HIA: Autonomic and General Collecting of Real-time Data Streams

Changfeng Chen[1, a] , Xiang Wang[2,b*] and Jinying Zhou[3,c]

[1,2,3] China Real-time Database Co. LTD, Software Avenue. 180, 210000 Nanjing, China

[a]chenchangfeng@sgepri.sgcc.com.cn, [b]wangxiang2@sgepri.sgcc.com.cn, [c]zhoujinying@sgepri.sgcc.com.cn

Keywords: Event Streams Collecting, Autonomic Computing, I/O Driver

Abstract. Power manufacturing enterprises process events by hundreds of devices interacting with their plants. Rich statistical data distilled from combining such interactions in real-time could generate much business value. In this paper, we describe the architecture of HIA, a system for ingesting multiple geographically distributed data source in real-time with high scalability and low latency, where the data streams may be real-time or batch. The system strongly self-manages infrastructure degradation without any manual intervention. HIA guarantees that there will be no data lost in the ingested output at any point in time, that most ingested events will be present in the output in soft real-time. Our product deployment ingests millions of events per second at peak with an average end-to-end latency of less than 1second. We also present challenges and solutions in maintaining large persistent ingesting state across geographically distant locations, and highlight the design principles that emerged from our experience.

Introduction

Collecting real-time streams has received considerable attention in the past three decades due to its importance in numerous applications (e.g. Device[1]monitor), where information generates in the form of very high-speed streams that need to be processed online to enable real-time[2] response.

With the explosive evolution of the power manufacturing enterprises in the last several years, the need for similar technologies has grown quickly as energy-based enterprises must process events generated by millions of devices interacting with their plants. Devices all over the enterprise measure the manufacture process on a daily basis, issuing power production, gas emit and materials cost.

Distilling rich statistical data from plant device interactions in real-time has a huge impact on business processes. The data enables decision maker to fine-tune production, budgets, campaigns and supervision, vary the parameters in real-time to suit changing market environment. It provides immediate feedback to decision maker on the effectiveness of their changes, and also allows the enterprise to optimize the budget spent for each power plant on a continuous basis.

To provide real-time statistical data, we built a system, called HighSoon Interface Adapter or HIA, that can ingest event[1,2] that generated by distributed plant devices. HIA ingest multiple real-time continuous streams of events.

Let us illustrate the operational steps in HIA with an example of ingesting a CO gas event:

- When a plant device record a measure and forms in an event. The Pull driver collets the event, where the event is forward to the HIA server.
- HIA server routes an event to destination Push driver
- Push driver push event to backing store or destination.

HIA ingest input event streams from plant device and writes collected events to backing stores. Collected events as shown in Fig.1 are used to generate rich statistical data.

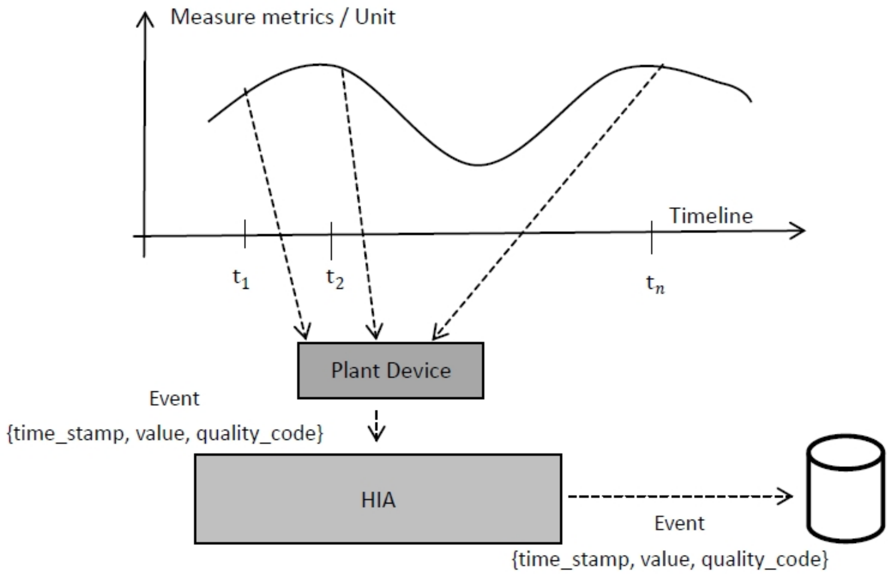

Fig. 1 Collecting event in HIA

Problem Statement

Formally, we express the data ingest task in *PullPush* model. *PullPush* is a similar MapReduce[3] programming model and an associated implementation for ingesting large data events. Programmers specify a *pull function* that poll events from a data device source to generate a set of intermediate events group by channel, and a *push function* that merges all intermediate events associated with the same channel key to writes collected events to backing stores.

