Legion 1.8

System Administrator Manual

The Legion Group

Department of Computer Science
School of Engineering & Applied Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740
legion@virginia.edu
http://legion.virginia.edu/

This work partially supported by DARPA (Navy) contract # N66001-96-C-8527, DOE grant DE-FD02-96ER25290, DOE contract Sandia LD-9391, Northrup-Grumman (for the DoD HPCMOD/PET program), DOE D459000-16-3C and DARPA (GA) SC H607305A

Before you start

1.0 Introduction

1.1 About this manual

This manual is for system administrators of a Legion system. It explains how to install, run, and manage a new Legion system, how to set security features, and how to use the Legion core objects. It also presents several sample system policies.

There are four Legion manuals, each aimed at a specific type of user, that can be consulted for more information. The others are:

- **Basic User Manual**: introductory information about the system.
- **Developer Manual**: information and documentation for programmers working in Legion, and includes information on languages, libraries, core objects, and implementing new Legion objects.
- **Reference Manual**: detailed information about specific elements of the Legion system.

There are also man pages for all Legion commands included with the system files, and on-line tutorials on the Legion web site (<http://legion.virginia.edu>).

1.2 Style conventions

The manuals at times refer to path names in Unix directory space and in Legion context space. To avoid confusion, the following style conventions are used throughout Legion documentation:

- Unix, DOS, and local path names appear in a serif typeface.
- Functions, method names, parameters, flags, command-line utilities (such as `rm`, `cp`, and `legion_ls`), and context path names appear in `fixed typeface`.

1.3 About Legion

Legion is a grid operating system at the University of Virginia. It is intended to support the construction of wide-area virtual computers, or metasystems, which will allow users to work on a variety of geographically distributed, high-performance machines and workstations. Legion is designed to support large degrees of parallelism in application code and to manage the complexities of the physical system for the user in order to take advantage of this enormous physical infrastructure.

Installing and running Legion

2.0 Downloading Legion 1.8

Legion 1.8 is available in binary form only. Information about downloading 1.8 is on the Legion web site (<http://legion.virginia.edu/download/index.html>) and below. Legion is currently available for the following platforms:

- Sun Workstation/Solaris 5.7 or later (`solaris`)
- SGI Workstations/IRIX 6.5 n32 build or later (`sgi_n32`)[1]
- SGI Workstations/IRIX 6.5 n64 build or later (`sgi_n64`)
- x86/Red Hat 6.x Linux (`linux`)
- DEC Alpha/Red Hat Linux 6.x (`alpha_linux`)
- DEC Alpha/OSF1 v4 (`alpha_DEC`)
- IBM RS6000/AIX 4.3 (`rs6000`)
- HPUX 11.x (`hppa_hpux`)[2]

Legion has been ported to several Cray IEEE architectures (C90, T90, T3E) using the native Cray C++ compiler. However, the binaries produced are excessively large, due to a compiler bug. Therefore, we are not releasing a binary version of Legion for the Cray platform. You can start a virtual host on these machines (see page 69).

We are no longer supporting the x86/FreeBSD 4.2 platform, although we will consider adding it back in if someone needs it. We are not currently supporting a Windows platform.

The Legion software is currently available to qualified educational, research, and commercial customers. However, we ask that any interested users submit the following information:

1. Your full name, e-mail address, phone number, and mailing address.
2. Your organization or university's full name and (if applicable) line of business.
3. A brief description of what you wish to do with Legion (applications you wish to use or develop, research you wish to pursue, etc.).

This will be used to determine appropriate licensing agreements only and will not be distributed or sold to any outside parties. Please send it via e-mail to <legion@virginia.edu>.

1 We have previously supported SGI Workstations IRIX o32 build (sgi_o32) but it is not available in this release. Please contact us at <legion-help@virginia.edu> if you have any questions.

2 The HPUX 11 platform is available upon request. We will include an HPUX 10 platform in a future release.

If you are interested in a corporate license, Avaki (<www.avaki.com>) is the corporate distributor of Legion software.

2.1 Legion structure

As of version 1.8, Legion has been restructured. It is now split into packages. This change does not affect any tools or context space, but you may need to update paths in makefiles or change library paths from previous Legion versions. You need to run extra commands to start up some of the packages (see page 21).

1. **Core**: This is the basic Legion package and the minimum for running a Legion system. It lets you start up and shut down Legion, work in context space, run Legion security, etc.
2. **Software Development Kit** (SDK): This contains development-oriented tools and libraries, such as the stub generator, Legion Grid library, LegionArray library, etc. This package is not necessary if you aren't planning on writing Legion applications.
3. **High-Performance Computing** (HPC): The HPC module lets you run your programs in Legion. It contains PVM and MPI tools, the two-dimensional FileObject interfaces, JobProxy and JobQueue objects, batch queue class and host object, and `legion_run` and `legion_run_multi`.
4. **Extra**: This adds functionality to the basic Legion package. It contains the round robin scheduler, simple k-copy class (SKCC), process control daemon host objects, etc. It is not necessary, but it gives you more control over your objects. You must have the HPC package in order to use the Apps package.
5. **Applications** (Apps): The Apps package also extends the basic Legion package. The current version gives you more flexibility in moving files between Legion and your desktop, via `legion_export_dir` and the Legion FTP daemon.

2.2 Space requirements

You will need about 250-300MB of free disk space and at least 256MB virtual memory (we suggest 512MB, if possible) on your bootstrap node in order to run Legion.

2.3 Software requirements

You must have `/bin/ksh` installed in your system. There are a number of Legion scripts that will look for `ksh`, and if it is not installed in your system you will get error messages.

Depending on your platform and package, you will need a set of GNU tools (all available from <http://www.gnu.org>). The NT/2000 platform doesn’t need any GNU tools. Section 2.5 lists tools for untarring binary files.

2.4 RSA and OpenSSL

Current Legion release use public key cryptography based on the RSA 2.0 algorithm, as implemented by OpenSSL. You will need to download OpenSSL 0.9.5 or higher from <http://www.openssl.org>. You’ll need to untar, configure, and compile it. Be sure that you set your $OPENSSL_INC and $OPENSSL_LIB variables to the correct directory. Suggested values are:

(ksh or sh users)

```
export OPENSSL_INC=<OpenSSL installation directory>/include
export OPENSSL_LIB=<OpenSSL installation directory>/lib
```

(csh users)

```
setenv OPENSSL_INC <OpenSSL installation directory>/include
setenv OPENSSL_LIB <OpenSSL installation directory>/lib
```

2.5 Downloading binary files

All Legion distribution binaries are compressed tar files created with GNU `tar` and `gzip`. Using a non-gnu tar program may result in some files names being truncated. You can download these tools for free from GNU at <http://www.gnu.org>.

Decide where you want your Legion root directory to reside and move the distribution file to that directory. Next, uncompress and untar the file (`<platform_name>` is one of {`solaris, sgi_n32, sgi_n64, linux, alpha_linux, alpha_DEC, rs6000, hppa_hpux`}). The binaries files are all compressed tar files created with GNU `tar` and `gzip`.

If you have GNU `tar` you can unzip and untar the binary by running:

```
tar zxvf Legion-binary-<platform_name>-V1.8.tar
```

If you do not have GNU `tar`, you must use `gunzip` (part of GNU `gzip` package). Run the following:

```
gunzip -c Legion-binary-<platform_name>-V1.8.tar.gz | tar -xvf -
```

This will create a root directory called “Legion” in the current directory, and will include all necessary sub-directories and files.

Note that the compressed binary files are large (ranging from 12 to 100+ MB), so it may take a few minutes for them to arrive, and that the uncompressed tar file will be about two to three times larger. The system will actually need even more space once it is running, since it will be making copies of some of the files. In addition, the binary tar files do not include intermediary object files, which will be created when the system is started.

3.0 Starting a new system

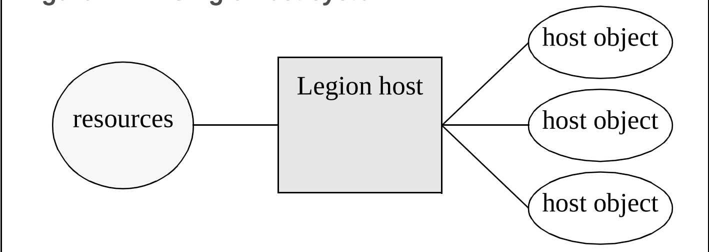
A summary of the start-up procedure is on page 28.

3.1 Before you start

Before you start a new Legion system, consider what type of set-up will best suit your needs. Primary considerations include:

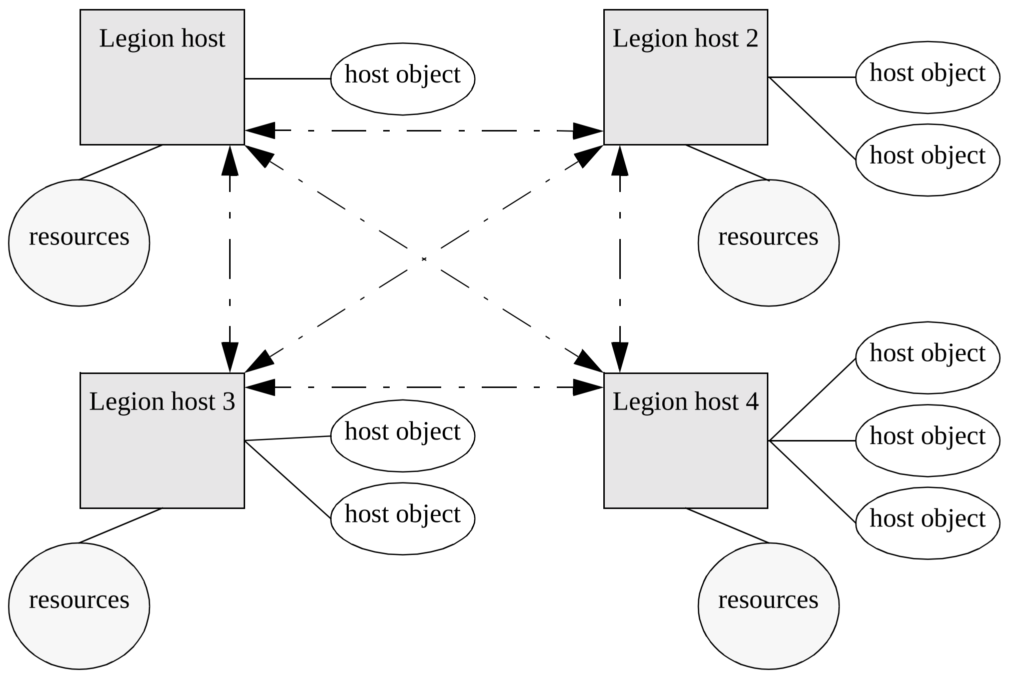
- What kind of system do you need? How many machines do you anticipate using? Do you only use local hosts or do you also use remote machines? Possible configurations might include:
    a. **A single host system**: one Legion machine with one or more host objects (Figure 1). This is the simplest system.

Figure 1: Single host system



    b. **A multihost system**: multiple Legion hosts linked together and sharing local resources (Figure 2).

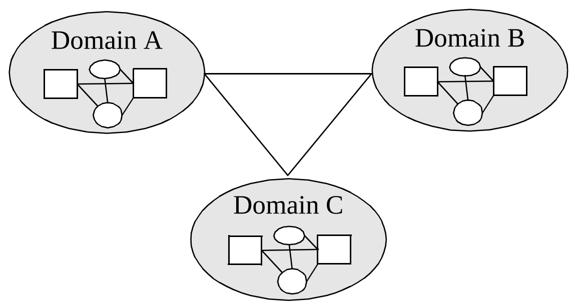
Figure 2: Multihost system



This can include homogeneous or heterogeneous platforms, as well as non-Legion machines. The machines do not need to be in physical proximity.

c. **A multidomain system**: multiple Legion domains connected together and sharing each others' resources (Figure 3).

Figure 3: Multidomain system



- Will you be using Legion security? This is an important consideration, since a secure Legion system cannot be cleanly shut down and restarted. If you are using security, you should decide who will be the "admin," the system administrator. The system should be started up by the admin, since he or she will have special privileges on the core objects created in the new system. (These privileges do not extend to other users' objects, however.)
- What kind of host objects will you be using? Options include:

a. **Basic host object**: it resides on its host and manages and guards its host's resources. This is the template for the other host objects. See "Host and vault objects" on page 30 in the Basic User Manual for information on basic host objects. In this manual, see section 11.0, starting on page 49, for a discussion of host-vault pairings and adding new hosts.

b. **PCD host object**: it resides on its host, manages and guards its host's resources, and uses a process control daemon to regulate ownership of all Legion processes executed on that host. If you use a PCD host as your bootstrap host, the start-up process will be slightly different. For more information, please see "Process control daemon host objects" on page 59.

The daemon requires root privileges to start and to run. The PCD host object is useful if outside users will be running processes on your host, but can only be used if Legion security is enabled. Each user's processes will be tracked and accounted for.

c. **Batch queue host object**: it resides on its host, manages and guards its host's resources, and submits Legion jobs to the local queueing system.

   This is the best choice for hosts that use a queue management system, although the PCD host object is more secure and has better accounting. For more information, please see "Batch queue host objects," page 66.

d. **Virtual host object**: it resides on a different host, represents and guards its host's resources, but does not run normal Legion objects. A virtual host cannot be used as a bootstrap host: it is added to an already running system.

   A virtual host object is used for running Legion jobs on unsupported platforms. The host object resides on a supported platform and runs native jobs with standard Legion tools on the target host machine. It can be used for scheduling, resource selection, and transparent execution on the target machine. For more information, please see "Virtual hosts," page 69.

3.2 Set up the environment

A properly set-up environment is crucial for working in the Legion system. The start-up process uses certain Legion-specific environment variables, which must be correctly set before starting applications and running command-line utility programs. You must set these variables each time you starting working in Legion. Without a properly set environment, programs cannot communicate with other objects in the system, and the program may terminate with an error, never return a value, or fail in a more spectacular fashion. If this occurs, try setting your environment properly and starting over.

You must have /bin/ksh installed in your system. There are a number of Legion scripts that will look for `ksh`, and if it is not installed in your system you will get error messages.

If you have not yet done so, set $OPENSSL_INC and $OPENSSL_LIB (see page 10). You must also set $LEGION_HOME and $LEGION_OPR and run the legion_profile.[c]sh script.[3] The environment must be properly set in each shell in which you plan to run Legion commands. Check to be sure that environment variables are properly set.

(ksh or sh users)

```
export LEGION_HOME=<Legion root dir path>
export LEGION_OPR=<Legion OPR root dir path>
export OPENSSL_INC=<OpenSSL installation directory>/include
export OPENSSL_LIB=<OpenSSL installation directory>/lib
. $LEGION_HOME/legion_profile.sh
```

3 Bourne Shell is not directly supported by our implementation of Legion, due to the use of `alias` to implement some Legion commands. Bash, however, is supported.

(csh users)

```
setenv LEGION_HOME <Legion root dir path>
setenv LEGION_OPR <Legion OPR root dir path>
setenv OPENSSL_INC <OpenSSL installation directory>/include
setenv OPENSSL_LIB <OpenSSL installation directory>/lib
source $LEGION_HOME/legion_profile.csh
```

We suggest $LEGION_HOME/../OPR for the OPR root directory path)

3.3 Starting a single host system

You must have the Legion module on your bootstrap host in order to start a new system. Once you've untarred it (see page 10) you'll need to configure, start, and then initialize the new system. You'll use three commands:

```
legion_setup_state
legion_startup
legion_initialize
```

3.3.1 Choose a bootstrap host

The bootstrap host is where you start and shut down your system.[4] It must be able to hold:

- The LegionClass object (the root of the Legion binding mechanism and the parent class of many metaclasses), and
- the $LEGION_OPR/LegionClass.config file.

