

Correlated Data in Java

Robert Goodwin

Tue, Mar 22, 2011

Data collection via the GETS32 protocol is widely supported in Acnet front ends. For Java programs, the Data Concentrator Engine (DCE) uses an RMI callback to deliver each piece of requested data soon after it arrives from a front end. Included in the callback is a time stamp to make it possible to correlate data so that one can work with data measured on the same 15 Hz beam cycle, say. But since the callbacks may come from several different DCEs, they can occur in any order. The problem for the Java program is how to “put Humpty Dumpty back together again” to recover a complete set of correlated data. This note describes a scheme for doing this, with the end result being a new callback that delivers an entire set of correlated data.

This scheme, written in C, involves a few functions that keep the correlation logic isolated so that it does not complicate the Java program logic. Here are the functions involved:

```
corr = CorrInit(int nDev, int nCopies, Func fun);      /* Initialize correlated data */
CorrData(CorrBlk *corr, int devx, int time, int dat); /* Table callback data */
CorrTerm(CorrBlk *corr);                             /* Terminate correlated data */
fun(int time, int count, int *data);                 /* user callback function */
```

The `CorrInit` function is called to set up support for managing the data from a set of devices. Memory is allocated, given the number of devices and the number of copies of the returned values. Each copy includes a time stamp key, a count of tabled devices with that time stamp, and the set of device values. The copies allow for different time stamps to be handled at the same time. The number of copies might be only 2 or 3.

The `CorrData` function is called for each callback that the Java program receives from the DCE. The data value is tabled associated with the given time stamp.

The `CorrTerm` function ends the correlated support for the set of devices in question. One may of course manage several active correlated data sets at once.

The callback function `fun` provides to the Java program a complete set of correlated data. Its parameters include the time stamp key, the count of devices whose data has been collected, and the array of data values. Here, the data and time stamps are assumed to be 32-bit integer values.

Review this scheme in more detail. The `CorrInit` function allocates memory sufficient to manage the sets of correlated data. A pointer to the allocated memory block is returned as `corr`. Note that `corr` must be passed to `CorrData` for each callback and finally to `CorrTerm`. This allows a program to use the same scheme for managing several sets of correlated device readings.

Assume that the Java program maintains an array of `device_ids` of some sort. When it gets a callback, it will convert the `device_id` into a device index `devx` by finding the `device_id` in this array. If one had a set of 10 devices being managed, the values of `devx` will range from 0–9.

The time is specified here as a 32-bit integer. In Java, it is actually an 8-byte value in millisecond units since the year 1970. To convert it to a 32-bit integer, one could establish a time base line when the request is first set up, and the values passed as time may be relative to that base value. In millisecond units, 32 bits can count to nearly 50 days.

The main logic is in the function `CorrData`. It takes the time stamp and finds a match against the copies of data sets that it manages in the allocated memory block pointed to by `corr`. On finding a match within an allowed range of `thr` milliseconds, it copies the data value into the `devx` element of the associated array. (The value of `thr` may be 5, say.) Each time it tables a data value, it checks to see whether all `nDev` devices have been collected, and if so, it makes a callback to the user function `fun`, then advances to the next data set, freeing up the one just passed to the user. If there are additional full data sets, it makes the corresponding callbacks for them as well.

If `CorrData` does not find a match against an existing data set, it looks for a free entry to use and initializes it accordingly. If `CorrData` cannot find a free entry to use for a new time stamp, it frees up the oldest data set by performing a callback with the incomplete data set, then frees that entry for use with the new time stamp.

Note that this scheme does not impose an explicit time-out in case a front end fails to deliver data that was requested to the DCE so that no callbacks come for data from that front end. What it does impose is a user-specified number of data sets, which means it can handle that many different time stamps at one time. If `CorrData` cannot find a free entry, it means that all data sets are full, which is only likely if one or more front ends is not returning the data requested. What will usually happen, with all front ends responding, is that the correlated data callback will be made as soon as a device has been placed in the array to comprise the complete data set. Only in the case that some front end is not replying with its requested data will the correlated callbacks be made late, depending on the number of data sets being maintained. The Java program can recognize the incomplete callback because the returned count is less than `nDev`.

The final result of this scheme is to provide a new correlated data callback to the user program that includes all of the data read for a single time stamp.

The reason for suggesting use of an event-based data request is that such requests have time stamps that mark when the event occurred, which all front ends can agree on. For periodic requests, in contrast, the time stamp will depend upon the time, μP Start, when a front end is scheduled to begin its cyclic (10Hz or 15Hz) work. If front ends are targeted that do not have the same μP Start, it is more difficult to match the time stamps to achieve correlated data.

To flesh out the details described here, see the file called `CorrData.c`.