M2M device redundant backup system based on ad hoc network system

Yichun Yu
School of Computer Science and Engineering
Beihang University
Beijing, China
yichunyu@buaa.edu.cn

Xiaoyi Yang*
School of Software
Beihang University
Beijing, China
xyyang001@buaa.edu.cn
*Corresponding author

Yuqing Lan*
School of Software
Beihang University
Beijing, China
Lanyuqing@buaa.edu.cn
*Corresponding author

Zhihuan Xing
School of Computer Science and Engineering
Beihang University
Beijing, China
xingzhihuan@buaa.edu.cn

Dan Yu
China Standard Intelligent Security
Beijing, China
yu_dan@csisecurity.com.cn

Han Zheng
School of Software
Beihang University
Beijing, China
zhenghan@buaa.edu.cn

Danxing Wang
China Standard Intelligent Security
Beijing, China
wang_danxing@csisecurity.com.cn

2024 IEEE 6th International Conference on Power, Intelligent Computing and Systems (ICPICS) | 979-8-3503-7431-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICPICS62053.2024.10796455

***Abstract*—With the rapid proliferation of IoT (Internet of Things) technologies, ensuring reliable and continuous operation of M2M (machine to machine) communications has become paramount. This paper presents the design and implementation of a redundant backup system tailored for M2M devices, utilizing an ad hoc network system. The proposed architecture is systematically divided into three layers: the physical layer, network layer, and application layer, each meticulously crafted to address the specific constraints and demands inherent to IoT devices. At the physical layer, considerations are given to the hardware specifications and energy consumption of M2M devices. The network layer focuses on establishing robust, self-organizing ad hoc network protocols to ensure seamless connectivity and data transmission between devices. Meanwhile, the application layer is designed to manage data redundancy, recovery processes, and system monitoring, ensuring that the backup mechanism operates efficiently under varying conditions. Through the integration of this multi-layered device redundant backup solution, the system guarantees continuous operation and enhances reliability in M2M scenarios. Extensive testing and simulation results demonstrate the effectiveness of the proposed system in maintaining data integrity and operational resilience, even in the face of network disruptions and hardware failures. This study offers a comprehensive framework for improving the robustness of M2M communications, contributing to the advancement of IoT technologies.**

Keywords—Ad Hoc Network, M2M, Redundancy

I. INTRODUCTION

In the era of the Internet of Things, ensuring the security and efficient management of numerous devices is a major challenge [1]. With the rapid increase in the number of IoT devices and the continuous expansion of application scenarios, device fault recovery and business continuity have become particularly important. Redundant backup solutions provide an efficient fault recovery method for M2M devices. By replicating and backing up device data, they can quickly recover when a device fails and ensure business continuity.

Existing redundant backup solutions are mainly used in resource-rich computing environments [3][4]. In the IoT environment, devices usually have limited resources, such as limited storage space, computing power, and power supply. This brings new challenges to the design and implementation of redundant backup solutions[5]. To solve these problems, this paper deeply explores the architecture design and implementation of self-organizing network systems. Through self-organizing network technology, automatic discovery and communication between devices are realized, reducing dependence on centralized control and management, and improving the flexibility and scalability of the system.

In this paper, we introduce the design of the system architecture in detail, including the functions and implementation methods of the physical layer, network layer, and application layer. The physical layer is mainly responsible for the hardware connection and basic communication of the device, the network layer provides data transmission and routing functions between devices, and the application layer implements specific backup and recovery operations. Each layer is designed for the unique limitations and requirements of IoT devices to ensure that the system can still operate efficiently in a resource-constrained environment.

By integrating device redundancy backup solutions, our system can automatically switch to backup devices in the event of device failure or communication interruption, ensuring operational continuity and reliability in M2M scenarios. This study not only provides a new solution for fault recovery of IoT devices, but also provides a useful reference for other application scenarios that require high reliability and high availability.