This file will be created on this host and will contain the LegionClass object address, which must be globally known to all Legion objects. The file must be available when other objects are started in the system.

If you choose a PCD host as your bootstrap, the start-up procedure is slightly different than for a basic host object.

3.3.2 Set up and configure

You must first set up the initial state for core Legion system objects. Legion system objects are persistent, and can save and restore their own state. Some of these objects must have their state initialized before they run for the first time. After the initial start-up, these objects will manage their own state and configuration, if the system is properly maintained.

4 Once fully operational, Legion does not automatically shut down and restart: the system is intended to stay up.

If you are booting on a PCD host, first run:

```
$ LEGION_HOST_BIN=PCDUnixHost
```

Then run

```
$ legion_setup_state
```

to configure the system. This program will return your start-up host name, a port number for the LegionClass object, and a time. If you do not want to use the default settings, use the `-i` flag to run the command in an interactive mode. Your output will look something like this:

```
$ legion_setup_state
Creating OPR directory /home/xxx/OPR/.
Saving LegionClass configuration file:
   /home/xxx/OPR/LegionClass.config
LegionClass host name  = your.startup.host.name
LegionClass port number = 7899
LegionClass timestamp  = 898198093
$
```

The script creates the $LEGION_OPR directory and several sub-directories, populating them with initial states for several core system objects. The timestamp sets the starting time for the system: Legion objects use a timestamp to guarantee each object’s unique identity. The current time is measured in seconds since January 1, 1970.

3.3.3 Start up

The `legion_startup` script provides prompts asking whether or not to start each component. It's best to answer “yes to all” (Y). The `verbose` option allows you to see more detailed information, as the script works, about debugging. (This can be large amounts of information, so use this option only if you are searching for a problem.)

To start the main core system objects, enter

```
$ legion_startup
```

Legion will start several classes on your host. The output shows major class objects starting up.

```
$ legion_startup
Starting meta-class object: LegionClass
Continue (startup) (y=yes, Y=yes to all, n=no, N=no to all,
   v=verbose, V=verbose all)? Y
```

```
    Starting meta-class object: BootstrapMetaClass
    Starting class object: DefaultBindingAgentClass
    Starting class object: CommandLineClass
    Starting class object: UnixHostClass
    Starting class object: UnixVaultClass
    Starting class object: DefaultImplementationClass
    Starting class object: DefaultImplementationCacheClass
    Starting class object: DefaultContextClass (SKCC enabled)
    Done with DefaultContextClass
    Legion first-time system startup complete
    $
```

The first object created, `LegionClass`, is the highest-level metaclass (the meta-metaclass) and the parent of every other object in the system. The next object, `BootstrapMetaClass`, is the class object for bootstrap class objects (i.e., class objects whose instances must be created in the initialization phase).

The bootstrap class objects are:

UnixVaultClass
UnixHostClass
UnixImplementationClass
UnixImplementationCacheClass

These are started up a bit further down, as the output shows.

The next new classes also start instances in the new system but are not bootstrap class objects: `BindingAgentClass` parents binding objects, `CommandLineClass` parents command-line objects, etc.

3.3.4 Initialize

Like `legion_startup`, the `legion_initialize` script will provide prompts asking whether or not to perform each task, and it is generally best to use the “yes to all” option (Y). To initialize Legion, enter:

```
        $ legion_initialize
```

The output shows the system creating and tagging the key ingredients of a new system. It is too long to reproduce in full here, but we’ll look at some selected actions.

```
    Creating host object BootstrapHostObject on
             “your.current.host.name”
    Continue (y=yes, Y=yes to all, n=no, N=no to all,
             v=verbose, V=verbose all)? Y
    Configuring wellknown binaries for host “1.01.07.0100...“
```

The first line shows the system creating a **bootstrap host object** on your current host (a host object manages a host, so the bootstrap host object manages the bootstrap host).

```
Creating vault object BootstrapVaultObject on
      “your.bootstrap.host.name”
Setting BootstrapHost and BootstrapVault restrictions
Added 1 host(s) to vault's compatibility set
Added 1 vault(s) to host's compatibility set
```

A **bootstrap vault object** is automatically created on your current host (a vault object manages a vault, which stores Legion object’s permanent states). This guarantees that the bootstrap host object has a compatible vault object. All host objects must be paired with at least one compatible vault object (i.e., a vault that it can “see”).[5]

```
Creating an ImplementationCache
Creating an implementation (ContextObject) for ContextClass
Creating the root context object
```

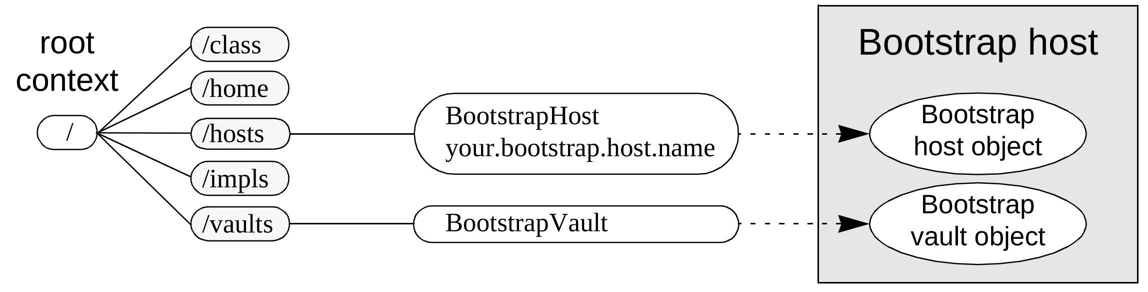
Implementation objects represent and manage the implementation cache (used to allow Legion processes to take place in different architectures) and context space.

```
Adding "BootstrapHost" to the hosts context
Adding the alias "your.bootstrap.host.name" for
      BootstrapHost to the hosts context
Adding "BootstrapVault" to the vaults context
```

More implementation objects are created as the process creates new object classes.

Two context names are added to the `/hosts` context, `BootstrapHost` and `your.bootstrap.host.name`. Both names refer to the Bootstrap host object but only one is added to the `/vaults` context (Figure 4).

Figure 4: Context paths for the bootstrap host and vault objects



This object will manage a portion of the persistent storage mechanism for the Bootstrap host. The `/impls` context contains names of the

5 For more about hosts and vaults, see section 6.0 in the Basic User Manual. For more about host-vault pairs, see page 49 in this manual.

default implementation objects (see "Implementation model," pg. 44). These can all be viewed with context-related commands or the GUI once the system has been completely started.

Before finishing, legion_initialize generates a set-up script for the new system. This script is placed in the $LEGION_OPR directory. Please see section 3.5 for more information on using this script.

The basic Legion system is now ready to go. If you wish to start security, follow the steps in section 3.3.5, below. If you have other modules, follow the steps in section 3.3.6 on page 21.

3.3.5 Set security

If you wish enable Legion security, run the `legion_init_security` command. You’ll have to decide now whether or not you want security, since the command will not run properly unless you run it immediately after initializing the system. If you don’t wish to use it, just skip over this section. However, none of your processes will be protected and you won’t be able to create individual accounts.

To use Legion security, run:

```
$ legion_init_security
```

Please note that this command is for the Legion module and will not work with all classes in the other modules. For best results, start each module’s security when you initialize it (see section 3.3.6 on page 21).

Several events take place when you run this command.

```
$ legion_init_security
Creating the context "/users" to contain user-objects
Creating the initial system-admin user object, "/users/admin"
Please select a Legion password for "/users/admin":
New Legion password: xxxx
Retype password: xxxx
1.399b330d.6f000000.01000000.000001fc0cd...
Please enter the Legion password for "/users/admin" to
     continue:
Enter Password: xxxx
You have successfully logged in.
```

First, Legion creates a `/users` context. This context contains all Legion user ids. Since you need a user id to work in a secure system, you are automatically assigned a system administrator user id called `admin`. Anyone logged in as `admin` has root privileges in the system and can create new users, modify security settings, etc. The admin user also has ownership of all *existing* objects in the new system but not any future objects that other users create.

You must create a password for admin. You’ll be asked to enter it three times during the `legion_init_security` process.

Once you're logged in, Legion gives you ownership of all existing objects in the system.

```
          Changing ownership of all objects to "/users/admin"
          1.3622260c.01..000001fc0cbe1846763f895a...
          1.3622260c.02..000001fc0b3b16eb8b2dde29...
          [...etc.]
          Changed ownership of 63 objects.
```

After this point, any new objects created will belong to whoever created them.

Legion then configures security for the new system's resources. It creates access control lists (ACLs) for all existing core classes and their instances.

```
     Configuring security for the default collection
     Creating initial ACLs files for all core objects in
          /home/spw4s/OPR/init_acls
     Creating ACL for /class/AuthenticationObjectClass class
     Creating ACL for /class/BasicFileClass class
     Creating ACL for /class/BasicSchedulerClass class
     Creating ACL for /class/BatchQueueMetaClass class
     [...etc.]
     Creating ACLs for instances of /class/BasicSchedulerClass
     Creating ACL for
          1.399b330d.68000000.01000000.000001fc0b3da560eff6580b
          840f3e7a76b4c82beb9b421ce47ff465557c914bc4bb3ba85140b
          3444091bdf45dca6e50deac309b02d420b631b886619ea276de13
          72260b
     Creating ACLs for instances of /class/BatchQueueMetaClass
     Creating ACL for
          1.399b330d.73000000..000001fc0cd9a1202afc0753365c8441
          c69ffcbcd9ccd235c5c72603707b855aaf543dc6632731d932286
          18948049c13dba35de9727993c1e4abe7467c232cb16c05c831
     ... already done
     Creating ACLs for instances of /class/BootstrapMetaClass
     Creating ACL for
          1.399b330d.04..000001fc0e608816a569dbc1d0503434eedcd9
          d7cbb97112871ff09e32482308466f094531d7b332007536be821
          a598bb4aa4cbdbd9731592bdc06167c028403f5ab8945
     ... already done
     [...etc.]
```

The access control lists (ACLs) protect objects against unauthorized use. Only an object's creator can use the object, unless the creator specifies otherwise. The initial ACL files allow only the `admin` to use the core objects.

```
          Setting ACL for /class/AuthenticationObjectClass class
          Setting ACL for /class/BasicFileClass class
          Setting ACL for /class/BasicSchedulerClass class
          [...etc.]
          All acls set.
```

```
You have successfully logged out.
$
```

When all necessary ACLs have been set, you are logged out.

At this point, security has been enabled and is running. ***You must now log in as `/users/admin`.***

```
$ legion_login /users/admin
Password: xxxx
$
```

Legion's security is now enabled. For more information about security see "About Legion security" on page 30 and "Using security features" on page 35.

3.3.6 Starting up other packages

The steps outlined in sections 3.3.2 - 3.3.5 initialized only the Legion package. If you are using any others, you must initialize them by hand by running the appropriate `legion_init_<module name>` tool.

If have run `legion_init_security`, Legion security will automatically be enabled in each package when you initialize it.

Due to internal dependencies, you need to initialize in a specific order: HPC, Extra, and Apps. So, if you have all of the packages, you would run the following tools, in this order:

```
$ legion_init_HPC
$ legion_init_Extra
$ legion_init_Apps
```

3.4 Starting a multihost system

This is a two-part process. First, you have to have a running single host system, as laid out in section 3.3. Second, you add new host objects on the desired machines. Since you will be starting processes on the target hosts from the bootstrap host be sure that you can run `rsh`/`ssh` on the bootstrap host as well as on the target hosts from the bootstrap host without having to enter a password. You can set up a .rhosts file for `rsh` or an authorized_keys files for `ssh` to accomplish this (see the `rsh` and `ssh` man pages for more information).

3.4.1 Set up

You'll need to set the proper environment variable on the bootstrap host and the remote host(s) so that you can run Legion commands on a remote host using `rsh` or `ssh`.

For sh, ksh, or bash:

```
LEGION_RSH=<rsh|ssh>
LEGION_RCP=<rcp|scp>
export LEGION_RSH LEGION_RCP
```

For csh:

```
setenv LEGION_RSH <rsh|ssh>
setenv LEGION_RCP <rcp|scp>
```

Set these variables on the bootstrap host before you start the new system (i.e., before you run `legion_startup`). Please note that you only need to follow these steps on the bootstrap host: you will need to install the Legion binaries on any other machines that you add to your system, but you do not need to start more Legion systems.

To add additional hosts and users, copy the $LEGION_OPR/setup.[sh|csh] scripts to a globally accessible location. Hosts can share an NFS-mounted Legion tree,[6] but for best results you should place the OPRs on a local disk.

3.4.2 Create a new vault

If the new host will not be compatible with your existing vaults, create a new vault object with the `legion_startvault` command.

```
$legion_startvault /hosts/Bootstrap
```

There are several flags that you can use to set $LEGION, $LEGION_OPR, architectures, etc. Please see page 43 in the Reference Manual for more information about this command.

See page 56 for more information on new vaults.

3.4.3 Create a new host

Use the `legion_starthost` command to create a new host object on the desired host. For example, to start a new host object on MyNewHost, you would enter:

```
$ legion_starthost myNewHost.DNS.name \
  /vaults/BootstrapVault /hosts/myNewHost
```

The same command, with `-B`, will start a new PCD host object.

```
$ legion_starthost -B PCDUnixHost \
  MyNewPCDHost.DNS.name /vaults/BootstrapVault \
   hosts/myNewPCDHost
```

6 See `rpc.lmountd` and `rpc.lnfsd` in the Reference Manual on page 109 for more information about NFS-mounted Legion systems.

See also page 59 for more information on adding PCD hosts.

The same flag can be used to start a new batch queue host object. You'll need to update the host object's attributes to include the queue type.

```
$ legion_starthost -B BatchQueueHost \
  MyNewBQHost.DNS.name /vaults/BootstrapVault \
  hosts/myNewBQHost
$ legion_update_attributes /hosts/myNewBQHost \
  -a "host_queue_type('<Queue_type>')"
```

See page 66 for more information on adding batch queue hosts.

There are several other flags and options with this command described on page 42 in the Reference Manual. See page 52 in this manual for more information on new hosts.

If the new host has a new architecture, you now need to add implementations of the core objects for the new architecture. Log in to the new machine and run the following:

```
$ source <path_to_globally_visable_setup_script>/setup.[sh|csh]
$ legion_login /users/admin   [run legion_login only if you
                               have enabled Legion security]
$ legion_init_arch
```

Repeat these steps for each additional host. We suggest that you customize these steps and write a script to simplify the process, especially if you need to bring up a big net.

3.4.4 Adding new users to a secure net

If you have not initialized security, there are no user accounts.

Only `admin` can add users to a secure net. Run the `legion_create_user` command with the new user's name. We suggest that you put all users in the `/users` context. I.e.,

```
$ legion_create_user /users/<new user name>
```

New user accounts are available immediately after creation.

If you are working on a PCD host, follow up with these steps:

```
$ for i in 'legion_ls /hosts'
$ do
$ legion_add_host_account /hosts/$i <unix_id> \
 /users/<new_user_name>
$ done
```

3.4.5 Working in the new net

Users can work in an insecure net by entering:

```
$ source <path_to_globally_visible_setup_script>/setup.[sh|csh]
```

Users can work in a secure net by entering:

```
$ source <path_to_globally_visible_setup_script>/setup.[sh|csh]
$ legion_login /users/<user_name>
```

Please note that neither of these procedures will open a separate shell.

