

# LFC User Guide

Version 2.0

Raoul A.F. Bhoedjang

Tim Rühl

Kees Verstoep

Henri E. Bal

October 1, 1999



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	LFC . . . . .	5
1.2	LFC structure . . . . .	5
1.3	Limitations . . . . .	5
1.3.1	Runtime environment and portability . . . . .	5
1.3.2	Device sharing . . . . .	6
1.3.3	Protection . . . . .	6
1.3.4	Multithreading . . . . .	6
1.4	Status and future work . . . . .	6
<b>2</b>	<b>LFC core</b>	<b>9</b>
2.1	Initialization and cleanup . . . . .	9
2.2	Configuration information . . . . .	11
2.3	Sending . . . . .	11
2.4	Receiving . . . . .	12
2.5	Interrupt management . . . . .	14
2.6	Memory allocation . . . . .	14
2.7	Timer support . . . . .	15
2.8	Fetch-and-add . . . . .	15
2.9	Statistics . . . . .	16
<b>3</b>	<b>Compiling LFC programs</b>	<b>17</b>
3.1	Include files . . . . .	17
3.2	Libraries . . . . .	17
3.3	Example Makefile . . . . .	17
<b>4</b>	<b>Running LFC programs</b>	<b>19</b>
4.1	Files . . . . .	19
4.2	Environment variables . . . . .	19
4.3	Running with prun . . . . .	19
4.4	Starting programs with rsh . . . . .	20
4.5	Debugging tips . . . . .	20
<b>5</b>	<b>LFC Example Application</b>	<b>21</b>



# Chapter 1

## Introduction

### 1.1 LFC

This document describes LFC, a low-level messaging layer for Myrinet [2]. LFC is intended to be used by the developers of runtime systems for parallel programming systems. It provides packet unicast, packet broadcast, fetch-and-add, interrupt management, and a microsecond timer.

### 1.2 LFC structure

LFC mainly provides a low-level packet interface. We have chosen this low-level interface because it can be implemented efficiently and because different higher-level interfaces can be layered on top of it quite easily. We have deliberately excluded such a higher-level interface, because no such interface suits all users. Our experience with various client systems indicates that this was a good choice.

For a generic higher level interface, we suggest looking at Panda, a portable communication/multithreading layer which has been ported to LFC. On top of Panda we implemented several communication layers (MPI, PVM, CRL) and languages (Orca and an efficient parallel version of Java called Manta). The disadvantage of layering software is usually some loss of performance, but when done carefully the resulting system can still be quite efficient. In our experience the advantages in software reuse can often outweigh the slight loss in performance.

### 1.3 Limitations

The following subsections describe the limitations of LFC.

#### 1.3.1 Runtime environment and portability

LFC was designed specifically for Myrinet. Our implementation therefore only runs on Myrinet. In addition, this implementation makes a number of assumptions that are satisfied on our DAS cluster system, but that will need extra effort on other systems.

- The implementation assumes that virtual memory pages can be locked in memory. Several, but not all, operating systems allow user-level processes to achieve this using the `mlock` system call.
- The implementation assumes that it can obtain the physical addresses of locked pages. To achieve this, it is usually necessary to add a small driver to the operating system. We have written such a pseudo device driver (*asmapi*) for Linux and BSD/OS.
- The implementation assumes that all host processors use the same byte order. The implementation does not assume that host processors use the same byte order as the LANai.

- The implementation assumes that the network hardware neither drops nor corrupts packets. LFC can be configured to test for CRC errors, but it does not recover from such errors.
- The implementation assumes a Myrinet device driver that transforms all interrupts generated by the network interface into a user-level SIGIO signal.
- The implementation assumes the presence of a startup daemon. This daemon is used to synchronize all participating processes before they send any messages over the Myrinet network. Without such a daemon, initialization messages could block the network for a sufficiently long time to cause problems for other jobs running on the cluster. (Currently the use of the startup daemon is a configuration option).

### 1.3.2 Device sharing

LFC is not capable to service more than one user process per host, mainly because LFC's LANai control program does not separate the packets from different users. We have modified the Myrinet device driver to return an error if a second user tries to obtain access to the Myrinet device.

### 1.3.3 Protection

LFC does not implement any form of protection. Specifically, users and administrators should be aware that:

- Users can freely modify the LANai control program that runs on the network interface. Once modified, the control program can both read and write **all** host memory locations.
- LFC's LANai control program does not make a serious attempt to reject network packets that do not originate from the current user.

### 1.3.4 Multithreading

LFC is not multithread-safe. LFC's initial clients were all single-threaded and we have been reluctant to make LFC dependent on specific thread packages. We do have a port of the (multithreaded) Panda communication system [1, 9] to LFC. With some careful locking, Panda avoids concurrent invocations of LFC primitives.