The main goal of HIA is to ingest continuous stream in real-time. We use this framework to ingest many different event streams in our products.

System Challenges

While building HIA to ingest continuous data streams, we face these challenges:

- Consistent semantics: The output produced by HIA is used for operating production, reporting revenue to senior officer in the enterprise, and so on. HIA must guarantee that a single input event is never lost, since this leads to misunderstand the real world.
- Automatic system: Service interruptions can last from a couple of minutes to a few days or even weeks. There are hundreds plant devices. The operations of the whole system are overhead. The volume and complexity may be managed by highly skilled humans; but the demand for skilled IT personnel is already outstripping supply. The system could make decisions on its own, using high-level policies; it will constantly check and optimize its status and automatically adapt itself to changing conditions. An autonomic computing framework is the solution.
- Multiple sources and different formats: How will we ingest data from multiple sources and different formats in an efficient manner?
- Multiple destinations: Should all the raw data be ingested only in a single data store?
- High scalability: Not only does HIA need to handle millions of events per second today, it must also be able to handle the ever-increasing number of events in the future.
- High extensibility: How to support the owner develops the custom driver for specify data source?
- Low latency: The output of HIA is used to compute statistics for decision maker on how their manufacturing processes are performing, detecting overhead cost, optimizing budget, etc. Having HIA perform the collect within a few cent seconds significantly improves the effectiveness of these business processes.
- Just-in-Time Transformation: Should preprocessing of data – for example, checking / cleansing / validation - always be done before ingesting data in data store.
- Hybrid collecting mode: What are the essential tools / frameworks required in ingestion layer to handle data in real-time or batch mode?

Our Contributions

The key contributions from this paper are:

- To the best of our knowledge, this the first paper to formulate and solve the problem of ingesting multiple streams[4] continuously under these system constraints: consistent semantics, autonomic system, high scalability, low latency, multiple sources and different formats, multiple destinations, Just-in-Time transformation, Hybrid collecting mode.
- The solutions detailed in this paper have been fully implemented and deployed in a live production for several months. Based on this real-world experience, we present detailed performance results, design trade-offs, and key design lessons from our deployment.

The rest of this paper is organized as follows. Section 2 describes the data model in more detail. Section 3 presents the detailed design and architecture of the HIA system. Section 4 describes our product deployment settings and measurements collected from the running system. Sections 5 and 6 describe related research, and summarize future work and conclusions.

Data Model

A HIA system is a set of processes. Each system serves a set of channels. A channel[5,6] in HIA is a sparse map, and an abstraction like as a process in operating system for fulfilling transfer data between backing stores. Channels are dynamic entities that usually have a limited life span with the HIA system.

A HIA relation is an information container structured as a sequence of events, and a mapping between a measure point in the plant device and a destination measure point in the backing store. The relation just like as a file in the UNIX-style operating system[7,8]. Data is organized into two dimensions: relations and timestamps.

(relation: string, time: int64)-->value

We ingest the raw data identified by a particular relation key and timestamp. Rows or relations are grouped together to form the unit of load balancing, called frames, and columns or timestamps also are grouped together to form the unit of transferring, called batches.

Table 1 A slice of an example channel that represents the collected data

{ t_1,$value_{11}$,$quality_{11}$}	{ t_2,$value_{12}$,$quality_{12}$}	… …	{ t_m,$value_{1m}$,$quality_{1m}$}
{ t_1,$value_{21}$,$quality_{21}$}	{ t_2,$value_{22}$,$quality_{22}$}	… …	{ t_m,$value_{2m}$,$quality_{2m}$}
… …	… …	… …	… …
{ t_1,$value_{n1}$,$quality_{n1}$}	{ t_2,$value_{n2}$,$quality_{n2}$}	… …	{ t_m,$value_{nm}$,$quality_{nm}$}

The row name is associated a measure point. The columns name is a sampling timestamp.

