Data Synchronization Method Between FLPs and EPNs

Summer Student Report | 6 August 2014

Sarunya Pumma
Department of Computer Engineering
King Mongkut's University of Technology Thonburi, Thailand

Supervisors
Sylvain Chapeland
ALICE, CERN

Assoc. Prof. Dr. Tiranee Achalakul
Department of Computer Engineering
King Mongkut's University of Technology Thonburi, Thailand

1. Overview

In the year 2018, the ALICE detectors will be upgraded and the amount of data that will be produced from the detectors will be higher. The data throughput will be approximately 1 TB per second. For this reason, the whole ALICE's system has to be improved. The constraint of the new ALICE experiments is the system should be able to handle a large amount of particles collision events that will be produced from detectors. The data acquisition comprises two computer clusters – First Level Processors (FLP) and Event Processing Nodes (EPN). The two clusters serve different purposes. The FLP cluster is responsible to readout the detector and to reduce by a factor 5x the data before event reconstruction because the data have to be streamed to a permanent storage. Then, the data will be sent through the network to EPNs for event reconstruction and further processing. The data flow of ALICE experiment is shown in Figure 1.

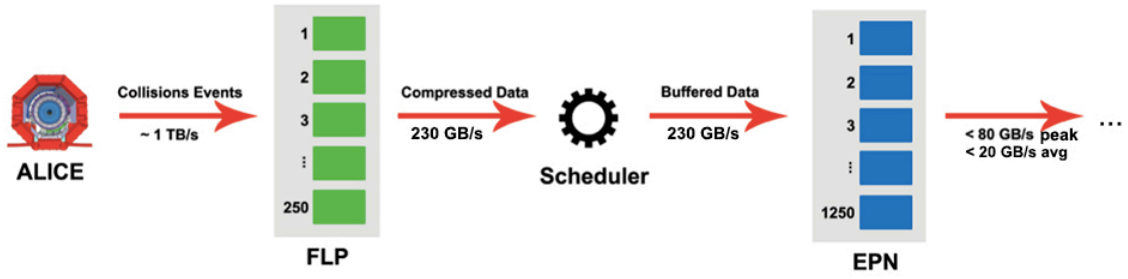

Figure 1 Data Flow of ALICE

In the new system, a scheduler between the two clusters is required in order to efficiently propagate jobs and data from FLPs to EPNs. The scheduler is responsible for matching between the collision events and EPN nodes. Each FLP node receives collision events which are grouped in time frames spanning 0.1 second. There are 10 thousand collision events per second (10 timeframes per second). EPN also has to forward the events to an assigned EPN node immediately. Scheduling process and data synchronization must be fast because the delay may cause severe bottleneck and provoke side effects like increasing the buffer space needed on FLPs. In this work, we have adopted Zookeeper, a data synchronization tool, to distribute events schedule (that is created by scheduler) to FLPs. This report presents the design and implementation of events schedule distribution protocol using Zookeeper.

2. Background Study: Zookeeper

As Zookeepr was used throughout this project, this section provides the general overview of this application. Zookeeper is a data synchronization service for distributed systems. It provides a service for sharing information among computers. Zookeeper is designed for serving heavy read operations, but low write operations. The shared data can be stored hierarchically in the same fashion as a file directory in most file systems where a single directory is allowed to have subdirectories. The directory in Zookeeper is called '*znode*', where the topmost *znode* is called 'root'. The illustration of data hierarchy is shown in Figure 2. With Zookeeper, data can be accessed using reference path as in a standard file system. As shown in Figure 2, the reference path for root is '/'. A *znode* consists of 2 parts, name and data. In order to refer to a specific znode, name is used as a reference.

