

Article

MSCFS-RP: A Colored-Petri-Net-Based Analysis Model for Master–Slave Cloud File Systems with Replication Pipelining

Wenbo Zhou ^{1,2,3} 

¹ School of Information Science and Technology, Northeast Normal University, Changchun 130117, China; zhouwb646@nenu.edu.cn

² Guangxi Key Lab of Multi-Source Information Mining & Security (Guangxi Normal University), Guilin 541004, China

³ Key Laboratory of Applied Statistics of Ministry of Education (Northeast Normal University), Changchun 130024, China

Abstract: As a typical information system, a cloud file system enables the storage, retrieval, and management of data on remote servers or server clusters. The reliable design of such systems is critical to ensure the security of data and availability of services. However, designing correct-by-construction systems is challenging due to the complexity of and concurrency inherent in cloud file systems. Further, existing works on cloud file system analysis often focus on specific systems or lack formal modeling and verification, leading to potential design flaws and security vulnerabilities. To address these issues, we propose MSCFS-RP, which is a formal analysis model based on colored Petri nets. Leveraging the strengths of colored Petri nets in representing diverse information types with colored tokens and defining explicit rules for concurrent interactions, our model captures the writing and reading processes of clients, meta servers, and clusters. With strong formalism and support for verification using CPN Tools, we rigorously evaluate key properties such as replication consistency under various scenarios. The results demonstrate that MSCFS-RP satisfies these properties, validating its effectiveness and trustworthiness in managing information within cloud storage systems.

Keywords: cloud; information system; Petri net; formal analysis; replication pipelining



Citation: Zhou, W. MSCFS-RP: A Colored-Petri-Net-Based Analysis Model for Master–Slave Cloud File Systems with Replication Pipelining. *Electronics* **2024**, *13*, 2852. <https://doi.org/10.3390/electronics13142852>

Academic Editor: Claus Pahl

Received: 22 June 2024

Revised: 12 July 2024

Accepted: 15 July 2024

Published: 19 July 2024



Copyright: © 2024 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Many information systems leverage the capabilities of cloud computing to deliver scalable, on-demand access to data and applications, enhancing their functionality and enabling seamless integration of various components for effective data management and decision-making. Cloud computing provides on-demand and measurable services based on resource pools with rapid elasticity through the internet [1]. Nowadays, cloud computing has become a common choice for enterprises to conveniently develop their various applications [2–4]. As an important component in cloud environments, a cloud file system plays an irreplaceable role in supporting reliable storage services. A cloud file system is a type of information system that utilizes remote servers to store, manage, and process data, providing users with the ability to access their information from anywhere with an internet connection. The application of information techniques enables cloud storage systems to handle large-scale data and provide data security [5], thus meeting users' needs for data storage and access.

In cloud computing, the master–slave framework is one of the most typical distributed file system architectures. For instance, the Hadoop Distributed File System (HDFS) is a representative system with such an architecture [6]. A master–slave cloud file system usually consists of a meta server and a cluster, in which the meta server is responsible for managing storage nodes and providing query services to clients, while the cluster is made up of a series of storage nodes for actual reading or writing tasks. Such systems need to process a mass of clients' requests, manage states of nodes, and maintain complex

consistency. Therefore, safeguard mechanisms are very necessary to ensure reliable data storage. Replication is a simple and common way to prevent information loss and recover data from node faults. In particular, replication pipelining adopted by HDFS is one of the most representative replication mechanisms [6,7]. The replication mechanism can be viewed as a method of creating redundant copies of information to improve reliability and availability. To further lay a foundation for such systems and release the power of this mechanism, it is necessary to construct a generic and abstract model for a master–slave cloud file system with a replication pipeline.

Correct-by-construction is one of the most effective ways to ensure the reliability and security of software systems from the source. It usually needs to construct formal models [8,9] and then verify or test them before further implementation. There are many advantages to doing it this way. On the one hand, simple errors can be found and corrected at an early stage, avoiding them being developed into complex and difficult faults that are very costly to diagnose and repair. On the other hand, formal system models (particularly, executable models that can be directly simulated) benefit from various proving or model checking techniques and explore more design options and possibilities for verification [10].

To construct such an abstract and executable model, the formal method to adopt must be appropriate and with good applicability. As one of the most widely used formal methods, colored Petri nets (CPNs) combine Petri nets with a programming language to obtain a scalable modeling language for concurrent systems [11]. They have been used to model and analyze many critical information systems as well as to assist in the generation of new methods [12–19], including various industrial applications [20]. One of the most important advantages of a CPN is good visualization, giving users an intuitive impression. The other advantage is that CPN models are executable, allowing developers to explore more scenarios and validate critical properties [20]. Moreover, mature support tools, e.g., CPN Tools [21], inject much power for practical modeling, simulation, and verification. Therefore, we use CPNs to model and analyze the target system in this paper.

