Network Server/Consolidator

B. Lublinsky

Fermi National Accelerator Laboratory

Abstract

During the last several years Fermilab's control system was undergoing significant changes. More and more individual subsystems have been taken out of Camac crates and implemented as standalone boxes connected directly to the network. This relieves the Camac Front Ends [1] from controlling those subsystems, but creates significant additional load on the network, and what is even worse, some of the functionality that was previously supported by Camac Front Ends is gone with this approach. A most important feature that was supported through Camac Front Ends is the ability to do "ring-wide" reading and setting (the same "device" around the ring, supported through the different subsystems could be manipulated by one request). The necessity to support this feature, without further abusing the network, forced the creation of the network server/consolidator that is described in this paper.

1. Major system requirements.

The major requirements for this type of system are as follows:
1. It has to be flexible enough to allow adding new nodes and new "associations" between nodes;
2. It has to minimize network traffic that it originates.

The first requirement can be satisfied relatively easily by using multiple node association tables (fig 1). Based on the table number that the request is referring to, internal software builds whatever amount of requests is necessary to the nodes specified in the association table, sends them out, compiles the replies together and sends results to the original requester.

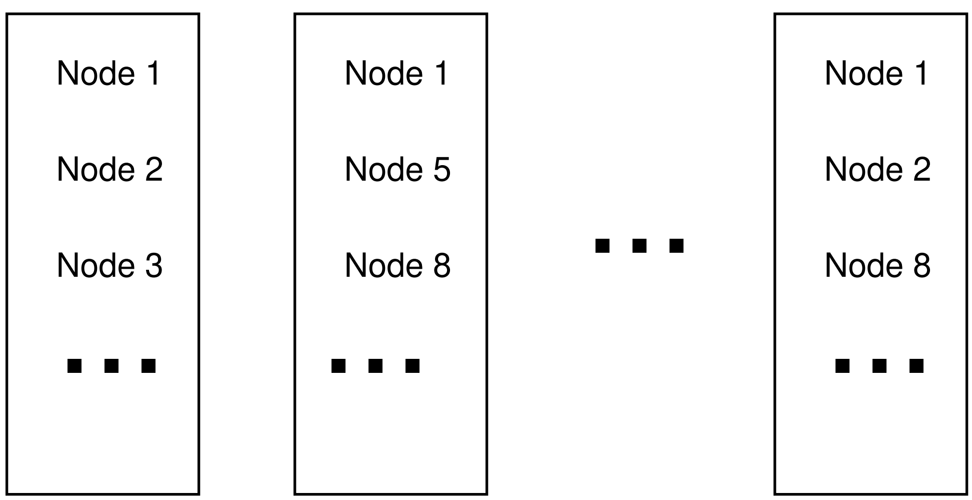

Fig 1. Association tables for Server/consolidator.

The second requirement is significantly tougher. If we examine the techniques that have been used for implementing microprocessor based systems at Fermilab, we will find two distinct approaches. One of them is the approach used in the Camac Front End [1]. Let us call it "no pool" approach. This approach is usually used in the case when machine is capable of front-ending very many devices, but only small part of them are accessed at a time. The advantage of this approach is that the machine is doing only what it has to do to satisfy current requests. The software implementation of this approach is fairly straightforward: you get the request, you go and get the data, you return it back. The obvious drawback is that for duplicated requests the data will be collected multiple times. This approach is usually acceptable if acquiring the necessary data is relatively "cheap" in terms of time and hardware efforts to get it. The second approach is usually used in the dedicated systems with relatively small amount of data [2]. In this case, the whole system usually runs from a fixed data pool that is collecting all system parameters. With organization like this, data is assumed always to be there, so any request from the outside can be satisfied immediately. This implementation usually implies two major levels of data processing: internal data collection that ensures data pool updates with the rate that is dictated by the system requirement, and replies to the rest of the world with the frequency specified by requests. The advantage of this approach is that the data access is always synchronous and straightforward. The biggest drawback is that usually the system is collecting more data that it really needs. But if the amount of this data is relatively small, pool paradigms are usually used.

For the consolidator system neither one of these two approaches will work satisfactorily. Running a fixed data pool is virtually impossible because we don't even know up front what will be the data that we will have to collect, and no pool solution can turn out to be too expensive from the point of view of the network traffic. The solution that has been used for this system is a dynamic pool, the kind of approach that is used on the consoles for request consolidation. When the system starts up, the pool is empty and no data collection is happening. When a request comes in, the first thing that is done is check whether this data is already available through the pool. If it is, then the request will be reusing existing data collection, if not, then new data collection will be started in the pool. The rules for appending of the requested item to the element of the pool are as follows:

1. Data collection frequency of the new request should not exceed data collection frequency of the existing element of the pool;
2. Length of the collected data for the new request should not exceed length of the collected data in the pool.

When all of the requests pointing to the specific element of the pool are canceled, this element of the pool will be deleted. This allows us to take advantage of both approaches described above. Only the data that is currently necessary is collected, every duplicated data collection is merged by the pool. An additional caveat here is that one request can build many requests going to different nodes. Having these small requests going back and forth can degrade significantly network performance. Even worse, this can saturate the network hardware. That is why, besides pooling of the requests themselves, we have to combine outgoing requests to different nodes as much as we can. For every new request we are comparing its reply frequency to the reply frequencies of the requests that already exist for this node. If the frequences are close enough and the combined reply length fits into the ACNET packet, the existing reply is killed. It is then combined with the new request and restarted.

