How To Write A Converse Load Balancer

Terry L. Wilmarth

June 22, 2000

1 Introduction

This manual details how to write your own load balancer in Converse. A Converse load balancer can be used by any Converse program, but also serves as the balancer of Charm++ chare creation messages. Specifically, to use a load balancer, you would pass messages to CldEnqueue rather than directly to the scheduler. This is the default behavior with chare creation message in Charm++. Thus, the primary provision of a new load balancer is an implementation of the CldEnqueue function.

2 Existing Load Balancers and Provided Utilities

Throughout this manual, we will occasionally refer to the source code of two provided load balancers, the random initial placement load balancer (`cldb.rand.c`) and the graph-based load balancer (`cldb.graph.c`). The functioning of these balancers will be described in detail later.

In addition, a special utility is provided that allows us to add and remove load-balanced messages from the scheduler's queue. The source code for this is available in `cldb.c`. The usage of this utility will also be described here in detail.

3 A Sample Load Balancer

This manual steps through the design of a load balancer using an example which we will call HELP! The HELP! load balancer has each processor periodically send half of its load to its neighbor in a ring. Specifically, for N processors, processor K will send approximately half of its load to (K+1)%N, every 100 milliseconds (this is an example only; we leave the genius approaches up to you).

4 CldEnqueue

The prototype for the **CldEnqueue** function is as follows:

`void` **CldEnqueue**`(int pe, void *msg, int infofn);`

Here, `pe` is the intended destination of the `msg`. It may take on the values of:

- Any particular processor number - the message must be sent to that processor
- `CLD_ANYWHERE` - the message can be placed on any processor
- `CLD_BROADCAST` - the message must be sent to all processors excluding the local processor
- `CLD_BROADCAST_ALL` - the message must be sent to all processors including the local processor

CldEnqueue must handle all of these possibilities. The only case in which the load balancer should get control of a mesage is when `pe = CLD_ANYWHERE`. All other messages must be sent off to their intended destinations and passed on to the scheduler as if they never came in contact with the load balancer.

The integer parameter `infofn` is a handler index for a user-provided function that supplies CldEnqueue with information about the message `msg`. We will describe this in more detail later.

Thus, an implementation of the **CldEnqueue** function might have the following structure:

```
void CldEnqueue(int pe, void *msg, int infofn)
{
  ...
  if (pe == CLD_ANYWHERE)
    /* These messages can be load balanced */
  else if (pe == CmiMyPe())
    /* Enqueue the message in the scheduler locally */
  else if (pe==CLD_BROADCAST)
    /* Broadcast to all but self */
  else if (pe==CLD_BROADCAST_ALL)
    /* Broadcast to all plus self */
  else /* Specific processor number was specified */
    /* Send to specific processor */
```

```
}
```

In order to fill in the code above, we need to know more about the message before we can send it off to a scheduler's queue, either locally or remotely. For this, we have the info function. The prototype of an info function must be as follows:
`void ifn(void *msg, CldPackFn *pfn, int *len, int *queueing, int *priobits, unsigned int **prioptr);`

Thus, to use the info function, we need to get the actual function via the handler index provided to **CldEnqueue**. Typically, **CldEnqueue** would contain the following declarations:

```
int len, queueing, priobits;
unsigned int *prioptr;
CldPackFn pfn;
CldInfoFn ifn = (CldInfoFn)CmiHandlerToFunction(infofn);
```

Subsequently, a call to `ifn` would look like this:

```
ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
```

The info function extracts information from the message about its size, queuing strategy and priority, and also a pack function, which will be used when we need to send the message elsewhere. For now, consider the case where the message is to be locally enqueued:

```
...
else if (pe == CmiMyPe())
  {
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
  }
...
```

Thus, we see the info function is used to extract info from the message that is necessary to pass on to **CsdEnqueueGeneral**.

In order to send the message to a remote destination and enqueue it in the scheduler, we need to pack it up with a special pack function so that it has room for extra handler information and a reference to the info function. Therefore, before we handle the last three cases of **CldEnqueue**, we have a little extra work to do:

```
  ...
  else
    {
      ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
      if (pfn) {
pfn(&msg);
ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
      }
      CldSwitchHandler(msg, CpvAccess(CldHandlerIndex));
      CmiSetInfo(msg,infofn);
      ...
```

Calling the info function once gets the pack function we need, if there is one. We then call the pack function which rearranges the message leaving space for the info function, which we will need to call on the message when it is received at its destination, and also room for the extra handler that will be used on the receiving side to do the actual enqueuing. **CldSwitchHandler** is used to set this extra handler, and the receiving side must restore the original handler.