In this paper, we propose a CPN-based analysis model for a master–slave cloud file system with replication pipelining called MSCFS-RP and analyze some important properties with the help of CPN Tools. We model the writing and reading processes for the client, meta server, cluster, and node, and we verify nine properties (i.e., deadlock freedom, writing correctness, reading correctness, mutual exclusion, writing once, minimal distance, replication consistency, writing robustness, and reading robustness) under different configurations. Note that we focus on an abstract and executable model from a design level but not the real implementations. It is assumed that the reader has basic knowledge about colored Petri nets (i.e., multiset, place, transition, etc.), and more precise definitions of syntax and semantics of CPN can be found in [20]. Our constructed CPN model can be found in [22]. The contributions of this paper are as follows.

- A generic framework for a master–slave cloud file system with replication pipelining (MSCFS-RP) is considered.
- A formal and executable analysis model with high concurrency for MSCFS-RP is constructed using CPNs.
- Several critical properties, such as replication consistency, are analyzed for different scenarios with CPN Tools.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 briefly presents the preliminary knowledge about CPNs. Section 4 introduces the MSCFS-RP architecture. Section 5 presents our CPN-based model for MSCFS-RP. Section 6 verifies important properties using CPN Tools. Finally, Section 7 concludes this paper and outlines future work.

2. Related Work

In this section, we present a brief review of relevant research focusing on two aspects: system modeling using CPNs and formal verification of distributed file/storage systems. CPNs are highly respected for their capability to comprehensively model complex systems;

they leverage hierarchical structures and colored tokens to describe diverse system states and behaviors. Meanwhile, formal verification of distributed file and storage systems employs rigorous mathematical techniques to guarantee compliance with specified functional and non-functional requirements, covering critical aspects such as reliability and fault tolerance. This summary emphasizes the crucial role of CPNs and formal verification methodologies in ensuring the reliability and security of complex distributed systems.

2.1. System Modeling Using CPNs

CPNs are usually used in system design and analysis due to their modeling power and tool support. In [12], collaborative safety-critical systems were modeled and analyzed using hierarchical colored Petri nets, where an automated human rescue robot system was taken as a case study and specified failures were injected to check the safety in unexpected scenarios. In [13], a colored Petri net was constructed to model an ETC control system, and a dynamic slicing method of CPN was presented to more efficiently analyze and modify the system model. In [14], a timed CPN model was proposed to evaluate the service composition in multicloud environments while minimizing the number of clouds involved for serving a composite service request. In [15], an online firmware functional validation scheme was proposed using automatically generated colored Petri net models, and simulation runs of the CPN presented the firmware execution path for identifying potential locations of error. In [16], an approach for qualitative and quantitative safety evaluation of the CTCSS-3 using stochastic colored Petri nets was proposed, considering scenarios, movement of trains, failures, and repairs of components. In [17], temporal role-based access control was modeled using hierarchical timed colored Petri nets and analyzed with CPN Tools. In [18], a framework using colored Petri nets to assess the energy consumption of LPWAN-based Internet of Things systems was proposed, with a particular emphasis on the CPU and communication transceiver. In [19], a timed colored Petri net (TCPN) for temporal performance evaluation of distributed control systems through a FIFO queue was presented, and the consistency of bounds in a numerical application of the TCPN model was tested.

In contrast, MSCFS-RP focuses specifically on advancing modeling and verification for cloud file storage systems. While previous studies focus on specific applications such as safety-critical systems, automatic controls, and service composition, MSCFS-RP introduces a framework tailored for cloud file storage systems. Existing models often lack scalability or fail to adequately address the fault tolerance and specific challenges of distributed file systems. MSCFS-RP enhances these aspects through its robust replication pipelining mechanism, which promises improved reliability in cloud storage environments. By focusing on these critical aspects, MSCFS-RP provides a robust solution that ensures efficient data replication and maintains system reliability under diverse operational conditions, distinguishing it from previous models.

2.2. Formal Verification of Distributed File/Storage Systems

Formal verification applies mathematical techniques to systems, ensuring the satisfaction of properties related to functional or non-functional specifications. Several typical distributed file systems have been verified with different formal methods, e.g., rewriting logic, process algebra, and Petri net.

