THE LINEAR BUFFER AND ITS ROLE IN THE WSR-88D OPEN SYSTEM RPG

by Zhongqi Jing[1] and Mike Jain[2]

[1]Cooperative Institute for Mesoscale Meteorological Studies,
University of Oklahoma, Norman, Oklahoma

[2]NOAA/ERL/National Severe Storms Laboratory

1. INTRODUCTION

The WSR-88D Open System Radar Product Generator (ORPG) (Jain, et al, 1997) is a distributed system composed of a number of processing tasks each of which is a stand-alone executable program performing a part of the ORPG functionality. Each task performs a well-defined, specialized and relatively simple function. The design goal is a well-partitioned system in which each task is relatively simple and, at the same time, interactions among the tasks are minimized.

The ORPG is "loosely coupled" in the sense that each task has minimum knowledge about how it interacts with other processes. There is no designed connection among ORPG tasks. Each task, for example, looks for its inputs without concerning who generates them. It produces outputs without caring about who (if any) will use them. Each task can be independently designed, implemented, tested, documented and maintained. The design and implementation details of each task are less important for the system level design and often left to its developer as long as the task performs the functionality correctly with reasonable efficiency. They can be implemented using any language and tools that are considered to be the best suitable.

To support the loosely coupled ORPG architecture, a software module called the Linear Buffer (LB) has been developed. The LB is the software component that supports inter-process communication (IPC), message passing, data buffering, data storage and configuration information management within ORPG. Although the LB is initially developed as part of the ORPG project, it is not ORPG specific. While ORPG development approaches its completion, the LB has evolved into a useful middleware tool for building extensible and portable real-time meteorological data processing systems. In the following we briefly describe the functionality and application programming interface (API) of LB and how it is used as the core software infrastructure in developing ORPG. Comparisons with other alternative technologies will be discussed.

2. THE LINEAR BUFFER

The LB is designed for message communication among multiple processes (A process is a running instance of a task). A message is a byte string of given length. Processes send messages to an LB and other processes can then receive those messages by reading them from the LB. An LB, like a file, is identified by a name (path). When using LB for IPC, processes access LBs just like reading/writing files. There is no connection (such as socket or pipe connections) between communicating processes. In a typical LB application, a task calls "LB_open" to open an LB and gets an LB descriptor. It then calls LB_read or LB_write to read or write messages.

An LB can hold multiple messages. Multiple writers and readers can access an LB simultaneously. Three functional types of LBs are supported - sequential LB, replaceable LB and message pool LB.

A sequential LB functions as a message queue in which the messages are stored sequentially. When a new message is written, it is put at the end of the queue. When the message in the front of the queue is no longer needed, it can be removed from the queue (first-in-first-out). The LB keeps track of all messages such that, from a reader's point of view, all messages are stored in a linear buffer (rather than a circular buffer) and every message, existing or removed, can be unambiguously identified.

In a replaceable LB, messages can be replaced (i.e. updated) with a new version. A message pool type LB can be used as a message pool in which message can be stored, retrieved, updated and deleted randomly.

A message in the LB has three attributes: The length of the message, the message ID (an unsigned integer) and a tag of up to 32 bits. When a message is written to the LB, the writer provides the message length and optionally assigns a message ID and a tag for the new message. The message ID can be used for random message access. The tag is not used by the LB, but an attribute attached to a message that the application can use for various purposes (e.g. as an object identifier, message processing info carrier and so on). It can be set, changed and read by any application (a task that uses LB) anytime in the message's life time. Other than the three attributes, the LB treats a message as a byte string and does not understand and process it.