Rows: HIA operates data in ascend order by row id. The row keys in a channel are arbitrary strings (currently up to 256 bytes in size).

Timestamps: Different relations in a channel can contain a sequence of events, where the events are indexed by timestamp. HIA timestamps are two 32-bit integers. They are assigned explicitly by plant device, in which case they represent real time in microseconds formed by Julian day. Plant devices that need to avoid collisions must generate unique timestamps themselves. Different events of a relation are represented in decreasing timestamp order, so that the most recent events can be seeing first.

Design Overview

We now describe the architecture of a pipeline at a single instance, as show in Fig. 2. In the following examples, we use current events as inputs, but the architecture applies to any similar event streams.

There are two major components in a single instance: the driver program to read/write current continuously and feed them to the HIA server; HIA sever orchestrates global operation of system.

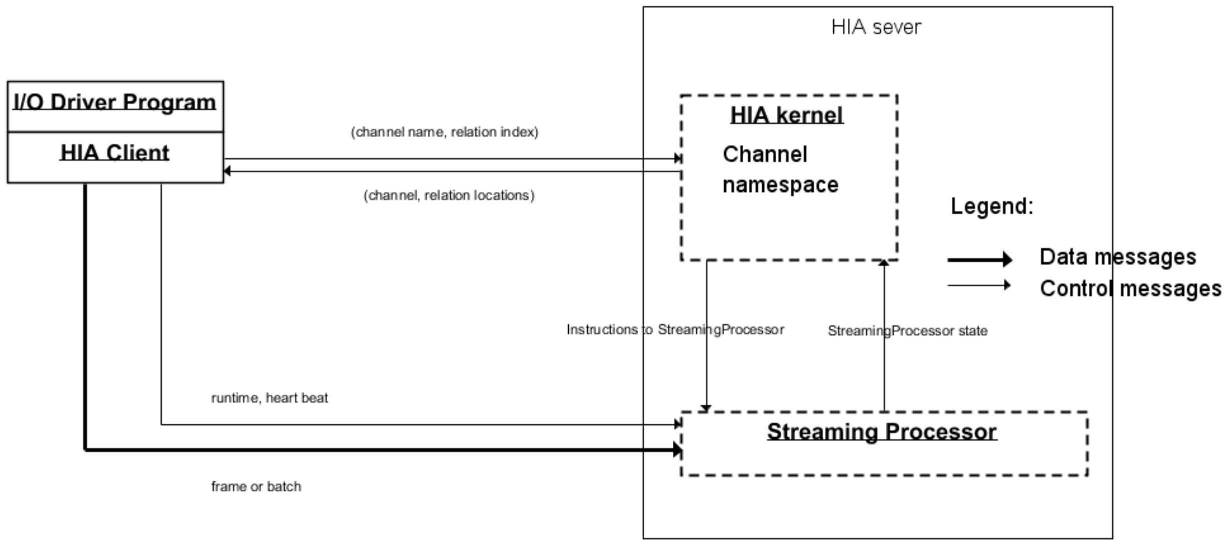

Fig. 2 HIA Architecture

Assumptions

In designing a collecting system for market's needs, we have been guided by assumptions that offer both challenges and opportunities. We alluded to some key observations earlier and now lay out our assumptions in more details[9].

- The system is built from low sustained technique environment that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly component failures on a routine basis.
- The system ingest a million number of data points. We expect a few million points, each event typically 3 seconds period.
- The workloads primarily consist of two kinds of ingestion: real-time streaming collect and batch streaming collect.

Interface

HIA provides a simple collection interface. We support the usual operations to get_snapshot, and get_archive_value events.

Moreover, HIA has device connection maintain operations. Plant device connection creates a link between I/O driver and plant data device.

Architecture

A HIA system instance consists of a single server with a kernel service and multiple streaming processor services and is accessed by multiple I/O driver programs, as shown Fig.2. Both HIA server and HIA I/O driver are user-level processes in a commodity Windows_NT machine.

