Design of Satellite Network Simulation Platform Supporting Distributed Controller

Haoxuan Lu[1,2], Dezhi Li[1,2], and Zhenyong Wang[1,2(✉)]

[1] School of Electronics and Information, Harbin Institute of Technology, Harbin, China
ZYWang@hit.edu.cn

[2] Songjiang Laboratory, Harbin Institute of Technology, Harbin 150001, China

Abstract. Satellite networks, integral to future communication, benefit from SDN's advantages. Combining SDN with satellite networks, especially leveraging distributed controllers, shows promise. To facilitate research, we propose a simulation platform. Integrating STK, Containernet, Docker, and ONOS, it supports scenario and topology simulation and distributed control. Python scripts drive STK for scenario creation, Containernet for SDN network setup, and ONOS for controller functionality. Tests validate platform efficacy, showcasing ONOS's robustness and Docker node compatibility. This platform fosters satellite network research with scalability and practical applicability.

Keywords: Network simulation platform · Satellite network · Software Defined networking · Distributed controller

1 Introduction

In recent years, the booming telecommunications sector, driven by the rapid development and deployment of 4G and 5G technologies, has significantly transformed our lives and societal dynamics. As we transition into the era of B5G and 6G, satellite communication has reemerged as a focal point of research, with projects like SpaceX's Starlink and Amazon's Project Kuiper showcasing the feasibility and value of large-scale low Earth orbit (LEO) satellite constellations. Satellite communication, once overshadowed by terrestrial mobile communication, is now regaining attention due to its potential to complement and enhance existing networks. With advancements in satellite technology enabling higher on-board processing capabilities and mature inter-satellite links, large-scale LEO satellite networks offer broad coverage, resilience against natural disasters, high-speed communication, and massive user capacity [1]. Moreover, the flexibility of Software Defined Network presents promising applications in satellite networks, simplifying device management, enhancing reliability, and facilitating upgrades and adjustments [2]. However, the deployment of SDN in satellite networks faces challenges [3], particularly concerning the reliability and scalability of centralized controllers [4]. In response, there is a growing need for distributed SDN controllers to ensure reliability [5],

H.-H. Chen and W. Meng (Eds.): WiSATS 2024, LNICST 605, pp. 26–41, 2025.
https://doi.org/10.1007/978-3-031-86196-3_3

scalability, fault tolerance, and load balancing in large-scale satellite networks. Furthermore, the high cost, maintenance challenges, and industry secrecy associated with satellite networks underscore the importance of developing simulation platforms to reduce costs, accelerate development, and facilitate testing. Designing a satellite network simulation platform that supports distributed control is thus crucial for advancing research and development in this field. In summary, the development of satellite communication, coupled with the integration of SDN and the need for distributed control, underscores the significance of designing and implementing a satellite network simulation platform to support research.

The paper proposes a satellite network simulation platform with distributed control, integrating STK, Containernet, Docker, and ONOS. It consists of three main components: scenario simulation using STK for constellation scenarios, topology simulation with Containernet, and a distributed ONOS controller cluster for network management. The platform facilitates research on constellation design, network performance, routing, and distributed control functionalities, with scalability to simulate virtualized Docker node connections. Functional verification and performance testing confirm the platform's effectiveness in simulating large-scale LEO satellite networks and supporting Docker nodes.

2 The Design of Simulation Platform

2.1 Overall Framework of Simulation Platforms

The overall design framework of the simulation platform is illustrated in Fig. 1, consisting of three main components. The Satellite Network Scenario Simulation component is responsible for creating satellite network constellations, calculating link parameters, and generating satellite network topologies. It is primarily based on the STK simulation tool and Python scripts. The Network Topology Simulation component constructs the satellite network topology, runs host and switch nodes, and conducts network simulation tests. It mainly relies on Containernet, Docker, and Python scripts. The SDN Distributed Controller component is tasked with building a controller cluster, obtaining network topology views, and performing control functions such as routing calculations. It is primarily based on ONOS.

The simulation platform described above enables simulation from satellite network constellation construction to network topology testing. It supports distributed controllers and can facilitate research in areas such as satellite network constellation design, satellite network performance simulation testing, satellite network routing, and distributed controllers. It is capable of simulating network layers and can simulate both virtualized Docker satellite function nodes and physical nodes, demonstrating good scalability. This platform holds significant research significance and practical value.

2.2 The Satellite Network Scenario Simulation Component

Using Python scripts to invoke the STK engine, the Satellite Network Scenario Simulation component creates satellite network constellations and conducts simulation calculations for ISL (Inter-Satellite Link) parameters. The obtained information regarding

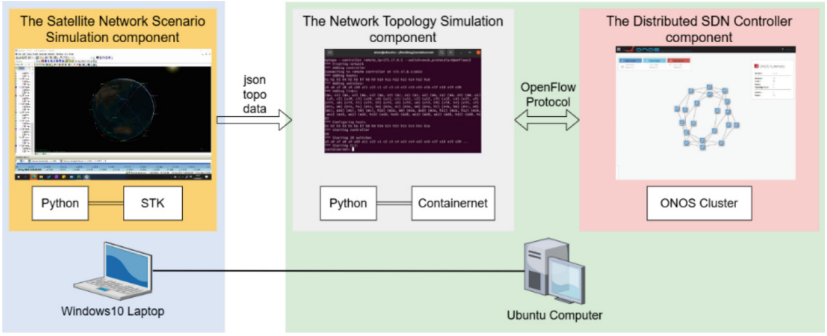

Fig. 1. The overall design framework of the simulation platform

satellite constellations and link parameters is utilized to generate a satellite network topology, which is then saved as JSON topology data and transmitted to the Network Topology Simulation component via Flask.