II. AD HOC NETWORK SYSTEM ARCHITECTURE AND DESIGN

Since most M2M devices have the characteristics of low performance, heterogeneity, and resource shortage [2], the system design should minimize the system resource usage and be able to access various heterogeneous IoT devices. As shown in Figure 1, the ad hoc network system is divided into three layers: physical layer, network layer, and application layer.

The physical layer is mainly used to adapt to IoT terminal

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on April 14,2025 at 09:53:27 UTC from IEEE Xplore. Restrictions apply.

devices of various communication methods. It can receive messages from communication methods such as Ethernet, Bluetooth, WiFi, etc., and pass them to the system for processing, so as to realize dynamic interaction between devices with different communication methods.

The main responsibilities of the network layer are unpacking, packet assembly, protocol conversion and data forwarding, which are used to transmit various data of the device and transmit control signals downward. The network layer designs and implements coroutine modules and network modules based on coroutine modules to provide efficient asynchronous programming methods, so that the system can make full use of the resources of a single thread and achieve high-performance network communication.

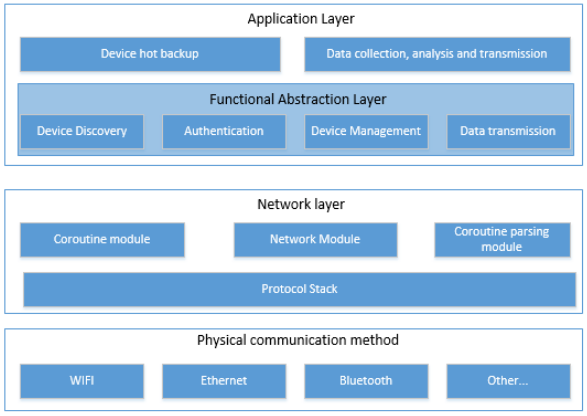

Fig. 1. Overall architecture diagram of self-organizing network system

The application layer includes functions such as device discovery, authentication, device management and data transmission. This layer supports business applications such as data collection, analysis, transmission and device hot backup, forming a complete IoT device management system.

III. Implementation Details

To meet the stringent requirements on resources, the system uses customized lightweight and efficient basic components to provide services for upper-layer business codes.

A. Coroutine Framework

In order to solve the problem of low support for multi-threading in the perception layer devices of the Internet of Things, this system implements a coroutine framework to avoid the overhead of frequent thread switching, so as to better support multi-threaded operations in the business. The traditional network framework is based on the multi-threaded model and is not applicable in the coroutine environment. Based on the coroutine framework, this system implements a customized network framework to support TCP and UDP connections with other devices. The network framework is optimized to efficiently handle network communications in a coroutine environment.

The overall coroutine framework can be divided into 4 layers, from top to bottom: coroutine API interface, two-layer scheduler, coroutine pool, system call layer, as shown in Fig. 2.

1) System call layer

The context switch is performed using ucontext.h of the GNU C library, and a Context class is encapsulated. This class contains the context structure and the corresponding stack space, and provides an upper-level interface to implement context transfer. From a design perspective, the Context class can be regarded as the smallest unit of context transfer, which corresponds one-to-one to the coroutine.

2) Coroutine Pool

The coroutine pool manages the coroutine objects in the system and is responsible for their lifecycle and scheduling. The coroutine object encapsulates the operation of the coroutine, is responsible for managing the context and state of the coroutine, and is the basic unit of scheduling and management. The operation of the coroutine object includes the yield, recovery and execution of the coroutine, and its state can be divided into ready state, running state, waiting state and dead state. The coroutine pool initially has 500 coroutine objects, and the capacity is doubled each time it is expanded to improve performance.

This system abstracts the coroutine object pool into a template class, uses deque as the underlying data structure, and manages idle objects and running objects respectively, as shown in Fig. 3. This system designs the CorManager class, which corresponds to each kernel thread one by one and is responsible for scheduling, running, giving up, and waking up the coroutine objects in the thread. The CorManager class contains components such as event scheduler, time scheduler, double buffer queue, and thread object.

