Postprint of LAMOST Control Node Distributed Status Collection and Monitoring System Based on Python Coroutine Technology

Authors: Tian Yuan1,3; Wang Feng2,3; Li Jian1,3; Wang Zheng1,3; Zhao Yongheng1,3

Date: 2018-06-22T00:00:00+00:00

Abstract

The control of modern large astronomical telescopes is generally accomplished by multiple independent computers (or computer clusters). The performance, efficiency, and reliability of these computers directly impact the stable operation of the entire telescope. Consequently, it is necessary to develop a software system capable of real-time collection, storage, and monitoring of hardware resource information from each control computer (node), along with providing certain early warning functions. Such a system can effectively eliminate potential hazards and thereby enhance the overall observation and operational efficiency of the telescope. This paper, based on an in-depth analysis of the observation and operational requirements of the LAMOST telescope, employs asynchronous coroutine technology to design and develop a hardware resource monitoring system software based on the Python language. The system can efficiently and stably acquire various states of the computers and obtain real-time information of corresponding components, while providing multiple human-computer interaction methods and extension interfaces for subsequent development. Deployed in the LAMOST environment, actual operation demonstrates that the entire system has achieved favorable results.

Full Text

Preamble

LAMOST Control Node Distributed Status Acquisition and Monitoring System Based on Python Coroutine Technology

Tian Yuan[1,3], Wang Feng[2,3], Li Jian[1,3], Wang Zheng[1,3], Zhao Yongheng[1,3]
[1]. Guo Shoujing Operation and Development Center, National Astronomical Observatories, Chinese Academy of Sciences, Beijing 100101

[2]. [Second affiliation not fully provided in original]
[3]. [Third affiliation not fully provided in original]

Abstract

Modern large-scale astronomical telescopes are typically controlled by multiple independent computers or computer clusters. The performance, efficiency, and reliability of these computers directly impact the stable operation of the entire telescope system. Consequently, a software system capable of real-time acquisition, storage, and monitoring of hardware resource information from each control computer (node) is essential, along with providing early warning functionality. Such a system can effectively eliminate potential risks and improve overall observational efficiency. Based on a thorough analysis of LAMOST's operational requirements, this paper presents the design and development of a hardware resource monitoring system implemented in Python using asynchronous coroutine technology. The system efficiently and stably collects various status information from computers, obtains real-time data for corresponding components, provides multiple human-computer interaction methods, and offers extension interfaces for future development. Deployed in the LAMOST environment, practical operation demonstrates that the system achieves satisfactory results.

Keywords: Resource monitoring; Asynchronous coroutines; LAMOST; Large-scale telescope operations and maintenance

1. Requirements Analysis and Tool Selection

The LAMOST telescope utilizes a diverse array of computers with significant variations in both hardware and software. In terms of equipment types, the system includes high-performance blade servers and workstations, industrial PCs adapted to harsh environments, and desktop computers for operator use. Geographically, data service computers (such as storage servers and data transmission devices) are housed in dedicated machine rooms, human-interaction desktops are located in the observation control room, hardware driver computers (such as camera control clusters and focal plane controllers) are installed within the telescope building itself, and auxiliary service computers (like meteorological station outdoor units and DIMM devices) are placed outdoors. Regarding operating systems, the cluster currently employs Windows series, Linux (RedHat or Ubuntu), and Apple macOS.

Given this heterogeneous cluster with varied types, complex distribution, and multiple operating systems, manual monitoring of hardware resource information for each node is not only inefficient but also highly prone to errors and omissions that can impact telescope operations. Clearly, constructing an automated hardware resource monitoring software system to acquire, store, monitor,

and analyze the status of various nodes—especially critical ones—is essential for modern large-scale telescopes.

In the engineering domain, mainstream open-source cluster monitoring solutions include Cacti, Nagios, and Zabbix, all of which play important roles in various applications. However, Cacti focuses primarily on network resource monitoring and cannot comprehensively monitor node status. While Nagios and Zabbix can fully monitor node resources and provide alerting mechanisms and plugin templates, their configuration is cumbersome and re-development is difficult due to their general-purpose nature, making integration into LAMOST' s existing observatory control system challenging. Therefore, based on project requirements, a customized monitoring system tailored specifically for LAMOST' s computer cluster is necessary.

To ensure good usability and stability, the hardware resource monitoring system should adopt a layered design. Considering practical engineering factors, the system is divided into three layers: information collection and transmission, information reception and processing, and information display and application. The information collection and transmission layer must account for significant node differences while maintaining unified transmission implementation. The reception and processing layer requires robust receiving capacity and stable processing capability, providing decoupled interfaces for upper-layer applications. The display and application layer should be modularized to achieve flexible and user-friendly human-computer interaction. The entire system must also consider operational efficiency to avoid imposing excessive load on nodes or the network.

