Informatica™

Ultra Messaging (Version 6.17)

Quick Start Guide

Contents

Chapter 1

Introduction

This document provides step-by-step instructions on how to rapidly get started using Ultra Messaging high performance message streaming. It gets you started using our pre-compiled evaluation binaries to test performance, and then walks you through the steps of writing minimal source (publisher) and receiver (subscriber) applications.

For policies and procedures related to Ultra Messaging Technical Support, see `UM Support`.

See **UM Glossary** for Ultra Messaging terminology, abbreviations, and acronyms.

Chapter 2

Ultra Messaging Binary Quick Start

Ultra Messaging software consists of the documentation package and one or more binary packages. These instructions help get you started evaluating Ultra Messaging quickly. You can run pre-compiled commands from an Ultra Messaging binary distribution to get some quick performance numbers in your environment. A subsequent section guides you through creation of simple programs for local compilation.

A binary package consists of:

- Binary versions of source code examples, in the bin directory.
- Header files, in the include directory.
- Link libraries, in the lib directory.
- Shared libraries, in the lib directory on Unix for .so files or the bin directory on Windows for .dll files.

The Ultra Messaging code and documentation are copyrighted and confidential information owned by Informatica and are covered under the terms of our Software License Agreement or Non-Disclosure Agreement as appropriate. Use of this code and documentation without a valid Non-Disclosure Agreement or Software License Agreement with Informatica Corporation is strictly prohibited. If this code and documentation is being supplied under the terms of a Non Disclosure Agreement, all copies, in any form, must be returned or destroyed at the end of the evaluation period or as requested by Informatica.

2.1 Binary Quick Start on Microsoft Windows

The following steps assume that the windows package is installed on all test machines in the standard place:

`C:\Program Files\Informatica\`*rel-id*`\`*platform-id*

where *rel-id* is the release identifier and *platform-id* describes the ABI (Application Binary Interface). For example:

`C:\Program Files\Informatica\UMQ_6.17\Win2k-x86_64`

It is also assumed that the `\bin` directory is included in the windows PATH environment variable. This is needed so that the `.EXE` and `.DLL` files can be found.

1. Open a command prompt window on the machine you want to use for receiving messages and enter the command lbmrcv topic to start a receiver. Note that topic can be any string. You should see output that looks something like this:

```
1.006 secs.   0.0 Kmsgs/sec.   0.0 Kbps
1.019 secs.   0.0 Kmsgs/sec.   0.0 Kbps
1.010 secs.   0.0 Kmsgs/sec.   0.0 Kbps
```

A new line will be printed about once per second showing the elapsed time, messages received, and data received. As long as there are no sources yet running on topic, the number of messages received will continue to be zero.

2. Open a command prompt window on the machine you want to use for sending messages and enter the command lbmsrc topic to start sending messages. The receiver will automatically discover the source, at which time its output will change to something like this:

```
1.010 secs. 451.9 Kmsgs/sec.  90.4 Mbps
1.010 secs. 451.4 Kmsgs/sec.  90.3 Mbps
1.010 secs. 445.1 Kmsgs/sec.  89.0 Mbps
```

3. With no options given, the source will send 10,000,000 small (25 byte) messages. If you would like to test different size packets or number of packets sent, you can set options. Enter lbmsrc -h at the command line, you will get a list of options you can control.

4. Press `Ctrl-C` to kill the source or receiver.

2.2 Binary Quick Start on Unix

The following steps assume that the Unix package is installed on all test machines under a normal user account:

`/home/`*user-id*`/lbmeval/`*rel-id*`/`*platform-id*

where *rel-id* is the release identifier and *platform-id* describes the ABI (Application Binary Interface). For example:

`/home/jsmith/UMQ_6.17/Linux-glibc-2.17-x86_64`

It is also assumed that the `/bin` directory is included in the PATH environment variable and the /lib directory is in the appropriate loader library search path environment variable (e.g. LD_LIBRARY_PATH for Linux; see `http://bhami.com/rosetta.html` for equivalences in other flavors of Unix).

1. Open a command prompt window on the machine you want to use for receiving messages and enter the command lbmrcv topic to start a receiver. Note that topic can be any string. You should see output that looks something like this:

```
1.006 secs.   0.0 Kmsgs/sec.   0.0 Kbps
1.019 secs.   0.0 Kmsgs/sec.   0.0 Kbps
1.010 secs.   0.0 Kmsgs/sec.   0.0 Kbps
```

A new line will be printed about once per second showing the elapsed time, messages received, and data received. As long as there are no sources yet running on topic, the number of messages received will continue to be zero.