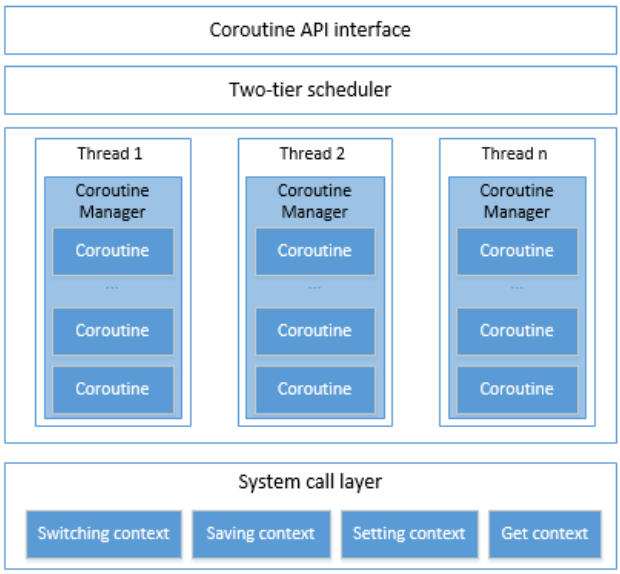

Fig. 2. Coroutine framework diagram

When CorManager is started, a thread object is created. This thread object is responsible for looping through the active coroutines and the ended coroutines in the current thread. Active coroutines are obtained in two ways: the time scheduler is responsible for waking up the timed coroutines, and the event scheduler uses epoll to listen to the IO events of the managed file descriptors and wake up the corresponding coroutines. For the executed coroutine objects, CorManager will set them to the dead state, initialize their context and other information, and then put them back into the coroutine object pool so that they can be reused the next time they are woken up.

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on April 14,2025 at 09:53:27 UTC from IEEE Xplore. Restrictions apply.

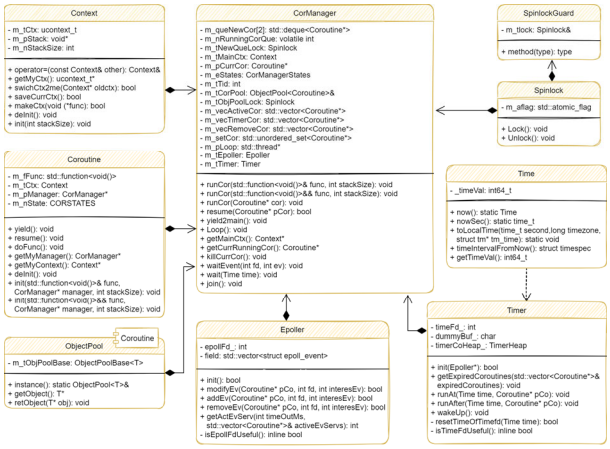

Fig. 3. Coroutine framework class diagram

3) Two-tier scheduler

The main object of the two-layer scheduler is CorManager, which is responsible for managing the life cycle of each CorManager. The main scheduler Schedule adopts the singleton mode. When it starts, it will create a corresponding number of CorManagers according to the number of kernel threads, and assign a unique ID to each thread for management. The main function of the second-layer scheduler is to schedule the coroutine to the corresponding CorManager for execution. This system implements the fair scheduling algorithm and the priority scheduling algorithm based on the operating system process scheduling strategy. For coroutine tasks that are not directly related, the scheduler will automatically assign them to a thread for execution to achieve parallel execution. In this case, coroutines in different threads can be executed simultaneously, improving the concurrent performance of the system.

4) Coroutine API interface

The coroutine API interface is the part that encapsulates the coroutine library. Its main purpose is to hide the details of the coroutine and only expose the control interface to the user. The API layer provides a series of interfaces, including initiating a coroutine, coroutine sleep, and obtaining the current CorManager.