Furthermore, to better serve telescope operations, the software should integrate into the telescope' s software architecture. LAMOST is currently upgrading its observatory control system using the Remote Telescope System 2nd version (RTS2). Therefore, the resource monitoring system must reserve interfaces for the RTS2 framework.

Based on this engineering requirements analysis and considering the actual working environment, a "client-server-application" architecture is adopted for the monitoring system. Python is selected as the primary language due to its strong versatility and excellent cross-platform capabilities, leveraging both Python' s standard libraries and various third-party libraries.

After thorough investigation and careful selection, the following Python libraries and modules are primarily utilized:

- **psutil module**: Enables easy acquisition of system processes and utilization information (including CPU, memory, disk, network, etc.), primarily applied in system monitoring, analysis, and management.
- **Python standard library asyncio module**: Supports asynchronous coroutines. Traditional server-side designs typically employ multi-way I/O multiplexing, multi-process (pools), or multi-thread (pools) technologies. While these improve server response efficiency, they have limitations. For

instance, multi-way I/O multiplexing becomes overly complex and difficult to maintain for large server programs. Multi-process technology consumes substantial system resources (memory footprint, context switching overhead) and involves complex inter-process communication. Multi-threading consumes fewer resources than multi-processing, but issues like synchronization, race conditions, and deadlocks create significant design challenges. Coroutines represent a user-controlled context switching technology in user space. Compared to processes and threads, coroutines are more lightweight and offer more flexible scheduling, making them ideal for network communication environments with small data volumes, simple logic, and high concurrency. In Python, coroutines are a relatively new concept, introduced in Python 3.4 and enhanced with async/await keywords and syntax in Python 3.5, along with the asyncio standard library.

- **curio library**: A wrapper for the asyncio module that further hides event loops and complex callback mechanisms, providing development tools such as Queue (for inter-coroutine communication), SignalQueue (for system signal processing), run_in_process (for subprocess scheduling), and run_in_thread (for subthread scheduling), making asynchronous coroutine programming more concise and secure.
- **PyZMQ library**: The Python version of the ZMQ communication library. ZMQ builds upon standard sockets to develop upper-layer protocols, providing advanced interfaces such as discontinuous retransmission, load balancing, and publish/subscribe, suitable for decoupling large software components and enhancing stability.
- **aiomysql module**: An asynchronous MySQL database interface for Python that enables asynchronous data interaction between Python applications and MySQL databases, while providing configurable thread pools to improve database operation performance.

2.1 Overall Design

The LAMOST resource monitoring software system is divided into three layers: information collector (client), central information processing server, and upper-layer application modules, as shown in Figure 1 [Figure 1: see original paper].

The information collector (client) deployed on each node computer in the LAMOST cluster periodically collects local hardware resource information based on default settings and configuration files that specify collection intervals and content, then transmits this information to the central information processing server.

The central information processing server receives information from numerous clients, performs processing (such as data validation, analysis, and storage), and provides data services to upper-layer application modules through its service interface.

Multiple independent application modules can be implemented for different functions (such as graphical interface interaction, web display, integration with the observatory control system, and detailed data analysis). All modules obtain data through the central server' s data provision interface to perform their respective tasks. Application modules and the central information processing server are independent of each other, with no required startup sequence or need to run on the same computer, maximizing system decoupling and thereby enhancing stability and usability.

2.2 Client

The client's primary responsibility is to periodically collect node system resource information and transmit it to the server. For code simplicity and maintainability, the Python psutil package is selected for information collection. Most node computers have Python interpreters and corresponding packages pre-installed. For nodes without psutil, installation via pip or source code is straightforward. For the few nodes where Python cannot be installed, C language is used to call native system library interfaces for information acquisition.

To ensure transmission universality and minimize additional library installation on node computers, the client employs the standard TCP communication protocol. The communication functionality is implemented in Python by wrapping collection and transmission operations in a loop with error tolerance thresholds. When communication failures occur, exceptions are caught and an error counter increments. If the counter hasn' t reached the threshold, the system sleeps for a specified time before retrying; if the threshold is reached, the program terminates and releases system resources.

2.3 Server

The central information processing server serves as the system hub with two main tasks: (1) receiving periodic resource information from numerous clients, and (2) processing and storing this information into a MySQL database. Consequently, high operational efficiency and stability are required. To achieve clear design and concise implementation, the solution employs two subprocesses: an asynchronous message receiver and a MySQL asynchronous processor. The server' s internal structure is shown in Figure 2 [Figure 2: see original paper].