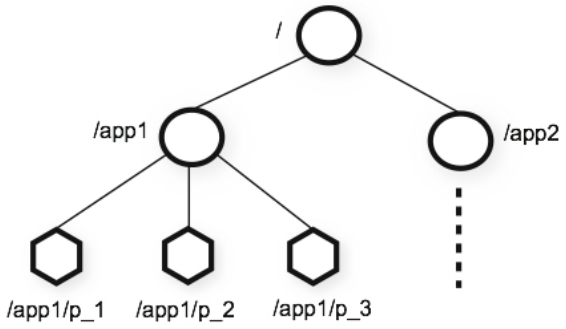

Figure 2 Zookeeper's Data Hierarchy
[Ref: http://zookeeper.apache.org/doc/trunk/zookeeperOver.html]

Zookeeper provides several useful functions for data synchronization. These functions can be called by using Zoopkeeper API, which supports only Java and C. The following list shows API of Zookeeper:

1. Create – create *znode* by specifying name, reference path of parent node, and data
2. Delete – delete *znode*
3. Get data – get data from *znode*
    a. Watch – this is an option for monitoring the change of a *znode*. If this option is enabled, Zookeeper will notify the process when the data of *znode* is changed.
4. Set data – set data of *znode*
5. Get children – obtain a list of children *znode* of a node

Znode can be created to be whether **permanent** or **ephemeral** node. An ephemeral *znode* will be deleted automatically when process dies. Unlike a permanent node that can be deleted only upon specific request. Moreover, there is another *znode* option called **sequential**. In the case that sequential option is selected, Zookeeper will automatically add numerical suffix to the name of *znode*, for example, <*name*>0000000001. The 10-digit suffix is calculated by a counter. This is useful for automatically generating the id for data, i.e., tasks in a queue.

In order to deploy Zookeeper, servers are required to host the service. From the requirement, 3 servers is a minimum. One server has to be selected as a leader who handles write requests. The topology of Zookeeper is shown in Figure 3.

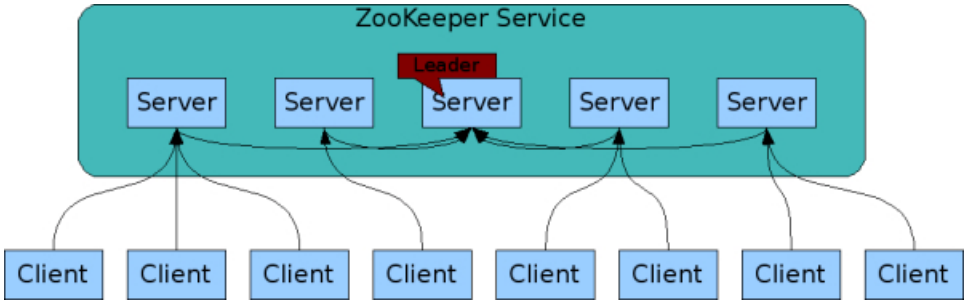

Figure 3 Zookeeper's Architecture
[Ref: http://zookeeper.apache.org/doc/trunk/zookeeperOver.html]

3. Scope of Work

In this section, the scope data synchronization between FLPs and EPNs are described. In order to understand the scope of work, the work flow and requirements of this work have to be known first. There are 3 main components in the work flow, which are FLP, Zookeeper, and EPN as illustrated in Figure 4.

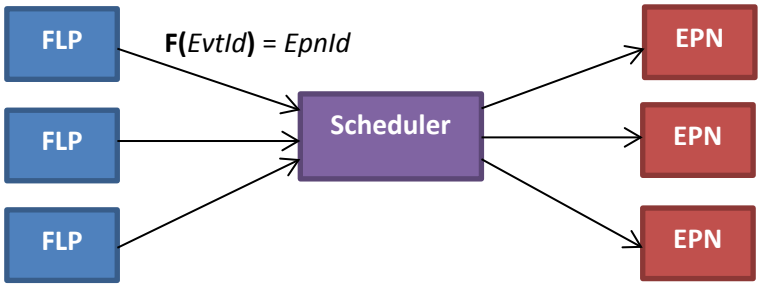

Figure 4 Work Flow

From Figure 4, data synchronization steps are as follows:

1. Scheduler keeps updating the available EPNs and events and sends this information to FLPs in the form of message. Notice that there is only one message at a time. The message consists of 3 parts as shown in Figure 5.