B. Optimize IO efficiency based on Hook technology

IO-intensive operations in the Linux environment can cause performance bottlenecks, especially the frequent switching between user mode and kernel mode and the blocking problem in the data copy stage. The paper proposes an innovative method that uses Hook technology[6] to transform the original synchronous blocking system calls into asynchronous operations in the coroutine, and implements the interception and encapsulation of system calls through the dlsym dynamic library link function. This solution allows data copying to be processed in user mode, effectively eliminating the blocking in the data transmission stage, thereby improving the efficiency of IO operations, and is particularly suitable for optimizing system performance in resource-scarce scenarios.

C. Optimizing concurrency efficiency based on spin locks

In a multi-threaded environment, it is a difficult problem to effectively manage critical resources. Researchers usually use techniques such as mutex locks and atomic variables to control the access order of threads. However, the switching of mutex locks will cause the switching between user state and kernel state, and this overhead is unacceptable in a coroutine environment. A lightweight lock that can only protect critical resources in user state came into being. To achieve this goal, this system uses the atomic variable sub-implementation Spinlock and SpinlockGuard provided by C++11 to protect critical resources.

D. Optimizing coroutine creation overhead based on coroutine stack allocation strategy

Since the coroutine needs to record information such as CPU registers and program counters when executing tasks, each coroutine needs to have its own independent memory stack. At present, there are two coroutine stack allocation strategies: independent stack and shared stack, corresponding to stacked coroutines and stackless coroutines respectively.

Based on this, this section proposes a coroutine stack allocation strategy based on the memory pool that combines the two schemes. In view of the lack of flexibility, memory fragmentation, size definition of stackless coroutines and the high creation overhead and difficulty in unified management of stacked coroutines, this article uses the memory pool as the coroutine stack for scheduling. When the program starts, you only need to create memory blocks of different sizes and organize them in the memory pool in the form of a linked list. When the coroutine is created/destroyed, there is no need to allocate/release memory, but only need to return the memory to the memory pool for unified management by the memory pool.

E. Network module scheduling and optimization

The network library provides functions such as establishing and managing network connections, sending and receiving data, and processing network events. The network framework uses coroutines instead of thread pools at the bottom layer to handle the reception of network connections, and uses hook technology to replace intensive system IO operations to fully utilize the performance advantages of coroutines and achieve efficient network connection management.

The entire network framework can be divided from bottom to top: basic component layer, connection layer, and service layer, as shown in Fig. 4.

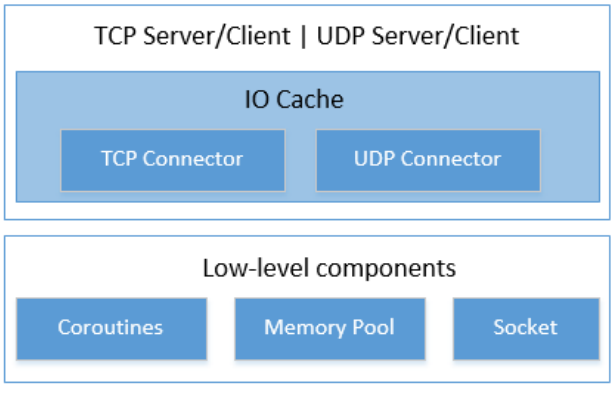

Fig. 4. Network framework diagram

The basic component layer includes three parts: coroutine framework, Socket class and memory pool. The coroutine framework provides the scheduling and management functions of coroutines. The Socket class is a component that encapsulates and manages network file descriptors. The

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on April 14,2025 at 09:53:27 UTC from IEEE Xplore. Restrictions apply.

memory pool is used to manage memory blocks. The second layer is the network connection layer, which inherits Socket and implements Tcp connector and Udp connector, and provides related data connection, sending and receiving interfaces.

IV. Equipment Redundancy Backup Design

This solution divides device identities into two categories: backup devices and deployment devices. Deployment devices will be responsible for specific services, including processing video streams, real-time alarms, algorithm processing, etc. Backup devices are not responsible for specific services, but only replace the work of backup devices when deployment devices fail.

Device redundancy backup organizes devices in a trusted domain, closely monitors the status of each device, and promptly detects device failures. At the same time, when the system starts, the configuration information of the device will be backed up to the root node, and the configuration information of the root node will be stored in the sentinel node. When a device fails and goes offline, the root device will select a backup device with a matching capability set to replace the offline device to complete its work, thereby achieving rapid switching of the device.