The asynchronous message receiver subprocess contains a TCP-Server coroutine and a ZMQ.PUB coroutine. The TCP-Server coroutine handles TCP connection requests from node collectors (clients) and concurrently receives messages from multiple clients. The ZMQ.PUB coroutine publishes received messages to a message bus for use by upper-layer application modules. Inter-coroutine communication within the single thread uses curio' s Queue, with the entire subprocess driven by the curio library' s event loop.

The MySQL asynchronous processor subprocess contains a ZMQ.SUB coroutine

and an aiomysql coroutine. The ZMQ.SUB coroutine subscribes to and retrieves messages from the message bus, while the aiomysql coroutine asynchronously stores messages into the MySQL database. Since the aiomysql module requires the standard Python.asyncio event loop (curio does not currently provide direct aiomysql interfaces), and the Python.asyncio event loop must exclusively occupy a thread, the implementation creates an independent subthread for the aiomysql coroutine. Asynchronous communication between these components uses curio' s Universal Queue, which unlike the regular Queue, enables asynchronous communication between coroutines belonging to different threads. The entire subprocess is driven by the curio library' s event loop.

Additionally, the central information processing server utilizes ZMQ' s publish/subscribe mechanism to form a message bus for providing data to upper-layer application modules. The ZMQ.PUB port publishes messages, while any number of ZMQ.SUB ports can subscribe and receive messages. ZMQ.PUB and ZMQ.SUB need not be deployed on the same computer and run without mutual interference. These ZMQ advantages greatly simplify service interface layer development and achieve complete decoupling between the central information processing server and application modules.

The internal component sequence diagram of the central information processing server is shown in Figure 3 [Figure 3: see original paper]. Since all components are coroutines using asynchronous communication, with coroutine logic execution and context switching precisely controlled by the user, the server' s parallel processing capability and data throughput are greatly enhanced without consuming additional operating system resources.

In practice, the central information processing server runs as a Linux background daemon process configured for automatic startup, ensuring more stable and efficient operation.

2.4 Application Modules

Application modules are responsible for data utilization and presentation. The resource monitoring system supports multiple independent application modules for different functions. The ZMQ message bus enables complete decoupling between the central information processing server and application modules, significantly reducing development difficulty.

Based on project requirements, the system currently provides three application modules: a PyQt-based graphical interface module, an RTS2 system sensor device (Rts2-sensord) module, and a Django-based web service module.

Figure 4 [Figure 4: see original paper] shows a screenshot of the local graphical interface module. The interface displays real-time resource information for each node computer, with color progress bars providing resource utilization status for maintenance engineers. In Figure 4, the hard disk usage of CCD computers 01 and 02 exceeds 80%, so the corresponding hard disk progress bars in the node

sub-interfaces turn red to alert observers, achieving an early warning effect.

Figure 5 [Figure 5: see original paper] shows a screenshot of the RTS2 system monitor receiving warning information after integration with the resource monitoring system' s sensor module RTS2-Sensord, enabling the RTS2 framework to receive resource warnings.

The web service module, deployed on LAMOST' s internal network, provides browser-based query services, allowing observers and maintenance engineers to query node resource information from computers on the internal network. Figure 6 [Figure 6: see original paper] shows the resource information page for CCD computer 01. The upper portion displays historical records and trends of various resource information as curves, with configurable dashed lines providing warning functionality. The lower portion shows a detailed information list including timestamp, CPU usage, memory_percent, disk_percent, network download speed (nic_in), and network upload speed (nic_out).

3.1 System Overhead and Network Load

The resource monitoring system software consumes system resources during operation. Therefore, during design and development, code should be simplified and optimized while ensuring usability. Deployment in the actual engineering environment yields the following performance measurements:

- Node message collector (client) resource consumption: CPU usage not exceeding 0.2%, memory not exceeding 4KB.
- Central information processing server resource consumption: CPU not exceeding 0.5%, memory not exceeding 40MB.

Thus, the system does not impose significant resource consumption on either node computers or the central server.

Regarding network transmission, current single-node information collectors transmit data as strings, with each message not exceeding 128 bytes and the default collection interval set to 10 seconds. Consequently, 100 node collectors generate no more than 1,280 bytes per second, meaning the entire system' s internal network bandwidth consumption does not exceed 1.25KBps. In the future, if increased per-node information volume or total node count leads to excessive bandwidth usage, binary data transmission may be considered instead of strings.

Therefore, the LAMOST resource monitoring system meets current engineering requirements in terms of both resource consumption and network load.