Rewriting logic is a suitable framework with good expressiveness and applicability for formally specifying and analyzing cloud storage systems [10,23]. In [24], an executable formal model of Megastore in Real-Time Maude was described, together with simulation for QoS estimation and verification of functional correctness. In [25], a formal executable model of Cassandra was presented in Maude, including formally modeling its main components as well as design strategies, and consistency properties were checked under various communication latency and consistency combinations. Compared with these works, our work does not focus on a specific system but a more generic model.

Process algebra uses algebraic approaches to model and analyze distributed, concurrent, or parallel systems. Yin et al. focused on the internal design of the Haystack architecture and used communicating sequential processes (CSPs) to model a browser, web server, directory, CDN, cache as well as storage, and they verified crucial properties, including basic properties and supplementary properties, with the model checker Process Analysis Toolkit (PAT) [26]. Xie et al. used CSPs to model a Hadoop Distributed File System, including reading files, writing files as well as the heartbeat mechanism, and they verified important properties with the help of the tool PAT [7]. Toward another distributed file system, Fang et al. modeled a Moose File System with CSPs, especially for reading and writing operations, and analyzed relevant secure properties using the model checker Failures Divergence Refinement [27]. Since we use colored Petri net, our work is more comprehensive as a benefit of good visualization. In addition, our model is more parameterized, e.g., it is easy to set the number of replications by only adjusting one parameter.

Petri net is an effective way to verify concurrent systems. In [28], a predicate Petri net model of a microservice-based cloud application was proposed, and a microservice reliability strategy using instance redundancy and circuit breakers was introduced. In [29], an approach combining a stochastic Petri net model with a cloud scoring system was proposed, where the Petri net model evaluates the availability of cloud application deployments, and the scoring system selects the optimal high-availability-aware deployment. In [30], a modeling and simulation framework based on a hierarchical colored generalized stochastic Petri net for evaluating the service reliability of cloud data centers was proposed, and the authors considered the three aspects of connectivity, performance, and service delivery. In [31], a colored-Petri-net-based formal model of Cache-A Replica On Modification (CAROM) was presented and verified through state space analysis and simulation, demonstrating improved performance in terms of storage cost and bandwidth latency.

As shown in Table 1, different from the above works that focus on cloud applications or cloud data centers, our work considers a generic replication pipelining mechanism and verifies some important properties under such a mechanism. This approach ensures a more comprehensive analysis of the data replication process, which is crucial for maintaining data consistency, fault tolerance, and system reliability in distributed file systems. By addressing these fundamental aspects, our MSCFS-RP model enhances the understanding of replication dynamics and provides a robust framework for ensuring the reliability and fault tolerance of distributed storage systems. This broader focus allows our work to apply to various types of cloud storage systems, offering potential improvements over the more specialized models discussed in previous studies.

Table 1. Comparison of formal verification methods for distributed file/storage systems.

References	Methodology	Key Features	Limitations
[10,23–25]	Rewriting logic	- Expressiveness and applicability for cloud storage systems - QoS estimation and functional correctness verification	- Specific to particular systems (e.g., Megastore and Cassandra) - Lack of generalizability to diverse distributed file systems
[7,26,27]	Communicating sequential processes	- Modeling and analysis of concurrent systems - Verification of basic and supplementary properties	- Focuses on specific file systems (e.g., Hadoop and Moose File System) - Challenges in scalability and parameterization

Table 1. Cont.

Reference	Methodology	Key Features	Limitations
[28]	Predicate Petri nets	- Reliability strategies for microservice-based applications	- Limited scope beyond microservices
[29]	Stochastic Petri nets	- Evaluation of cloud application availability and high-availability deployment	- Complexity in modeling connectivity and performance
[30]	Colored stochastic Petri nets	- Evaluation of cloud data center service reliability	- Focus on connectivity, performance, and service delivery metrics
[31]	Colored Petri nets	- Modeling and simulation of Cache-A Replica On Modification - Optimization of storage costs and bandwidth latency	- Specific to Cache-A Replica On Modification
This paper	Colored Petri nets	- Comprehensibility and visualization benefits - Parameterization flexibility (e.g., adjustable replication number) - Addresses generic replication pipelining mechanisms	- Potential challenges in performance optimization

3. Preliminary Knowledge

A colored Petri net is a graphical modeling language used for the design, specification, and analysis of systems where concurrency, communication, and synchronization are crucial [20]. Each token in a CPN represents a value (or color) and can have complex data types. The various data types, hierarchical structures, and support tools allow CPNs to model systems at a higher abstraction level, making them suitable for various applications, including communication protocols, distributed systems, workflow management, and manufacturing systems. The graphical nature of CPNs, combined with their mathematical foundation, provides a powerful tool for both qualitative and quantitative analysis of system behaviors [20]. We introduce the four core definitions of a CPN as follows.