Using Python scripts to invoke STK for scenario simulation. Starting from version 11, STK introduced the capability to connect with Python, offering a new option alongside Matlab. It provides a rich Python API interface, enabling direct access to the STK Engine for scenario simulation and parameter calculations. Python also boasts abundant module resources, leading to higher development efficiency. In comparison to joint simulation with Matlab, although there are fewer reference documents due to its later introduction, it requires more in-depth research and investment, resulting in a higher learning curve. However, using Python scripts to call STK in satellite network simulation platforms offers greater flexibility, allows for programming language unification, facilitates platform integration, and avoids the limitations of Matlab.

2.3 The Network Topology Simulation Component

Using Python scripts, the satellite network topology JSON data is transformed into an SDN architecture network topology. This operates within the Containernet simulation tool based on Mininet, generating a simulated network environment consisting of satellite switches and nodes. These nodes can run Mininet hosts, virtualized Docker satellite function nodes, or physical nodes, and network performance testing is conducted using tools such as iperf or sflow. The network topology is connected to the SDN distributed controller component, which can retrieve the topology view of the simulated network and perform control actions.

To simulate the SDN satellite network, the satellite network simulation platform has chosen Containernet, a simulation tool based on Mininet. Containernet is a fork of Mininet, which is the most widely used simulation tool in SDN research. It offers more comprehensive support for SDN compared to other network simulation software and has the advantage of being able to connect with real devices. Containernet further enhances this capability by adding support for Docker, allowing containers to run as host nodes. This enables highly customizable node functionalities while improving operational and simulation efficiency.

To simulate satellite switching functionality within the SDN satellite network, the satellite network simulation platform has chosen to deploy OpenvSwitch virtual switches. OpenvSwitch is currently the most comprehensive multi-layer virtual SDN switch, boasting excellent scalability and programmability. It can be extensively deployed on Containernet through virtualization techniques, as illustrated in Fig. 2.

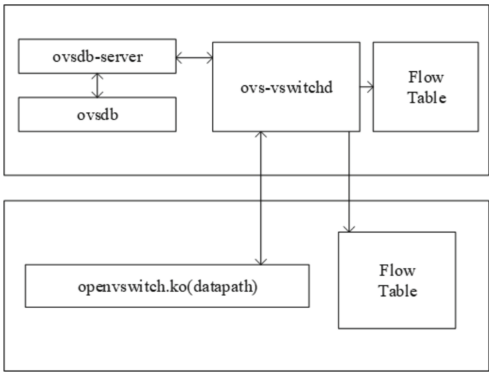

Fig. 2. OpenvSwitch structure

The SDN satellite network topology is created within Containernet using Python scripts. Upon receiving satellite network topology JSON data from the satellite network scenario simulation component, it generates the SDN satellite network topology. It creates satellite OpenvSwitch switch nodes and host nodes, then initiates the simulation network by invoking the APIs provided by Containernet and Mininet. Various network tests can be conducted within the simulation network. It is connected to the SDN distributed controller component, receiving control commands from the distributed controller.

With the Containernet tool, the design of the satellite network simulation platform supports the deployment of Docker nodes, enabling the simulation of Docker applications. Compared to ordinary host nodes, Docker nodes can implement a wider range of network functionalities, offering high flexibility to meet various research needs. This allows for the implementation of custom networks or protocol stacks.

2.4 The SDN Distributed Controller Component

Through comparison with other distributed SDN controllers like OpenDaylight, the design of the satellite network simulation platform has opted for the ONOS controller due to its excellent scalability and ease of development for research purposes. Being a project with collaborations from companies like Google, Amazon, and Microsoft, ONOS enjoys high recognition and practicality. Its modular structure facilitates customization and allows for the addition and removal of functionalities.

The architecture of the ONOS controller, as depicted in Fig. 3, interfaces with the southbound layer and connects to the simulation network running on Containernet using the OpenFlow 1.3 protocol. It communicates with the OpenvSwitch virtual switches,

retrieves the overall topology view of the simulation network, and issues network control commands, performing tasks such as route computation and flow table installation.