3.2 Network Communication Architecture: Centralized vs. Distributed Design

Current mainstream network communication architectures are divided into centralized and distributed designs. Centralized design requires a robust central

server as the communication system hub. Its advantages include mature technology and ease of management and maintenance, but it depends heavily on the central server' s stability and processing capacity. Distributed design disperses information or functions across different computers that collaborate to complete tasks, thus requiring less performance from individual machines. Its advantages include significantly lower server costs, but the design is more complex with increased collaborative communication overhead.

For this project, although the cluster involves numerous computers, both communication volume and data processing volume are small. Therefore, centralized design in a "client-server-application" pattern is more reasonable, avoiding data synchronization issues in distributed systems and reducing development and maintenance complexity.

3.3 Communication Protocol Selection Between Client and Server

After collecting information, clients must transmit it to the server over the network. Implementation options include UDP transmission, TCP transmission, ACE communication library, and ZMQ communication library. UDP communication is relatively unreliable and therefore not considered. The ACE communication library is a comprehensive C++ library with complete object-oriented design and multiple design patterns, but its complexity and high development/maintenance difficulty make it unsuitable for this project' s practical engineering requirements. While ZMQ communication (see server internal communication implementation) is convenient and flexible, it requires installing the ZMQ library on each client. As previously mentioned, significant hardware differences and diverse operating systems across nodes would make installing appropriate ZMQ versions labor-intensive, operationally difficult, and 不利于 future equipment updates and expansion. TCP transmission, as the most common 底层 communication protocol, is implemented in libraries included with all operating systems, eliminating additional installation and configuration. Therefore, TCP is adopted for client-server communication.

3.4 Extension of System Alert and Analysis Functions

The LAMOST resource monitoring system has been initially completed and deployed in the actual engineering environment, effectively collecting and storing resource information from cluster nodes. However, two limitations remain:

(1) **Human-computer interaction alerting**: As described in Section 2.4, current alerts are limited to color changes in local graphical interface resource bars (green, blue, red) and web-based warning lines. Future improvements could 借鉴 Zabbix' s approach by providing email push notifications and WeChat messages based on warning types or error levels to enhance system friendliness.

(2) **Post-storage data analysis and processing**: This functionality is not yet implemented, primarily because it requires continuous accumulation of large telescope operation and maintenance experience.

Nevertheless, the server' s internal use of ZMQ publish/subscribe message bus technology completely decouples warning components and data analysis components from the software framework, greatly reducing the technical difficulty of further development. Consequently, the resource monitoring system will continue to improve, ultimately providing strong support for efficient telescope operation.

Conclusion

The LAMOST telescope, independently developed by China, is currently the world' s most efficient large-scale astronomical spectroscopic survey facility. Its internal management and control involve a vast, diverse, and numerous computer cluster. This system' s design enables real-time collection, storage, and analysis of hardware resource information from each node, effectively improving operational stability and observational efficiency. Operational experience demonstrates the system' s successful design and development, while this approach also provides valuable references for observation control and maintenance of other large telescopes.

References

[1] Zhao Yongheng. Technology of automatic observation of astronomical telescope [J]. E-Science Technology & Application, 2012, 3(4): 11-16.

[2] Luo Ali, Tian Yuan, Song Jiang, et al. Design and implementation of LAMOST observatory control system [J]. E-Science Technology & Application, 2012, 3(4): 76-85.

[3] Zhao Weirui, Lu Zhigang, Jiang Zhengwei, et al. The Design and Development of Network Security Comprehensive Monitoring System[J]. Nuclear Electronics & Detection Technology, 2014, 4(4): 419-424.

[4] Guo Xiaohui, Li Runzhi, Zhang Qian, et al. Application research on distributed Zabbix network monitoring system[J]. Journal on Communications, 2013, 9(Z2): 94-98.

[5] Zhao Zhe, Tan Haibo, Zhao He, et al. Network Monitoring System Based on Zabbix[J]. Computer Technology and Development, 2018, 1(1): 144-149.

[6] Fan Yufeng, Xin Yuxin, Bai Jinming, et al. An overview of the BOOTES-4 at the Lijiang observatory[J]. Astronomical Research & Technology, 2015, 12(1): 78-88.

[7] Li Jian, Cui Chenzhou, Zhao Yongheng, et al. A secondary development of an RAO management system based on the Remote Telescope System-2nd version[J].

Astronomical Research & Technology—Publications of National Astronomical Observatories of China, 2013, 10(3): 264-272.

[8] Wei Shoulin, Chen Yajie, Liang Bo, et al. Methods of constructing new CCD classes in the RTS2[J]. Astronomical Research & Technology—Publications of National Astronomical Observatories of China, 2014, 11(3): 281-286.

Note: Figure translations are in progress. See original paper for figures.