## 1.4 Status and future work

Using LFC, we have developed the following client systems:

- CRL [5]. CRL is a distributed shared memory system. The CRL implementation only uses LFC's core.
- MPI [4]. MPI is a standard message passing interface. Our port used to be based on the FM-2 port of MPICH by Mario Lauria [7]. (MPICH is a publicly available MPI implementation.) We now have an implementation that uses Panda as its intermediate layer (see below).
- TreadMarks [6]. TreadMarks is page-based distributed shared memory systems. The LFC port uses two extra modules: one dedicated to upcall handling and one implementing streams efficiently.
- Panda [1, 9]. Panda is a multithreaded communication library that provides reliable message passing, remote procedure call, and totally ordered group communication. The Panda port only uses LFC's core. With Panda as intermediate layer we have implemented various other systems, e.g., MPI, PVM and Orca.

Other LFC related topics we are working on include:

- A zero-copying interface. Currently LFC uses Programmed I/O on the send side and a restricted form of DMA at the receiving side. It would be interesting to see to what extent a zero-copying interface would gain (especially at the application level).
- A remote read/write interface. It is fairly simple to add a remote read and write interface to LFC. We are already experimenting with such an extension to be used in a parallel game tree searching system.
- Alternatives for the credit based “careful” implementation of LFC. We are currently experimenting with various other implementations, including a sliding window protocol that is able to recover from packet corruption and message loss.



# Chapter 2

## LFC core

### 2.1 Initialization and cleanup

#### `lfc_init`

---

```
typedef enum { LFC_NO_INTR, LFC_INTR } lfc_intr_t;
```

```
void lfc_init(int *argc, char **argv);
```

```
int lfc_group_create(unsigned *members, unsigned nr_members);
```

```
void lfc_start(void);
```

```
void lfc_exit(void);
```

---

The call to `lfc_init` should precede all other calls to LFC functions. Argument vector `argv` is a list of string arguments, usually the one passed as argument to `main`. Function `lfc_init` parses `argv` and removes all arguments that it recognizes. `*argc` contains the initial argument count and is decreased by `lfc_init` each time that it removes some argument. As usual, `argv[0]` contains the application name.

Function `lfc_init` is able to derive the information about the participating nodes based on an environment variable `HOSTS` or by means of the parameter `-lfc-hosts=<hostlist>` supplied to the application.

If the space (or comma) separated host list is specified by means of the environment variable `HOSTS` (see Section 4.2), `lfc_init` expects that:

- `argv[1]` contains the rank of the invoking process; this is the number that will be returned by `lfc_my_proc`. Each application process should be given a unique rank in the range `0 ... lfc_nr_procs()-1`.
- `argv[2]` contains the number of participating processes; this is the number that will be returned by `lfc_nr_procs`.

In this case `lfc_init` removes both `argv[1]` and `argv[2]` from the argument list.

Alternatively, the participating nodes can be specified by the `-lfc-hosts=<hostlist>` option. Here, `<hostlist>` is a comma (or space) separated list of node names. The rank of a process is determined by the index of its host in the hostlist.

In addition, `lfc_init` recognizes the following arguments:

<code>-lfc-use-hosts</code>	In case environment variable <code>HOSTS</code> is used, derive rank information from that alone. The rank and number of nodes arguments are not be passed separately in this case.
<code>-lfc-verbose</code>	Show LFC configuration information during initialization.
<code>-lfc-ring-frames=&lt;size&gt;</code>	Change the size of the host-resident receive queue. LFC increases the size on demand, so this normally should not be necessary
<code>-lfc-no-intr</code>	Disable all receive interrupts. This can be slightly more efficient than the default behaviour, where the LCP polls the host to see if interrupts should be generated.
<code>-lfc-interval=&lt;usec&gt;</code>	Initial and subsequent interrupt delay in microseconds; see Section 2.4.
<code>-lfc-intr-first=&lt;usec&gt;</code>	Initial interrupt delay in microseconds, see Section 2.4
<code>-lfc-intr-first=&lt;usec&gt;</code>	Subsequent interrupt delay in microseconds, see Section 2.4
<code>-lfc-stats</code>	Collect and print statistics, both on the host and the LCP; see Section 2.9. This option forces the use of specially instrumented version of the LCP, with a slight loss in performance.
<code>-lfc-lcp=&lt;LCPfile&gt;</code>	Use the LFC LCP file instead of the default one configured during the compilation of the LFC library. Note that both the library and the LCP contain a version string that LFC checks to ensure compatibility.
<code>-lfc-tree=&lt;tree&gt;</code>	Specify an explicit multicast tree topology. The allowed types are <code>binary</code> , <code>bst</code> (binomial spanning tree), and <code>chain</code> (linear forwarding chain). The default multicast topology is <code>binary</code> which offers a good compromise in latency and throughput (due to the constant fan-out of two).
<code>-lfc-routes=&lt;file&gt;</code>	Use a different routing file than the default (usually <code>/usr/local/package/lfc/etc/routes.lfc</code> ).