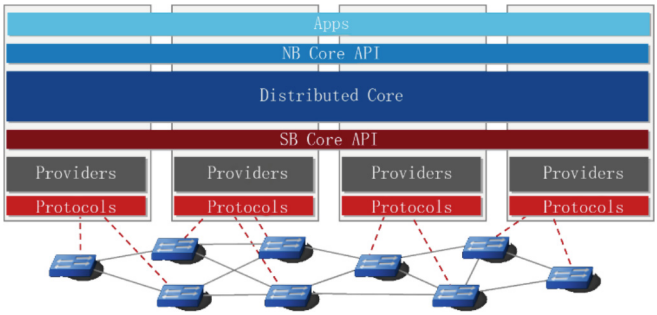

Fig. 3. ONOS structure

ONOS has undergone a series of version updates and has now iterated to version 2.7.0. Significant functional differences exist between versions, necessitating careful consideration based on specific needs. Regarding build tools, prior to version 1.7, ONOS controller compilation utilized Maven. In version 1.7, Buck was introduced as a compilation tool, while Bazel, adopted as the latest version's compilation method, replaced Buck in version 1.14. Differences in compilation tools affect the implementation of custom controller app functionalities, requiring the use of the corresponding version's compilation tool for construction. Developed by Google, Bazel boasts excellent scalability and robust functionality, with abundant reference materials available for research. Additionally, it has been the ONOS compilation tool of choice for a considerable period, thus selecting the ONOS controller based on Bazel compilation from version 1.14 onwards.

The current implementation of final consistency in the ONOS controller is achieved through Gossip, while strong consistency relies on Raft. Atomix, a distributed framework developed within the ONOS project, employs the Raft algorithm, offering better performance and higher availability compared to earlier versions which utilized Zookeeper and Hazelcast.

Regarding cluster deployment methods, starting from version 1.14, ONOS has transitioned to deploying controller clusters through independent Atomix clusters, as illustrated in the diagram. In contrast to the previous built-in approach, the data architecture of the distributed controller is now implemented through external Atomix, with ONOS internally retaining only replication protocol functionality, thereby achieving improved consistency and reliability.

3 Implementation of the Simulation Platform

3.1 The Simulation Process of the Platform

Based on the design proposed in the previous chapter, the implementation of a satellite network simulation platform supporting distributed control was conducted. The simulation process of the simulation platform, as illustrated in Fig. 4, consists of three

parts: satellite network scenario simulation, network topology simulation, and the ONOS distributed controller.

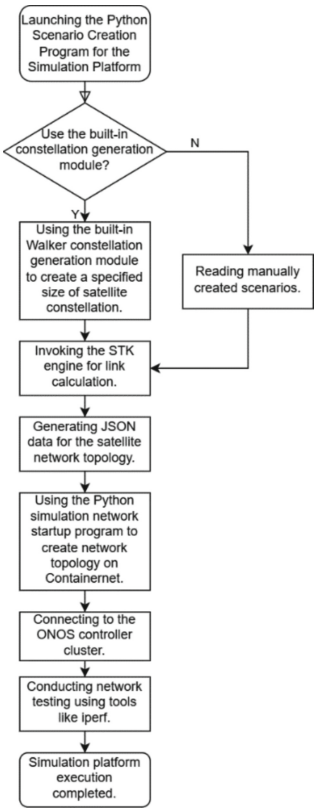

Fig. 4. The simulation process of the platform

The satellite network scenario simulation section utilizes Python programs to invoke the STK engine for the creation of satellite constellations. It completes the simulation calculation of ISL (Inter-Satellite Link) parameters and generates the satellite network topology based on the obtained satellite constellation and link parameter information. The resulting topology is saved as JSON topology data and transmitted to the network topology simulation section via Flask.

In the network topology simulation section, Python programs convert the satellite network topology JSON data into an SDN (Software Defined Network) architecture network topology. This runs on the Containernet simulation tool based on Mininet, generating a simulated network environment consisting of satellite switches and nodes. Nodes can run Mininet hosts, Docker virtualized satellite function nodes, or physical nodes. Network performance testing is conducted using tools such as iperf or sflow. The network topology is then connected to the SDN distributed controller section.

The ONOS distributed controller creates a controller cluster through Atomix, retrieves the switch and host node views generated by the network topology simulation section, and performs network management and control.

The satellite network scenario simulation section runs on a Windows 10 host to facilitate the use of the STK tool. The network topology simulation section and the ONOS distributed controller section run on Ubuntu 20.04 hosts or virtual machines. Virtual machines are used for development and testing, and Ubuntu hosts are utilized for higher simulation performance, thus enhancing simulation efficiency.

3.2 Implementation of Satellite Network Scene Simulation Module

The satellite network scenario simulation section is implemented through Python programming, comprising four modules: STK connection and scenario reading, Walker constellation creation, link calculation, and topology data generation. It invokes the STK engine to simulate satellite scenarios, generates network topology JSON data after link calculation, and transfers it to the network topology simulation section via Flask.

Two scenario options are provided: using the built-in Walker constellation creation module in Python programming or reading manually created satellite constellation scenarios with TLE data imported in the GUI. The number of ISLs (Inter-Satellite Links) supported ranges from 0 to 4, enabling connections between adjacent satellites in the Walker constellation.

Since version 11, the STK tool has supported integration with Python. Considering both stability and the richness of available references, version 11.6 was selected for implementing the simulation platform. The necessary library functions and modules, such as the Comtypes module for calling COM components, were configured through Anaconda. Following this, initialization Python scripts were used in the PyCharm environment to configure the connection with STK, importing the required STK modules into Python to ensure subsequent access to the corresponding API interfaces.

