the right packer, compressor, and encryptor implementations.

8.4.2 Operations on the associated LegionStorage

A `LegionStorage` exports member functions to read and write untyped characters from and to a logical buffer. These operations are also exported by `LegionBuffer`, but the user should be warned that the "better" way to put data into a buffer is through the `LegionPacker` interface; `read()`and `write()` will not perform appropriate data format conversions, but `put_int()` and `get_int()` will.

`size_t read()`

```
Parameters: size_t num_bytes
            void *data
```

Reads num_bytes bytes from the buffer starting at the current location of the buffer pointer. Copies the bytes into the space pointed to by data_dest. Returns the number of bytes actually read, and positions the buffer pointer immediately after the last byte that was read. `Read()` will not read past the end of the buffer.

`size_t write()`

```
Parameters: size_t num_bytes
            void *data
```

Writes num_bytes bytes pointed to by data into the buffer starting at current position of the buffer pointer (overwriting existing data). Returns the number of bytes actually written, and positions the buffer pointer immediately after the last byte that was written. Writing past the end of the buffer causes it to expand.

`size_t seek()`

```
Parameters: seek_start whence
            int bytes_away
```

Changes the position of the buffer pointer. "`Whence`" can be BEGINNING (0), CURRENT (1), or END (2), and bytes_away tells how many bytes away from `whence` to set the pointer. Seeking past the end of the buffer causes it to expand and to be filled with NULL bytes. Seeking to negative logical positions the buffer does not cause the buffer to expand; the buffer pointer is placed at logical position 0.

`size_t size()`

```
Parameters: None
```

Returns the current size of the buffer in bytes.

`size_t current_byte()`

```
Parameters: None
```

Returns the number of the byte to which the buffer pointer currently points. `Current_byte()` returns 0 when the buffer pointer is at the beginning of the buffer, and `current_byte()` == `size()` when the buffer pointer is at the end of the buffer.

`char *linearize()`

```
Parameters: int pack_metadata=0
```

Returns a pointer to the beginning of the buffer's data. This pointer is guaranteed to point to data that is contiguous in memory. Depending on the implementation of the `LegionStorage`, this function may or may not need to return a pointer to a copy of the data.

`void setMetaData()`

```
Parameters: LegionMetaData md
```

Sets the meta data associated with the `LegionStorage`.

`LegionMetaData getMetaData()`

```
Parameters: None
```

Returns the meta data associated with the `LegionStorage`. This is useful for instantiating an appropriate `LegionBuffer` based on a given `LegionStorage`.

8.4.3 Operations on the associated LegionPacker

A `LegionPacker` exports operations for packing and unpacking the basic C++ data types into and out of a `LegionBuffer` in a particular data format. A `LegionPacker` exports

put_ZZZ() and get_ZZZ() for all ZZZ in {char, short, ushort, int, long, ulong, float, double}.

size_t put_ZZZ()

```
Parameters: ZZZ *source
            int how_many
```

Assumes that "source" points to an array of "how_many" instances of type ZZZ. Copies this data into the buffer after first performing the appropriate data conversion operation if necessary and appropriate for the type of LegionPacker that is instantiated.

size_t get_ZZZ()

```
Parameters: ZZZ *source
            int how_many
```

Assumes that "source" points to enough space for an array of "how_many" instances of type ZZZ. Copies the next data from the LegionBuffer into this space after first performing the appropriate data conversion operation if necessary and appropriate for the type of LegionPacker that is instantiated.

8.4.4 Operations on the associated LegionEncryptor

Since no encryption algorithms have been implemented, the current encryption operations, encrypt() and decode(), are merely placeholders until the right set of encryption operations are defined.

8.4.5 Operations on the associated LegionCompressor

Since no compression algorithms have been implemented, the current compression operations, compress() and decompress(), are merely placeholders until the right set of compression operations are defined.