The kernel maintains all relation system metadata. This includes namespace, the mapping from a source measure point to a destination measure point, and the channel owner of relations. It also controls system-wide activities such as loading I/O driver program[10], linking channels, keeping track of channel list[11,12], Keeping track of the I/O driver program[13], Watching the production environment[13], Scanning the directory of drivers[7]. The HIA kernel periodically communicates with each driver in Heartbeat message[9] to give it instructions and collect its state.

The programmer writes code to implement the collecting interfaces in a *PullPush* specification custom by practical data sources, and products a dynamic library. The user's code is linked together with the *PullPush* library (or HIA driver framework, implemented in C++). The HIA client as a C++ framework then invokes the *PullPush* function running in a user-level process. HIA driver program communicates with the HIA kernel and HIA streaming processor to pull or push data on behalf of the data device. Drivers interact with the kernel for metadata operations, but all real-time and batch data communication goes directly to the streaming processor.

The driver doesn't cache event data. Driver caches offer little benefit because it increases the complex of driver. Streaming processor needs to cache event data because the backing store will be offline.

Single Kernel

Having a single kernel or server vastly simplifies our design and enables the kernel to make sophisticated taking care of channels running using global knowledge. HIA system provides an execution environment in which channel may run. The task of creating, eliminating the channels is delegated to a group of routines in the kernel. Kernel itself is not a channel but a channel manager.

To let the kernel manage channels, each channel is represented by a channel descriptor that includes information about the current state of the channel. When the kernel stops the execution of a channel, it saves that current content in the channel descriptor. When the kernel decides to resume executing a channel, it uses the proper channel descriptor fields to load the streaming processor's registers. Because the stored value of the program counter points to the message following the last message collected, the channel resumes execution at the point where it was stopped.

The kernel interacts with plant's I/O devices (e.g. PI system[14]) by means of device drivers.

Channel Size

Channel Size is one of the key design parameters. We have chosen 40,000 relations per channel, which is larger than typical plant's need. The numbers of channels can up to 256 in our system. In other words, our ingesting system can support to collect 10 million measure points in real-time.

A suitable channel size offers several important advantages. First, it reduces driver's need to interact with the kernel because pulls and pushes on the same channel require only initial request to the kernel for channel information. Second, it reduces Administrator's need to interact with the HIA system because a suitable channel can cover all measure points in a plant device. Third, since on a large channel, a driver is more likely to perform many operations on a given channel, it can reduce network overhead by keeping a persistent TCP connection to the streaming processor over an extended period of time.

Metadata

The kernel stores three major types of metadata[1,4,9]: the list of relation, channel and driver. All metadata is maintained by the kernel's tracker service.

Consistency Model

HIA has a relaxed consistency model that supports our high performance ingesting well but remains relatively simple and efficient to implement.

Every channel can isolated acquire resource. There are no relations between channels at runtime. The metadata update is marked by version, it can avoid conflicts[2,10].

Performance Result

In this section we present a few micro-benchmarks to illustrate the bottlenecks inherent in the HIA architecture and implementation, and also some numbers from real clusters in use at *State Power Investment Corporation*.

We measured performance on a HIA instance consisting of one HIA server and16 plant devices. Note that this configuration was set up for ease of testing. Typical instance have hundreds of plant data devices.

All the machines are configured with dual 2.0 GHz Intel Core i5 processors, 4 GB of memory, two 320 GB 7200 rpm disks, and a 1Gbps full-duplex Ethernet connection. Event throughput results with different channel numbers is shown in Fig. 3, Fig. 4 and Fig. 5.