The built-in Walker constellation creation module in Python can generate a Walker constellation with specified parameters including orbit altitude, inclination, number of orbital planes, and number of satellites per orbital plane. Initially, within a newly created satellite scenario, a constellation is established. Satellites are then added to each orbital plane of the constellation following the Walker constellation pattern. The satellites are configured with Twobody propulsion and their initial states are set using classical orbital parameters, specifically the six orbital elements: semi-major axis, orbit altitude, eccentricity, inclination, right ascension of ascending node, and true anomaly. Once the constellation is established, the Propagate interface is invoked to compute the satellite orbits.

After creating or reading satellite constellation scenes using the built-in Walker module in Python, the link calculation module adds 0 to 4 inter-satellite links between satellite nodes and sets the link parameters. It then calls the STK engine to compute the status of the links, obtaining information such as distance, delay, energy, and error rate, which are saved in a dictionary and added one by one to a data list.

The topology data generation module saves the satellite network constellation and the obtained link information data in JSON format for the network topology simulation part to generate the simulation network. Using the Flask module, the JSON data is

sent to Ubuntu virtual machines or hosts running the network topology simulation part and the ONOS distributed controller part. In the Ubuntu virtual machines or hosts, the Flask module receives the data and reads it using the JSON module for generating the simulation network topology.

3.3 The Implementation of the Network Topology Simulation Section and ONOS Controller Section

The network topology JSON data generated by the satellite network scene simulation section includes information about satellite nodes and their connections. The network topology simulation section converts this data into an SDN network topology consisting of switches and hosts through a Python program, which is then run on Containernet.

The program utilizes Containernet's Topo class to transform the network topology. It employs the 'addSwitch' method to create OpenvSwitch virtual switches corresponding to satellite nodes and uses either the 'addHost' method or the 'addDocker' method to create host nodes for each satellite node, simulating users or ground gateway stations.

Once the topology creation is complete, the Containernet tool is invoked to start the simulation network topology, enabling network simulation tests to be conducted and connecting to the ONOS controller cluster.

In Python programs, the Containernet is invoked and the satellite network topology is constructed using the Topo class through the Python API interface inherited from Mininet. The Containernet method is then used to start the Containernet simulation network. This method theoretically can achieve all the functionalities of the 'mn' command start mode, but it requires calling interfaces provided in modules like the node class on your own. There is limited reference material available, and the official API documentation does not provide detailed explanations, so research and experimentation are needed to determine the usage method.

To start the network topology and establish connection with the controller in the upper Docker, the key aspects are the utilization of the RemoteController class and the addSwitch method. By specifying the RemoteController class and setting the address and port number of the ONOS controller node in Docker, normal controller connection can be achieved. By specifying the call to the OVSSwitch class and setting the OpenFlow protocol version to 1.3, correct deployment of OpenvSwitch is achieved.

During the implementation process, three methods of deploying the ONOS controller were attempted, including running it natively on the host machine, within a virtual machine, and inside Docker containers. Running the controller natively on the host machine proved inconvenient for creating a controller cluster, so it was not adopted. Running it within a virtual machine allowed for the formation of a cluster with multiple controllers, but the overhead of virtual machines was significant and severely impacted the performance of the simulation platform. Running it inside Docker containers allowed for rapid creation and configuration of the controller cluster, while incurring much lower performance overhead compared to virtual machines, thus enhancing the performance of the simulation platform.

Multiple versions of ONOS, ranging from 2.7.0 to 1.14.0, were attempted during the deployment process. Incompatibilities were encountered with versions like 2.7.0,

leading to the final selection of ONOS versions 2.1.0 and 2.2.2, which were able to fulfill the required functionalities while demonstrating good stability.

Within the deployed ONOS controllers, the openflow module is responsible for establishing connections with OpenvSwitch switches in the simulation network according to the OpenFlow 1.3 protocol. The fwd module handles routing decisions based on the obtained topology view, calculating and installing flow rules on the switches. The path visualization module combines the calculated routing paths with a classic GUI, presenting them within the topology view.

4 Platform Simulation Testing

After completing the implementation of the satellite network simulation platform supporting distributed control, the functionality and performance overhead were first tested. Subsequently, large-scale low earth orbit satellite network simulation tests were conducted to compare different scales of satellite networks. Following this, high availability testing was performed on the ONOS distributed controller cluster, and finally, testing was conducted on the deployment of Docker nodes.

Test Environment: The satellite network scenario simulation runs on a Windows 10 laptop with a 10th generation Intel Core i5 1065G4 processor (4 cores, 8 threads) and 16 GB of RAM. The network topology simulation and ONOS distributed controller components run on a Ubuntu host with an AMD Ryzen 5950X processor (16 cores, 32 threads) and 32 GB of RAM.

4.1 Functional Validation of the Simulation Platform

First, functional validation was conducted on the satellite network scenario simulation component. Through testing, it was determined that to ensure the visibility of inter-satellite links, the simulation platform requires a minimum number of orbital planes, set at 8. Additionally, a minimum of 8 satellites per orbital plane is required. Attempting to use parameters below these thresholds would result in errors in the link parameter calculation module. Testing involved creating a Walker constellation composed of 18 orbital planes, each containing 11 satellites, with an orbit altitude of 550 km and an inclination angle of 53°. The resulting satellite constellation scene is illustrated in Fig. 5.