3.5 Making a set-up script for users

We strongly suggest that Legion system administrators use a set-up script for users to source when starting work in Legion. In version 1.8 and forward, Legion will automatically generate a setup script (called setup.[c]sh) in your $LEGION_OPR directory when you run `legion_initialize`. It contains information about your Legion environment variables and should be run before you start working in a new shell. You can edit the script as necessary and distribute it to other users in your system.

You can also run the `legion_make_setup_script` command to generate a set-up script. The usage is:

```
legion_make_setup_script
    [-o <script basename>]
    [-OPR <OPR dir name>]
    [-L <$LEGION dir name>]
    [-debug] [-help]
```

Supported options are:

-o `<script basename>`	Specify the basename for the resulting setup scripts ($LEGION_OPR/setup is the default). This command will generate two setup scripts, one for /bin/sh derivative users and one for csh-derivative users. The scripts will be named `<basename>.sh` and `<basename>.csh`, respectively.
-OPR `<OPR dir name>`	Specify the OPR directory name that will be set up when the resulting scripts are run. This directory will contain the user's local copy of LegionClass.config (default is `<user>`-OPR). The user's local version of the directory will be placed in the user's $HOME.

-L `<$LEGION dir name>`	Specify the value of $LEGION, which is the directory where the resulting scripts are run. The default is the current value of $LEGION.
-debug	Catch and print Legion exceptions.
-help	Print command syntax and exit.

4.0 System shutdown

Once the system is running, Legion can be safely shut down and restarted without a loss of state. Creators of user object classes can choose whether or not to maintain the state of their objects. A summary of the steps for shutting down is on page 29.

4.1 Shutting down an insecure system

If you did not enable Legion security, issue the shutdown command from the bootstrap host:

```
$ legion_shutdown
```

It may take several minutes to shut down the system. Be patient, and do not try to quit the process with ^C. When completed the entire system, with the notable exception of any extra hosts and vaults that were started separately, will be deactivated.

4.2 Shutting down a secure system

If you are using a PCD host as your bootstrap, the shutdown process is a bit complicated. There is no “root” user and each user owns his or her own objects. Please contact us at <legion-help@virginia.edu> if you have any questions about this.

To shut down a secure system, run `legion_shutdown` while logged in as `admin`. You may need to clean up after the system by hand, i.e. kill the processes one by one from the command line (you can use `ps` to check that all Legion processes have been killed).

5.0 System restart

Once a system has been safely shut down, it can be restarted without repeating the configuration and first-time initialization process. If the system was not safely shut down, you cannot restart it but must start again with the configuration and start-up procedure (i.e., run `legion_setup_state`, etc.). In that case be sure to remove the $LEGION_OPR directory and to kill any extraneous Legion processes (use `ps` to check this).

For a normal system restart, check to be sure that environment variables are properly set. If necessary, run the following:

(ksh or sh users)

```
export LEGION_HOME=<Legion root dir path>
export LEGION_OPR=<Legion OPR root dir path>
export OPENSSL_INC=<OpenSSL installation directory>/include
export OPENSSL_LIB=<OpenSSL installation directory>/lib
. $LEGION_HOME/legion_profile.sh
```

(csh users)

```
setenv LEGION_HOME <Legion root dir path>
setenv LEGION_OPR <Legion OPR root dir path>
setenv OPENSSL_INC <OpenSSL installation directory>/include
setenv OPENSSL_LIB <OpenSSL installation directory>/lib
source $LEGION_HOME/legion_profile.csh
```

Go to the start-up host and run the start-up command:

```
$ legion_startup
```

Do not rerun `legion_initialize`. The objects created when you first ran it are still in the system, just in an inert state until the system is restarted. They will be reactivated and their state reloaded as necessary.

A summary of restarting is on page 29.

6.0 Summary of commands

6.1 Starting a new system

1. Set the following environment variables: $LEGION_HOME, $LEGION_OPR, $OPENSSL_LIB, and $OPENSSL_INC.
2. Run the legion_profile.[c]sh script (page 14)
3. Start and initialize the system (page 15):

   $ LEGION_HOST_BIN=PCDUnixHost
   [If using a PCD host as bootstrap]
   $ legion_setup_state
   $ legion_startup
   $ legion_initialize

4. If desired, start security (page 19). You must login in as admin after running `legion_init_security`:

   $ legion_init_security
   $ legion_login /users/admin

5. Start other packages, as necessary (page 21):

   $ legion_init_HPC
   $ legion_init_Extra
   $ legion_init_Apps

6.2 Start working in a running system

1. If the Legion environment has not yet been set, run the setup.[c]sh script:

   $ $LEGION_OPR/setup.[c]sh

   OR

   Set the following four environment variables and run the legion_profile.[c]sh script:

   ```
   $LEGION_HOME
   $LEGION_OPR
   $OPENSSL_LIB
   $OPENSSL_INC
   ```
   $LEGION/bin/legion_profile.[c]sh

2. Login if necessary.

   $ legion_login /users/<user id>

6.3 Shutdown

1. Run the shutdown command (page 26):

```
$ legion_shutdown
```

2. If security was enabled, you'll have to clean up some objects by hand.

6.4 Restart

If your system shut down successfully

1. Run the startup command (page 27):

```
$ legion_startup
```

2. Run the setup.[c]sh script, which may be as simple as:

```
$ setup.[c]sh
```

OR

Set the following four environment variables and run the legion_profile.[c]sh script:

```
$LEGION_HOME
$LEGION_OPR
$OPENSSL_LIB
$OPENSSL_INC
$LEGION/bin/legion_profile.[c]sh
```

3. Login if necessary.

```
$ legion_login /users/<user id>
```

If your system did not shut down successfully

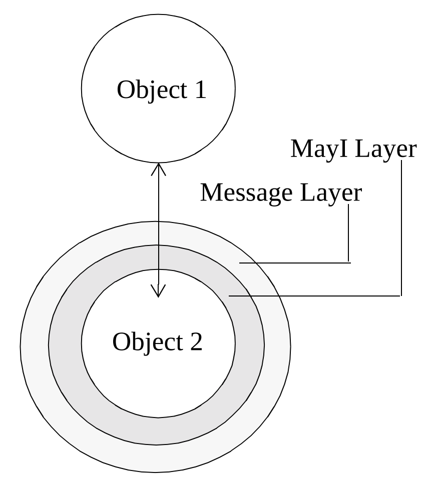
1. Kill any lingering Legion processes and remove the $LEGION_OPR directory.
2. You must start again from scratch. Repeat the steps outlined in section 6.1 above.

Legion security

7.0 About Legion security

Legion's security model has two layers, shown in Figure 5. The message layer is responsible for ensuring that communications between Legion objects are secure. The MayI layer is responsible for access control, and it determines what objects (or users) are allowed to call a particular object's methods. It also relies on the message layer for some services, such as encrypting rights certificates.

Figure 5: Legion security model



7.1 Message layer

The message layer intercepts every message that is sent from or received by an object. For outgoing messages, the layer uses the implicit parameters associated with the message to determine what security measures to apply. Implicit parameters here are similar to Unix environment variables, although their values are not restricted to strings.

An outgoing message can be sent in three ways: in the clear, in *protected mode,* or in *private mode*. When a message is sent in the clear no encryption or other security processing is applied to the body of the message. An eavesdropper can extract any information from the message, and if certificates are included (explained below) it can use them in constructing fraudulent methods calls.

In protected mode, the body of the message is not encrypted. However, any certificates sent with the message are encrypted, as are the tags uniquely identifying the method call with which this message is associated. The body of the message is cryptographically digested. These transformations yield several guarantees. First, the certificates cannot be extracted and used by an attacker in another message. Second, the attacker cannot modify the message in any way, although he can copy it and replay it. Finally, the attacker cannot forge a reply message (e.g., the return value for a Legion method call) because he cannot access the tags.

Private mode encrypts the entire message. Full encryption provides the same features as protected mode as well as privacy (in protected mode, an attacker can still read the messages going by, even if he can't modify or forge them). For small messages full encryption is not appreciably slower than protected mode and may even be faster.

The choice of message security mode is stored in the current implicit parameters. These parameters are inherited through call chains, so if one object calls another object in private mode, the called object will use private mode for any messages it sends on behalf of the original caller. There is also a special implicit parameter for message security that is *not* inherited; it can be used to send an encrypted message and receive an unencrypted reply.

You can use the `legion_configure_profile` command to change the security mode.

The basis for all of these security mechanisms is public key encryption. The default size for public/private keys is now 1024. In addition, the `keypair_len` attribute can be set for any class object (like the AuthenticationClassObject) and future instantiations of that class object will be created with the `keypair_len` keysize. When an object's LOID is created, typically by a class object, it is given a newly generated public key pair. The public part of the key becomes part of the object's LOID, and it can be used by others to encrypt their communications to that object or to verify that received messages were actually generated by the object. The private key of the object is never revealed. The object's class, host, and vault all could potentially access it, although in the University of Virginia (UVa) Legion implementation they do not. In Legion an object implicitly trusts its class, host, and vault.

Because public key encryption is expensive, caching is used so that objects engaged in repeated communications can reuse a DES session key. These cached keys eventually time out (after thirty minutes in the current release) and are refreshed.

7.2 MayI layer

Though the message layer can protect individual messages, it cannot stop an attacker from simply calling the methods of an object. The MayI layer fills this role.

When an object that has a MayI layer is called, MayI examines the method call before the method is actually invoked. (The name of MayI comes from the idea that the caller is asking “May I call this method?”) If the call passes the access control policy being enforced by MayI, it is allowed. Otherwise, a security exception is returned to the caller.

Legion objects are automatically built with MayI, although you can turn off or alter the default MayI or write a new one. Note that if you do not enable the security features when you first start your system (”Set security,” pg. 19) the default MayI will be turned off.

If an object with MayI is created but has no special access-control information in its implicit parameters, it will be “fail safe”: It will only accept calls from itself or its class. Of course, the object's first action when it begins running might be to modify its access control policy. Normally, however, the creator of the object will pass an access control policy in the initial implicit parameters.

The UVa MayI is very flexible and supports many access policies. The MayI for an object can maintain separate access control information for each method as well as a catch-all that applies to any method not otherwise listed. The access control information records who is granted the right to call a method, who is explicitly denied this right, and who may generate a certificate for the method.

The “deny” list exists because groups may be specified in the access control lists. Adding the name of a context to the “allow” list, for example, permits every object whose LOID is stored in that context to call that method. However, if the “deny” list for that method contains the caller's name or a context containing his name, the call will be denied. Because resolving groups is somewhat time consuming, MayI caches the results of its lookups. This means a user added to a group will not be able to access an object until the object's cache entry for that group expires. The default expiration time is five minutes.

Certificates are another means of granting and obtaining access to an object. A certificate consists of a list of methods, an optional timeout, and an optional class LOID that restricts the certificate to instances of a particular class. It is cryptographically signed by a particular object. Any object may be given a certificate.

If an object presents a certificate to another object’s MayI (i.e., includes the certificate in the implicit parameters of the method call), MayI will check that the certificate is properly signed, that the method being called is named in the certificate, and that the maker of the certificate has the right to grant access to that method. This last information is checked in the per-method lists maintained by MayI. MayI also checks the timeout and class information in the certificate.

Certificates form the base for the concept of users in Legion. A Legion user is represented by an *AuthenticationObject*, which supports a “log in” method. The `legion_login` utility can be used to obtain a password from an unknown person and then send it to a specified AuthenticationObject for verification. If the password matches, the AuthenticationObject creates a certificate and sends it back to `legion_login`, where it is placed in the implicit parameters. All utilities run during the session will inherit these implicit parameters and thus the certificate, and they can then be used to do work on the AuthenticationObject’s (and therefore the user’s) behalf.

7.3 Special implications of security

Because Legion is a distributed system, some familiar concepts of traditional monolithic system security are different. For example, there is no central password file: each user’s password is stored in the corresponding AuthenticationObject. Furthermore, any user can create an AuthenticationObject and create objects that no one else can call. There is still some control, though. For example, the system administrator may be the owner of an object that provides a resource such as printing. If that object looks in a particular group to determine access, and the system administrator has not added a user to that group, access will be denied.

We should note that release 1.8 of the system has not been hardened to withstand attack. For example, by sending an appropriately mangled message, a sender can crash an object because the low-level message processing layers will not understand the headers. These changes are currently in progress.

7.4 Legion and Kerberos

Legion 1.8 includes the necessary support to operate in environments that require Kerberos authentication. The guiding design principle of this Kerberos support is that no process should be created by Legion without Legion first presenting to the underlying operating system valid Kerberos credentials for the target user.

There are two fundamental components of Legion’s Kerberos support. First, each Legion user creates a Kerberos proxy object and uploads her valid Kerberos credentials into her proxy object. Second, the target machine’s host object explicitly contacts the target user's Kerberos proxy object whenever a process must be created on that machine for the target user; the host object then uses the Kerberos credentials stored in the user’s Kerberos proxy object when performing a Kerberos `ksu` command in order to create the process. Of course, Kerberos credentials are encrypted for transmission from the Kerberos proxy object to the target host object (i.e., they do not travel across the network in the clear).

Legion’s Kerberos support is not currently fully documented; a small collection of sites that use Kerberos are working with the Legion developers to define and improve Kerberos support in Legion. In the near future, we will include detailed information regarding the creation and use of the Kerberos proxy objects from both the user perspective and the system administration perspective. If you currently require Kerberos support in Legion, e-mail us at <legion-help@virginia.edu>.

7.5 Session file

As of version 1.8, your credentials, current and root context LOIDs, and other relevant session information are stored in a Session file. On Unix, this file is located in /tmp and is named legioncc_p<user number>.<shell pid>. The file will be deleted when the user logs out with `legion_logout`. If the user exits without logging out, the session file will remain.

8.0 Using security features

You are not required to use any of Legion's security options. We realize that not all systems will benefit from our security and Legion can run with or without security. However, you must decide whether or not to enable Legion security before you use your new system: the command-line tool that starts the security mode (`legion_init_security`) will not run correctly if you have started to work in your system (i.e., if you have created new objects, changed context space, run classes, etc.).

If you are running a multi-architecture system, you will need to register other implementations for each additional architecture (an implementation for your current architecture is automatically created when the system is first initialized). Use the `legion_create_implementation` command.

```
$ legion_create_implementation \
$LEGION/bin/$LEGION_ARCH/AuthenticationObject \
$LEGION_ARCH /class/AuthenticationObjectClass
```

If you choose to enable the security features (see "Set security," pg. 19) you must run `legion_init_security` immediately after you have started a new system and you must log in as `admin`. If you do not enable security, Legion will run normally but none of your processes will be protected.

8.1 Authentication objects

When you create a user account in a running Legion net, Legion creates an AuthenticationObject, which holds the user's credentials and represents an individual user id in Legion. AuthenticationObjects are displayed in context space as `/users/<user id>`.

As of version 1.8, AuthenticationObjects also hold the user's home context (`/home/<user id>`) in the "home_dir" attribute.[7] They are also allowed to have any number of attributes with the name "legionrc_file." These attributes are assumed to point to LegionRC files in context space. LegionRC files are scripts that contain instructions for when the user logs in. The files are downloaded and executed on your local machine.