8.4.6 LegionPackable

int pack()
int unpack()

```
Parameters: LegionBuffer &lb
```

LegionBuffers are themselves packable.

8.4.7 Other functions

show()

```
Parameters: None
```

Prints the contents of the LegionBuffer to stderr. This is done however the associated LegionStorage sees fit.

8.4.8 LegionLOID

LegionLOID is intended to be a base class for LOID's that enforce a particular structure on the fields of the LOID. An LOID contains four private data members, (1) an integer that holds the type of LOID, (2) an integer that indicates how many fields the LOID contains, (3) an array "field_size[]" of integers that holds the sizes in bytes of the LOID fields, and (4) an array "field_value[]" of pointers to the field data. Currently, the only derived class is called LegionGeneralPurposeLOID, which simply exposes the protected members to the public interface.

`LegionLOID()` `(protected)`

    Parameters: int ltype

Sets the type to ltype, sets all other fields to zero.

`LegionLOID()` `(protected)`

    Parameters: int ltype,
                short nfields

Sets the type to ltype. Sets num_fields to nfields. Allocates the field_size[] and field_value[] arrays. Sets all field_size[]'s to 0, sets all field_value[]'s to NULL.

`LegionLOID()` `(protected)`

    Parameters: int ltype,
                short nfields,
                short *fld_size

Sets the type to ltype. Sets num_fields to nfields. Allocates the field_size[] and field_value[] arrays. Sets all field_size[]'s to the values contained in the fld_size[] array. Allocates the field_value[] entries to the right size and zeros them out. This constructor assumes the fld_size array has at least nfields elements.

`LegionLOID()` `(protected)`

    Parameters: int ltype
                short nfields
                short *fld_size
                char **fld_value

Sets the type to ltype. Sets num_fields to nfields. Allocates the field_size[] and field_value[] arrays. Sets all field_size[]'s to the values contained in the fld_size[] array. Allocates the field_value[] entries to the right size and copies the values from fld_value[] array into the field_value[] array. This constructor assumes that the fld_size[] and fld_value[] arrays have at least nfields elements, and that each fld_value[i] points to at least fld_size[i] bytes of space.

`set_field_size()` `(protected)`

    Parameters: short field_num
                short fsize

Sets the appropriate field_size element to fsize and makes sure that the corresponding field_value element is at least fsize bytes. If it is not, it deletes the old field_value entry and allocates a new one. Only derived classes should be allowed to call this member.

`set_type()` `(protected)`

    Parameters: int new_type

Sets the type entry to be new_type. Can only be called from within the code of derived classes.

`LegionLOID()`

    Parameters: LegionBuffer &lb

`LegionLOID()`

    Parameters: LegionLOID &otherLOID

These two constructors are public because neither allows the caller to violate the structure of any particular type of LOID - both just copy the LOID from the parameter, either a `LegionBuffer` or another `LegionLOID`.

`LegionLOID()`

    Parameters: None

Not a very useful constructor, so it prints a warning and assigns all data members to zero. Useful constructors should at least say what the type is.

Accessors

Methods for getting and setting the type and all field values, by field number and field name, exist.

Overloaded operators

The ==, !=, and = operators are overloaded appropriately. An LOID is equal to another only if all fields are identical in size and value.

int is_empty()

`Parameters: None`

Returns 1 only if the LOID is of type `UVaL_LegionLOID_type_EMPTY` (zero).

int is_class()

`Parameters: None`

Returns 1 only if the LOID seems to refer to a class object, i.e. the instance number field is empty.

int same_class_as()

`Parameters: UVaL_Reference<LegionLOID> other_loid`

Returns 1 if the `class_id` field matches that of other_loid.

int pack()

`Parameters: LegionBuffer &lb`

Type and num_fields are packed first, *in network order*. Next num_fields shorts are packed, *in network order*. Next, `num_fields` values are packed.

int unpack()

`Parameters: LegionBuffer &lb`

The `type`, `num_fields`, and `field_size[]`'s are unpacked into host order. The `field_value[]`'s are unpacked without switching the byte order.

int show()

`Parameters: None`

Prints the contents of the LOID to stderr for debugging purposes.