2. Implementation.

Consolidator/Server is implemented using a 68040 VME-based system with two networking interfaces: Token Ring, used for communication with the cryogenic control system [2], and Ethernet, used for communication with the rest of the systems; it utilizes the VX-Works real time kernel. It uses Fermilab's "standard" timing and communication support [3]. The whole system is organized around three major queues (fig 2).

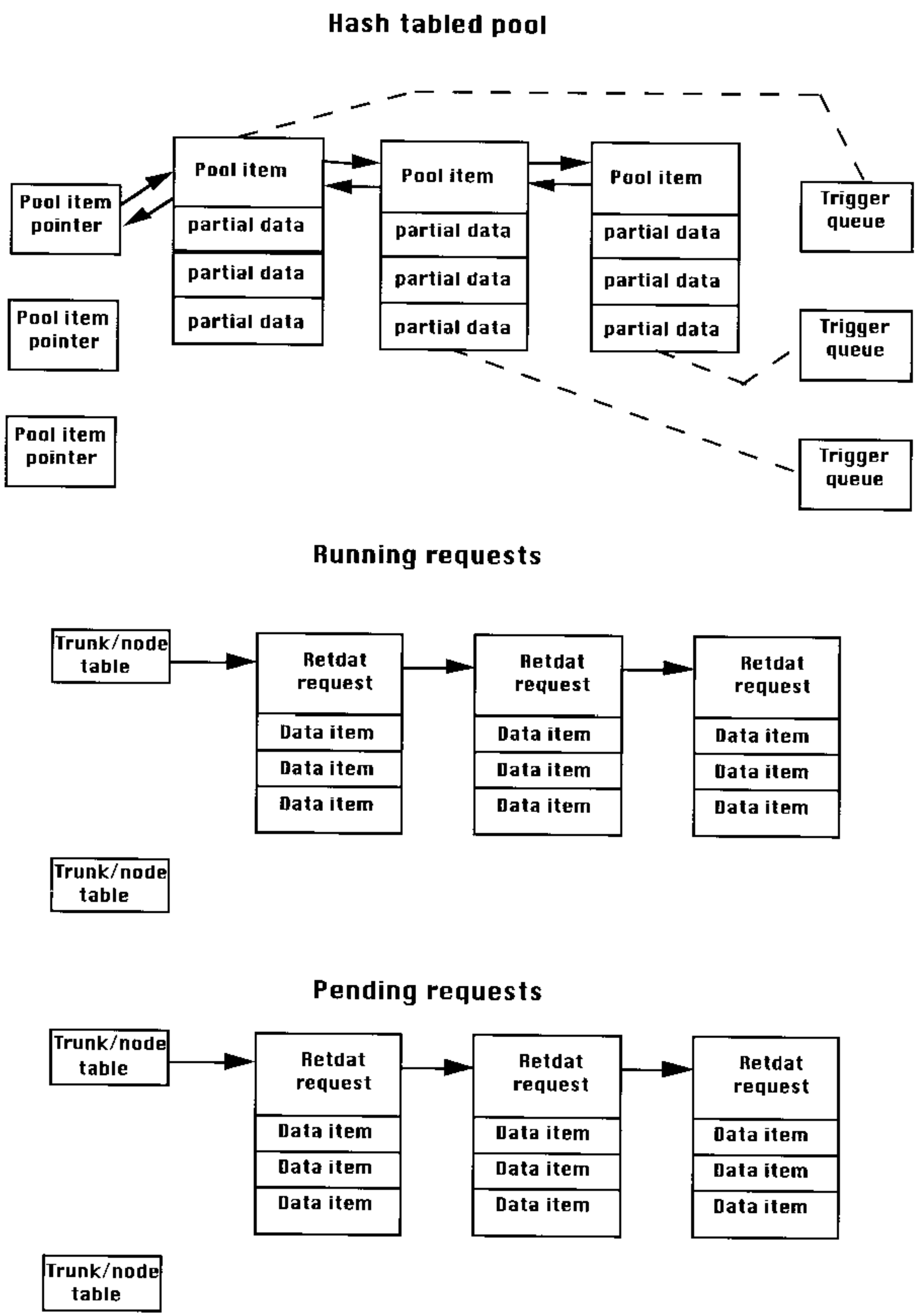

Fig 2. Server/Consolidator main queues

The most complex part of the system is the pool itself. In order to speed up the access every element of the pool is hashed based on the Device Index (DI) and Property Index (PI) that identify uniquely every Fermilab device. Every incoming request is parsed for participating device requests and every device is ether linked with the existing element of the pool or a new pool element is created for this device. Two other queues have to do with the requests built out of original requests. Both of them are organized on the basis of the node number of the system that they are talking to. One of them is the queue of the active (serviced) requests; another one is the queue of pending requests. Usually pending requests are the result of one of the nodes being down. Server/consolidator will periodically try to restart pending requests, so that if the node that was down is rebooted, data will automatically start to come back.

3. Conclusion.

The system is completed and tested. It has been heavily used for cryogenic system support for the last several months.

References

1. M. Glass et al, The upgraded Tevatron Front End. Nucl. Instr. and Meth. a293(1990) 87-90
2. B. Lublinsky, J. Firebaugh, J. Smolucha, New Tevatron Cryogenic Control System, "The proceedings 1993 International Conference on High Energy Physics", V3, p1817.
3. B. Lublinsky. Shell software for Smart subsytems with front-end capabilities. Nucl. Instr. and Meth A 352 (1994) 403 - 406