Configuration information mainly includes: the camera connected to the device, the algorithm module running on the device, and the alarm type being uploaded by the device. The following will introduce the device redundancy backup solution from four perspectives: inter-process communication, obtaining configuration information, updating configuration information, and configuration loading.

A. Inter-process communication

The smart station (hereinafter referred to as the station) in this experiment has a native MiniChallenge process, while the functions related to the ad hoc network system are in the Adhoc process, so the first problem faced is how to communicate efficiently and reliably between the two processes. In order to balance communication efficiency and security, we choose to use shared memory to exchange data between the two processes. To solve the data consistency and synchronization problems when multiple processes access shared memory, we use semaphores to protect the memory area and ensure the correct read and write order, thereby improving communication efficiency and data security, and ensuring the reliability and real-time performance of the device redundant backup system.

B. Get configuration information

In the device redundancy backup business process, devices in the trusted domain need to back up configuration information with each other. When a new device joins a trusted domain, it first undergoes two-step authentication and four-way handshakes to establish a trust relationship with the root node. Then the device identity will be determined: if it is a backup device, it can be directly added to the trusted domain; if it is a deployment device, its related configuration information needs to be saved.

Specifically, the MiniChallenge process first obtains the configuration information of the current device and writes it into the configuration file in the form of Json. Then the file is packaged and compressed, and then sent to the Adhoc process in the form of shared memory. Finally, the Adhoc process uploads the configuration file to the root node for backup, as shown in Fig. 5.

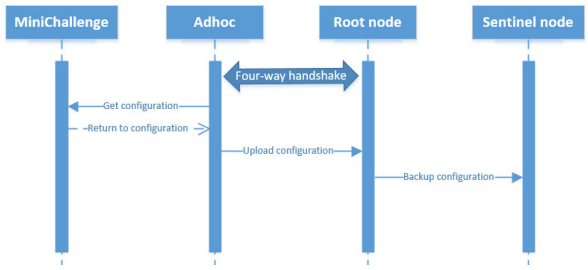

Fig. 5. Update configuration information

Considering that the root device may fail, resulting in the loss of all device configuration information, this article uses the sentinel device as a backup of the root device. That is, the configuration information of each device will be stored in two copies, one in the sentinel device and the other in the root device. In this way, no matter which device fails, there is no need to collect configuration information again, achieving the effect of rapid replacement.

C. Update configuration information

The configuration file includes all the information of the current device operation. Therefore, once the working content of a device changes, the configuration information needs to be updated. The process is similar to obtaining information, as shown in Fig. 6.

Considering the data consistency of configuration information, this article designs two synchronization strategies according to different scenarios: real-time synchronization and scheduled synchronization. For scenarios with low data consistency and real-time requirements, a scheduled synchronization strategy can be considered. This strategy is implemented by regularly collecting configuration files of updated devices in the trusted domain at predetermined time intervals to reduce network pressure and device burden and prevent redundant backup from affecting other business logic. For scenarios with strong data consistency requirements, a real-time synchronization strategy is required. In this strategy, when the configuration information of a device is updated, the configuration file must be synchronized to the root device and sentinel device immediately to ensure data consistency and avoid the risk of configuration loss.

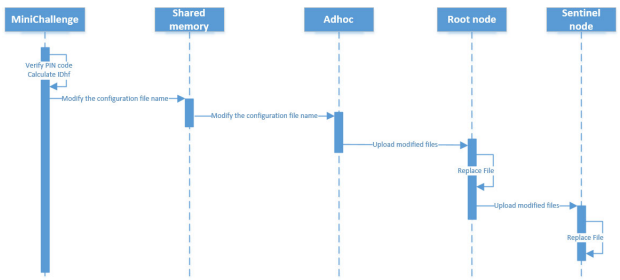

Fig. 6. Configuration modification and update flow chart

D. Load the configuration file

In the entire system, if a device's heartbeat is lost for a long time, the device will be judged to be offline and deleted from the trusted domain. Similarly, in the device redundancy backup solution, there are two ways to determine device

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on April 14,2025 at 09:53:27 UTC from IEEE Xplore. Restrictions apply.

failure. The first is that its parent device manages the online status (the root device is managed by the sentinel device). If it is offline, it is considered that the device has failed; the second is that the MiniChallenge process actively monitors the internal status of the device, such as memory usage, CPU load, etc. If an indicator is abnormal, it will actively mark itself as a faulty state.