The LB is a reliable message passing channel supporting certain ACID (Atomicity, Consistence, Independence and Durability) features. No messages written to an LB are ever lost or corrupted as long as the operating system (OS) is running. A reader or a writer can start, terminate or crash at any time without destroying any LB that is been accessed, leaving corrupted messages in it or requiring any exception handling in any other accessing processes. If a writer is killed before it completes a message update, the message being updated keeps intact. The LB supports concurrent access of multiple readers and writers, in which case each reading and writing call is an atomic procedure as seen from the applications. A reader will never see any partially written messages. In LB, messages are stored either in a file on in shared memory. In both cases the

Corresponding author address: Zhongqi Jing, National Severe Storms Laboratory, 1313 Halley Circle, Norman, OK 73069; e-mail <jing@nssl.noaa.gov>

messages will survive process termination. Messages in a file LB will survive host shutdown and reboot.

The LB supports network transparent remote access. All LB functions work for LBs located locally as well as on remote hosts. To specify a remote LB, we need to give both host name and the LB name. For example "rhost:/users/orpg/data/basedata.lb" specifies an LB named "/users/orpg/data/basedata.lb" on host "rhost". Data compression and progress reporting is supported when accessing through low bandwidth connections.

The LB provides memory leak protection. The LB contents are protected from being corrupted by memory leaks in the applications that open the LB.

Message locking is supported by LB for implementing transaction type of functions. For example, if two processes must update the same LB message, they can, with LB message locking, lock each other from simultaneous accesses.

The LB supports arbitrary message sizes. Very large messages can be stored in a file LB. The maximum number of messages, however, is determined when an LB is first created. Both sequential and random (through message ID) accesses are supported.

The LB provides notification service. An application can register callback functions in an LB such that they are called when specified events occur. Supported events include message update, message expiration (deletion) and application defined events. An application defined event is identified by an event ID (a number) and generated (posted) by any application process. An application defined event can carry a message of any size with it. Application defined events are not related to any LBs. The LB notification service is reliable, efficient and non-blocking, by which we mean that any slow event receiver will not block any event senders.

The LB is an efficient message passing facility. Messages are stored in shared memory or memory mapped files and, for local LB, accessed directly without passing through an agent. LB uses balanced tree data structures to manage free segments in the LB message space. Binary search is used, whenever possible, to randomly locate the messages. Access lock is not invoked between readers and the writer. It is only used between multiple writers. An efficient indexed linked list table is used in managing notification registrations.

Multiple threaded programming is supported except notification functions (Only one thread should receive incoming events).

The LB is available through a shared library (libinfr.so) and a server called "rssd". They also contain, in addition to the LB functions, some other support functions for distributed application development. The currently supported platforms are Solaris, HPUX and Linux. Client side support on Windows is under development.

The LB uses a simple Application Programming Interface (API), which is described in the next section. The "rssd" server uses a simple configuration file listing all participating hosts and directories that are permitted to access. There are no other system administration works involved. Several command line tools have been developed for assisting system development and testing. They can be used for creating or destroying LBs, listing LB information, viewing and writing messages, posting and monitoring events and others.

The sequential LB is often called message queue. Refer to "MQSeries: An Introduction to Messaging and Queuing" (http://www.software.ibm.com/ts/mqseries/library/horaa101) for discussions on why and how to build distributed systems using message queues.

3. LB API AND SIMPLE EXAMPLES

The following is a list of basic LB functions:

int **LB_open** (char *lb_name, int flags, LB_attr *attr) - Creates a new LB or opens an existing LB for access. It returns an LB descriptor (lbd) for later LB access.

int **LB_close** (int lbd) - Closes an open LB.

int **LB_remove** (char *lb_name) - Removes an LB.

int **LB_read** (int lbd, void *buf, int buflen, LB_id_t id) - Reads a message from an LB.

int **LB_write** (int lbd, char *message, int length, LB_id_t id) - Writes a message to an LB.

int **LB_seek** (int lbd, int offset, LB_id_t id, LB_info *info) - Moves read pointer for sequential access control.

int **LB_lock** (int lbd, int command, LB_id_t id) - Locks a message in an LB.

int **LB_UN_register** (int fd, LB_id_t msgid, void (*notify_func)()) - Registers an LB update notification callback function.

int **LB_AN_register** (LB_id_t event, void (*notify_func)()) - Registers an application defined notification callback function.

int **LB_AN_post** (LB_id_t event, char *msg, int msg_len) - Posts an application defined notification.