2. Open a command prompt window on the machine you want to use for sending messages and enter the command lbmsrc topic to start sending messages. The receiver will automatically discover the source, at which time its output will change to something like this:

```
1.010 secs. 451.9 Kmsgs/sec.  90.4 Mbps
1.010 secs. 451.4 Kmsgs/sec.  90.3 Mbps
1.010 secs. 445.1 Kmsgs/sec.  89.0 Mbps
```

3. With no options given, the source will send 10,000,000 small (25 byte) messages. If you would like to test different size packets or number of packets sent, you can set options. Enter lbmsrc -h at the command line, you will get a list of options you can control.

4. Press Ctrl-C to kill the source or receiver.

2.3 Binary Evaluation Notes

1. There are many other example programs available. See:
    - `C examples`
    - `Java examples`
    - `.NET examples`
2. For general information on measuring performance, see `LBM Reference Performance Tests` and `UMP Performance Testing Sample Applications`.
3. For information on testing the LBT-RM (multicast) transport, see the following UM Knowledgebase articles:
    - `Testing with LBT-RM`
    - `Interpreting LBT-RM Source Statistics`
    - `Tuning LBT-RM Questions`

Chapter 3

Ultra Messaging Programming Quick Start

The C programs below contain the minimum code and supporting material. Their purpose is to verify that the user's build and run-time environments are set up correctly. They also give a basic introduction to the Ultra Messaging API.

We also have equivalent Java and C# programs available in source form. See `MinSrc.java` and `MinRcv.java` or `MinSrc.cs` and `MinRcv.cs`. We also have an example of how application callbacks are coded in C++ programs. See `minrcv.cpp`.

(Most browsers let you right-click on a link and use the "save link target" function, or some variation.)

Note

> There are build .NET instructions in the initial comment blocks of MinSrc.cs and MinRcv.cs which are oriented towards the Windows development environment. For information related to .NET on Linux, see `UM .NET on Linux`.

Note that these programs do not allow the user to override any of the default configuration values. As a result, operation is fixed according to the normal LBM defaults; for example TCP is the transport protocol, topic resolution is performed using multicast, etc. See the `Ultra Messaging Configuration Guide`.

The Source Code Examples tab on the left panel provides a much richer set of source files that use a wide variety of features. However, those programs double as performance testing tools, so they tend to be more complex than just demonstrating the features. We recommend to first build and run these minimal examples.

This source code example is provided by Informatica for educational and evaluation purposes only.

Error handling in these programs is primitive. A production program would want to have better error handling, but for the purposes of a minimal example, it would just be a distraction. Also, a production program would want to support a configuration file to override default values on options.

3.1 Building Notes: C on Windows

When building on Windows, the following notes are applicable.

- Make sure the preprocessor variable "WIN32" is defined.
- Add the '`...\include\lbm`' (under the package install directory) as an additional include directory.
- Add lbm.lib and wsock32.lib as Object/library modules.
- Add the '`...\lib`' as an additional library path.

- The install procedure should already have added the LBM bin directory to the Windows PATH. This is necessary so that `lbm.dll` can be found when a program is run.

3.2 Building Notes: C on Linux

When building on Linux, the following notes are applicable.

- Sample build command:

  ```
  gcc -I$HOME/UMS_6.17/Linux-glibc-2.17-x86_64/include -I$HOME/UMS_6.17/
      Linux-glibc-2.17-x86_64/include/lbm \
      -L$HOME/UMS_6.17/Linux-glibc-2.17-x86_64/lib -llbm -pthread -lm -lrt -o
          minsrc minsrc.c
  ```

  (For programs that use more features, additional UM libraries might be required. See the contents of your installed UM "lib/" directory.)

- The appropriate library search path should be updated to include the installed UM "lib/" directory. For example:

  ```
  export LD_LIBRARY_PATH="$HOME/lbm/UMS_6.17/Linux-glibc-2.17-x86_64/
      lib:$LD_LIBRARY_PATH"
  ```

  Alternatively, the shared library can be copied from the LBM `lib/` directory to a directory which is already in your library search path.