A default LegionRC file is created when you create a new user (page 37). Its context path is `/home/<user id>/.legionrc` and its contents are:

7 Use `legion_list_attributes` to view an object's attributes and `legion_update_attributes` to change them (see pages 10 and 12 in the Reference Manual).

```
$ legion_cat /home/<user id>/.legionrc
legion_cd ${home_dir}
$
```

This means that when the user logs in, she will automatically be moved to her home context. Users can edit their own AuthenticationObject and LegionRC files at any time, however, so they can easily remove or change this behavior.[8]

A LegionRC file MUST use the following syntax:

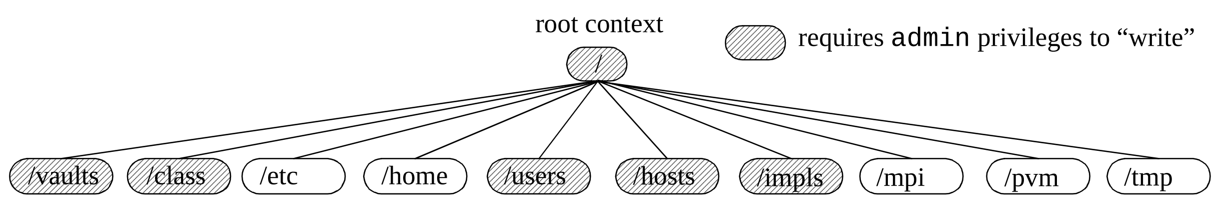
- Anything following a # on a line is a comment
- Any value included in $() is assumed to be a local environment variable. For example, $(LEGION) refers to $LEGION in the current shell.
- Any value included in ${} is assumed to be a remote environment variable. For example, ${home_dir} refers to the "home-dir" attribute in the user's AuthenticationObject.
- All remote variables come from your Authentication object's attributes
- All variables are replaced with their values BEFORE the script is run

Aside from the semantic elements described, the file MUST contain nothing else except programs to execute on the command line, one per line in the script.

8.2 Security and context space

While all users can "read" (i.e., look at and move to) all of the new context space, non-`admin` users can "write" (i.e., create new context objects) only in the `/home`, `/etc`, `/tmp`, `/mpi`, and `/pvm` contexts. Only `admin` can "write" in the all parts of context space (Figure 6). Log out is achieved by exiting the shell.

Figure 6: Context space access in a secure Legion system



8 In this case, she could use `legion_update_attributes` to change `/users/jill`'s "home_dir" attribute to a different context path. For example:

```
$ legion_update_attributes -c /users/jill -r "home_dir('/home/spw4s')" \
  "home_dir('/home')"
legion_update_attributes: Replaced 1 attributes(s) in object
$
```

The next time Jill logs in, she will automatically be moved to the `/home` context. She can edit her `/home/jill/.legionrc` file to remove to `legion_cd` command altogether, if she prefers, in which case she will start at `/`.

The `legion_change_permissions` command can be used to alter "read," "write," and "execute" object permissions so that other users can use those objects (see page 11 in the Basic User Manual).

8.3 Creating new users

You add users to your system by creating new *user ids*. A user id is an entry in context space that represents an AuthenticationObject (see section 8.1, page 35). The user id signifies ownership of all objects that a logged in user creates.

If you have enabled security you must be logged in as `admin` in order to create new users.

The `legion_create_user` command creates new user ids. This command is actually a simple wrapper around the `legion_create_user_object` command (see page 66 in the Reference Manual). The latter command gives more control in creating AuthenticationObjects, so that you can choose a particular host or vault, or (if you have another class that can create AuthenticationObjects) specify the new object's class. We strongly suggest that you put all new users in the `/users` context.

You will be asked to assign a password. The user can change it later on with the `legion_passwd` command (page 38).

New users are assigned a "home" context in `/home/<user id>`. They are also given a LegionRC file (page 35).

```
$ legion_create_user /users/jill
New Legion password: xxxxx
Retype password: xxxxx
1.3a8ba36a.6b000000.02000000.0000...
Creating a Home context: /home/jill
Changing ACLs on /home/jill
Setting up initial LegionRC file
legion_update_attributes: Added 1 attributes(s) to object
legion_update_attributes: Added 1 attributes(s) to object
$
```

If you are working on a PCD host, follow up with these steps:

```
$ for i in 'legion_ls /hosts'
$ do
$ legion_add_host_account /hosts/$i <unix_id> \
  /users/<new_user_name>
$ done
```

8.4 Logging in users

Please allow about five minutes for a new user id to propagate in your system after creating it. If a new user tries to log in too early, he or she will get security errors when trying to create objects.

The new user can then log in with `legion_login`, using the new user id's context path (note that you need to include `/users`):

```
$ legion_login /users/jill
Password:xxxxx
$
```

On a successful login, a credentials file (a user read-only file) is created in the local /tmp directory (see page 68 in the Reference manual). The user will automatically be moved to his or her home context (unless the default LegionRC file has been edited).

8.5 Changing user passwords

Use `legion_passwd` to change passwords. You must be logged in as either `admin` or the user to change a user password. Note that you need to use `/users/<user id>` as the argument.

```
$ legion_passwd /users/jill
New Legion password: xxxx
Retype new password: xxxx
Password changed.
$
```

If you are logged in as `admin` you can change all passwords. Otherwise, you can only change your own password.

8.6 Changing implicit parameters and ACL information

There is a suite of four commands for manipulating implicit parameters and access control information:

`legion_set_implicit_params`
`legion_get_implicit_params`
`legion_set_acl`
`legion_get_acl`

To set the implicit parameters for a user id, run `legion_set_implicit_params` and specify the user and the file containing the parameters.

```
$ legion_set_implicit_params /home/<user_name> <file_name>
```

Below is an example implicit parameters file that sets message security and an access control policy.

```
## Set up message level protection. Legal values are
## “Protected”, “Private”, and “Off”.
String MessageSecurity = “Protected”

## Set up an access control set and store it in the
## specified implicit parameter.
```

Notes:

- An object instance gets both the access control lists for its specific class and the default lists (if defined). The specific class lists override the default ones on a per-method basis. For example, if the method `read()` is in each, only the access control information in the specific class definition for `read()` will be kept and used.
- Every object instance and its class are automatically allowed to call all of the object's methods and to grant certificates for them. That is not affected by the access control set in the implicit parameters. These permissions can be modified by using the explicit SetACL method for the object.
- On log in, an authentication object returns the implicit parameters. It adds on to the access control set in those parameters the following right: for all methods in all classes, the authentication object is a valid granter of certificates. Subsequent objects are created with this right, so a user holding a certificate from the authentication object (after `legion_login`) can manipulate all objects that he creates.
- The access control for the default method is only applied if there is no control information for the specific method being called.
- Access is denied by default. So `deny unknown` (though valid) is unnecessary. The main purpose of deny is to deny access to specific individuals or subgroups who would otherwise have access because of their membership in a larger group.
- Names that only include letters and digits, plus underscore, dot, slash, and comma, do not need to be in quotes. Others must be in quotation marks. The reserved name `unknown` can be made unreserved by surrounding it in double quotation marks.
- The default cases can be in any order with the others. Anything between braces can be empty.

```
AccessControlSet {
   instanceOf /class/BasicFileClass {
      Method “read()” {
       allow bob fred
       ## Only bob and fred can call this method
       ## (the class and instance can call it and
       ## grant certificates for it, too, of
       ## course–see note above).
      }

      Method “    LegionLOID ping();” {
         allow group
      deny fred
      }
      ## The function identifier for this method
      ## includes the spaces within the quotation marks
```

```
            Default {
              allow bob unknown
              ## All other methods are covered by this case.
              ## Suppose that bob is the one setting up this
              ## file. He probably wants to put himself in the
              ## allow list for every method he lists, as well
              ## as for the default. Of course, if he doesn't
              ## want access of a particular type (e.g., write
              ## access), he can leave himself off. For any
              ## method where he grants access to unknown, he
              ## doesn't have to list himself. However, it
              ## doesn't hurt.
            }
        }

        Default {
            Method "    LegionLOID ping();" {
              allow bob
              ## Nobody can ping objects except bob
            }
            Default {
              allow unknown
            }
        }
    }
```

Once the implicit parameters for a user have been set, you must log out and log in again for them to take affect. Alternatively, `legion_set_implicit_params` can be used to change the implicit parameters of the current session (if you do not specify a file name the command sets the implicit parameters of the current environment). If you do this, make sure that the implicit parameters contains a certificate definition for your AuthenticationObject, or you will have to log out and log in again in order to execute any further commands as an authenticated user. This documentation does not yet include an example of defining a certificate.

To show the current implicit parameters or the parameters for a particular user, use `legion_get_implicit_params`. Use `legion_set_acl` to change the access policy of an existing object: this is not the same as changing implicit parameters, which (in the case of access control) will only affect the creation of new objects. The `legion_set_acl` tool can have the same input file as `legion_set_implicit_params` but it only uses the access control information.

The `legion_change_permissions` command manipulates an object's ACL so that other users can call methods on that object.

If you have created a system with secure files, try creating a file as a logged-in user (the executable `testBasicFiles` will create a sample one for you). You can experiment with permissions and access control.

Run `legion_get_interface` to get the names of methods available on an object. The method names follow the output line titled `Object Interface`. Some of the methods are Legion object-mandatory functions, while others (usually listed at the end) are particular to objects of that class. You can cut and paste lines from the output into an implicit parameters file: put the names in double quotation marks in the `Method` definitions. Be sure to make your cut be from the real start and end of the line output by `legion_get_interface`, since some method names have leading or trailing tabs and spaces. Directing the output of `legion_get_interface` into a file then editing the file may be helpful.

If a user's AuthenticationObject is deleted there is no way to regenerate an equivalent AuthenticationObject; the user must be re-created from scratch. The reason is that the private key of the original AuthenticationObject cannot be recovered, so the same LOID cannot be used for the object.

Legion system management

9.0 Legion core objects

The Legion core object model specifies the composition and functionality of Legion's core objects. These objects create, locate, manage, and remove objects in the Legion system. Legion provides implementations of core objects but you are not obligated to use them.

Although the object model includes and relies on a few single logical Legion objects, access to these objects is limited because of heavy caching and hierarchical organization of lower level objects. Objects can be replicated to reduce any contention. Increasing the number of Legion computing resources will not increase competition for the few "centralized" Legion objects.

In this object model, each Legion object belongs to a class and each class is itself a Legion object. All Legion objects export a common set of object-mandatory member functions, such as `save_state()` and `restore_state()`. Class objects export an additional set of class-mandatory member functions, such as `create()`, `derive()`, and `inherit_from()`. The object model's power comes from the Legion classes. Much of what is usually considered system-level responsibility is delegated to user-level class objects. Legion classes are responsible for creating and locating their instances and subclasses, and for selecting appropriate security and object placement policies. Core Legion objects provide mechanisms for user-level classes to implement policies and algorithms that they choose. Assuming that we define the operations on core objects appropriately (i.e., that they are the right set of primitive operations to enable a wide enough range of policies to be implemented), this philosophy effectively eliminates the danger of imposing inappropriate policy decisions and opens up a much wider range of possibilities for the applications developer.

9.1 Core objects classes

There are six core objects

- LegionClass
- LegionBindingAgent
- LegionHost
- LegionVault
- ContextObject
- ImplementationObject

From these, the core class types – hosts, vaults, and binding agents – are derived. The core classes set the minimal interface that the core objects export. Every core object is an instance of a class that is itself eventually derived from one of the core object classes.

- *LegionClass*: The LegionClass object is the common unit of the Legion system. The core LegionClass provides the fundamental characteristics and object-mandatory functions of all Legion objects. There is only one LegionClass object in a system.
- *LegionBindingAgent*: Binding agents are Legion objects that map an object's LOID to its LOA (Legion Object Address). A `<LOID, LOA>` pair is called a *binding*. Binding agents cache bindings and organize themselves in hierarchies and software combining trees in order to implement the binding mechanism in a scalable and efficient manner.
- *LegionContextObject*: Context objects map context names to LOIDs, allowing users to assign arbitrary high-level string names to Legion objects. These objects also enable multiple disjoint name spaces to exist within Legion. All objects have a current context and a root context, which define parts of the name space in which context names are evaluated.
- *LegionHostObject*: Host objects represent processors. One or more host objects run on each computing resource which is included in Legion. Host objects create and manage processes for active Legion objects on their hosts. Classes invoke the member functions on host objects in order to activate instances (see page 143 in the Developer Manual). Representing computing resources with Legion objects abstracts the heterogeneity which results from different operating systems using different mechanisms for creating processes. Further, it provides resource owners with the ability to manage and control their resources as they see fit.
- *LegionVaultObject*: Just as a host object represents computing resources and maintains active Legion objects, a vault object represents persistent storage, but only for the purpose of maintaining the state, in OPRs, of the inert Legion objects that the vault object supports.
- *LegionImplementationObject*: Implementation objects allow other Legion objects to run as processes in the system. An implementation object typically contains machine code that is executed when a request to create or activate an object is made. More specifically, an implementation object is generally maintained as an executable file that a host object can execute when it receives a request to activate or create an object. An implementation object (or the name of an implementation object) is transferred from a class object to a host object to enable the host to create processes with the appropriate characteristics.

For more information on the Legion core objects, please see "Core objects," page 144 in the Developer Manual.

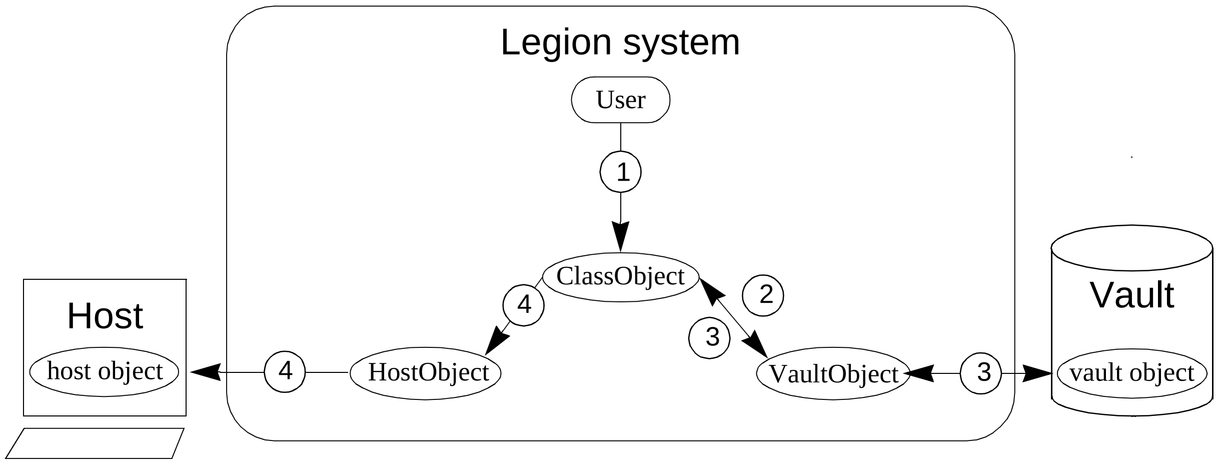
10.0 Implementation model

When a user asks a class object to create an instance on another host, the class must do the following:

- Determine what kind of architecture the new host has,
- Contact the correct vault object to request persistent storage space,
- Assign the new object a LOID and an *Object Persistent representation Address* (OPA: see "Object states," pg. 135 in the Developer Manual for further information), and
- Contact the correct host object and ask it to start the new object on the host using a particular implementation object.

Figure 7 shows the different steps in the procedure.