There are about a dozen of other LB functions that allow the applications to retrieve info about an LB and its messages, set/reset tags, control notification delivery and delete messages. We do not have space to list them in this report. The following code writes a message to an LB on a remote host.

```
#include <stdio.h>
#include <infr.h>
#include <rss_lb.h>
Int main () {
   int fd, ret;
   char *my_message = “This is my message”;
   fd = LB_open ("rhost:/users/orpg/data/my_data.lb", LB_WRITE, NULL);
   if (fd < 0) {            /* check error */
      fprintf (stderr, "LB_open failed (return %d)\n", fd);
      exit (1);
   }
   ret = LB_write (fd, my_message, strlen (my_message) + 1, LB_ANY);
   if (ret < 0) {           /* check error */
      fprintf (stderr, "LB_write failed (return %d)\n", ret);
      exit (1);
   }
   exit (0);
}
```

The following example opens the same LB, registers for message update events and waits for any event to come. It then reads any message that is written to the LB.

```
#include <stdio.h>
#include <infr.h>
#include <rss_lb.h>
#define BUF_SIZE    128
static void my_callback (int fd, LB_id_t id, int msg_len, void *arg) {
   char buf[BUF_SIZE];
   int ret = LB_read (fd, buf, BUF_SIZE, id);
   if (ret < 0)             /* check error */
      fprintf (stderr, "LB_read failed (return %d)\n", ret);
   else
      fprintf (stderr, "read a message (id %d): %s\n", id, buf);
   return;
}
```

```
int main () {
   int fd, ret;
   fd = LB_open ("rhost:/users/orpg/data/my_data.lb", LB_READ, NULL);
   if (fd < 0) {            /* check error */
      fprintf (stderr, "LB_open failed (return %d)\n", fd);
      exit (1);
   }
   ret = LB_UN_register (fd, LB_ANY, my_callback);
   if (ret < 0) {           /* check error */
      fprintf (stderr, "LB_UN_register failed (return %d)\n", ret);
      exit (1);
   }
   while (1)                /* wait for events to come */
      sleep (10);           /* will be wake up by any event */
}
```

4. HOW IS LB USED IN ORPG

ORPG product generation part uses a data driven model. Radar data processing algorithms and product generation procedures are implemented by many tasks each of which generates a set of related products or intermediate products. All meteorological data (RDA base data, RPG base data, dealiased base data, intermediate products, final products and others) are stored in separate LBs. An algorithm/product task knows its input and output LBs. It waits until all inputs are available for a given radial, elevation or volume. It, then, processes the data, writes its results into its output LBs and go back to wait for the next set of inputs. The input data ready is detected by either LB update notification or polling (periodically perform LB_read). Polling, which is more efficient for high frequency inputs, is used for base data. The meteorological data LBs are sequential (i.e. message queues) and act as a data buffer between processes. They have typically single writer and multiple readers. For example, about two dozens of tasks read dealiased base data as input. Each data consumer can have its own data reading rate without affecting others. The base data writer will automatically remove the oldest message in the queue when the queue is full and a new one has to be written. This means that no down stream process will block the data flow. If some task does not keep pace with input, it will loose data but not anybody else. This makes the system more robust in case of running less perfect algorithm software or on hardware with limited resources.

Communications manager and data base server tasks use a client/server model, in which case a server sets up a request LB to receive client requests. Response LBs are set for putting response messages by the servers. A separate sequential response LB is set for each communications manager client because sufficiently large buffering is needed to hold unsolicited responses (incoming data and server events). Request/response LBs provide necessary buffering such that servers can run asynchronously in terms of their clients. The ORPG data base server uses a simple one-response-per-request model. Only one LB is needed for receiving requests from multiple clients, which is a case of multiple writers. One replaceable LB is set for storing responses for all clients. LB update notification is used for both request and response.