## **lfc\_group\_create**

Processes can create static multicast groups by calling `lfc_group_create`. For each group to be created, all participating processes must invoke `lfc_group_create`. `lfc_group_create` must also be called by processes that are not a member of the group to be created. `lfc_group_create` should be considered a collective operation, although the current implementation does not synchronize the calling processes. All groups must be created before invoking `lfc_start`.

All callers must specify the same list (`members`) of group members. This list contains `nr_members` process ranks. The ranks should be unique and all calling processes must specify the same ranks in the same order.

`lfc_group_create` returns a globally unique multicast group identifier which can be passed to `lfc_mcast_launch`.

## **lfc\_start**

`lfc_start` synchronizes all participating processes. No packets will be delivered before all participating processes have called `lfc_start`. Interrupts are always disabled when `lfc_start` returns.

## **lfc\_exit**

`lfc_exit` cleans up LFC. No LFC functions should be invoked after `lfc_exit`.

## 2.2 Configuration information

---

```
unsigned lfc_my_proc(void);

unsigned lfc_nr_procs(void);

unsigned lfc_packet_size(void);

void lfc_print_config(void);
```

---

### **lfc\_my\_proc**

Returns the rank of the invoking process. This rank is in the range 0 ... `lfc_nr_procs()-1`. The value returned is a runtime constant.

### **lfc\_nr\_procs**

Returns the number of participating processes. The value returned is a runtime constant.

### **lfc\_packet\_size**

Returns the maximum user payload in bytes of both send and receive packets. This is a runtime constant.

### **lfc\_print\_config**

Prints configuration information on `stdout` (mainly compile-time configuration information).

## 2.3 Sending

The send interface is described below. To avoid unnecessary copying, LFC gives users direct access to send packets in network interface memory.

LFC only preserves FIFOness between packets that are transmitted by the same send primitive. For example, FIFOness is preserved between all packets sent by `lfc_ucast_launch`, but no guarantees are given when one packet is transmitted using `lfc_bcast_launch` and the next packet is transmitted using `lfc_ucast_launch`. The destination of the unicast packet may receive the unicast packet before it receives the broadcast packet.

---

```
void *lfc_send_alloc(int upcalls_allowed);

void lfc_ucast_launch(unsigned dest, void *packet,
                    unsigned size, int upcalls_allowed);

void lfc_mcast_launch(int gid, void *packet,
                    unsigned size, int upcalls_allowed);

void lfc_bcast_launch(void *packet,
                    unsigned size, int upcalls_allowed);
```

---

All calls listed below that take a parameter named `upcalls_allowed` potentially drain the network while waiting for resources. `upcalls_allowed` indicates whether LFC is allowed to invoke `lfc_upcall` while draining the network. A nonzero value means that upcalls are allowed; a zero value

means that no upcalls should be made while this function executes. This feature should be used with great care. In general, allowing upcalls is the safest way to go, even if it complicates your code. You should only disable upcalls when you know that LFC has enough free packets to buffer incoming data.

### **lfc\_send\_alloc**

`lfc_send_alloc` allocates a send packet on the network interface and returns a pointer to the data part of the packet. The size (in bytes) of this data part is given by `lfc_packet_size`. Users can copy their data into the data part of the packet.

**Warning** Send packets are allocated in a special virtual memory segment. Writes to this segment are not guaranteed to be performed in program order. Users should therefore not read from send packets.

### **lfc\_ucast\_launch**

`lfc_ucast_launch` transmits the first `size` bytes in `packet` to `dest`. `Size` should not exceed `lfc_packet_size()`. Packet must have been allocated using `lfc_send_alloc`. `lfc_ucast_launch` returns ownership of the packet to LFC; users cannot use a packet after it has been launched.

### **lfc\_mcast\_launch**

`lfc_mcast_launch` multicasts the first `size` bytes in `packet` to all processes in group `gid` except the sender. `Size` should not exceed `lfc_packet_size()`. Packet must have been allocated using `lfc_send_alloc`. `lfc_mcast_launch` returns ownership of the packet to LFC; users cannot use a packet after it has been launched.

### **lfc\_bcast\_launch**

`lfc_bcast_launch` broadcasts the first `size` bytes in `packet` to all participating processes except the sender. `Size` should not exceed `lfc_packet_size()`. Packet must have been allocated using `lfc_send_alloc`. `lfc_bcast_launch` returns ownership of the packet to LFC; users cannot use a packet after it has been launched.

## **2.4 Receiving**

Packets can be received in two ways, through explicit polling or by means of a network interrupt which is transformed to a signal (SIGIO). In both cases, LFC passes an incoming packet to a user-supplied upcall function named `lfc_upcall`.