When the root device finds that a device in the trusted domain has failed, it will actively delete the device from the trusted domain. If the device is a backup device, only the trusted domain topology needs to be adjusted; if the device is a working device, in addition to adjusting the topology, the backup device needs to be switched. That is, the root device will select a device with a matching capability set from the backup device and send the backup file of the failed device to the backup device.

If there is no additional backup device in the trusted domain, the root device will issue an early warning to the user, but will not merge the backup information of the failed device into a certain deployed device. This is because after the normal deployment device merges the configuration of the faulty device, insufficient resources may occur, causing the device to crash continuously.

V. Experimental Results

A. Application Verification

The following will take the intelligent safety monitoring scenario as an example to verify the performance, reliability and real-time performance of the equipment redundancy backup system in the M2M environment. In this scenario, the small station mainly serves as an edge computing node to connect and manage the IoT perception layer devices in the M2M scenario. The data information that the small station needs to store and backup includes: the camera managed by the device, the running algorithm module and the recording, storage, alarm, control information, etc. The average data file occupies 19KB of memory.

The system divides the small stations into two categories: the deployment small station is responsible for specific services, including processing video streams, real-time alarms, algorithm processing, etc.; the backup small station only replaces the work of the deployment small station when the deployment small station fails. The system will closely monitor the status of each small station, back up the data information of the small station and detect faults in time. When the small station fails and goes offline, the root device selects a small station with a matching capability set to replace the offline device to complete its work, realizing the rapid switching of the device.

The test environment is shown in Table I. Among them, the fault recovery time refers to the time required for the system to detect the faulty device and restore it to normal state. In the test, since the online status of ordinary devices is managed by the heartbeat mechanism of the ad hoc network system, and the heartbeat interval can be flexibly adjusted according to the actual situation, the test results do not include the heartbeat waiting time. As shown in Table II, under normal working conditions, manually triggering the failure of the main device (such as power failure or communication interruption), the average fault recovery time of ordinary devices is 174 milliseconds, while the average fault recovery time of the trusted root device is 432 milliseconds. In the event of a device failure, the CPU occupancy rate of the device redundancy backup strategy does not exceed 1.8%, and the memory occupancy rate does not exceed 2%.

TABLE I. System Operating Environment Configuration

Configuration	**Operating Environment**
CPU	12th Gen Intel(R) Core(TM)i7-12700CPU
Memory	32.0GB
operating system	64-bit Ubuntu Linux 20.04

TABLE II. Experimental Results

experimental project	Testing frequency	Mean time to restore faulty equipment	Data consistency
Common equipment	10	172 ms	100%
Root of Trust Device	10	432 ms	100%

B. Coroutine Framework Performance Comparison

In this section, Libco and Libgo are selected from the C++ coroutine library[7] for comparative testing with the coroutine library. The stack size is set to 64K. The creation time of 1, 1000, 2000, 4000, 8000, 10000, 20000, 40000, and 80000 coroutines are tested respectively. Google benchmark is used to test the performance, and the average value is taken after five tests. The test environment is shown in the Appendix, and the results are shown in Fig. 7. Our coroutine creation speed (0.75ms/1W) is about 40 times higher than Tencent Libco (34.10ms/1W) and about 30 times higher than Libgo (21.17ms/1W).