Other ORPG tasks, RDA_control, the product schedulers, the product server and the HCI (human-computer interface) use more generic data passing structures. Each of them has multiple inputs and outputs. A number of LBs are set for all data passing needs. Application defined events are used for various ORPG events such as volume start, a piece of adaptation data updated, a new product generated, an ORPG exception (such as loading shed) occurred and so on. ORPG external actions are performed in a single place in order to simplify synchronization and improve modularity. For example, all RDA commands are issued from RDA_control. When the operator sends a RDA control command from HCI, a message is generated by the HCI and written to the RDA control command LB. RDA_control then reads commands from the queue and process them sequentially. If an acknowledgment is needed, an event is posted.

An application defined event can also be used as a message queue. Each event posting carries a message and multiple senders and receivers are supported. How do we, then, make a choice between using an application defined event or an LB? An application defined event does not need creation and clean-up. It is very convenient to use (post/register). However, it does not support persistent data storage and has much limited buffering capability. An event is a one-time notification and does not provide state information. For example, if we send RDA control commands via an application defined event, any command before RDA_control started will be lost. ORPG is designed such that all its operational processes can be started without any specific sequence and any of them can be terminated and restarted at anytime without shutdown other processes. Using events, however, for command acknowledgments do not have the same problem.

The ORPG uses message pool type LBs for its build-in data bases. Currently implemented data bases are the product data base and the product user data base. When a product is generated, it is written into the product data base LB. A unique message ID is returned from the LB that can be used later for accessing, updating and deleting the message. A task called data base manager is developed to provide data base query service for all ORPG data bases. For each data base, it registers for the update event of any message in the data base LB in order to build and update index tables for the data base. For product data base, the data base manager maintains the data base by deleting products in the data base LB at scheduled times. When a task in ORPG needs to query a data base, it sends a request to the data base server and the server returns the message IDs found in responding to the request. The message pool type LB manages large number of randomly stored messages of arbitrary sizes efficiently in terms of searching speed and space utilization .

ORPG configuration data, internal tables, adaptation data and status information are stored in LBs of replaceable type. A message in a replaceable LB can be updated anytime with a different size. A replaceable LB is useful for storing information that can be updated in real-time while only the latest version of it is of interest. With the LB notification, it supports the publish/subscribe type of messaging model. For example, a piece of adaptation data can be stored as a message in a replaceable LB. Tasks that need this data read it when they start. They also register for the update notification of the data. Later, when the operator updates this adaptation data, all interested processes are notified and they all re-read the data and update their own copy. Related pieces of adaptation data are stored in the same LB for easier management. ORPG status information are published in a similar way. The publish/subscribe mechanism is implemented here with the loosely couple model. The publisher (message writer) does not care who will use it and any process in the system can access any published data without contacting with any other processes (e.g. the publisher, the messaging server, or a system manager). If multiple processes

are going to update the same piece of data, unlike the message queue model, a locking mechanism must be used. LB_lock is used for this purpose. The following procedure can be used: Lock the message to be updated; Read the message; Update the message; Write back the message; And release the lock.

Each ORPG task stores its log/error messages in an LB. Using LB for log/error reporting, we can store binary messages of variable sizes since the LB provides the message boundary support. The log file size is automatically controlled by the LB and, thus, no clean-up work is needed and the chance that disk space is exhausted by a buggy process generating uncontrolled number of messages is eliminated.

The LB isolates ORPG applications from direct access to many of the OS services. Most ORPG applications only need ANSI C library functions. ORPG tasks don't directly use signals, networking services, sockets and pipes. Files are used only for certain configuration data that are read-only and never change while ORPG is running. This makes ORPG more portable.