In the above code, we call the info function again because some of the values may have changed in the packing process.

Finally, we handle our last few cases:

```
  ...
      if (pe==CLD_BROADCAST)
CmiSyncBroadcastAndFree(len, msg);
      else if (pe==CLD_BROADCAST_ALL)
CmiSyncBroadcastAllAndFree(len, msg);
      else CmiSyncSendAndFree(pe, len, msg);
    }
}
```

5 Other Functions

A CldHandler function is necessary to receive messages forwarded by CldEnqueue:

```
CpvDeclare(int, CldHandlerIndex);

void CldHandler(void *msg)
```

```
{
  CldInfoFn ifn; CldPackFn pfn;
  int len, queueing, priobits; unsigned int *prioptr;

  CmiGrabBuffer((void **)&msg);
  CldRestoreHandler(msg);
  ifn = (CldInfoFn)CmiHandlerToFunction(CmiGetInfo(msg));
  ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
  CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
}
```

Note that the **CldHandler** properly restores the message's original handler using **CldRestoreHandler**, and calls the info function to obtain the proper parameters to pass on to the scheduler.

Also required is a CldModuleInit function:

```
void CldModuleInit()
{
  CpvInitialize(int, CldHandlerIndex);
  CpvAccess(CldHandlerIndex) = CmiRegisterHandler(CldHandler);
  CldModuleGeneralInit();

  /* call other init processes here */
  CldGraphModuleInit();
}
```

Here's an example of an additional init function:

```
void CldGraphModuleInit()
{
  CpvInitialize(int, CldRelocatedMessages);
  CpvInitialize(int, CldLoadBalanceMessages);
  CpvInitialize(int, CldMessageChunks);

  CpvAccess(CldRelocatedMessages) = CpvAccess(CldLoadBalanceMessages) =
    CpvAccess(CldMessageChunks) = 0;

  CldBalance();
}
```

You may want to provide the three status variables, which get initialized in your own module init function (called CldGraphModuleInit above). These can be used to keep track of what your LB is doing (see usage in cldb.graph.c and itc++queens program).

```
CpvDeclare(int, CldRelocatedMessages);
CpvDeclare(int, CldLoadBalanceMessages);
CpvDeclare(int, CldMessageChunks);
```

A method for queueing balanceable messages is provided in cldb.c. That file contains instructions for its use, and examples of its use can be found in cldb.graph.c. Its primary function is to provide a way to retrieve messages from the scheduler queue that have not yet been processed, so that they may be moved to another processor.

6 The HELP! Load Balancer

The HELP! Load Balancer is available in charm/src/Common/conv-ldb/cldb.test.c. To try out your own load balancer you can use this filename and SUPER_INSTALL will compile it and you can link it into your Charm++ programs with -balance test. (To add your own new balancers permanently and give them another name other than "test" you will need to change the Makefile used by SUPER_INSTALL. Don't worry about this for now.) The cldb.test.c provides a good starting point for new load balancers.

Look at the code for the HELP! balancer, starting with the **CldEnqueue** function. This is almost exactly as described earlier. One exception is the handling of a few extra cases: specifically if we are running the program on only one processor, we don't want to do any load balancing. The other obvious difference is in the first case: how do we handle messages that can be load balanced? Rather than enqueuing the message directly with the scheduler, we make use of the token queue. This means that messages can later be removed for relocation. **CldPutToken** adds the message to the token queue on the local processor.

Now look two functions up from **CldEnqueue**. We have an additional handler besides the **CldHandler**: the **CldBalanceHandler**. The purpose of this special handler is to receive messages that can be still be relocated again in the future. Just like the first case of **CldEnqueue** uses **CldPutToken** to keep the message retrievable, **CldBalanceHandler** does the same with relocatable messages it receives.

Next we look at our initialization functions to see how the process gets started. The **CldModuleInit** function gets called by the common Converse initialization code, and in turn, it calls our **CldHelpModuleInit**. This function starts off the periodic load distribution process by making a call to **CldDistributeTokens**. This function computes an approximation of half of its total load (**CsdLength**()), and if that amount exceeds the number of movable messages (**CldCountTokens**()), we attempt to move all of the movable messages. To do this, we pass this number of messages to move and the number of the PE to move them to, to the **CldMultipleSend** function.

CldMultipleSend is generally useful for any load balancer that sends multiple messages to one processor. It takes parameters *pe* and *numToMove*, and handles the packing and transmission of as many messages up to *numToMove* as it can find, to the processor *pe*. If the number and/or size of the messages sent is very large, **CldMultipleSend** will transmit them in reasonably sized parcels.

That's all there is to the HELP! balancer. Make the test version of itc++queens and try it out.