After successfully creating the satellite constellation scene, the link calculation and topology generation modules were able to generate satellite network topology JSON data. The network topology simulation could convert the JSON data into an SDN satellite network topology and successfully initiate the simulation network on Containernet. The ONOS distributed controller component successfully started a controller cluster comprising three controller nodes. The network topology view obtained after connecting to the simulation network is illustrated in Fig. 6.

In Containernet, the ‘pingallfull’ command was used to conduct network connectivity tests and round-trip time tests between hosts, verifying the connectivity and link status of the simulated network. Simultaneously, tests were conducted on the routing calculation functionality of the ONOS distributed controller cluster. Results indicated that

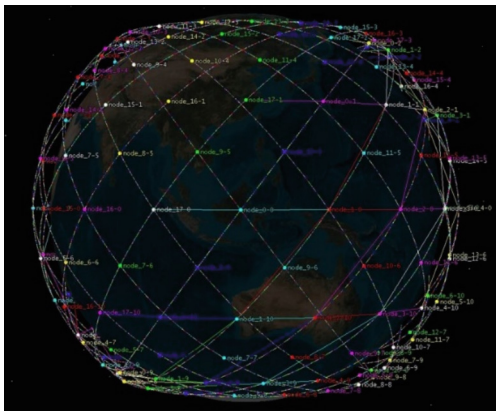

Fig. 5. Satellite constellation scene

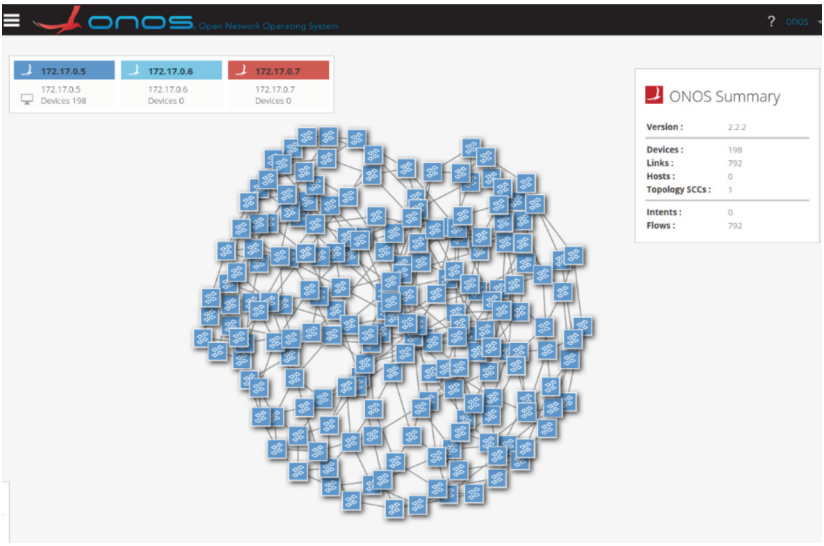

Fig. 6. ONOS topology view

all host nodes could communicate normally, the ONOS controller could perform routing calculations using the minimum path algorithm based on the acquired global view, and it successfully installed flow tables for OpenvSwitch switches. Graphical comparisons were made using the obtained data. Due to the symmetry of the network topology, analysis of the data from one node to other nodes was sufficient. The round-trip time curve graph, as depicted in Fig. 7, illustrated the differences in delay between nodes at different distances, confirming the authenticity of the network topology. The round-trip time between nodes and other hosts exhibited significant differences based on varying distances, aligning with the characteristics of a global low earth orbit satellite network.

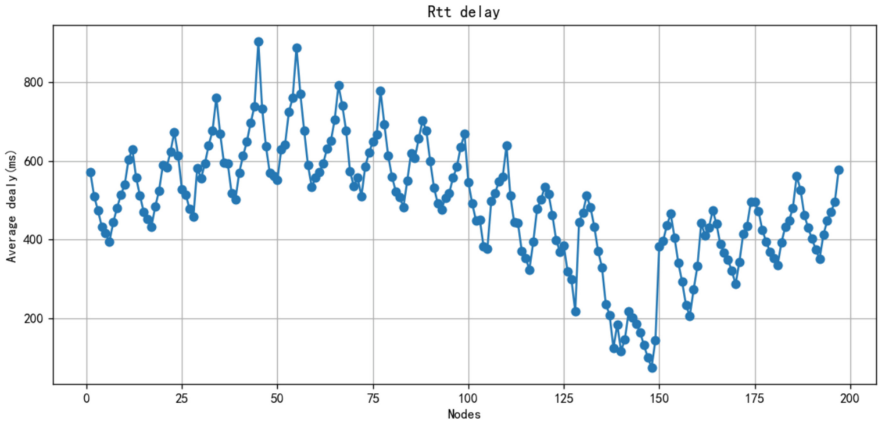

Fig. 7. The delay between nodes

To further ascertain the status of the links, the bandwidth of the links between hosts was tested using the iperf tool. Similarly considering the symmetry of the network topology, the results of analyzing and comparing the transmission bandwidth data from one node to other nodes are shown in Fig. 8. The data results confirm that parameters for all links have been correctly configured.