EvtId1 64 bits 20 digits	**EvtId2** 64 bits 20 digits	**EpnBitMask** 2048 bits

Figure 5 Message

   - EvtId1 – the smallest event id that available at that time
   - EvtId2 – the largest event id that available at that time
   - EpnBitMask – bit mask for indicating the available EPNs (1 denotes available)
     - The leftmost bit represents status of EPN 0
     - The rightmost bit represents status of EPN 2047
2. Each FLP receives collision events from the detectors (10 timeframes/second)

3. Each FLP calls function *f* to get an id of EPN that will receive a specific event to process
    - Input of function *f*: event id (*EvtId*)
    - Output of function *f*: EPN id (*EpnId*)
    - Calculation of function *f*: it adopts simple round robin algorithm to calculate *EpnId*
        - *EpnId* = *EvtId* % *nEvt* where % is modulo and *nEvt* is a number of available events
    - The calculation result from function *f* has to be logged in a file
4. Each FLP sends the event to EPN based on the calculation result from function *f*

In this work, only step 1 to step 3 were implemented and tested. The scope of this work includes implementation of message transferring method, implementation of function *f*, and implementation of logging method.

4. Design of Data Synchronization

This section provides the detailed design of the method for data synchronization between FLPs and EPNs. The design divided into 3 parts: data hierarchy design on Zookeeper, processes design, and Zookeeper's architecture design.

4.1. Data Hierarchy Design

In order to send message from a scheduler to FLPs, a *znode* for storing message is required. This znode will be henceforth called '*/msg*'. Moreover, another two *znodes* are required because the scheduler needs to update a list of available events and EPNs, and FLPs have to update the collision events from the detectors. These two *znodes* are called '*/uepn*' and '*/evt*' are used as information sources for the scheduler and FLPs. Notice that */uepn* and */evt* can be any other type of information sources in the real situations. Since there is only one message at a time, data design is simple. All *znodes* are under root node as shown in Figure 6.

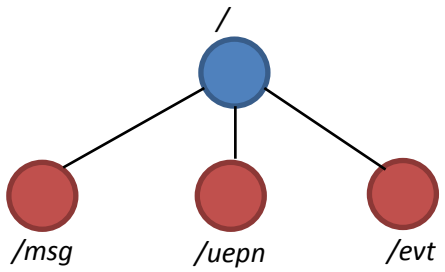

Figure 6 Data Architecture

The functions of these *znodes* are as follows:

- */msg* – stores a 2088-digit message (Figure 5)
    - Write by *Scheduler*
    - Read by *FLP* when they call function *f*
- */uepn* – stores list of available events and EPNs
    - Write by *ListUpdater* which is a process that keeps updating the list of available events and EPNs (the details of this process will be provided in the next section)
    - Read by *Scheduler*

- */evt* – stores an incoming event from the detector for a specific FLP
    - Write by *EvtUpdater* which is a process that keeps sending an event to FLP (the details of this process will be provided in the next section)
    - Read by *FLP*

4.2. Processes Design

Processes are the programs written in C for performing data synchronization as in the workflow shown in Figure 4. There are 4 processes in this work. *Scheduler* and *FLP* are the main processes, while *ListUpdater* and *EvtUpdater* are the assistive processes. The details of each process are as follows:

- *Scheduler*
    1. Gets the list of available events and EPNs from */uepn* when Zookeeper notified that information that is stored in */uepn* has been changed
    2. Posts this information as a message on */msg.*
- *FLP*
    1. Receives events from */evt* (10 events/second)
    2. Gets the message from */uepn*
    3. Checks the validity of the obtained event against the message
        - If valid,
            - Calls function *f* with *EvtId* as an input
            - Logs *EpnId* for the event to log file
        - If not valid, logs error
- *ListUpdater*
    1. Randomly adjusts the list of available events and EPNs
    2. Posts this information on */uepn* every 0.1 or 1 second (based on specification)
- *EvtUpdater*
    1. Increases event id by 1 (starting from the specified value)
    2. Posts this information on */evt* every 0.1 second

4.3. Zookeeper Architecture Design

In this section, the deployment of Zookeeper is presented. In our work, 3 Zookeeper services were installed on the same computer, and one service was selected as a leader. The client processes (*Scheduler*, *FLP*, *ListUpdater*, *EvtUpdater*) also located on the same computers as well. Clients and Zookeeper services communicate with each other by using C API. The server computer consists of 4 Intel Core i5 CPUs, 4 GB memory, and 29 GB storage. The deployed architecture is shown in Figure 7.

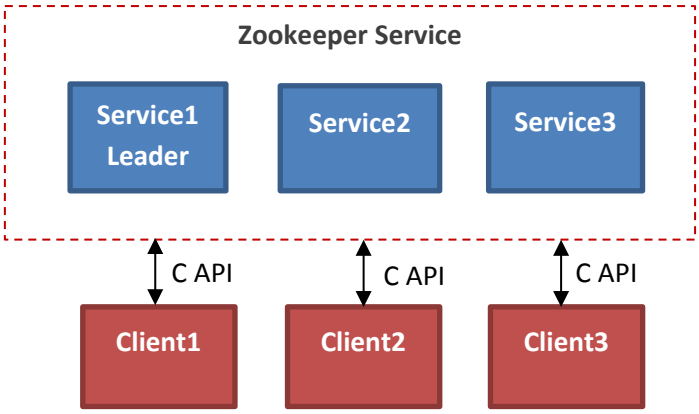

Figure 7 Architecture

5. Experiment and Result

The experiment aims to measure the average time that FLP used for calling function *f*. The experiment was initially set up on one computer. There were 3 Zookeeper services and 4 client processes, *Scheduler*, *FLP*, *ListUpdater*, and *EvtUpdater*, running on it. In the experiment, only one *FLP* was implemented. *EvtUpdater* sent events to FLP with a constant rate at 10 events per second, while *ListUpdater* sent the list of available events and EPNs to scheduler at two different rates, every 0.1 and 1 second.

To measure the time, a bunch of events was sent to FLPs. For each event, time used for calling function *f* was measured from the time when an event arrived at FLP to the time when EPN id was derived. Then, the average time for all events was calculated. Time measurement was performed by a timer in C. Note that a measured time includes time for events logging. In order to avoid the anomaly in measurements, each experiment was run 5 times and a mean of measured time of 5 runs was used. The results are presented in Table 1 and Figure 8.

Table 1 Average Time for Calling FLP

Number of events in a batch	**Average time (seconds)**	
	Available events and EPNs update every 0.1 seconds	**Available events and EPNs update every 1 second**
10	0.02492	0.01726
100	0.02063	0.01308
1000	0.02540	0.01148
10000	0.02200	0.01060
100000	0.02088	0.01239