Using other flavors of Unix can introduce differences. For example, not all Unixes use "LD_LIBRARY_PATH". For a great resource for learning how Unix flavors differ, see: `http://bhami.com/rosetta.html`.

3.3 Building Notes: Java

Starting in UM version 6.14, the necessary Java JAR files are included in the main package under "java".

See **Using UM Java on Unix** and **Using UM Java on Windows**.

For example: "UMP_6.17/java"

3.4 Building Notes: Dotnet

For .NET Core on Linux, the necessary .NET files are included in the main package under "bin/dotnet".

For example: "UMP_6.17/Linux-glibc-2.17-x86_64/bin/dotnet"

See `Using UM .NET on Linux`.

For .NET Framework on Windows, the necessary .NET files are included in the main package under "bin/dotnet".

For example: "UMP_6.17\Win2k-x86_64\bin\dotnet"

See `Using UM .NET on Windows`.

3.5 Minimal Ultra Messaging Source Implementation

This is a source code listing of a minimal source (sender) program. Examples also include equivalent Java and C# programs available in source form. See MinSrc.java and MinSrc.cs. (There are build .NET instructions in the initial comment blocks of MinSrc.cs and MinRcv.cs which are oriented towards the Windows development environment. For information related to .NET on Linux, see UM .NET on Linux.)

```
/*
"minsrc.c: minimal application that sends to a given topic.

 * (C) Copyright 2005,2025 Informatica Inc. All Rights Reserved.
 * Permission is granted to licensees to use
 * or alter this software for any purpose, including commercial applications,
 * according to the terms laid out in the Software License Agreement.

 * This source code example is provided by Informatica for educational
 * and evaluation purposes only.

 * THE SOFTWARE IS PROVIDED "AS IS" AND INFORMATICA DISCLAIMS ALL WARRANTIES
 * EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF
 * NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR
 * PURPOSE.  INFORMATICA DOES NOT WARRANT THAT USE OF THE SOFTWARE WILL BE
 * UNINTERRUPTED OR ERROR-FREE.  INFORMATICA SHALL NOT, UNDER ANY CIRCUMSTANCES, BE
 * LIABLE TO LICENSEE FOR LOST PROFITS, CONSEQUENTIAL, INCIDENTAL, SPECIAL OR
 * INDIRECT DAMAGES ARISING OUT OF OR RELATED TO THIS AGREEMENT OR THE
 * TRANSACTIONS CONTEMPLATED HEREUNDER, EVEN IF INFORMATICA HAS BEEN APPRISED OF
 * THE LIKELIHOOD OF SUCH DAMAGES.
 */

/* Explanatory notes referenced in this file are in the Quick Start guide. */

#include <stdio.h>

#if defined(_MSC_VER)
/* Windows-only includes */
#include <winsock2.h>
#define SLEEP(s) Sleep((s)*1000)
#else
/* Unix-only includes */
#include <stdlib.h>
#include <unistd.h>
#define SLEEP(s) sleep(s)
#endif

#include <lbm/lbm.h>

main()
{
  lbm_context_t *ctx;    /* pointer to context object */
  lbm_topic_t *topic;    /* pointer to topic object */
  lbm_src_t *src;        /* pointer to source (sender) object */
  int err;               /* return status of lbm functions (true=error) */

#if defined(_MSC_VER)
  /* windows-specific code */
  WSADATA wsadata;
  int wsStat = WSAStartup(MAKEWORD(2,2), &wsadata);
  if (wsStat != 0) {
    printf("line %d: wsStat=%d\n",__LINE__,wsStat); exit(1);
  }
#endif

  err = lbm_context_create(&ctx, NULL, NULL, NULL);  /* See Note #1 */
  if (err) {
    printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);
  }

  err = lbm_src_topic_alloc(&topic, ctx, "Greeting", NULL);  /* See Note #2 */
  if (err) {
    printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);
  }

  err = lbm_src_create(&src, ctx, topic, NULL, NULL, NULL);  /* See Note #3 */
  if (err) {
    printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);
  }

  SLEEP(1);  /* See Note #4 */

  err = lbm_src_send(src, "Hello!", 6, LBM_MSG_FLUSH | LBM_SRC_BLOCK);  /* See Note #5 */
  if (err) {
    printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);
  }
  SLEEP(2);  /* See Note #6 */

  /* Finished all sending to this topic, delete the source object. */

  err = lbm_src_delete(src);

  /* Do not need to delete the topic object - LBM keeps track of topic
   * objects and deletes them as-needed.  */

  /* Finished with all LBM functions, delete the context object. */
  err = lbm_context_delete(ctx);

#if defined(_MSC_VER)
  WSACleanup();
#endif
}  /* main */
```

Notes:

1. Create a context object. A context is an environment in which LBM functions. Note that the first parameter is a pointer to a pointer variable; **lbm_context_create()** writes the pointer to the context object into "ctx". Also, by passing NULL to the context attribute parameter, the default option values are used. For most applications only a single context is required regardless of how many sources and receivers are created.