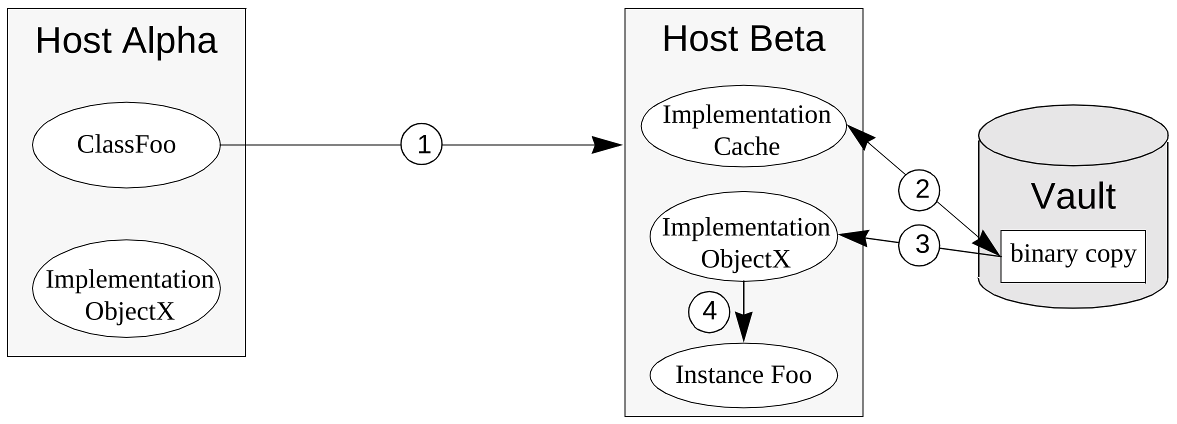
Figure 7: Legion object creation



1. User wants to create an instance on a particular host
2. Class contacts a vault object
3. Vault passes the new instance's OPR, OPA to the class object
4. Class passes a LOID and OPA to a host object and tells it to start the new instance on the requested host

When a class object asks a host object to start an instance (Figure 7 step 4), it gives the host object the LOID for an appropriate implementation object. An implementation object typically contains executable object code for a single architecture and operating system platform, as well as any other information that might necessary for instantiating an object on a particular host object (Java code, Perl script, etc.). There are different implementation objects for different architectures, and each class maintains implementation objects for all of the architectures on which it might run its instances. The host must have a copy of a appropriate implementation object in order to start the instance.

Figure 8: The implementation cache object



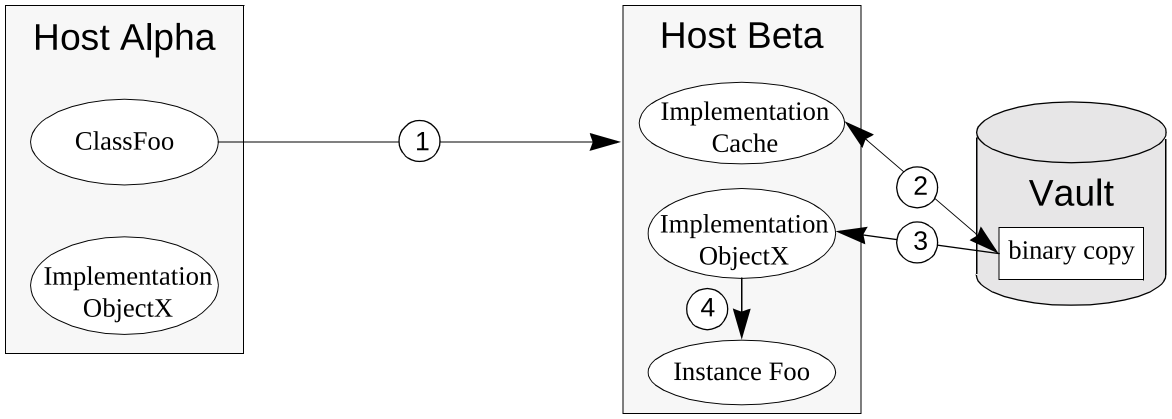
1. Class Foo tells Beta to create instance Foo using ImplementationObjectX
2. Beta asks its Implementation Cache Object to find a copy of ImplementationObjectX
3. Cache finds copy of ImplementationObjectX in the vault.
4. Beta runs its copy of ImplementationObjectX to activate instance Foo.

Figure 8 shows how the host object accomplishes this. When class `Foo`, sitting on host Alpha, sends a call (step 1) to create instance `Foo` on Beta, it gives Beta the LOID for ImplementationObjectX. Beta uses the LOID to find ImplementationObjectX on Alpha (step 2) and makes a binary copy for its own vault (step 3). Beta can then create instance `Foo` (step 4).

10.1 Implementation caches

While this procedure is a reasonable investment if a host object requires a particular implementation object once, it becomes expensive when repeated. An *implementation cache* circumvents the problem by acting as an intermediary between the host object and the class object. The implementation cache object is responsible for finding and keeping implementation objects on behalf of its host object. We can update the scenario in Figure 8, since Beta can now ask its implementation cache object to locate a copy of ImplementationObjectX, as show in Figure 9, below.

Figure 9: The implementation cache object



1. Class Foo tells Beta to create instance Foo using ImplementationObjectX
2. Beta asks its Implementation Cache Object to find a copy of ImplementationObjectX
3. Cache finds copy of ImplementationObjectX in the vault.
4. Beta runs its copy of ImplementationObjectX to activate instance Foo.

10.2 Implementation tools

When new host objects are added, the `legion_init_arch` tool will register implementation objects of that architecture for commonly used classes and objects. The tool is run on the new host, so as to create the objects in the proper place. The sample below was run on a Linux host.

```
$ legion_init_arch
Initializing Legion implementations for “linux”

Creating an implementation (ContextObject) for ContextClass
Continue (y=yes, Y=yes to all, n=no, N=no to all, v=verbose,
          V=verbose all)? Y
Creating an implementation (MetaClassObject) for LegionClass
Creating an implementation (ClassObject) for VanillaMetaClass
Creating an implementation (BindingAgent) for
BindingAgentClass
Creating an implementation (BasicFileObject) for
BasicFileClass
Creating an implementation (ttyObject) for ttyObjectClass
Creating an implementation (StatTreeObject) for StatTreeClass
$
```

You can see existing implementation objects in the `/impls` context.

```
$ legion_ls -la /impls
.                                  (context)
..                                 (context)
AuthenticationObject.linux.1       (implementation)
BasicFileObject.linux.1            (implementation)
BatchQueueClassObject.linux.1      (implementation)
BindingAgent.linux.1               (implementation)
ClassObject.linux.1                (implementation)
JobProxyObject.linux.1             (implementation)
MetaClassObject.linux.1            (implementation)
StatTreeObject.linux.1             (implementation)
StatelessProxyClassObject.linux.1 (implementation)
legion_make_backend.linux.1        (implementation)
ttyObject.linux.1                  (implementation)
$
```

The default context names for all implementation objects consist of a class name, architecture, and encoded architecture number (names ending in `*.1` are the first implementation object of that architecture for that class).

You can create Implementation objects for a specific binary executable with `legion_create_implementation`. The new implementation object is marked as usable whatever architecture you specify. The syntax is:

```
legion_create_implementation
   <binary path name> <architecture>
   {[-c] <class context name> | -l <class LOID>}
   [-c <object context path>] [-nc] [-v]
   [-a <attribute>] [-debug] [-help]
```

Please see page 23 in the Reference Manual for a list of possible `<architecture>` values and explanation of the flags.

The new implementation object will be associated with the class object named in `<class LOID>` or `<class context path>`. You must provide a path for the binary executable that will run on your specified architecture. The new object will be assigned the context path `/impls/<class_name>.<architecture>.#` unless you specify otherwise in the `<object context path>` parameter or use the `-nc` flag.

The example below creates a Linux implementation object for `my_class`. The new object will automatically be assigned the context path `/impls/my_class.linux.1`.

```
$ legion_create_implementation Legion/bin/linux/my_class \
 linux my_class
```

If you ran the example a second time, the second implementation object would be called `/impls/my_class.linux.2`.

Use `legion_list_implementations` to see a list of which implementation objects have been assigned to a particular class. The output will be each object's LOID and architecture. The example below lists seven implementation objects for the tty class.

```
$ legion_list_implementations -c /class/ttyObjectClass

alpha_linux  1.3933cb3f.08.42000000.000001fc0bc...
solaris 1.3933cb3f.08.51000000.000001fc0bc...
sgi   1.3933cb3f.08.92000000.000001fc0bc...
rs6000  1.3933cb3f.08.07010000.000001fc0bc...
x86_freebsd   1.3933cb3f.08.20010000.000001fc0bc...
linux  1.3933cb3f.08.49010000.000001fc0bc...
hppa_hpux    1.3933cb3f.08.be010000.000001fc0bc...
```

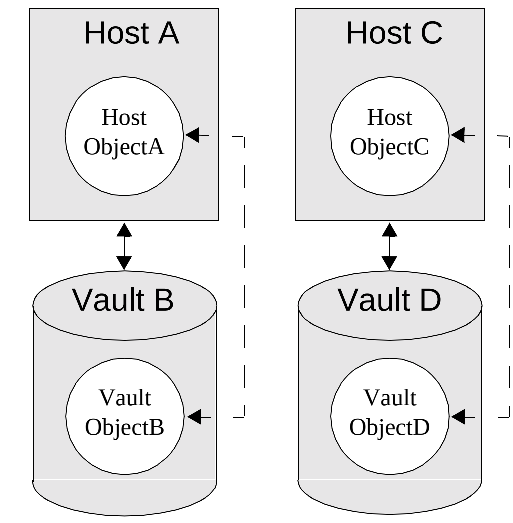
11.0 Host and vault objects

Please see section 6.0 (page 30) of the Basic User Manual for information about Legion host and vault objects and an introduction to some basic host- and vault-related commands.

11.1 About host-vault pairs

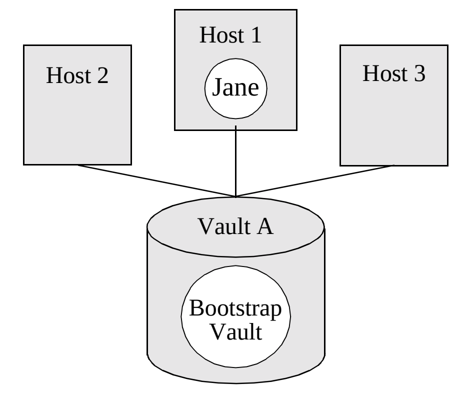
Adding new hosts and vaults to your system makes multiple processors and storage space available to your system, but before you start expanding be aware that Legion hosts and vaults must work in compatible pairs. Figure 10, right, shows two pairs of compatible host-vaults: Host A and Vault B can "see" each other and Host C and Vault D can "see" each other.

Figure 10: Compatible host-vault pair



All Legion host objects must be paired with at least one compatible vault object in order to carry out Legion processes: all Legion objects maintain an OPR on a vault and objects must have access to their inert state in order to function properly. Therefore, before you add a new host object or vault to your system you must consider any possible compatibility problems. An incompatible host object and vault object will not work together. `HostObjectA` in Figure 10 is compatible with `VaultObjectB` but not with `VaultObjectD`, while `VaultObjectB` is not compatible with `HostObjectC`.

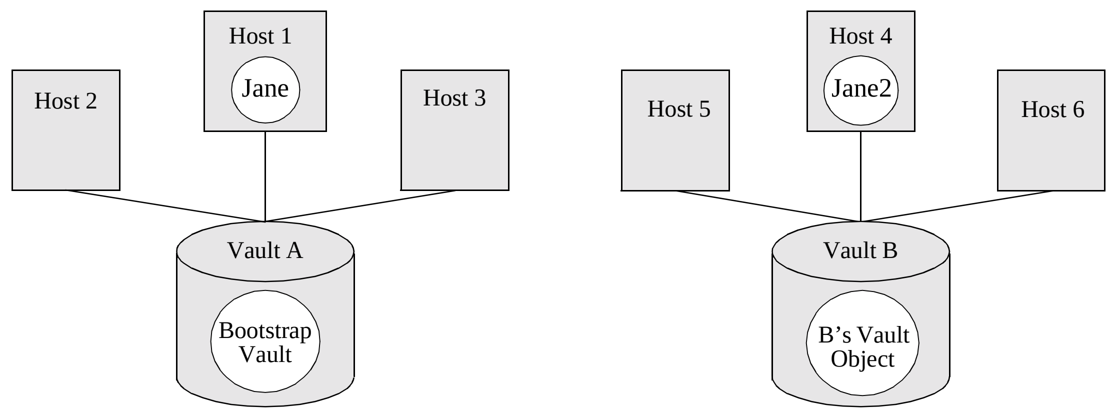
Figure 11: Common persistent storage system



This is not a concern in systems that use a single shared vault (e.g., a networked file system, database system, tape drive, CD-ROM, etc.), as in Figure 11, left. Here, all vault objects will represent space in the only available disk storage space (Vault A). They will therefore all be accessible to any host object created on Hosts 1, 2, or 3. If Jane, working on Host 1, wishes to create a new host object on Host 2 she can either pair the new host object with the currently existing `BootstrapVault` or create a new vault object on Vault A. Either way, there is no need to worry about incompatible pairing.

On the other hand, if Jane wants to add a new host from a foreign system (i.e., her bootstrap host object cannot "see" the new system's persistent storage space) she must create a new vault object in the foreign system and pair it with her new host object. Figure 12 (below) shows an example of this situation, with two different file systems and multiple hosts.

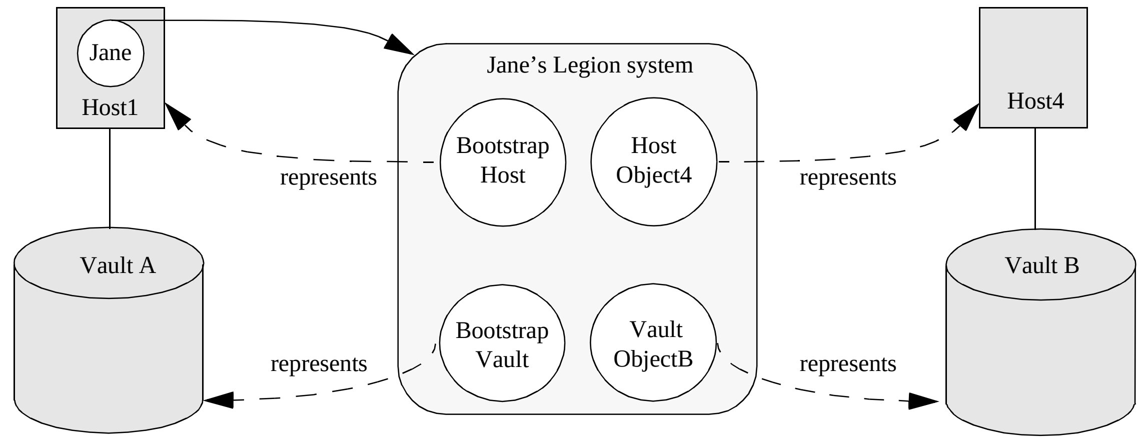
Figure 12: Multi-host and multi-vault system



Hosts 1-3 can see Vault A, and Hosts 4-6 can see Vault B. If Jane wants to create a host object on Hosts 2 or 3 she can pair it with `BootstrapVault` or create another vault object on Vault A. Either way, she does not need to worry about host-vault compatibility. However, if she wants to create a host object on Host 4 she must pair it with a compatible vault object on Vault B, either by creating a new vault object or by getting permission to use a currently existing vault object.

Figure 13 (below) shows how this might work. Jane creates `HostObject4` on Host 4 and `VaultObjectB` on Vault B. `HostObject4` will manage her Legion work on Host 4, and `VaultObjectB` will manage the persistent storage of `HostObject4`'s object.

Figure 13: Adding new resources to a Legion system



Assuming that there are no conflicts in architecture, environment, etc., you can add a new host to your system with the `legion_starthost` command and a new host object will be created on the new host using the current environment values of $LEGION and $LEGION_OPR. There are a variety of options in case the new host has a different architecture or different Legion environment variables or if you need to specify a different user id. This process will be discussed in section 11.3.