LFC generates interrupts as long as there are packets that have not yet been passed to `lfc_upcall`. Compared to a successful poll, an interrupt is very expensive. Therefore, LFC delays interrupts for a short while, optimistically assuming that the user will soon poll. (This mechanism, proposed in [8], is called a **polling watchdog**.) Users can override the default delay by means of the `-lfc-interval` option (see Section 2.1).

In some cases, users cannot completely process a packet in the context of `lfc_upcall`. To avoid copying, users may hold on to their receive packets until they can process them. In such cases, it is the user's responsibility to return the packet to LFC explicitly, using `lfc_packet_free`.

---

```
typedef enum { LFC_UPCALL_MCAST = 0x1 } lfc_upcall_flags_t;

extern int lfc_upcall(unsigned src, void *packet, unsigned size,
                    lfc_upcall_flags_t flags);

void lfc_poll(void);

void lfc_packet_free(void *packet);
```

---

## **lfc\_upcall**

`lfc_upcall` should be defined by the user. LFC calls this function exactly once for each incoming packet and only (!) in one of the following cases:

- as the result of a user's call to `lfc_poll` (see below);
- as the result of a SIGIO signal, unless LFC's signal handler has been replaced.

In both cases LFC disables network interrupts before calling `lfc_upcall` and re-enables network interrupts when `lfc_upcall` returns.

The parameters of `lfc_upcall` provide the following information about the packet:

- `Src` is the rank of the process that sent the packet.
- `Packet` is a pointer to the data part of the packet. Users can directly read data from this data part.
- `Size` is the size of the valid data part in bytes. This size cannot exceed `lfc_packet_size()` bytes.
- `Flags` is a combination of the flags specified in `lfc_upcall_flags_t`. These flags have the following meaning:
  - `LFC_UPCALL_MCAST` The packet is a multicast packet.

The return value indicates whether the user wants to keep the packet. When 0 is returned, LFC assumes the user no longer needs the packet and recycles it. Otherwise (nonzero), the user remains the owner of the packet. When the user has finished processing the packet, he should return it to LFC by means of `lfc_packet_free`.

**Warning** Receive packets are a relatively scarce resource, because LFC stores them in pinned memory. Users should therefore return receive packets to LFC as soon as possible.

**Warning** LFC separates the draining of the network and user processing of network packets [3]. While this is usually convenient in that no unexpected upcalls occur, it also implies that LFC will run out of receive packets if the user continuously injects packets into the network without consuming incoming packets. Therefore, users should either enable interrupts or poll frequently (unless the user knows that there cannot be any pending packets).

## **lfc\_poll**

Users can explicitly poll for incoming packets. `lfc_poll` will check if new packets have arrived. It will invoke `lfc_upcall` once for each new packet. This invocation runs in the context of the user's call to `lfc_poll`, so blocking in `lfc_upcall` will also block the thread/process that invoked `lfc_poll`.

## **lfc\_packet\_free**

The packet is returned to LFC. `lfc_packet_free` should only be used if the `lfc_upcall` that delivered the packet returned nonzero.

## 2.5 Interrupt management

LFC allows clients to receive messages in an interrupt-driven way, by means of the Unix SIGIO signal. By default, LFC catches this signal and processes it as follows.

- If the client system runs with interrupts disabled, the signal is ignored.
- Otherwise, the signal handler checks if any new packets have arrived. For each packet, it invokes `lfc_upcall`.

Clients can disable all network interrupts at startup time (see `lfc_start`). To disable interrupts temporarily, either to achieve atomicity or to avoid being interrupted while polling the network, use `lfc_intr_disable` and `lfc_intr_enable` (described below).

Clients that replace LFC's SIGIO handler by a handler of their own, can no longer rely on `lfc_intr_disable` and `lfc_intr_enable` (see below).

---

```
void lfc_intr_disable(void);
```

```
void lfc_intr_enable(void);
```

---

### `lfc_intr_disable` / `lfc_intr_enable`

`lfc_intr_disable` and `lfc_intr_enable` should always be called in pairs, with `lfc_intr_disable` going first. For convenience, such pairs may be nested in time, but only the outermost pair will actually affect the current interrupt status.

An outermost call to `lfc_intr_disable` disables network interrupts; when `lfc_intr_disable` has returned, LFC will no longer invoke `lfc_upcall` in the context of its SIGIO signal handler. SIGIO signals may still be generated, but they will be silently discarded. With interrupts disabled, `lfc_upcall` can still be invoked as the result of a user call to `lfc_poll`.

An outermost call to `lfc_intr_enable` re-enables network interrupts. After `lfc_intr_enable` has returned, `lfc_upcall` can again be invoked in the context of LFC's SIGIO signal handler.

The cost of using `lfc_intr_disable` and `lfc_intr_enable` is modest; these routines merely set a global flag in local memory.