The LB provides only basic message communication/storage services. Messages managed by LB have only a few attributes. This makes the LB easy to customize, configure, use and administrate. For example, the LB does not support data format conversion when messages are passed between machines of different byte orders and data representations. This, however, makes the LB independent of application message formats and eliminates the need of telling the LB all application data formats. If we only need to support homogeneous environment, the applications will be easier to develop and run more efficiently. In cases we do need to support heterogeneous environment, we can build another software level on top of LB to take care of the data format issues. In an object oriented system, objects can be stored and passed around by using the LB. Object oriented data access/communication interface can be built on top of the LB.

The ORPG has many application level libraries built on top of LB. This further modularizes the software and simplifies task development. For example, most of ORPG tasks access data through a library module called ORPGDA (OPRG data access) where data are accessed through logical data name instead of physical LB locations. This data location transparency makes dynamic resource (processes and data stores) relocation possible. Most of ORPG internal data accesses, such as base data, products, product attributes and so on, are through library modules. There are library functions that write products, access data bases, report status, set alarms and many other services.

5. COMPARISONS WITH OTHER TECHNIQUES

Files are sometimes used for message passing or IPC. One task creates a file and puts data in it. Then another task opens the file and reads data from it. NFS (Network File System) files may be used if remote access is required. Using plain files for IPC is simple and convenient especially if concurrence control and efficiency are not concerns. File creation and removal are slow processes. NFS does not work well for IPC because its cashing mechanism is not designed for IPC. As a matter of fact, the LB is an extension of the idea of using files for IPC.

Sockets and pipes are OS services for IPC. They are very efficient. They use a connection based, process-to-process IPC model and, thus, do not support multi-readers/writers accesses. Persistent data storage is not supported and the data buffering capability is often quite limited. Connection maintaining, exception handing and notification service have to be handled by the applications. Sockets and pipes are convenient tools for building simple client/server applications in which a client makes a connection to a well-known server, sends requests to the server, receives the response from the server and completes the transaction. To build a complex software system like ORPG, where every task plays both roles of client and server and data often need to be shared by multiple tasks with concurrent accesses, using sockets and pipes would make all ORPG tasks more complicated and difficult to develop. The LB implements more generic IPC models, hides complex code that uses the raw OS services and helps developing loosely couple systems.

Many operating systems support Remote Procedure Call (RPC) as an IPC mechanism. RPC often provides support for machine dependent data format conversions (byte order, floating point number representation and so on). RPC, however, uses a different distributed system model than message passing based loosely coupled model.

Commercially available IPC/message passing software products are often called middleware. CORBA is the object oriented extension of the RPC. For supporting message passing based distributed systems, there are many products available on the market. Examples are MQSeries from IBM and MSMQ from Microsoft. MSMQ is a part of the Windows operating system. It is not readily available on other platforms. MQSeries is available on many different platforms. MQSeries's functionality is similar, in many regards, to that of the sequential LB. It, however, does not support the functionality of the replaceable and message pool types of LBs. We don't know if it supports arbitrary message sizes and data compression. It has very limited event notification support. An important design consideration for ORPG is the data passing efficiency because the radar base data is a relatively high band width real-time data flow. More than two dozens of processes must read/write base data concurrently. We have not had an opportunity to find out whether these products can support such data rate demand.

6. ACKNOWLEDGMENTS

ORPG developers, Steve Smith, David Priegnitz, Hoyt Burchan, Arlis Dodson, Eddie Forren and others made numerous contributions to the ORPG software design.

7. REFERENCES

Jain, M., Z. Jing, A. Zahrai, A. Dodson, H. Burchan, D. Priegnitz, S. Smith: Software Architecture of the NEXRAD Open Systems Radar Product Generator (ORPG), Preprints 13 -th IIPS, Longbeach, CA, Amer. Meteor. Soc., paper 8.4.