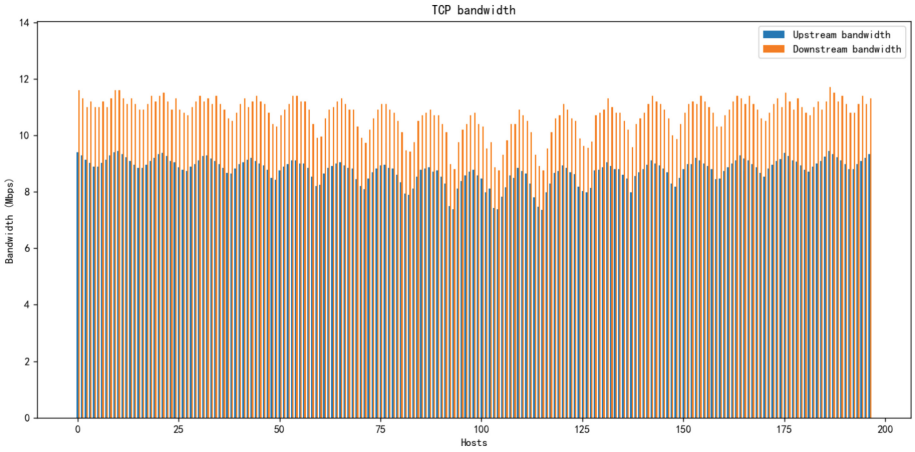

Fig. 8. TCP bandwidth

4.2 Analysis of System Error Characteristics

To validate the practicality of the simulation platform, performance overhead testing was conducted on simulation platforms of varying satellite constellation sizes, ranging from 8×8, 18×11, 18×22, 36×22 to 72×22 constellations.

Firstly, the CPU and memory usage during platform operation were tested. The CPU usage results are illustrated in Fig. 9.

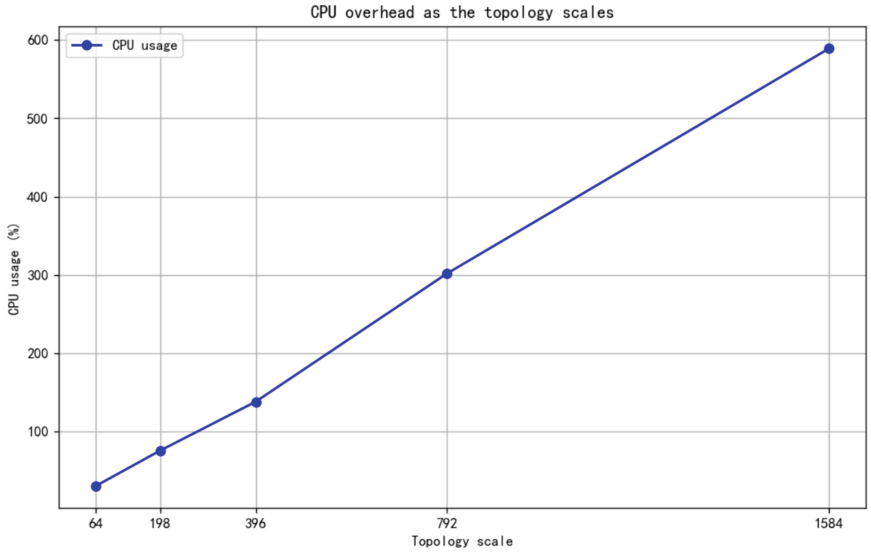

Fig. 9. CPU overhead as the topology scales

In the Ubuntu system, CPU utilization for each logical processor is calculated separately, thus for the test environment's host, the maximum CPU resource is 3200%, not 100%. Memory usage results are depicted in Figure Fig. 10.

CPU and memory usage include the performance overhead of OpenvSwitch switches, Containernet, and ONOS. Before the 36x22 topology, the Ubuntu simulation host ran smoothly without noticeable lag. Occasional lags were observed when simulating the 72 $\times$ 22 scale network topology, but simulation operations continued normally. Hence, on an Ubuntu host configured with a 5950X processor and 32 GB of RAM, it was possible to simulate a network topology with 1584 satellite nodes. For larger scale simulations, it is advisable to use more powerful hosts or implement optimizations.

Next, the time overhead of generating and launching the simulated network on Containernet was tested, with the results shown in Fig. 11. It was quickly completed when the scale was relatively small, but it took considerably longer for the 72 $\times$ 22 topology.

4.3 Comparison of Performance Tests for Different Scales of Satellite Constellations

The Starlink deployment process involves launching satellites one orbital plane at a time, gradually increasing the number of orbital planes. Additionally, different satellite constellations utilize various sizes and scales. Therefore, the simulation platform was utilized to conduct network performance tests on satellite constellations of different