### `lfc_intr_status`

Returns 0 iff interrupts are disabled and nonzero otherwise.

## 2.6 Memory allocation

It is unsafe to call `malloc` and family in the context of a signal handler. Since it is sometimes convenient to dynamically allocate memory in packet handlers, we have written a version of `malloc` and family that operate on a **small** heap that is not shared with the standard C allocation routines. These routines can be used safely in the context of `lfc_upcall`, even if `lfc_upcall` is invoked by LFC's SIGIO signal handler.

---

```
#include <stddef.h>
```

```
void *lfc_malloc(size_t size);
```

```
void *lfc_calloc(size_t nmemb, size_t size);
```

```
void *lfc_realloc(void *ptr, size_t size);
```

```
void lfc_free(void *ptr);
```

---

These functions behave just like their C library counterparts, but operate on another, independent heap. (These routines are currently not part of LFC's core distribution).

## 2.7 Timer support

LFC provides a single timer with microsecond granularity. In addition, there is a routine to delay a process for a fixed amount of time. We found this useful in the implementation of several benchmarks.

---

```
void lfc_timer_start(void);

void lfc_timer_stop(void);

double lfc_timer_elapsed(void);

void lfc_timer_delta(unsigned microsec);
```

---

### **lfc\_timer\_start**

`lfc_timer_start` restarts the timer.

### **lfc\_timer\_stop**

`lfc_timer_stop` stops the timer.

### **lfc\_timer\_elapsed**

`lfc_timer_elapsed` returns how many microseconds elapsed between the last `lfc_timer_start-lfc_timer_stop` pair.

### **lfc\_timer\_delta**

`lfc_timer_delta` spin-loops until `microsec` microseconds have passed.

## 2.8 Fetch-and-add

The fetch-and-add function is useful for obtaining global sequence numbers efficiently.

---

```
unsigned lfc_fetch_and_add(unsigned dest, int upcalls_allowed);
```

---

### **lfc\_fetch\_and\_add**

`lfc_fetch_and_add` performs an atomic fetch-and-add operation on a per-LANai variable. The variable is stored in the memory of the LANai attached to the processor with rank `dest`. All variables are initialized to 0 (zero). While waiting for the result, `lfc_fetch_and_add` will drain the network. If and only if `upcalls_allowed` is nonzero, `lfc_fetch_and_add` will also make upcalls while draining the network.

## 2.9 Statistics

LFC collects two types of statistics: host statistics and network interface statistics. Host statistics are always collected, while network interface statistics are only collected by a special LANai control program. This control program can be selected in two ways:

- by setting the environment variable **LFC\_STATS**;
- by passing `-lfc-stats` to `lfc_init`.

Note that the selection is done at runtime. It is not necessary to recompile or relink to collect statistics.

Users can collect the values of the per-processor statistics in a single statistics buffer by calling `lfc_stats_gather`. The contents of the buffer can be dumped to a file using `lfc_stats_dump`.

---

```
#include <stdio.h>

void lfc_stats_reset(void);

int lfc_stats_create(void);

void lfc_stats_gather(int statbuf);

void lfc_stats_dump(FILE *fp, int statbuf, char *hdr, char *ftr);
```

---

### **lfc\_stats\_reset**

`lfc_stats_reset` is a collective operation that should be called by all participating processes. It synchronizes all processes (like a barrier) and resets their local host and network interface statistics.

### **lfc\_stats\_create**

`lfc_stats_create` creates a statistics buffer on processor 0. It returns a unique identifier for the buffer. This routine should be called in the same order by all participating processes, so that all processes agree on the identifiers of different buffers. This routine does not synchronize the calling processes.

### **lfc\_stats\_gather**

`lfc_stats_gather` is a collective operation that should be called by all participating processes. It synchronizes all processes (like a barrier) and collects the current values of all processes' host and network interface statistics in the statistics buffer identified by `statbuf`. All processes should specify the same `statbuf`.

### **lfc\_stats\_dump**

`lfc_stats_dump` prints the contents of the statistics buffer identified by `statbuf` to the stream named `fp`. `lfc_stats_dump` should only be called on processor 0. Before printing the statistics, `lfc_stats_dump` prints the header string `hdr` and the footer string `ftr`. A newline is appended to both strings. If `hdr` is a null pointer, no header is printed. If `ftr` is a null pointer, no footer is printed.

## Chapter 3

# Compiling LFC programs

The following explains how to compile programs that use LFC. We use  $\${LFC}$  to refer to the root directory of the LFC installation.

### 3.1 Include files

Every program that uses LFC must include  $\${LFC}/include/lfc.h$ . Each module supplies its own include file in  $\${LFC}/include$ .