2. Allocate a topic object. A topic object is little more than a string (the topic name). During operation, LBM keeps some state information in the topic object as well. The topic is bound to the containing context, and will also be bound to a source object. Note that the first parameter is a pointer to a pointer variable; **lbm_↩ src_topic_alloc()** writes the pointer to the topic object into "topic". Also, by passing NULL to the source topic attribute, the default option values are used. The string "Greeting" is the topic string.

3. Create the source object. A source object is used to send messages. It must be bound to a topic. Note that the first parameter is a pointer to a pointer variable; **lbm_src_create()** writes the pointer to the source object to into "src". Use of the third and fourth parameters is optional but recommended in a production program - some source events can be important to the application. The last parameter is an optional event queue (not used in this example).

4. Need to wait for receivers to find us before first send. There are other ways to accomplish this, but sleep is easy. See `Avoiding or Minimizing Delay Before Sending` for details.

5. Send a message to the "Greeting" topic. The flags make sure the call to **lbm_src_send()** doesn't return until the message is sent. Note that while this ensures low latency, flushing every message carries a heavy efficiency cost. See **Intelligent Batching** for more information.

6. For some transport types (mostly UDP-based), a short delay before deleting the source is advisable. Even though the message is sent, there may have been packet loss, and some transports need a bit of time to request re-transmission. Also, if the above **lbm_src_send()** call didn't include the flush, some time might also be needed to empty the batching buffer.