8.5 LegionMessage

LegionMessage()

```
Parameters: UVaL_Reference<LegionLOID> src
            UVaL_Reference<LegionLOID> dest
            int fnum
            int parms_to_expect
            UVaL_Reference<LegionComputationTag> tag
            UVaL_Reference<LegionParameterList> plist
            UVaL_Reference<LegionContinuationList> lcontList
            UVaL_Reference<LegionEnvironment> lenv
```

Creates a `LegionMessage` from the constituent parts passed as parameters.

Accessor functions

LegionMessage exports public member functions to get and set all of its constituent parts .

Overloaded operators

The equality operators (== and !=) are overloaded. Two LegionMessages are deemed equal only if each of the constituent parts are equal, as determined by the equality operators of their respective classes.

show()

Parameters: None

Prints the contents of the LegionMessage to the stderr stream.

int pack()

Parameters: LegionBuffer &lb

int unpack()

Parameters: LegionBuffer &lb

LegionMessage is packable.

8.5.1 LegionParameter

LegionParameter()

```
Parameters: int param_number
            UVaL_Reference<LegionBuffer> lb
```

Constructs a new parameter whose value is assumed to be in lb, and whose number is set to param_number.

Other constructors

The default constructor creates a parameter with a negative parameter number and an empty LegionBuffer. A constructor that takes only a LegionBuffer as a parameter unpacks the contents of the LegionParameter from that buffer. The copy constructor is also overloaded.

Accessor functions

LegionParameter exports public member functions to get and set both the parameter number and the buffer that contains the value of the parameter.

Overloaded operators

The == operator is overloaded to return 1 when the parameter numbers are the same, and 0 otherwise.

show()

Parameters: None

Prints the contents of the LegionParameter to the stderr stream.

int pack()

Parameters: LegionBuffer &lb

int unpack()

Parameters: LegionBuffer &lb

LegionParameter is packable.

8.5.2 LegionParameterList

Constructors

The default constructor creates an empty parameter list, and a constructor that takes a LegionBuffer as an argument unpacks the contents of the LegionParameterList from that buffer.

Set operations

LegionParameterList is derived from templated class UVaL_PackableSet_LinkedList, and therefore exports the full interface of UVaL_PackableSet.

UVaL_Reference<LegionParameter> find()

Parameters: int parameter_number

Augments the UVaL_Set operations to allow parameters to be looked up by number. Returns a reference to the parameter, if found, or a null reference if not.

show()

Parameters: None

Prints the contents of the LegionParameterList to the stderr stream.

int pack()

Parameters: LegionBuffer &lb

int unpack()

Parameters: LegionBuffer &lb

LegionParameterList is packable.

8.5.3 LegionComputationTag

LegionComputationTag simply maintains a glorified interface to a long integer. The Library also contains a class—LegionComputationTagGenerator—that creates random computation tags. See the source code or on-line documentation for a description of that class.

Constructors

The default constructor creates a LegionComputationTag with an uninitialized initial value. A constructor that takes a LegionBuffer as an argument unpacks the contents of the computation tag from that buffer.

Accessor functions

LegionComputationTag exports public member functions to allow the value of the tag to be set and retrieved as a long integer.

show()

Parameters: None

Prints the contents of the LegionComputationTag to the stderr stream.

int pack()

Parameters: LegionBuffer &lb

int unpack()

Parameters: LegionBuffer &lb

LegionComputationTag is packable.

8.5.4 Other fields

A LegionMessage also contains a LegionContinuationList, and a LegionEnvironment. A LegionContinuationList is simply a UVaL_PackableSet of LegionContinuations, and a LegionEnvironment is a UVaL_PackableSet of LegionEnvironmentItems. Please refer to the online documentation and source code for the interface to these classes.