Definition 1 (Colored Petri net [20,32]). *A colored Petri net is a tuple $CPN = (P, T, A, \Sigma, V, C, G, E, I)$, where*

- P is a finite set of places;
- T is a finite set of transitions, and $P \cap T = \emptyset$;
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs;
- Σ is a finite set of types, each one being a non-empty “color set”;
- V is a finite set of typed variables, i.e., $type(v) \in \Sigma$ for all $v \in V$;
- $C : P \rightarrow \Sigma$ is a coloring function that associates a type to each place;
- $G : T \rightarrow Expr_V$ is a labeling function that associates expressions (called guards) with free variables from V to transitions such that $type(G(t)) = Bool$;
- $E : A \rightarrow Expr_V$ is a labeling function that associates expressions with free variables from V to arcs such that $type(E(a)) = C(p)_{MS}$, where p is the place connected to the arc a , and $C(p)_{MS}$ is a multiset type;
- $I : P \rightarrow Expr_{\emptyset}$ is an initialization function that associates an expression without free variables to each place such that $type(I(p)) = C(p)_{MS}$.

Definition 2 (Marking [20,32]). *Given a CPN, a marking is a function M mapping p to multisets of tokens such that $type(M(p)) = C(p)_{MS}$. The initial marking is assigned by I .*

Definition 3 (Enabled [20,32]). Given a marking M , $t \in T$, its preset $pre(t) = \{p | (p, t) \in A\}$ (respectively, its postset $post(t) = \{p | (t, p) \in A\}$) and a binding b of its variables ($\langle b \rangle$: evaluation in b), (t, b) is enabled for M if $M(p)$ satisfies (i.e., is greater than or equal to) $E(p, t)\langle b \rangle$ for all places $p \in pre(t)$ and $G(t)\langle b \rangle = \text{true}$.

Definition 4 (Firing [20,32]). Given $t \in T$ and a binding b , if (t, b) is enabled for M , t may fire and yield a new marking M' , defined for all $p \in pre(t) \cup post(t)$ by $M'(p) = M(p) - E(p, t)\langle b \rangle + E(t, p)\langle b \rangle$, where $E(p, t)\langle b \rangle = 0$ (respectively, $E(t, p)\langle b \rangle = 0$) for every $p \notin pre(t)$ (respectively, $post(t)$). Here, $-$ (respectively, $+$) indicates the subtraction (respectively, the sum) of multisets [20].

Figure 1 illustrates an example in CPN Tools. The textbox with a green background represents the current marking of each place. Before firing transition T , place $P1$ holds the multiset “3’2 ++ 4’1 ++ 1’5” (and $P2$ holds “1’6”), indicating three occurrences of the value 2, four occurrences of 1, and one occurrence of 5. If the value “5” is randomly chosen to bind to variable x , the guard “ $x > 2$ ” is satisfied, enabling T to fire. Post-firing, T produces 1’15 into $P2$ as per expression $E(T, P2)$. For precise definitions of semantics, readers are encouraged to refer to [20], which explains how CPN models execute and effectively model behaviors of concurrent systems. This highlights their applicability in analyzing distributed systems such as cloud file systems with replication pipelining.

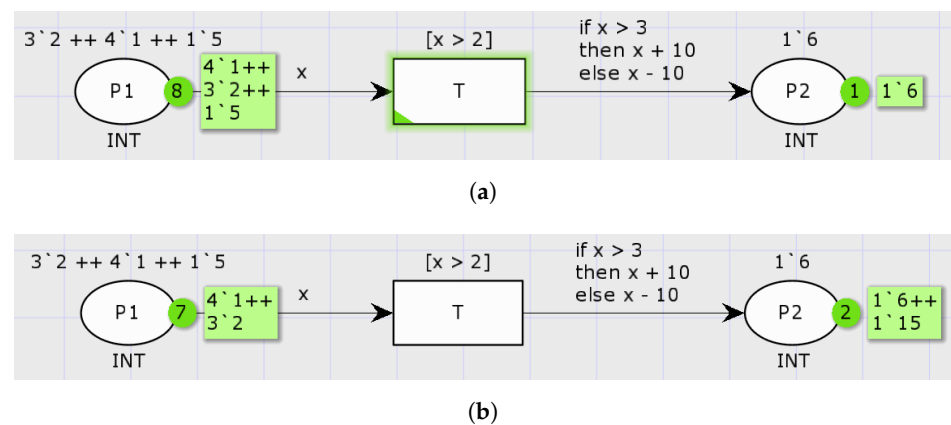


Figure 1. An example in CPN Tools. (a) Before firing. (b) After firing.