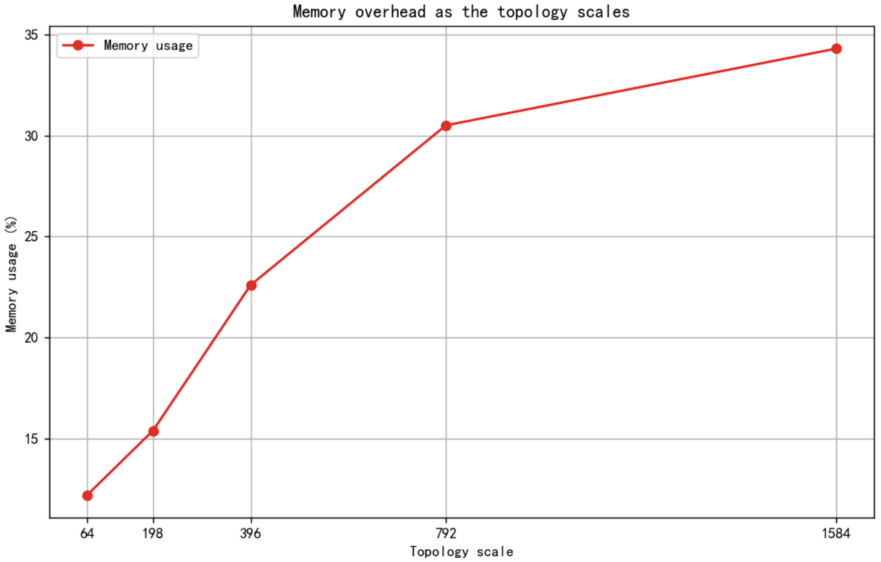

Fig. 10. Memory overhead as the topology scales

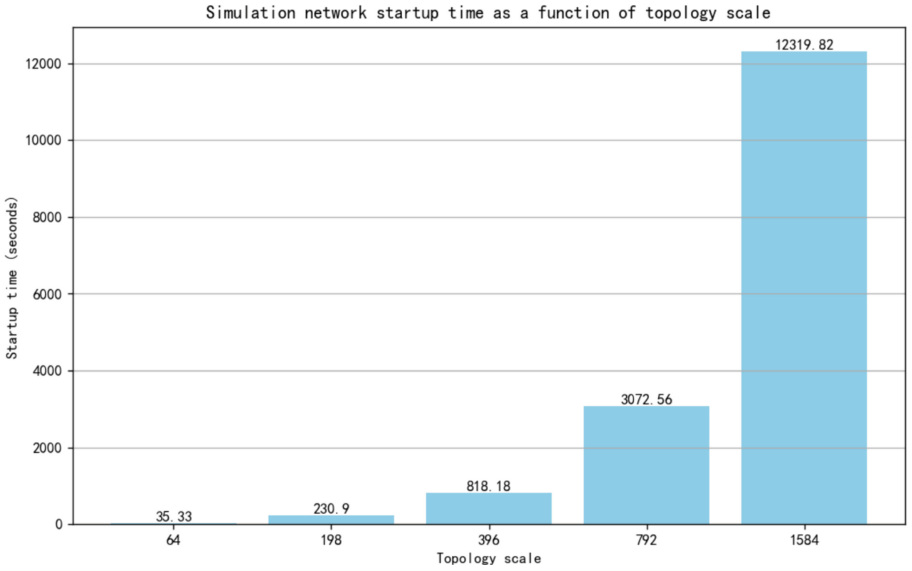

Fig. 11. Simulation network startup time as a function of topology scale

scales, including 8 × 8, 18 × 11, 18 × 22, 36 × 22, and 72 × 22 constellations, to facilitate simulation testing for deployment processes and different constellations.

The round-trip delay data obtained using the 'pingallfull' command on Containernet for constellations of different scales is illustrated in Fig. 12. It can be observed that as

the constellation scale increases, there is a noticeable increase in maximum delay. This is because with a larger scale, more satellite nodes imply the possibility of traversing more hops, resulting in increased delay at remote ends. However, this does not necessarily indicate a decrease in performance, as smaller satellite constellations have poorer coverage capabilities and cannot provide continuous service.

The transmission bandwidth data obtained through the use of the 'iperf' tool for different constellations on Containernet is depicted in Fig. 13.

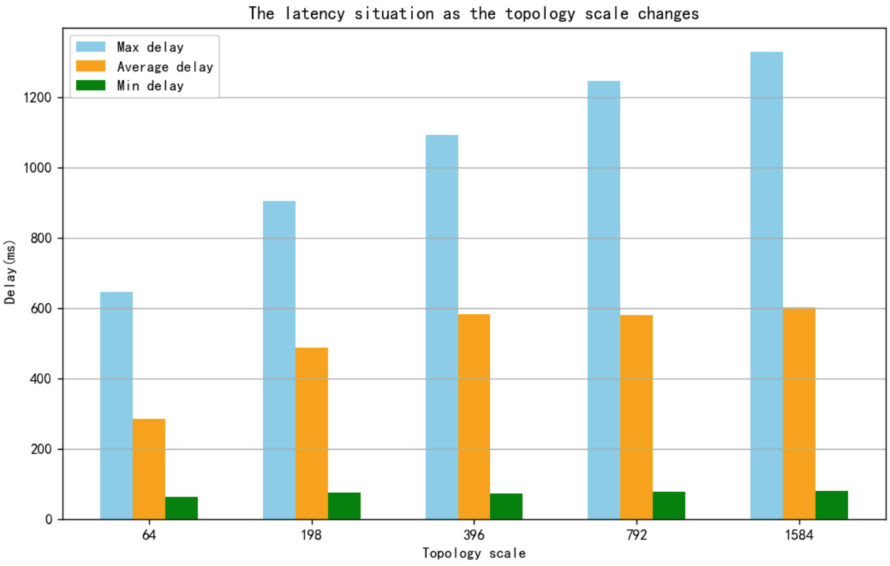

Fig. 12. The latency situation as the topology scale changes

It can be observed that there is no variation with the increase in satellite constellation scale. This is because the network topology tested only involved Container hosts and OpenvSwitch switches, without conducting tests involving the entire protocol stack or incorporating Docker nodes to enable additional network functionalities. Additionally, the number of hosts was relatively small, and congestion did not occur, resulting in consistent transmission bandwidth data.