### 3.2 Libraries

Every program that uses LFC must be linked against an LFC library. Libraries are found in subdirectories of  $\${LFC}/lib$ . By default, two versions of the LFC library are available: `optimized` and `debug`. Usually, library `optimized` should be used, since it offers the best performance. However, if you expect something is wrong with your application, it may be useful to link with the `debug` version, since provides additional assertion checks in the LFC library, and gives more useful stack traces when examining the process state or core file with a debugger.

In addition, the following DAS and Myrinet libraries are needed. These libraries are used by LFC for initializing the network interfaces with LFC's firmware and setting up the Myrinet routes.

```
libdas.a  
libDpi.a  
libLanaiDevice.a  
libbfd.a  
libliberty.a
```

### 3.3 Example Makefile

The GNU Makefile in Figure 3.1 can be used to compile the example program `latency.c`.

```
CONF      := optimized          # which LFC library version to use
LFC       := /usr/local/package/lfc
MYRINET   := /usr/local/package/myrinet
DASLIB    := $(LFC)/support/das

CC        := gcc
CFLAGS    := -g -Wall -Wmissing-prototypes -Wstrict-prototypes
CPPFLAGS  := -I$(LFC)/include

LD        := $(CC)
LDFLAGS   := -L$(LFC)/lib/$(CONF) -L$(MYRINET)/lib/intel_linux -L$(DASLIB)/lib
LDLIBS    := -llfc -ldas -lDpi -lLanaiDevice -lbfd -liberty

vpath %.a $(LFC)/lib/$(CONF)

latency: latency.o -llfc

.PHONY: clean
clean:
    $(RM) latency latency.o
```

Figure 3.1: Example Makefile

# Chapter 4

## Running LFC programs

### 4.1 Files

LFC uses several files at runtime.

- The **routing file** contains all routes between all cluster nodes. Users can force the library to use another routing file by means of the environment variable **ROUTES**. Under normal circumstances, you should never (have to) do this. Note that all users should use the same routing file to avoid network deadlocks.
- The **LANai control program** or **lcp** contains the executable program that will be run on the network interface. This file is usually named `lcp.lfc`. By default, it resides in the same directory as the matching LFC library. When loading the control program onto the network interface, the LFC library will look for the control program in this directory. Users can force the library to use another control program by means of the environment variable **LFC.LCP**.

### 4.2 Environment variables

All runtime options to LFC can also be passed by setting environment variables (see Section 2.1). In general any LFC option `param` which can be set using runtime flag `-lfc-param[=value]` can also be set using environment variable `LFC_PARAM` (i.e., in capitals) with the same value.

The only exceptions to this rule are `HOSTS` and `ROUTES`, which correspond to `-lfc-hosts` and `-lfc-routes` respectively. (The only reason for this difference is that on our system these environment variables can be used for non-LFC programs as well).

### 4.3 Running with prun

On our own platform, the Distributed ASCI Supercomputer (DAS), the easiest way to run programs that use LFC is to start all processes by means of the **prun(1)** program. Prun automatically assigns a rank to each process, selects hosts, and stores the names of the selected hosts in the **HOSTS** environment variable (see below). When the program is started using prun, the application's `main` function can pass its argument count and vector directly to `lfc_init`.

On platforms that don't have Prun installed, it should be possible to configure the native cluster management utilities to supply the required arguments and environment parameters to the application. This can be done using the `-lfc-hosts` parameter or `HOSTS` environment parameter mechanisms described in Section 2.1.

For the simple test programs included with the LFC distribution we include a simple example using plain `rsh` below.

## 4.4 Starting programs with rsh

LFC's latency test program can be started on nodes `node0` and `node1` using the following commands:

```
$ rsh -n node0 `pwd`/latency -lfc-hosts=node0,node1 -nsend 1000 &  
$ rsh -n node1 `pwd`/latency -lfc-hosts=node0,node1 -nsend 1000
```

The `rsh -n` option is used to specify that there is no terminal input (redirection from `/dev/null` achieves the same). The `-lfc-hosts` option specifies the participating nodes to the LFC library; the node's index in the host list determines its rank. The additional parameters `-nsend 100` are left to be processed by the application.

## 4.5 Debugging tips

You should be able to use any debugger to debug a program that uses LFC.

One problem that we have encountered during debugging (using **`gdb(1)`**) is that it is not possible to print the contents of data in LANai memory directly. With `gdb`, you can circumvent this problem by calling a function in the program being debugged. Within the program's context, it is possible to read LANai locations, so you can write a function that prints what you need to know (and then recompile, rerun, etc.).