3.6 Minimal Ultra Messaging Receiver Implementation

This is a source code listing of a minimal receiver program. Examples also include equivalent Java, C#, and C++ programs available in source form. See `MinRcv.java`, `MinRcv.cs`, and `minrcv.cpp`. (There are build .N↩ ET instructions in the initial comment blocks of MinSrc.cs and MinRcv.cs which are oriented towards the Windows development environment. For information related to .NET on Linux, see `UM .NET on Linux`.)

```
/*
"minrcv.c: minimal application that receives messages from a given topic.

 * (C) Copyright 2005,2025 Informatica Inc. All Rights Reserved.
 * Permission is granted to licensees to use
 * or alter this software for any purpose, including commercial applications,
 * according to the terms laid out in the Software License Agreement.
 * This source code example is provided by Informatica for educational
 * and evaluation purposes only.

 * THE SOFTWARE IS PROVIDED "AS IS" AND INFORMATICA DISCLAIMS ALL WARRANTIES
 * EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF
 * NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR
 * PURPOSE.  INFORMATICA DOES NOT WARRANT THAT USE OF THE SOFTWARE WILL BE
 * UNINTERRUPTED OR ERROR-FREE.  INFORMATICA SHALL NOT, UNDER ANY CIRCUMSTANCES, BE
 * LIABLE TO LICENSEE FOR LOST PROFITS, CONSEQUENTIAL, INCIDENTAL, SPECIAL OR
 * INDIRECT DAMAGES ARISING OUT OF OR RELATED TO THIS AGREEMENT OR THE
 * TRANSACTIONS CONTEMPLATED HEREUNDER, EVEN IF INFORMATICA HAS BEEN APPRISED OF
 * THE LIKELIHOOD OF SUCH DAMAGES.
 */

/* Explanatory notes referenced in this file are in the Quick Start guide. */

#include <stdio.h>

#if defined(_MSC_VER)
/* Windows-only includes */
#include <winsock2.h>
#define SLEEP(s) Sleep((s)*1000)
#else
/* Unix-only includes */
#include <stdlib.h>
#include <unistd.h>
#define SLEEP(s) sleep(s)
#endif

#include <lbm/lbm.h>

/*
 * A global variable is used to communicate from the receiver callback to
 * the main application thread.
 */
int msgs_rcvd = 0;

int app_rcv_callback(lbm_rcv_t *rcv, lbm_msg_t *msg, void *clientd)  /* See Note #1 */
{
  /* There are several different events that can cause the receiver callback
   * to be called.  Decode the event that caused this.  */
  switch (msg->type)
  {
  case LBM_MSG_DATA:    /* a received message */
    printf("Received %d bytes on topic %s: '%.*s'\n",  /* See note #2 */
           msg->len, msg->topic_name, msg->len, msg->data);

    /* Tell main thread that we've received our message. */
    ++ msgs_rcvd;
    break;

  case LBM_MSG_BOS:
    printf("[%s][%s], Beginning of Transport Session\n", msg->topic_name, msg->
      source);
    break;

  case LBM_MSG_EOS:
    printf("[%s][%s], End of Transport Session\n", msg->topic_name, msg->source);
    break;

  default:    /* unexpected receiver event */
    printf( "Unhandled receiver event [%d] from source [%s] with topic [%s]. Refer to
       https://ultramessaging.github.io/currdoc/doc/example/index.html#unhandledcevents for a detailed description.\n", msg->
      type, msg->source, msg->topic_name);
    break;
  }  /* switch msg->type */

  return 0;
}  /* app_rcv_callback */


main()
{
  lbm_context_t *ctx;    /* pointer to context object */
  lbm_topic_t *topic;    /* pointer to topic object */
  lbm_rcv_t *rcv;        /* pointer to receiver object */
  int err;               /* return status of lbm functions (non-zero=error) */

#if defined(_MSC_VER)
  /* windows-specific code */
  WSADATA wsadata;
  int wsStat = WSAStartup(MAKEWORD(2,2), &wsadata);
  if (wsStat != 0) {
    printf("line %d: wsStat=%d\n",__LINE__,wsStat); exit(1);
  }
#endif
```

```
  err = lbm_context_create(&ctx, NULL, NULL, NULL);  /* See note #3 */
  if (err) {
    printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);
  }

  err = lbm_rcv_topic_lookup(&topic, ctx, "Greeting", NULL);  /* See note #4 */
  if (err) {
    printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);
  }

  err = lbm_rcv_create(&rcv, ctx, topic, app_rcv_callback, NULL, NULL);  /* See note #5 */
  if (err) {
    printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);
  }

  while (msgs_rcvd == 0) {
    SLEEP(1);
  }

  /* Finished all receiving from this topic, delete the receiver object. */
  err = lbm_rcv_delete(rcv);
  if (err) {
    printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);
  }

  /* Do not need to delete the topic object - LBM keeps track of topic
   * objects and deletes them as-needed.  */

  /* Finished with all LBM functions, delete the context object. */
  err = lbm_context_delete(ctx);
  if (err) {
    printf("line %d: %s\n", __LINE__, lbm_errmsg()); exit(1);
  }

#if defined(_MSC_VER)
  WSACleanup();
#endif
}  /* main */
```

Notes:

1. LBM passes received messages to the application by means of a callback. I.e. the LBM context thread reads the network socket, performs its higher-level protocol functions, and then calls an application-level function that was set up during initialization. This callback function has some severe limitations placed upon it. It must execute very quickly; any potentially blocking calls it might make will interfere with the proper execution of the LBM context thread. One common desire is for the receive function to send an LBM message (via **lbm_src_send()**), however this has the potential to produce a deadlock condition. If it is desired for the receive callback function to call LBM or other potentially blocking functions, it is strongly advised to make use of an event queue, which causes the callback to be executed from an application thread. See the example tool `lbmrcvq.c` for an example of using a receiver event queue.

2. Note - printf can block, which is normally a bad idea for a callback (unless an event queue is being used). However, for this minimal application, only one message is expected.

3. Create a context object. A context is an environment in which LBM functions. Note that the first parameter is a pointer to a pointer variable; **lbm_context_create()** writes the pointer to the context object into "ctx". Also, by passing NULL to the context attribute parameter, the default option values are used. For most applications only a single context is required regardless of how many sources and receivers are created.

4. Lookup a topic object. A topic object is little more than a string (the topic name). During operation, LBM keeps some state information in the topic object as well. The topic is bound to the containing context, and will also be bound to a receiver object. Note that the first parameter is a pointer to a pointer variable; **lbm_rcv_topic_lookup()** writes the pointer to the topic object into "topic". Also, by passing NULL to the source topic attribute, the default option values are used. The string "Greeting" is the topic string.