11.2 Manipulating host-vault pairing

The `legion_host_vault_list` command manipulates a given host object's list of compatible vaults. Its usage is:

```
legion_host_vault_list
  {[-c] <host context path> | -l <host LOID>}
  [{-a | -d | -t} <vault1> <vault2> ... <vaultn>]
  [-p] [-debug] [-help]
```

The example below lists the compatible vaults for `BootstrapHost`. Note the use of `-p`: this signals that the list should be printed to standard output.

```
$ legion_host_vault_list hosts/BootstrapHost -p
** COMPATIBLE VAULT LISTING:
**    1.01.03.3cb53908.000001fc0bb4fef12ecf6cc...
**    1.01.03.3db53908.000001fc0dd5621fadf70b0...
**    1.01.03.3eb53908.000001fc0d6e9041e262126...
$
```

There are three vaults listed here: use `legion_list_names` to see their context names (see page 31 in the Reference Manual).

The `legion_vault_host_list` command manipulates a vault's list of compatible host objects.

```
legion_vault_host_list
   {[-c] <vault context path> | -l <vault LOID>}
   [{-a | -d | -t} <host1> <host2> ... <hostn>]
   [-p] [-debug] [-help]
```

The example below shows `BootstrapVault`'s compatible hosts.

```
$ legion_vault_host_list vaults/BootstrapVault -p
** COMPATIBLE HOST LISTING:
**   1.01.07.3cb53908.000001fc0c29636eee98d...
**   1.01.07.3eb53908.000001fc0d9b155044fb5...
$
```

Both of these commands can also add and delete compatible hosts or vaults with `-a` and `-d`. For example, to remove `aNewVault` from `BootstrapHost`'s list of acceptable vaults and then see the adjusted list you would enter the following:

```
$ legion_host_vault_list hosts/BootstrapHost \
  -d vaults/aNewVault -p
Deleted 1 vault(s) to host's compatibility set
** COMPATIBLE VAULT LISTING:
**   1.01.03.3cb53908.000001fc0bb4fef12ecf6cc...
**   1.01.03.3db53908.000001fc0dd5621fadf70b0...
$
```

To add a host and then see the adjust list, you would enter the following:

```
$ legion_vault_host_list vaults/BootstrapVault \
  -a hosts/AHost -p
Added 1 host(s) to vault's compatibility set
** COMPATIBLE HOST LISTING:
**   1.01.07.3cb53908.000001fc0c29636eee98d...
**   1.01.07.3eb53908.000001fc0d9b155044fb5...
**   1.01.07.3fb53908.000001fc0c96beaba5730...
$
```

11.3 Adding a new host

The main system must be active in order to add a new host. The new host machine must also have the Legion binaries installed (or visible via NFS).

The `legion_starthost` command is run from your current machine, not on the new host. This command uses remote shell (`rsh` or `ssh`) classes to start a new host object on a specified host. You can start new host objects on your current host as well as on other hosts, since a single machine can contain more than one host object. Please note that you *must* be able to run `rsh`/`ssh` on the target host from your current machine without having to enter a password. You can set up an .rhosts file for `rsh` or an authorized_keys file for `ssh` to accomplish this: see the `rsh` and `ssh` man pages for further information.

You can run Legion commands on a remote host using `rsh` or `ssh`, once you set the proper environment variables. For sh, ksh, or bash, use:

```
LEGION_RSH=<rsh|ssh>
LEGION_RCP=<rcp|scp>
export LEGION_RSH LEGION_RCP
```

For csh, use:

```
setenv LEGION_RSH <rsh|ssh>
setenv LEGION_RCP <rcp|scp>
```

Normal usage is below (please see page 42 in the Reference Manual for flags and default settings).

```
legion_starthost
   [<flags>] {<new host name>}
   [<compatible vault list>]
```

You should specify a compatible vault whenever you create a new host object: you can run `legion_starthost` without a vault name and then use `legion_host_vault_list` to add a vault to the new host object's list of compatible vaults, but it is simpler to specify one or more compatible vault when you first create the new host object. In the example below, the default `BootstrapVault` is the new host object's compatible vault.

```
$ legion_starthost new.host.DNS.name /vaults/BootstrapVault
Creating a Legion host object with the following attributes:
  Host             = “new.host.DNS.name”
  Context name     = “/hosts/new.host.DNS.name”
  $LEGION          = “/home/xx/Legion”
  $LEGION_OPR      = “/home/xx/OPR”
  $LEGION_OPA      = “/home/xx/OPR/new.host.DNS.name.OPA”
  Architecture     = “linux”
  User id          = “xx”
  Binary path      = “/home/xx/Legion/bin/linux/UnixHostObject”
  Compatible vaults= “vaults/BootstrapVault”
Transferring configuration files to
   “xx@new.host.DNS.name:/home/xx/OPR”
Creating an instance of “/class/UnixHostClass”
1.01.07.44b53908...
Adding “/hosts/new.host.DNS.name” to the host list for
  “vaults/BootstrapVault”
Added 1 host(s) to vault's compatibility set
Adding “vaults/BootstrapVault” to the vault list for
  “/hosts/new.host.DNS.name”
```

```
Added 1 vault(s) to host's compatibility set
Configuring well-known binaries for “/hosts/new.host.DNS.name”
$
```

There is a substantial amount of information returned. Legion first prints out the attributes of the newly created host object, which include its name, context name, local OPR and OPA path names, architecture, your Unix user id, local path name, and any compatible vault(s). It also shows the binary executable files for basic Legion objects (e.g., an implementation object) being added and configured to the new host. These files allow the new host to start new Legion objects as necessary. The output then shows the creation of the object: the new object is an instance of the UnixHostClass. Optional flags will let you change some of these attributes.

The output also lists the new host-vault pairs that were formed: the new host object is now on `BootstrapVault`'s list of compatible hosts and `BootstrapVault` is on the new host object's list of compatible vaults.

Note that the new host object is automatically assigned a context path, in this case `/hosts/new.host.DNS.name`. Use the `-N` flag to place the new host object in a different context or would assign it a specific context name.

```
        $ legion_starthost -N /hosts/aNewHost new.host.DNS.name \
          /vaults/BootstrapVault
```

This assigns the new host object the context name `aNewHost` (the example specifies that the context name be put in the `/hosts` context path: otherwise it will be put in the current context). If the `-N` flag is used, the new host object will not be assigned the default DNS name.[9]

11.3.1 `legion_starthost` flags

Supported `<flags>` for `legion_starthost` are:

`-L` `<$LEGION>`Specify $LEGION for the new host
(default is local $LEGION value)

`-O` `<$LEGION_OPR>`Specify $LEGION_OPR for host
(default is local $LEGION_OPR value)

`-A` `<$LEGION_ARCH>`Specify the host's architecture type
(default is local $LEGION_ARCH value)

`-B` `<path>`Specify the host binary's basename
(default is UnixHostObject)

`-N` `<context name>`Specify the host object's context name
(default is `/hosts/<new host name>`)

9 An alternative procedure, using command-line utilities, is explained in the Reference Manual on page 119. If possible, we recommend using the `rsh` procedure explained here, since it is faster and easier.

`-U` `<user id>`Specify a Unix user id for the host
(default is current Unix user id)

`-C` `<host class>`Specify the host class's context name
(default is `/class/UnixHostClass`)

This command creates a new host object on the specified host object. It selects the following additional default values for the new object:

<$LEGION_OPA> = $LEGION_OPR/Host-$HOST.OPA
<binary path> = $LEGION/bin/$LEGION_ARCH/UnixHostObject

The OPA (Object Persistent representation Address) is used to track a specific OPR (Object Persistent Representation, the inert state of an object): in this case, the OPA will represent the persistent state of the new host object. The binary path is the remote path of the program that will start the new host object on the new (remote) host. I.e., it is a path on the remote host.

These flags and defaults need to be carefully considered when adding new hosts and vaults to your system. If the new host has a different architecture or a different directory structure, use `-A`, `-L`, or `-O` to specify these parameters.

The `-B` flag allows you to specifies the basename of the executable host program that will be started on the target host: this file should be located in the target host's $LEGION/bin/$LEGION_ARCH directory. Note that a single class can manage instances with different implementations as long as all of the instances support the same interfaces (e.g., there are two implementations for `/class/UnixHostClass`, `UnixHostObject` and `PCDUnixHost`).

The `-U` flag allows you to specify a Unix user id for the new object, so that a system administrator can add host objects to another user on the same Unix system. This can be useful should you wish to create a guest user id that has limited access privileges to the new host or if you need to work under a different user id on the host.

The `-C` flag allows you to start an instance of a different class, so that users can create new host classes and have more flexibility in managing their resources. Legion currently comes with only one host class, `UnixHostClass`, but users can add more host classes as necessary, either by creating instances of the UnixHostClass or by writing new classes.

If the new host object's architecture is not the same as the current host object's architecture, you should run the `legion_init_arch` tool in order to create implementation objects to match the new architecture (see section 10.0 for information about implementations and section 10.2 for information about running this command).

11.3.2 The host object's log

Legion maintains a log containing information about all processes that are executed on your host objects in the $LEGION_OPR directory. Each host object has a separate log, called $LEGION_OPR/<host object name>.log. The log includes information about the process's LOID, owner (if applicable), binary executable, OPR, status, and start/stop times.

11.4 Adding a new vault

Starting a new vault is similar to starting a new host object. The `legion_startvault` command usage is: (see section 11.4.1, below, for `legion_startvault`'s flags and default settings)

```
legion_startvault [<flags>] {<host name>}
  [<compatible host list>]
```

The example below creates a vault object on the host we created above (`aNewHost`) and uses `-N` to assign the new vault the context path `/vaults/aNewVault`.

```
$ legion_startvault -N /vaults/aNewVault new.host.DNS.name \
  /hosts/BootstrapHost /hosts/aNewHost
Creating a Legion vault with the following attributes:
  Host            = "new.host.DNS.name"
  Context name    = "/vaults/aNewVault"
  $LEGION         = "/home/xx/Legion"
  $LEGION_OPR     = "/home/xx/OPR"
   $LEGION_OPA    = "/home/xx/OPR/vault-aNewVault.OPA"
  Architecture    = "linux"
  User id         = "xx"
   Binary path    = "/home/xx/Legion/bin/linux/UnixVaultObject"
  Compatible hosts= "/hosts/BootstrapHost /hosts/aNewHost"
Transferring configuration files to
   "xx@new.host.DNS.name:/home/xx/OPR"
Creating an instance of "/class/UnixVaultClass"
  1.36188412.03.04...
Adding "/vaults/aNewVault" to the vault list for
   "/hosts/BootstrapHost"
Added 1 vault(s) to host's compatibility set
Adding "/hosts/BootstrapHost" to the host list for
  "/vaults/aNewVault"
Added 1 host(s) to vault's compatibility set
Adding "/vaults/aNewVault" to the vault list for
"/hosts/aNewHost"
Added 1 vault(s) to host's compatibility set
Adding "/hosts/aNewHost" to the host list for
"/vaults/aNewVault"
Added 1 host(s) to vault's compatibility set
$
```

The compatible hosts are `aNewHost` and `BootstrapHost`. The `<host name>` parameter uses the host's DNS name but the compatible host list uses the host objects' context path (i.e., `/hosts/aNewHost`). ***If you do not specify any compatible host objects, the new vault's list of compatible hosts will be empty and the new vault will be unusable.***

The output is similar to the `legion_starthost` output, and includes the new vault's attributes and LOID. Two compatible host objects were added to the `aNewVault`'s list of compatible hosts and the new vault was added to `BootstrapHost` and `aNewHost`'s lists of compatible vaults.

You can use `legion_vault_host_list` to add and remove hosts from a vault's list of compatible hosts (see page 55 in the Reference Manual), and you can add hosts to this list after creating the vault, but if possible it is simpler to specify at least one compatible host when running `legion_startvault`. To add more than one host to the vault object's compatibility list, just add the names of the host objects.

11.4.1 `legion_startvault` flags

Supported `<flags>` for `legion_startvault` are:

`-L` `<$LEGION>`	Specify $LEGION for the vault's host (default is the local $LEGION value)
`-O` `<$LEGION_OPR>`	Specify $LEGION_OPR for the vault's host (default is the local $LEGION_OPR value)
`-A` `<$LEGION_ARCH>`	Specify the architecture of the vault's host (default is the local $LEGION_ARCH value)
`-N` `<context name>`	Specify the vault object's context name (default is `/vaults/vault-<host name>`)
`-U` `<user id>`	Specify a Unix user id (default is current Unix user id)
`-C` `<vault class>`	Specify the vault object's context path (default is `/class/UnixVaultClass`)

This commands creates a new vault object in the storage system of a specified host (named in `<host name>`). The flags are similar to the `legion_starthost` flags. The `-L`, `-O`, or `-A` flags can be used to specify a different architecture or a different directory structure. The `-N` flag allows you to specify a context name. The `-U` flag allows you to specify a Unix user id for the new object. The `-C` flag allows you to start an instance of a different class. This flag allows users to create new vault classes, so as to give users more flexibility in managing their resources.

11.5 Backup vaults

In the current releases, instances of the BasicFileClass, ContextClass, UserAuthenticationObject, and ImplementationClass classes can have their state replicated on backup vaults. If an instance's primary vault is dead or unavailable during the instance's reactivation, a copy of the instance's state can be retrieved from one of its backup vaults. Please see section 6.6 on page 32 in the Basic User Manual for more information.

12.0 Process control daemon host objects

In a normal host object all objects run under the same Unix user id, making it difficult to isolate and account for different objects: in other words, if an outside user runs processes on your host, his processes will run under the same Unix user id as your processes. To solve this problem, Legion lets you create a second type of Unix host object, a process control daemon (PCD) host object. A PCD host object uses the services of a daemon, which executes as root in order to provide the host object with controlled access to a limited set of privileged operations. That is, the daemon oversees the host object's processes, regulating ownership of each process. This daemon must be started by someone with root privileges on the host (such as a system administrator). Typically, the PCD is configured to start through inetd.

Legion users who have Unix accounts on the host are tracked by their Unix user ids and guest users can be assigned a temporary Unix guest account user id. The PCD host object assigns guest user status to outside users and tracks each process's owner. This prevents malicious users from interfering with other users' processes.

12.1 Adding a PCD host object

12.1.1 Configure the daemon

Before you start up a PCD host object, you must start the process control daemon if it is not already running. You can install a daemon with inetd (explained below). These steps only need to be run once.

The daemon is able to carry out the following operations:

- Spawn a given process, with a given environment, with a given user id. This user id must be listed in a file of authorized user ids called PCD_readUserFile.
- Kill a process. The process must be owned by a user listed in the authorized user id file PCD_readUserFile. The implementation of this operation currently depends on the `/proc` file system.
- Kill all of a given user's processes. The user must be listed in the authorized user id file PCD_readUserFile.
- Recursively change directory ownership to a given user. The user id must be listed in the authorized user id file PCD_readUserFile.

If you wish to use a PCD host object as your net's BootstrapHost, the Legion administrator must set `LEGION_HOST_BIN=PCDUnixHost` in his/her environment **before running** `legion_initialize`.

To install the Legion process control daemon on a host, perform the following steps while logged in as root:

1. Add the following line to your Unix /etc/services file:

   ```
   legion_host   4000/tcp   # Legion procControlD
   ```

2. You need to set values for `procControl-d`. This daemon lives on the host: when a Legion user wants to start a process on the host, the PCD host maps the user's Legion user to a Unix user and passes the information to `procControl-d`, which then creates the requested process under the user's Unix account. It takes the following arguments:

`-m` `<user file>`	Names a local file containing a list of Unix user accounts that `procControl-d` can spawn processes under.
`-c` `<client file>`	Names a local file containing a list of Unix users who have permission to access `procControl-d`. We recommend that this file contain only the Unix account under which the PCD host is running.
`-s` `<spawn dir>`	Names a local directory under which the PCD can spawn processes. We recommend that this directory be the same as the host/vault pair's $LEGION_OPR directory.
`-l` `<core dir>`	Name the local directory under which the PCD will spawn core Legion objects. We recommend that this be the same as the $LEGION/bin directory.