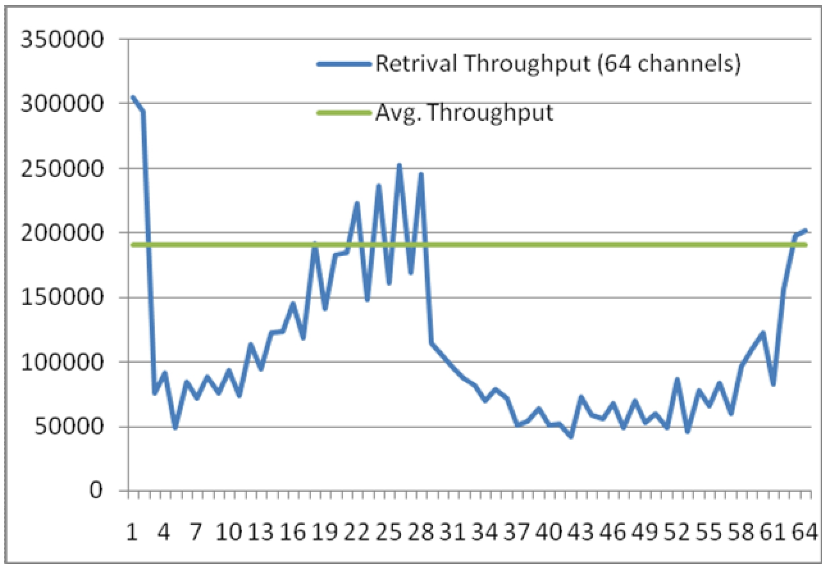

Fig. 3 Event throughputs when run 64channels

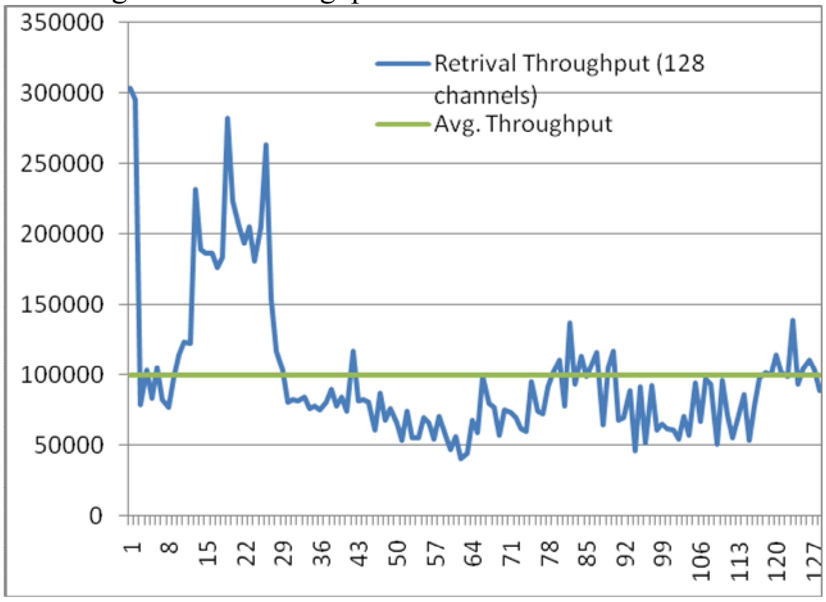

Fig. 4 Event throughputs when run 128 channels

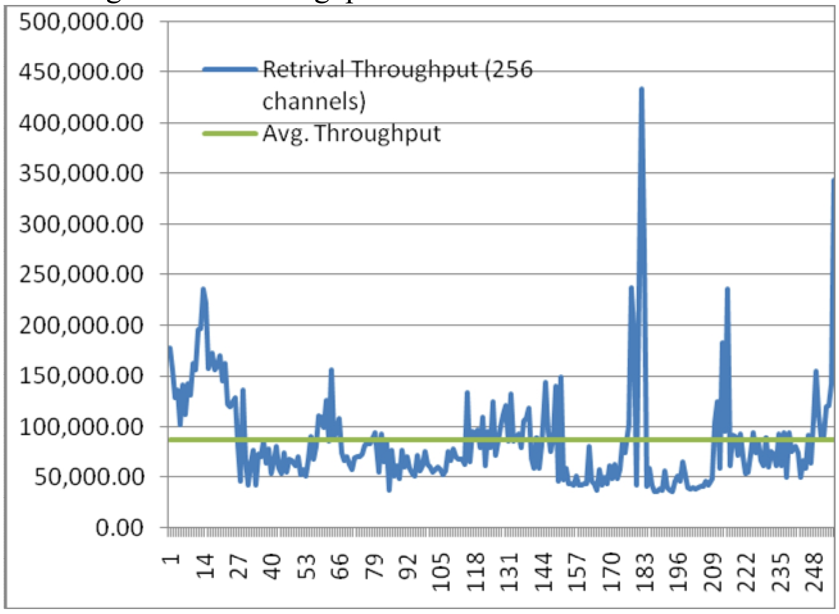

Fig. 5 Event throughputs when run 256 channels

We can see that the peak event throughput occurs in a single channel, the average throughput is decreasing by the number of channels increase. According to the average throughput, the total event throughput could over 20 million per second.