5. Create the receiver object and bind it to a topic. Note that the first parameter is a pointer to a pointer variable; **lbm_rcv_create()** writes the pointer to the source object to into "rcv". The second and third parameters are the function and application data pointers. When a message is received, the function is called with the data pointer passed in as its last parameter. The last parameter is an optional event queue (not used in this example).

Chapter 4

Unicast-Only Operation

One of Ultra Messaging's great strengths is its use of the network hardware and protocols to achieve very high performance and scalability. By default, Ultra Messaging uses multicast for topic resolution so that data sources and receivers can find each other.

In general, we recommend the use of multicast whenever possible because it provides the best performance and scalability. However, we recognize that it is not always possible to provide multicast connectivity between the machines. For those cases, we support unicast-only operation.

There are two parts to unicast-only operation. One is to use a unicast form of transport. Ultra Messaging uses TCP by default for transport, but LBT-RU is also available and has some performance advantages. For initial "quick start" testing, we recommend not changing the default TCP.

The second part of unicast-only operation is to use the lbmrd (UM Resolver Daemon) for topic resolution. (NOTE: this does not route message data through a daemon. This is a helper process that lets data sources and receivers find each other.) To enable unicast topic resolution, do the following:

1. Choose one machine to host the resolver daemon and enter the lbmrd command (use lbmrd -h for full instructions).
2. Create a Ultra Messaging configuration file containing:
   `context resolver_unicast_daemon [Iface[:Src_Port]->]IP:Dest_Port` where:
   - Iface is the interface to use (previously set via resolver_unicast_interface).
   - Src_Port is the source port to use (previously resolver_unicast_port).
   - IP is the resolver daemon's IP address (previously resolver_unicast_address).
   - Dest_Port is the resolver daemon's UDP port (previously resolver_unicast_destination_port).

   For more information, see **resolver_unicast_daemon (context)**.
3. Run the test applications with the option -c filename where filename is the configuration file created in step 2.

Due to the fact that the minimal source programs presented in this document (minsrc.c and minrcv.c) do not allow the use of a configuration file, it is not possible to configure them for unicast-only operation. If multicast operation is not possible on your network, then please use the binary test programs (described in Ultra Messaging Binary Quick Start) which which do allow unicast-only configuration.

Chapter 5

Starting Ultra Messaging Daemons

5.1 Starting Ultra Messaging Dynamic Routing Option

The **DRO** bridges disjoint topic resolution domains by forwarding multicast and/or unicast topic resolution traffic. Before installing and starting a DRO daemon (tnwgd) clear objectives and proper planning are very important. Approach this planning with the consideration that a DRO condenses your network into a single process. You must be clear about the traffic you expect to forward through a DRO. The following highlights some other specifics.

- Know your Topic Resolution Domains. See **Topic Resolution Domains**.
- Consider the size and quality of network paths into and out of your DROs. For example, DROs cannot efficiently forward messages from a 1GB path to a 100MB path.
- Fully examine any use of Late Join, taking care to configure retransmission options correctly.

5.2 Persistent Store Daemon

The daemon, umestored, provides persistent store services. To start umestored, perform the following steps:

1. Create the cache and state directories.
   ```
   $ mkdir umestored-cache ; mkdir umestored-state
   ```
2. Create a simple umestored XML configuration file or use a sample configuration, `ume-example-config.↩ xml`. For full details on Persistence, see `Ultra Messaging Guide for Persistence`.
3. Start the daemon.
   ```
   $ umestored config.xml
   ```

See also **Umestored Man Page**.

Chapter 6

Next Steps

The `Ultra Messaging Concepts Guide` introduces the important fundamental concepts you will need to design and code your applications.

See `C example` programs for descriptions and source files for the binary tools used for evaluation and performance measurements. They can be a valuable resource for seeing how various features can be implemented. Also available are `Java` and `.NET` example programs.

The `Ultra Messaging Guide for Persistence` contains concepts, tutorials and configuration information for UMP, along with discussions about designing persistent applications. It also contains detailed configuration information for the umestored daemon.

The `Ultra Messaging Guide for Queuing` contains concepts, tutorials and configuration information for UMQ, along with discussions about designing queuing applications.

The `C API`, `Java API`, `.NET API` documentation provide reference guides for the Ultra Messaging libraries.