## Chapter 5

# LFC Example Application

In this section we will discuss an example application: a simplified version of the standard LFC latency test program (file `test/latency/latency_simple.c` in the source distribution). The application performs an LFC latency test using two different ways to receive messages: using *polling* and using *interrupts*.

```
#include <stdio.h>
#include <stdlib.h>
#include <lfc.h>

#define UPCALLS_ALLOWED 1

static char *programe;
static unsigned nsend = 10000;
static volatile unsigned nreceived;
static unsigned last_nr_received;
static unsigned pktsize;
static unsigned msgsize = 16;

static void
usage(void)
{
    fprintf(stderr, "Usage: %s <options>\n"
               "\t[-nsend <nrsends>]      number roundtrips\n"
               "\t[-size <msgsize>]      message size\n", programe);
    exit(EXIT_FAILURE);
}

static void
command_line(int argc, char **argv)
{
    unsigned i;

    programe = argv[0];
    for (i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-nsend") == 0) {
            if (++i >= argc) usage();
            nsend = atoi(argv[i]);
        } else if (strcmp(argv[i], "-size") == 0) {
            if (++i >= argc) usage();
            msgsize = atoi(argv[i]);
        } else {
            break;
        }
    }
    if (argc - i != 0) usage();
}
```

Figure 5.1: Example application: argument processing

Figure 5.1 shows the include files needed by the program (in particular `lfc.h`), global definitions

```

int
lfc_upcall(unsigned src, void *data, unsigned size, lfc_upcall_flags_t flags)
{
    nreceived++;
    lfc_packet_free(data);      /* free packet ourselves */
    return 1;                  /* to indicate that we are freeing it */
}

static inline void
await_messages(unsigned n, int poll)
{
    while (nreceived - last_nr_received < n) {
        if (poll) lfc_poll();
    }
    last_nr_received += n;
}

```

Figure 5.2: Example application: receiving packets

and functions for argument processing. The application’s argument processing does not have to deal with LFC-specific options; these are all handled and removed by the call to `lfc_init` from `main` (as shown later in Figure 5.4). The application has two options `-nsend` and `-size` that may be used to change the number of iterations and message size respectively.

Figure 5.2 shows how received packets are handled by the application. As discussed earlier, when a packet arrives, the LFC layer can perform an *upcall* to the application’s `lfc_upcall` function in two ways:

- as a result of an explicit call to `lfc_poll` (in the example this happens in `await_messages`);
- from a SIGIO signal handler caused by an interrupt triggered by the network interface.

Receiving packets by means of a signal handler call is significantly more expensive than using polling, but may be convenient for handling unexpected requests while the application is busy computing, and regular polling is inconvenient, impossible (e.g., while in a mathematical library call) or when regular ineffective polling calls cause too much overhead. For many applications a combination of polling and interrupts is ideal.

In the example, `lfc_upcall` just increments a global counter when receiving the packet with contents `data` and size `size`. A more real-life application would obviously use the contents of the packet (e.g., copying it to a destination datastructure), possibly perform upcalls to additional software layers or send back a reply to the sender (the `src` argument to `lfc_upcall`). The final argument `flags` is not used in this case; it only needs to be inspected in applications that use a mixture of unicast and multicast communication.

Function `await_messages` simply spins until the required number of packets have arrived. It has two modes of operation: if parameter `poll` is true, it repeatedly calls `lfc_poll` to explicitly read packets away; otherwise it depends on `lfc_poll` being called by the SIGIO signal handler.

Figure 5.3 shows how messages are sent by the application. LFC has a maximum packet size that is reported by `lfc_packet_size`. Messages larger than that should be fragmented into separate packets; this is done by `send_large` in the example. Sending a packet in LFC is done by the following three steps:

Allocate a packet:	<code>void *pkt = lfc_send_alloc(upcalls_allowed);</code>
Copy the data:	<code>lfc_memcpy(pkt, databuffer, pktsize);</code>
Trigger transmission:	<code>lfc_ucast_launch(dest, pkt, pktsize, upcalls_allowed);</code>