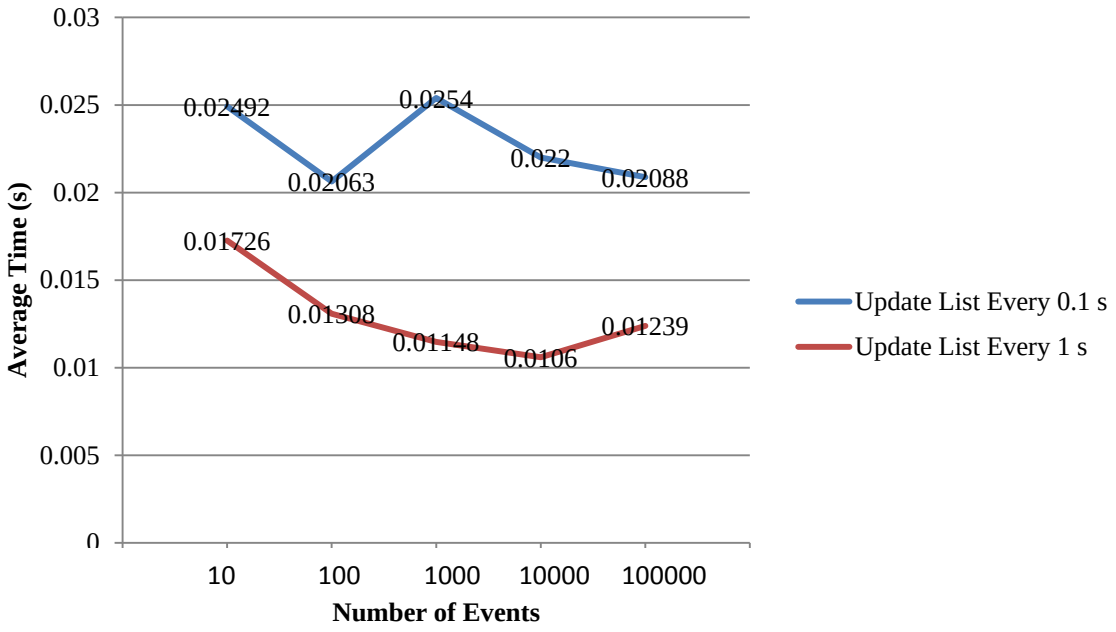

Figure 8 Chart Compare Between To Data Update Frequencies

According to the experimental results, a frequency of available events and EPNs refreshing directly affected to the time used to call function *f*. The more frequent the list was updated, the higher amount of time was required on FLP to process function *f*. However, number of events did not significantly affect the FLP calling time because the average FLP processing times for the experiments that had different number of events are similar.

With Zookeeper, FLP could obtain a message and process function *f* within a limited time of 0.1 second. We have monitored two factors in this experiment. One was a number of events and another one was a frequency of events and EPN updating. It was obvious that the frequency of events and EPN updating significantly affected the average FLP processing time. This may be caused from the *scheduler* process because it had to work harder when update occurred more frequent. Since both *scheduler* and *flp* had to read and write on Zookeeper, the processing powers of Zookeeper were shared among them. Therefore, the overhead that Zookeeper performed processing for *scheduler* could delay the processing for *flp*, and vice versa.

6. Conclusion

Due to the update of the ALICE detectors in the year 2018, entire ALICE system is needed to be developed. The two clusters, FLPs and EPNs, play an important role in data acquisition. Between these clusters a scheduler is required to efficiently propagate collision events from FLPs to EPNs. Moreover, to broadcast the information from the scheduler to FLPs, data synchronization protocol is needed.

In this work, we designed the data synchronization method in the middle of FLPs and EPNs. Zookeeper, which was data synchronization tool, was adopted. The scheduler was responsible for sending updated information of available events and EPNs to FLPs via Zookeeper. FLPs would receive collision events from detectors and fetch information from the scheduler via Zookeeper as well. In order to match an event to EPN, function *f* was implemented. The input and output of this function were event id and EPN id that the event will be sent to, respectively.

Once the framework was implemented, it was tested by measuring time used for calling function *f*. In the experiment, two factors, which were a number of events and a frequency of information update, were observed. Only the frequency of available events and EPNs update had significantly effect on the time used to call function *f*. However, all FLP calling time was satisfied the time constraint at 0.1 second (the period before new event arrives at FLP).

To improve the performance of Zookeeper, more Zookeeper services on separated machines will be required. This would help Zookeeper to be able to process better because computing resources are not shared. However, the communication overhead among the servers has to be determined closely.