Framework implementation details lead to differences in coroutine switching speeds. First, the context switching rate is an important factor affecting the coroutine switching rate. Libco uses assembly operations to switch contexts, which is extremely fast, but not cross-platform; while our coroutine uses the ucontext library to switch contexts, and the call stack is longer than the assembly code, resulting in a slower switching speed, but its advantage is that it can be cross-platform. At the same time, Libco uses a large number of handwritten linked lists, timers and other components, and does not use frameworks such as STL commonly used in C++; Our coroutine uses a ready-made framework, and does not customize the development and optimization of data structures such as linked lists, which increases redundant functions and affects the switching speed.

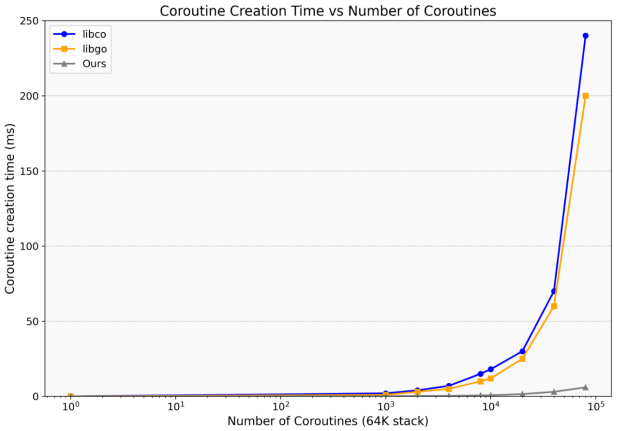

Fig. 7. Number of coroutines-Coroutine creation time

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on April 14,2025 at 09:53:27 UTC from IEEE Xplore. Restrictions apply.

The difference in coroutine creation speed is mainly caused by different memory allocation strategies. When creating stacked coroutines, Libco and Libgo will temporarily allocate corresponding stack space for each coroutine, with a default size of 2K. Our coroutine uses the object pool and coroutine pool strategy to allocate a large block of memory when the entire system starts. When a coroutine is created, only the corresponding context information needs to be saved, and the coroutine stack space is directly allocated from a large block of memory. When the coroutine is released, there is no need to return the memory directly to the operating system, but to return the coroutine to the coroutine pool. The next time a coroutine is created, it will be obtained from the coroutine pool first, avoiding the overhead of multiple memory allocation and release, and also avoiding problems such as memory fragmentation.

VI. Conclusion

This paper introduces a design and implementation of M2M device redundancy backup based on ad hoc network system. The system adopts modular design, combined with custom coroutine framework and robust network module to ensure high performance and low coupling, which is suitable for resource-constrained IoT devices. The integrated device redundancy backup solution further enhances the reliability of the system and reduces maintenance costs, which is suitable for large-scale IoT deployment.

References

[1] Laghari A A, Wu K, Laghari R A, et al. A review and state of art of Internet of Things (IoT)[J]. Archives of Computational Methods in Engineering, 2021: 1-19.

[2] Verma P K, Verma R, Prakash A, et al. Machine-to-Machine (M2M) communications: A survey[J]. Journal of Network and Computer Applications, 2016, 66: 83-105.

[3] Zhang Y, Zhu W, Feng D, et al. A fragmentation-aware redundancy elimination scheme for inline backup systems[J]. Future Generation Computer Systems, 2024, 156: 53-63.

[4] Walia G K, Kumar M, Gill S S. AI-empowered fog/edge resource management for IoT applications: A comprehensive review, research challenges and future perspectives[J]. IEEE Communications Surveys & Tutorials, 2023.

[5] Kamath V, Renuka A. Deep learning based object detection for resource constrained devices: Systematic review, future trends and challenges ahead[J]. Neurocomputing, 2023, 531: 34-60.

[6] Zhao Zhiheng, Yu Xiushan, Huang Song, et al. GUI test operation capture method based on Windows Hook[J]. Computer Engineering and Design, 2016, 37(3): 660-664.

[7] Ding Liaoyuan. Design and implementation of a coroutine-based web server prototype[D]. Dalian University of Technology, 2017.

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on April 14,2025 at 09:53:27 UTC from IEEE Xplore. Restrictions apply.