   We recommend that you set all four of these arguments. Add a line to your Unix /etc/inetd.conf file, replacing the /home/legion-admin/OPR argument with the location of your OPR directory and /home/legion-admin/Legion/bin argument with the location of your Legion bin directory.

```
legion_host stream tcp nowait root /etc/procControl-d
procControl-d -m /etc/LegionUsers -c /etc/LegionClients -s
/home/legion-admin/OPR -l /home/legion-admin/Legion/bin
```

3. Create the Unix file /etc/LegionUsers. List the user-ids of managed accounts in this files (the user-ids that the daemon will be able to spawn processes as), one user-id per line. This file must be owned by root and have mode 0640 (grant `read` permissions to the group). Be sure to set the file's group to a group that contains the Legion system administrator. E.g.,

```
$ chgrp legion-group /etc/LegionUsers
```

   where `legion-group` is the group that the system administrator's account belongs to).

4. Create the Unix file /etc/LegionClients. List the user-ids that will be able to connect to the daemon. This should probably contain a single user-id: the account that the UnixHostObject is running on. This file must be owned by root and have mode 0640 (grant read permissions to the group). Be sure to set the file's group to a group that contains the Legion system administrator. E.g.,

```
$ chgrp legion-group /etc/LegionUsers
```

   where legion-group is the group that the system administrator's account belongs to).

5. Copy the executable program `procControl-d` into /etc/procControl-d (in your Unix directory). This executable file can be obtained from the local Legion administrator. It resides by default in $LEGION/bin/$LEGION_ARCH/procControl-d under the home directory of the Legion administrator. Make sure that /etc/procControl-d has mode 0500.

6. Restart inetd.

```
$ killall -HUP inetd
```

7. Run `pcdCheckConfig` to make sure that all is well. This binary executable checks that `procControl-d` has a valid configuration. If the configuration is incorrect the host object will not function.

   It does not need to be run by root, but it does need to be run with read permissions to the /etc/LegionUsers and /etc/LegionClients files. E.g., if these files can be read by a group that includes the Legion system administrator, `pcdCheckConfig` can be run by the system administrator.

Note that you can change /etc/LegionClients and /etc/LegionUsers after you have created them. You must tell `procControl-d` to read the files once you have edited them. You can send a SIGHUP to the daemon to force it to reread these files.

```
$ kill -HUP <procControl-d PID>
```

12.1.2 Start the daemon and the host object

You need to be logged in as `/users/admin` to start up a PCD host object. To start up a PCD host object on a PCD host run `legion_starthost` with the `-B` flag (see page 54) on the host.

```
$ legion_starthost -B PCDUnixHost PCD.host.DNS.name \
  /vaults/vault_name
```

This starts the PCD host, which in turn starts the daemon. The daemon checks its configuration to make sure that it is valid (which it should be, if you ran `pcdCheckConfig`) and establishes a connection with its host. The host object will then report to the host class and run normally.

Note that the example above assumes that you have already started a compatible vault object. We recommend that the vault reside on the PCD host.

Once you have started the PCD host object and (if necessary) the accompanying vault, you must change the following file permissions on the node that is actually running the PCD host.

$LEGION_OPR should be set to 755
$LEGION_OPR/LegionClass.config* should be set to 644
$LEGION_OPR/BootstrapVaultOPR should be set to 777
(If your bootstrap host is a PCD host)
$LEGION_OPR/<vault_name>.OPA should be set to 777
(If the bootstrap host is not a PCD host)

The $LEGION home directory is set to mode 755. These changes should be made by the Legion administrator.

A PCD host object will behave very much like a normal Unix host object, so most users do not need to know whether or not their processes are running on one or the other.

12.2 PCD host commands

There are three Legion commands for PCD host objects:

1. `legion_add_host_account`, for adding new accounts to the list of available accounts,
2. `legion_list_host_accounts`, for viewing the list of available accounts, and
3. `legion_remove_host_account`, for removing an account from the list of available accounts.

12.2.1 Adding a new account

The `legion_add_host_account` command adds a mapping between a Legion account and a Unix account and adds this mapping to a PCD host object's list of available accounts.

The user's Unix user id is named in the `<Unix user id>` parameter.

```
legion_add_host_account
  {-l <host object LOID> | -c <host object context path>}
  {[-f <mapping file name>] | [<Unix user id>
      [-l <owner LOID> | -c <owner context path>]]
  [-debug] [-help]
```

The host object is named in the `<host object LOID>` or `<host object context path>` parameter.

The user's Legion user id can be given in the `[-l <owner LOID> | -c <owner context path>]` parameter or listed in a file: this parameter designates the ownership of the `<Unix user id>`, so that when a Unix user creates Legion processes on the host object the processes will automatically run under the proper Unix user id. If the Legion id parameter is left empty, the Unix user will be treated as a guest user. This command does not create a new Legion user id: use the `legion_create_user` command to create an id if necessary.

Alternatively, you can create a local mapping file that contains a list of Unix-Legion account mappings. This file contains a list of Unix user ids (one per line) and any corresponding Legion user ids. There is no limit on the number of mappings that can be listed. If no Legion account is named, the account will be treated as a guest account. Suppose that you want to map three accounts: one guest account, and one each for your Unix users John and Lucy. The mapping file below shows how to do this.

```
guest
unixLucy  -c /users/lucy
unixJohn  -c /users/john
```

Of, you could do this from the command line. The examples below map the accounts for a PCD host object called `myPCDhost`.

```
$ legion_add_host_account /hosts/myPCDhost guest
$ legion_add_host_account /hosts/myPCDhost unixLucy \
    -c /users/lucy
$ legion_add_host_account /hosts/myPCDhost unixJohn \
  -c /users/john
```

In John's case, a mapping for `unixJohn` would be created on `myPCDhost` and `john` would be its owner. When John (logged in to his Legion account) asks to runs a process on `myPCDhost`, the PCD demon will automatically execute it on his `unixJohn` account. If, on the other hand, you did not name John as the account owner:

```
$ legion_add_host_account /hosts/myPCDhost unixJohn
```

A guest mapping for `unixJohn` will be added. If any Legion user who does not already have a mapping runs a process on `myPCDhost` he or she will run under a guest account that is not currently in use. If John runs a process on `myPCDhost` the process will execute on a guest account, not the `unixJohn` account.

12.2.2 Removing an account

The `legion_remove_host_account` command removes one or more account mappings from the host object's list of available accounts.

```
legion_remove_host_account
   {-l <host object LOID> |
      [-c] <host object context path>}
   <user id> [-debug] [-help]
```

As with `legion_add_host_account`, the `<user id>` parameter is the user's Unix user id. If no host is named in the `<host object LOID>`/`<host object context path>` parameter, your current host object is the default.

12.2.3 Viewing available accounts

The `legion_list_host_accounts` command lists the available accounts on a host object. If no host object argument is provided, your current host object will be used as a default.

```
legion_list_host_accounts
   [-l <host object LOID> |
     [-c] <host object context path>]
   [-debug] [-help]
```

12.3 How the PCD host object works

When an object creation request arrives at a PCD host object as a normal method invocation, the host object checks the request's credentials against the user's LOID and the list of groups that are allowed to create objects on the host object. If the request's credentials pass inspection, the host object selects an account for the new object. Depending on its credentials, the request may be given a local user account or a generic (i.e., guest) account. Accounts are subject to scheduling and resource control (CPU time, memory usage, etc.), so an object's lease on an account, especially a generic account, is limited.

When a class object sends an object creation request to the host object it includes the new object's OPA as a parameter (see Figure 7, step 4, on page 44). The OPA contains the new object's vault directory (i.e., where the new object's persistent state will be stored), so before starting the creation process the PCD host object must switch the ownership of the new object's vault directory from the vault user id to the newly allocated user id. This switch gives the new object access to its persistent state and protects it against other objects (who will be running under different user ids).

The host object can then start the creation process, which will execute the object on the appropriate account. This involves some privileged operations (listed on page 59, above). The host object does not execute with root permissions: access to privileged operations is encapsulated in the PCD that runs on the host object. The PCD is configured to allow only the host object to have access to these operations. Two of its key functions are permitting the host object to change directory ownership and creating new processes on a designated account only. The PCD limits the accounts in which these two functions can be done to a set designated by the local system administrator. This set includes any generic (guest) Unix accounts and local Unix users that the administrator wishes to add.

PCDs can be used in two ways. First, they can multiplex objects onto multiple user accounts, providing a level of protection for user objects and, when combined with user logins, making it possible to audit a user's actions. Second, they can match an object's effective user id to the user's Unix user id, making it easier to track user actions: Legion maintains logs for all host objects in the $LEGION_OPR directory (see section 11.3.2), and the PCD host object logs include information about when different Unix users ids were used by Legion users.

12.4 Using a PCD host as your bootstrap host

You can use a PCD host as your bootstrap host. Before you run `legion_initialize` set the LEGION_HOST_BIN variable:

```
        export LEGION_HOST_BIN=PCDUnixHost
```

13.0 Batch queue host objects

The standard Legion host object creates objects using the process creation interface of the underlying operating system. However, some systems require using a queue management system to take full advantage of local resources. For example, some parallel computers contain a small number of “interactive” nodes, which can be accessed through normal means, and a large number of “compute” nodes, which can only be reached by submitting jobs to a local queue management system.

To make use of hosts that are managed by local queuing systems, Legion provides a modified host object implementation called the BatchQueueHost. BatchQueueHost objects submit jobs to the local queuing system, instead of using the standard process creation interface of the underlying operating system.

13.1 Starting a batch queue host object

To start a BatchQueueHost object, use `legion_starthost` with the `-B` flag to indicate the desired host object implementation. It would look something like this:

```
$ legion_starthost -B BatchQueueHost -N /hosts/SP2 \
  SP2.university.edu
```

Please see page 22 for more on starting new hosts.

13.2 Setting the local queue

A BatchQueueHost can be used with a variety of queue systems (LoadLeveler, Codine, PBS, and NQS are the currently supported queue types). You can specify what type of local queue a given BatchQueueHost object by editing the host object’s `host_queue_type` attribute. For example, if you want your new BatchQueueHost object to use the local “LoadLeveler” queue, you would run `legion_update_attributes` and add the “LoadLeveler” attribute:

```
$ legion_update_attributes /hosts/SP2 \
  -a “host_queue_type('LoadLeveler')”
```

Currently, each BatchQueueHost can use only one queue type at a time (i.e., if multiple local queuing systems are available, they can not all be used by the same BatchQueueHost: an individual BatchQueueHost would need to be started to represent each queue). Typically, though, individual machines are managed by a single queue.

13.3 Before running objects on the new host

By default every Legion class contains a `desired_host_property` attribute specifying that it be run on an interactive host. You can use the `legion_list_attributes` command to check this particular attribute:

```
$ legion_list_attributes -c /class/my_class \
 desired_host_property

/class/my_class:
 (desired_host_property)
 Total attributes retrieved 1
  desired_host_property('interactive')
```

This signals the scheduler that the class's instances should not run on BatchQueueHosts. This is based on the conservative assumption that any class can run on interactive hosts, but not all classes can run on batch hosts.

To allow instances of your class to run on BatchQueueHosts, you can just remove this attribute:

```
$ legion_update_attributes /class/my_class -d \
 “desired_host_property('interactive')”
```

13.4 Troubleshooting

If you are having trouble creating objects on a BatchQueueHost, there are several points of possible trouble. First be sure that you've removed the problem class's interactive `desired_host_property` (section 13.3). If you still have trouble, you may have a misconfigured host object. Check the following points to be sure that your host object is set up correctly.

- **The right "queue type" attribute should be set on the host**. You can use the `legion_list_attributes` command to check this.

```
$ legion_list_attributes -c /hosts/my_host host_queue_type
```

If the output shows the wrong queue type or no queue type, run `legion_update_attributes` to set `host_queue_type` correctly. For instance, if the host uses a LoadLeveler queue you could run the following:

```
$ legion_update_attributes-c /hosts/my_host -a \
 “host_queue_type('LoadLeveler')”
```

This "queue type" attribute points the host object to the location of the local Legion queue management scripts in the $LEGION_HPC/bin/QueueManagementScripts directory. The above command tells the host to look in $LEGION_HPC/bin/QueueManagementScripts/LoadLeveler. If the queue type attribute were set to Codine instead, the host would look for the queue management scripts in $LEGION_HPC/bin/QueueManagementScripts/Codine.

- **The appropriate corresponding directory must be in the host's $LEGION_HPC/bin/QueueManagementScripts directory.** It should contain the following queue management scripts.

```
legion_proxy_queue_load
legion_queue_cancel
legion_queue_load
legion_queue_status
legion_queue_submit
```

- **These scripts should have execute permissions set for the user-id that will be running the BatchQueueHost.** If all of this is set up correctly, the host should be calling the local scripts. If objects are still not being created correctly there may be a problem in the scripts.

You can get a better idea of whether or not the local scripts are being called and what they're doing by looking in the log file maintained by the scripts (look in $LEGION_OPR/Legion-BatchLog). You'll find this log on the host where the BatchQueueHost object is running. If the logs indicate that the scripts are never called there may be a scheduling problem.

There is also a six minute delay after you add a new host to the system before which it will not be selected for scheduling, so you may need to wait a few minutes before you can test a new batch queue host.

14.0 Virtual hosts

To support the use of resources for which there is no full port of the Legion system, Legion supports the notion of “virtual hosts,” host objects that run on a fully-supported Legion platform and represent a resource on an unsupported platform (e.g., a Cray T3E). A virtual host object cannot be used to run normal Legion objects (since by definition there is no Legion port for the represented machine). Instead, it is used to run native jobs, such as existing serial and MPI programs, with the standard Legion tools (`legion_run`, `legion_run_multi`, and `legion_native_mpi_run`[10]). The virtual nature of the host objects therefore remains transparent. The benefits of incorporating virtual hosts into the Legion system are many: transparent, simplified remote execution on the target machine; resource selection and scheduling of the machine through Legion mechanisms; etc.

To configure a virtual host object, use the following three steps:

1. **START A HOST OBJECT**

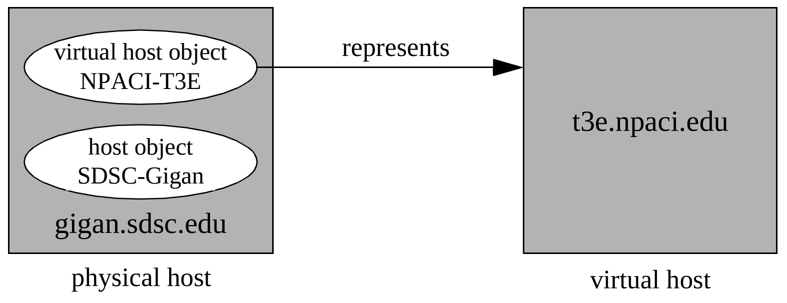
   Start a normal host object (any variety), with the standard `legion_starthost` command. Instead of starting the host on the desired target machine, however, start it on another machine that can conveniently be used to start jobs on the target machine (e.g., through a queue system, ssh, etc.). This machine is called the *physical host*. The target machine is called the *virtual host*.

   For example, start a virtual host object to represent the host t3e.npaci.edu on the physical host gigan.sdsc.edu you would run:

```
$ legion_starthost -N /hosts/NPACI-T3E gigan.sdsc.edu \
  /vaults/BootstrapVault
```

   This gives you a normal host object, except that it is not on its host (Figure 14). It uses the physical host’s bootstrap vault.