4.4 High Availability Testing of ONOS Controller Clusters

The ONOS controller cluster is capable of providing backup functionality between controller nodes. Each switch maintains a standby list of backup controllers to connect to when connecting to the primary controller. In the event of a primary controller failure, the standby controller will take over the switches it was controlling, thus preventing network paralysis and ensuring continuous and stable service delivery to achieve high availability. Using a 72×22 topology, controller node backup takeover tests were conducted. Simulating the failure of the third ONOS controller, once it was taken offline,

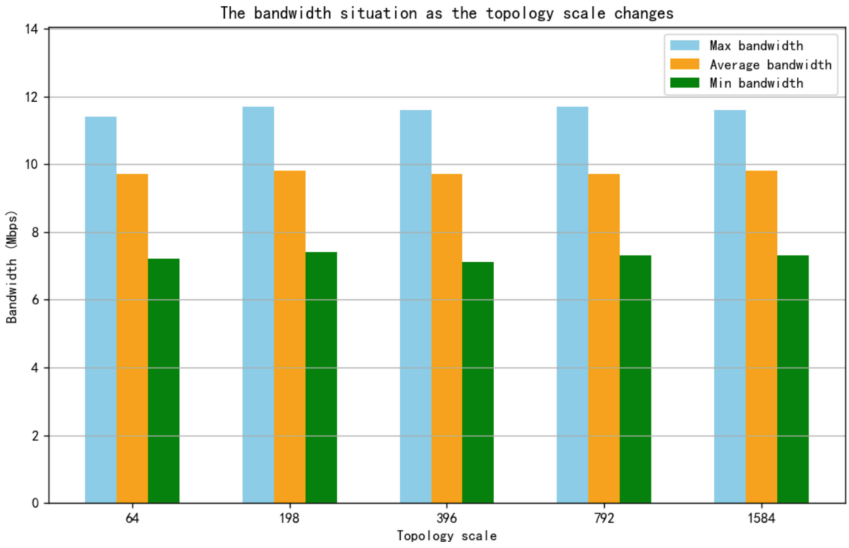

Fig. 13. The bandwidth situation as the topology scale changes

the remaining two controllers successfully took over control of the switches, as depicted in Fig. 14.

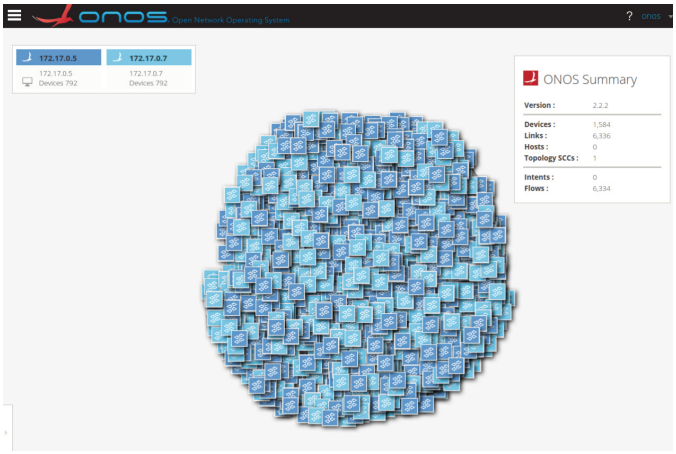

Fig. 14. The topology view after the takeover is completed.

5 Conclusion

This paper designs a satellite network simulation platform that supports distributed control, implements the simulation platform, and tests its functionality and performance. Furthermore, it conducts simulation tests on large-scale low Earth orbit satellite constellations using the simulation platform, validating the high availability of the ONOS distributed controller cluster.

Acknowledgment. This paper is supported by the research project fund of Songjiang Laboratory (No. SL20230104).

References

1. Zhan, Y., Wan, P., Jiang, C., Pan, X., Chen, X., Guo, S.: Challenges and solutions for the satellite tracking, telemetry, and command system. IEEE Wirel. Commun. **27**(6), 12–18 (2020)
2. Kreutz, D., Ramos, F.M.V., Veríssimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: a comprehensive survey. Proc. IEEE **103**(1), 14–76 (2015)
3. Kim, H., Feamster, N.: Improving network management with software defined networking. IEEE Commun. Mag. **51**(2), 114–119 (2013)
4. Xia, W., Wen, Y., Foh, C.H., et al.: A survey on software-defined networking. Commun. Surv. Tutorials IEEE **17**(1), 27–51 (2015)
5. Su, B., Zhao, L., Zhang, M., et al.: Research on the on-demand scheduling algorithm of intelligent routing load based on SDN. Int. J. Internet Protoc. Technol. **14**(1), 23 (2021)