Occasionally `lfc_send_alloc` and `lfc_ucast_launch` might have to wait until enough resources are available (e.g., because all send buffers are occupied). During this time they will poll for incoming messages. If at that point the application is able to process the resulting upcalls, it should pass 1 as `upcalls_allowed` parameter; otherwise upcalls will be delayed until later.

```

/* Fragment the message buffer (sendptr) */
static void
send_large(unsigned dest, void *sendptr, unsigned size)
{
    void *pkt;

    while (size > pktsize) {
        pkt = lfc_send_alloc(UPCALLS_ALLOWED);
        lfc_memcpy(pkt, sendptr, pktsize);
        lfc_ucast_launch(dest, pkt, pktsize, UPCALLS_ALLOWED);
        size -= pktsize;
        sendptr += pktsize;
    }
    pkt = lfc_send_alloc(UPCALLS_ALLOWED);
    lfc_memcpy(pkt, sendptr, size);
    lfc_ucast_launch(dest, pkt, size, UPCALLS_ALLOWED);
}

static void
do_test(void *send_buf, int size, int poll)
{
    unsigned nr_packets;
    double elapsed;
    unsigned dest;
    unsigned i;
    void *pkt;

    if (! poll) lfc_intr_enable();

    dest = (lfc_my_proc() == 0 && lfc_nr_procs() > 1) ? 1 : 0;
    nr_packets = (size + pktsize - 1) / pktsize;
    if (nr_packets == 0) nr_packets = 1;

    if (lfc_my_proc() == 0) {
        /* wait till receiver is ready too: */
        if (lfc_nr_procs() > 1) await_messages(1, poll);

        lfc_timer_start();
        for (i = 0; i < nsend; i++) {
            send_large(dest, send_buf, size);
            await_messages(nr_packets, poll);
        }
        lfc_timer_stop();

        elapsed = lfc_timer_elapsed();
        elapsed /= nsend;
        if (lfc_nr_procs() > 1) elapsed /= 2;
        printf("Size: %4u, polling: %d, latency: %3.1f microseconds\n",
              size, poll, elapsed);
    } else if (lfc_my_proc() == 1) {
        /* tell sender I'm ready: */
        pkt = lfc_send_alloc(1);
        lfc_ucast_launch(dest, pkt, 0, UPCALLS_ALLOWED);

        for (i = 0; i < nsend; i++) {
            await_messages(nr_packets, poll);
            send_large(dest, send_buf, size);
        }
    }

    if (! poll) lfc_intr_disable();
}

```

Figure 5.3: Example application: sending packets

```

int
main(int argc, char **argv)
{
    void *sendbuf;

    lfc_init(&argc, argv);
    command_line(argc, argv);
    lfc_start();

    pktsize = lfc_packet_size();
    if (msgsize > 0) {
        sendbuf = malloc(msgsize);
        if (sendbuf == NULL) {
            fprintf(stderr, "%s: out of memory\n", progname);
            exit(EXIT_FAILURE);
        }
    } else {
        sendbuf = NULL;
    }

    do_test(sendbuf, msgsize, 1);
    do_test(sendbuf, msgsize, 0);

    if (sendbuf != NULL) free(sendbuf);

    lfc_exit();
    return 0;
}

```

Figure 5.4: Example application: main routine

If the example application runs on two nodes, it first synchronizes the participating processes, and then performs a timed roundtrip test for a given number of times. If the application only runs on one node, it will send messages to itself without needing prior synchronization.

Note that while message delivery using interrupts is used, network interrupts are explicitly enabled using `lfc_intr_enable`. By default network interrupts are disabled in LFC.

Finally, Figure 5.4 shows the proper initialization and termination of LFC. Function `lfc_init` is called first. It takes care of processing any LFC-specific runtime options passed to the application. The remaining arguments are processed by the application itself in the call to `command_line`. The call to `lfc_start` really causes LFC to initialize the communication over Myrinet. When the function returns, all participating network interface have exchanged synchronization messages, and the LFC module is ready to be used.

The application then performs the two versions of the latency test (polling and interrupts) by calling `do_test`. Finally, the application properly terminates the LFC module by calling `lfc_exit`.

# Bibliography

- [1] R.A.F. Bhoedjang, T. Rühl, R.F.H. Hofman, K.G. Langendoen, H.E. Bal, and M.F. Kaashoek. Panda: A Portable Platform to Support Parallel Programming Languages. In *Proc. of the USENIX Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 213–226, San Diego, CA, September 1993.
- [2] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.
- [3] E.A. Brewer, F.T. Chong, L.T. Liu, S.D. Sharma, and J.D. Kubiatowicz. Remote Queues: Exposing Message Queues for Optimization and Atomicity. In *Proc. of the 1995 Symp. on Parallel Algorithms and Architectures*, pages 42–53, Santa Barbara, CA, July 1995.
- [4] J.J. Dongarra, S.W. Otto, M. Snir, and D.W. Walker. A Message Passing Standard for MPP and Workstations. *Communications of the ACM*, 39(7):84–90, July 1996.
- [5] K.L. Johnson, M.F. Kaashoek, and D.A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proc. of the 15th Symp. on Operating Systems Principles*, pages 213–226, Copper Mountain, CO, December 1995.
- [6] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 Usenix Conf.*, pages 115–131, San Francisco, CA, January 1994.
- [7] M. Lauria and A.A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997.
- [8] O. Maquelin, G.R. Gao, H.H.J. Hum, K.B. Theobald, and X. Tian. Polling Watchdog: Combining Polling and Interrupts for Efficient Message Handling. In *Proc. of the 23rd Int. Symp. on Computer Architecture*, pages 179–188, Philadelphia, PA, May 1996.
- [9] T. Rühl, H.E. Bal, R. Bhoedjang, K.G. Langendoen, and G. Benson. Experience with a Portability Layer for Implementing Parallel Programming Systems. In *The 1996 Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, pages 1477–1488, Sunnyvale, CA, August 1996.