Related Work

Like other ingest systems such as Kafka[15,16], PI ICU[14], HIA provides an abstraction which enables data to be moved between backing stores.

Unlike Kafka, HIA spreads collecting data across backing store servers in a way distributed by internet style, provides more precise operating model to control data flow, and also supplies highly efficiency because HIA system complete implements using C++. In contrast to system like Kafka, HIA does not provide large cluster architecture, since HIA's technical environment is constrained by resource.

In compare to systems like PI, HIA provide more flexible ingesting pattern using *PullPush* model.

Conclusions

HIA system demonstrates the qualities essential for supporting large-scale data ingesting workloads on loose couple environment. While some design decisions are specific to our unique setting, many may apply to data ingesting tasks of a similar magnitude and cost consciousness. We started by reexamining the popular ingesting system assumptions in light of our current and anticipated application workloads and technological environment. Our observations have led to radically different points in the design space. We treat component failures as the norm rather than the exception using autonomic computing paradigm. Our system provides fault tolerance by constant monitoring, and fast and automatic recovery. Our design delivers high aggregate throughput to many concurrent channels performing a variety of collecting tasks. We achieve this by separating channel control, which passes through the master, from data transfer, which passes directly between streaming processors and drivers. Kernel involvement in common operations is minimized by a suitable channel size. This makes possible a simple, centralized kernel that does not become a bottleneck. HIA has successfully met our ingestion needs and is widely used as the ingesting platform for production data processing. It is an important tool that enables us to continue to integrate in other high-level applications.

References

[1] D. C. Schmidt, T. Suda, The Performance of Alternative Threading Architectures for Parallel Communication Subsystems, Journal of Parallel and Distributed Computing, 1996.

[2] C. D. Gill, Flexible Scheduling in Middleware for Distributed Rate-Based Real-Time Applications, Doctoral Dissertation,2002.

[3] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, http://labs.google.com/papers/ mapreduce.html, 2005.

[4] R. Ananthanarayana, V. Basker, S. Das et al, Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams, SIGMOD'13, New York, 2013.

[5] D. C. Schmidt, Applying a Pattern Language to Develop Application-level Gateways, Theory and Practice of Object System, Wiley & Sons, Vol. 2, No. 1, 1996.

[6] D. C. Schmid, I. Pyarali, The Design and Implementation of the Reactor: An Object-Oriented Framework for Event Demultiplexing, http://www.dre.vanderbilt.edu/~schmidt/PDF/Reactor2-93.p df, 2010.

[7] D. P. Bovet, M. Cesati , Understanding the Linux Kernel, 3rd ed., O'Reilly, New York, 2005.

[8] W. Mauerer , Professional Linux Kernel Architecture. Wrox, New York, 2008.

[9] S. Ghemawat, H. Gobioff, S. T. Leung, The Google file system. In 19th Symposium on Operating Systems Principles, pages 29–43, Lake George, New York,2003.

[10] D. C. Schmidt, T. Suda, The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons. Proceedings of the IEEE Second International Workshop on Configurable Distributed Systems, Pittsburgh, PA, 1994.

[11] J. O. Kephart, D. M. Chess, The vision of autonomic computing, Computer, 36(1):41–52, 2003.

[12] D. C. Schmidt DC, Reactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events, http://www.yeolar.com/note/2012/12/09/reactor/, 2012.

[13] E. Curry, P. Grace, Flexible Self-Management Using the Model-View-Controller Pattern, IEEE Software, vol. 25, no.3, pp. 84-90, 2008.

[14] Information on http://www.osisoft.com/.

[15] K. Goodhope K, Koshy, J. Kreps et al, Building LinkedIn's Real-time Activity Data Pipeline, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2012.

[16] J. Kreps, N. Narkhed, J. Rao, Kafka: a Distributed Messaging System for Log Processing, http://kafka.apache.org /documentation.html, 2011.