Figure 14: Newly created virtual host object



10 See page 100, page 104, and page 92, respectively, in the Reference Manual.

2. SET VIRTUAL ARCHITECTURE FOR THE HOST OBJECT

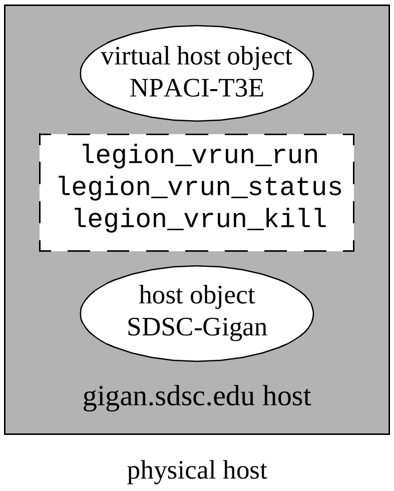
When the host object is first created, it is assumed to represent the architecture of the physical machine on which it resides. You must tell Legion that the host object will actually represent a machine of a different architecture. The `legion_set_varch` command sets a virtual architecture for a host object.

Continuing the previous example, then, you must set the virtual architecture for host object `NPACI-T3E` to `t3e`:

```
legion_set_varch /hosts/NPACI-T3E t3e
```

3. CONFIGURE VIRTUAL RUN SCRIPTS FOR THE HOST

Figure 15: Virtual host object scripts



When `legion_run` and other commands make use of a virtual host object to start native jobs, they require a mechanism for starting and managing jobs on the virtual host. To fill this need, there are three scripts that the virtual host object can call on the physical host to make use of the virtual host:

`legion_vrun_run`
`legion_vrun_status`
`legion_vrun_kill`

These scripts and the virtual host object are located on the physical host (Figure 15). Examples of these scripts are in the following location:

$LEGION_HPC/src/Tools/VirtualArchitecture

These versions use simple Unix `fork/exec` to demonstrate the required interface.

To configure the host with its required scripts, use the `legion_set_vrun` command, indicating the path at which the physical host can find the scripts. Continuing the above example:

```
$ legion_set_vrun /hosts/NPACI-T3E $LEGION/src/T3E/SDSC
```

The virtual host can be used normally for native jobs by registering programs for the (virtual host's) appropriate architecture and running them on the virtual host object. Only native jobs can be run on a virtual host. You can not run remote programs, because virtual hosts have no Legion binaries.

From the user's perspective, virtual and physical host objects are indistinguishable. For example, here we register a program for the T3E and run it on t3e.npaci.edu:

```
$ legion_register_program a.out /home/andrew/my_program t3e
$ legion_run /home/andrew/my_program
```

Notice that the program was registered and run from the physical host. In this case, there was no need to specify which host executes my_program, but you can use `legion_run`'s `-h` flag to specify a virtual host, if necessary.

15.0 Setting up a native MPI host

If you or your users are running native MPI code through Legion (via `legion_native_mpi_run`) you will need to install one class and set certain properties on the host.

To install the class, called `legion_native_mpi_backend`, run `legion_native_mpi_init`. This will install it in `/class`.

```
$ legion_native_mpi_init [<architecture>]
```

If you wish, you can specify an architecture for which an implementation for this class can be registered. You can run the command multiple times to specify multiple architectures.

To set native MPI properties on a host, run the `legion_native_mpi_config_host` command.

```
$ legion_native_mpi_config_host [<wrapper>]
```

If you wish, you can specify a wrapper script that locates `mpirun` on the host. If you do not, the command will use the `legion_native_mpich_wrapper` script, which is for an MPICH implementation. The script is in the HPC package in $LEGION_HPC/src/Tools/MPI/NativeMPI/.

16.0 Legion domains

Legion supports the concept of linking together discrete Legion systems. Previously, systems were isolated entities: objects running in one system can not communicate with objects running in another system. Now Legion treats an individual system as a *domain*: a domain contains all features necessary for running any Legion application and can be run autonomously. However, multiple domains can be combined to form a larger virtual machine. Objects created in one of these domains can communicate with and use the services of other objects in connected domains.

16.1 Naming Legion domains

A domain is automatically assigned system-level a *domain identifier* when it is first created (i.e., when a new Legion system is configured with the `legion_setup_state` command). The domain identifier is a variable-length field of bytes. It is embedded into the second field of all of its LOIDs. For example, the domain below has a one-byte domain-id, `c8`:

```
        1.c8.07.0800000.000001...
```

A different domain has a four-byte domain-id, 35d82a07:

```
        1.35d82a07.05.03000000...
```

All class objects in a domain will use the same domain identifier when assigning new objects LOIDs. You can specify the identifier if `legion_setup_state` is run interactively (with `-i`). Otherwise Legion will select a four-byte domain-id. Once the domain-id has been assigned, it cannot be changed.

You must have root user privileges in your current domain in order to connect with other domains (i.e, you must be logged in as `/users/admin`).

16.2 Domains and binding services

To locate an object in another domain you must contact the domain's binding services, which can then track down the object's LOID and location. However, if you know the domain's LegionClass's binding (i.e., the LOID and Object Address of the domain's metaclass) you can find the domain's binding services.

You can use the `legion_print_config` tool to display your current domain's LegionClass binding.

```
$ legion_print_config
 - LegionClass Configuration -

LOID = 1.35e09dfb.01..000001fc0b347...
OA  = [128.143.63.50: 6384: 903989954]
$
```

The LegionClass's binding is found in a file called LegionClass.config in each domain's $LEGION_OPR directory. The LegionClass.config file is a LegionBuffer, however, and to mask the data format of the file's contents the file is accompanied by a file called LegionClass.config._LegionStorage _MetaData_. If the metadata file is not present the LegionClass.config file may be unreadable.

16.3 Joining domains

Legion domains can be combined together to form larger systems with `legion_combine_domains`. This tool connects your current domain (i.e., the one in which you execute the command) to a specified target domain. If other domains have already been connected to either your current domain or to the target domain, they will be part of the new multidomain system as well.

To perform this operation, the tool makes the domains' LegionClass objects aware of one another. In practice, this involves determining the bindings of all of the involved LegionClass objects and broadcasting the complete binding set to all of the LegionClass objects. Once this operation has been performed, the binding trees of the different domains will be connected. In effect, the set of joined LegionClass objects represent a distributed class-map. Binding traffic that reaches LegionClass in your current domain but is related to another domain will be forwarded to the LegionClass object in the appropriate domain. Binding caches and the class-of operations involved in the binding process will minimize the need for interdomain binding-related traffic between LegionClass objects). As in earlier versions of the system, the global (now distributed) class-map in Legion is protected from contention by heavy caching.

To do its job, `legion_combine_domains` needs to have information about each domain that it will be linking. If Legion security is turned on in any of the involved domains it will also need security credentials for each secure domain in order to authorize it to link external domains. This domain information is stored in the form of a *domain cookie*. A domain cookie is simply a file holding the needed binding information and security credentials for a single Legion domain. The `legion_generate_domain_cookie` command creates a cookie file and `legion_print_domain_cookie` displays it.

16.4 Related commands

There are four commands related to Legion domains. The `legion_list_domains` command lists the set of domains currently connected to your domain, `legion_combine_domains` connects domains, `legion_generate_domain_cookie` generates a domain cookie for a domain, and `legion_print_domain_cookie` displays the cookie.

16.4.1 Listing currently connected domains

You can use the `legion_list_domains` command to view a list of those domains connected to your current domain. The output will list your current domain's binding and any domains linked to your current domain.

```
$ legion_list_domains
Current Legion domain root:
Type 302 binding:[ 1.35d82a07.01..000001fc0
  c0e21f57326b63336de9fc4d88d7bf5a314d9f1df
  1079abb0938b29b3643e6c9a8413ea6fd584f82be
  29b0ba56cdd0d421a609a4ba9ecf995c8ddb20b16
  d6df : [128.143.63.51 : 19870 : 903621581 ] ]

Linked external Legion domain roots (1):
Type 302 binding:[ 1.c8.01..000001fc0a533f0
  8413082b08857f283c8a0aa34193ea7478b2c6081
  63414ca5f13939bb0e5d48788b543d5fddd05e497
  35487150edf8256d78002bb04454da7eae82697 :
  [128.143.63.52 : 16022 : 903624927 ] ]
$
```

This output shows that there are two domains and lists each domain's binding (its LegionClass object's LOID and OA). The current domain is listed first. Note that the domains' identifier can be seen in the second field of the two LegionClass LOIDs: the first is 35d82a07 and the second is c8.

If you are in a single domain system, the output will simply list your current domain's binding.

```
$ legion_list_domains
Current Legion domain root:
Type 302 binding:[ 1.35d82a07.01..000001fc0
  c0e21f57326b63336de9fc4d88d7bf5a314d9f1df
  1079abb0938b29b3643e6c9a8413ea6fd584f82be
  29b0ba56cdd0d421a609a4ba9ecf995c8ddb20b16
  d6df : [128.143.63.51 : 19870 : 903621581 ] ]

No linked external Legion domains.
$
```

16.4.2 Generating cookies

As the name suggests, the `legion_generate_domain_cookie` command generates a cookie file for your current Legion domain. Usage is:

```
legion_generate_domain_cookie [-help]
   [-o <cookie output filename>]
```

The cookie contains binding information for your domain's LegionClass, security credentials, and information about your domain's context space. By default, the new cookie file will be named `LegionDomainCookie.<domain-id>`. Use the `-o` flag to specify a different name.

If security has been turned on, you must be logged in as `/users/admin` in your current domain in order to ensure that the proper credentials are generated and saved in the cookie file.

16.4.3 Displaying cookies

The `legion_print_domain_cookie` command will display the contents of a Legion domain cookie file. By default, the command will display the contents of `LegionDomainCookie.<domain-id>`. Use the `-i` flag to specify a different cookie filename.

16.4.4 Connecting domains

The `legion_combine_domains` tool connects Legion domains together into a single, larger Legion system. Once joined, objects in connected domains can communicate with each other as easily as objects in a single domain communicate with each other. Usage of this command is:

```
legion_combine_domains [-help] [-v]
   <list of domain cookie files>
```

Before you run `legion_combine_domain`, you must obtain a copy of the domain cookie files from all of the domains involved (i.e., if you wish to join a multidomain system you must have copies of all of the domains' cookie files).

In this example, two domains are connected together.

```
$ legion_combine_domains LegionDomainCookie.35d82a07 \
 LegionDomainCookie.c8
Created 2 new domain interconnections
$
```

Note that the number of interconnections includes connecting the new domain to each previously linked domain: if you added another domain to this group you'd make four new interconnections.

17.0 Resource management

A primary motivation in Legion's design is flexibility and transparency: programs can be distributed and run on widely distributed resources without the user having to engage in complex, time-consuming negotiations with individual site administrators. However, in order to allow sites to protect their resources against unauthorized or malicious use, Legion provides tools to allow system administrators can maintain their local policies. Final authority over the use of a resource remains with each resource's administrators.

There are three objects for managing your Legion resources: the *collection*, the *scheduler*, and the *enactor*. There are a corresponding set of command-line tools to control the objects. A more detailed discussion of these objects is in section 8.0 of the Developer Manual, but in brief they carry out the following functions.

- The **collection** collects and maintains information about its assigned resources. It constantly monitors its host and vault objects and knows which resources are in use and which are available for what kind of tasks.
- The **scheduler** takes this information and produces lists of possible resources for specific tasks.
- The **enactor** negotiates with those resources to reserve blocks of time and space.

Resource managers can use scheduling-related commands (below) to set up system, class, and instance level scheduling policies.

17.1 Scheduling-related commands

There are a several commands that can be used to set up an individual Legion's scheduling process and a class's or instance's host and vault placement policy. Please see page 45 in the Reference Manual for details of these commands' syntax and usage.

17.1.1 Configuring the scheduler

The `legion_config_scheduler` utility configures a basic Legion scheduler's helper objects. Use it to assign a particular collection and enactor to a basic Legion scheduler or vice versa. It can also be used to query which helper objects have been set for a basic Legion scheduler. The example below shows the LOID of the default scheduler object's enactor.

```
$ legion_config_scheduler /etc/DefaultScheduler
  -get_enactor
Current enactor is: 1.36baeb09.66000000.01000000.00...
$
```

17.1.2 Setting a class's default scheduler

This `legion_set_scheduler` command sets a specific class's default scheduler. The class will then use its assigned scheduler object to determine which hosts and vaults should manage its instances (i.e., determine placements for the class's instances). The example below sets `SchedulerFoo` as the default scheduler object for `ClassFoo`.

```
$ legion_set_scheduler /class/ClassFoo SchedulerFoo
```

All of `ClassFoo`'s instances will be placed with `SchedulerFoo`.

17.1.3 Setting scheduler policy

The `legion_set_scheduler_policy` command sets a class object's policy for using its default scheduler. There are two policy options, which determine whether or not the class uses its default scheduler if the scheduler object is not active. Depending on its type, a class may require a policy which does not use an inert scheduler. If not, classes should have a default placement available

17.1.4 Adding resources to a collection

There are three commands for controlling the collection object. Objects can be added with `legion_join_collection`. The example below adds `HostFoo` to the default collection, although any Legion object can be added to a collection.

```
$ legion_join_collection /etc/Collection /hosts/HostFoo
```

The `legion_leave_collection` command removes objects from the collection object.

The `legion_query_collection` prints a list of which objects are currently part of a given collection. The `legion_query_collection` command uses MESSIAHS Interface Language (MIL) query strings (see page 51 in the Reference Manual for query string examples and page 127 in the Developer Manual for relevant MIL interfaces). The example below returns the list of Linux host objects that are part of the default collection.

```
$ legion_query_collection /etc/Collection \
 'match(host_os_name,”Linux”)'
2 hits:
1.36baeb09.07.01000000.000001fc0b54bbc102...
1.36baeb09.03.01000000.000001fc0f4b64b072...
$
```

17.1.5 Subcollections

A subcollection is a normal collection object that is polled by another collection object (called the parent collection). This arrangement allows for faster and more efficient resource information gathering, since you can have several subcollections monitoring small groups of resources. A parent collection can have more than one subcollection and a subcollection can have more than one parent as well as its own subcollections. A parent runs one or more MESSIAHS-type queries on a subcollection (see page 51 in the Reference Manual for query string examples).

To create a parent-subcollection arrangement, use the `legion_add_sub_collection` command. You must specify a parent and a subcollection (both collections must already exist). You can also specify a query to be started by the parent on the subcollection. If no query is specified, the default value of 'true' will be used. You can run this command multiple times to start multiple queries on a single subcollection.

You can use the `collection_update_frequency_secs` attribute to adjust how often a collection polls its resources. The default setting is 300 seconds. Use the `legion_update_attributes` command:

```
$ legion_update_attributes -c /etc/Collection \
  “collection_update_frequency_secs(600)”
```

This will set the default collection to update itself every 10 minutes.

There are two other subcollection commands:

- `legion_list_sub_collections`, which displays your existing parent-subcollection relationships and queries
- `legion_remove_sub_collection`, which can be used to stop specific queries or to end a parent-subcollection relationship.

Please see section 2.6 in the Reference Manual for more information about using these commands.

Getting help

Please contact us at one of the addresses listed below if you have trouble with the system. Please be sure to include any relevant information about the state of your system before and during the problem.

We would greatly appreciate hearing from you whenever you find errors or bugs in Legion, so that we can avoid similar problems in future releases.

E-mail

For bug reports, help, and general Legion information, please send an e-mail message to <legion-help@virginia.edu>.

Contact Information

Legion Group
Department of Computer Science
School of Engineering & Applied Science
University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, VA 22904-4740
http://legion.virginia.edu/
fax: 434-982-2214

On-Line Help

A variety of technical notes, reports, and on-line tutorials are available on the Legion web site, at <http://legion.virginia.edu/>.

Index