4. Master–Slave Cloud File System with Replication Pipelining

In this paper, we consider a cloud file system architecture as shown in Figure 2. There are three main roles in such a system, i.e., client, meta server, and cluster. Through their communication and coordination, reading or writing tasks can be successfully completed. The roles and functions of different entities are outlined as follows.

- **Client:** A client controls the generic processing of reading or writing requests proposed by multiple tenants.
- **Meta server:** A meta server provides locations for requests about where to write and read as well as manages other meta information and some synchronization processes, e.g., informing, acknowledgment, and confirmation.
- **Cluster:** A cluster consists of a series of nodes, each of which actually stores tenants’ data independently.

The most critical operations in such a file system are reading and writing, as these are fundamental to the system’s functionality and performance. Efficient and reliable handling of these operations ensures that users can swiftly access and store their data, which is essential for maintaining the integrity and responsiveness of the cloud file system. The mechanisms for reading and writing are designed to optimize data retrieval and storage while ensuring data consistency and redundancy, which are vital for providing a robust and scalable cloud storage solution.

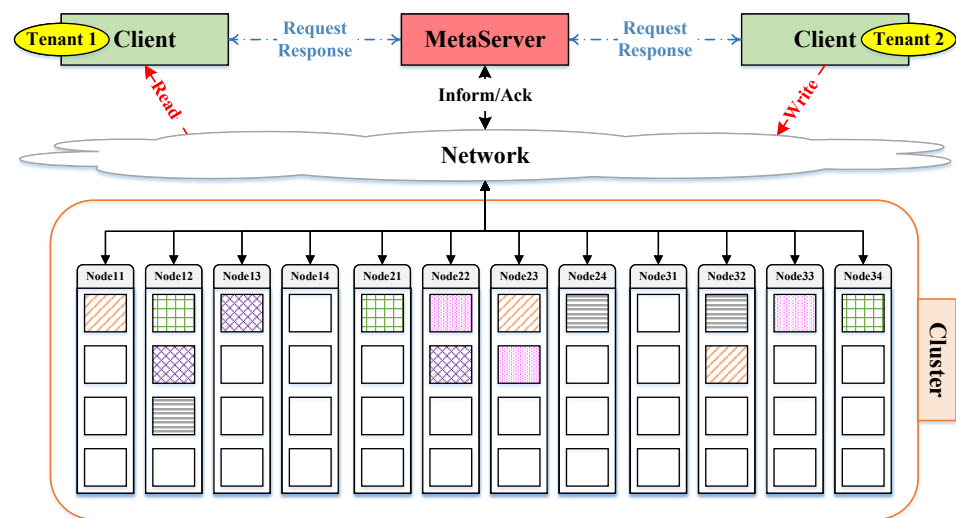


Figure 2. Master–slave cloud file system architecture.

- **Reading scenario:**

1. A client first sends a reading request to the meta server.
2. The meta server queries the location information about where the file to read is stored and informs the corresponding node.
3. After having received an acknowledgment from the node, the meta server sends a response with the corresponding location to the client.
4. The client communicates with the corresponding node according to the location, and the node returns the data required by the client.
5. After receiving the data to read, the client confirms it with the meta server. The reading process is shown in Figure 3.

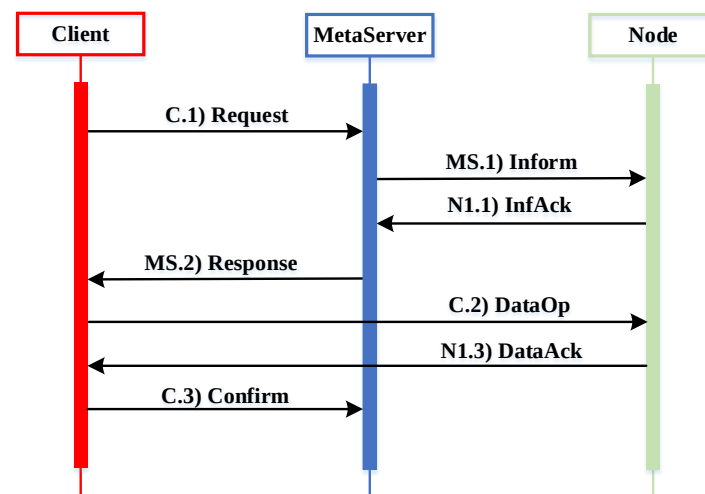


Figure 3. Reading process.

- **Writing scenario:**

1. A client first sends a writing request to the meta server.
2. The meta server informs each node about where to receive and send data and forms a link (see the orange rectangle called “Replication Pipelining” in Figure 4). In Figure 4, we set the number of replications to three. Thus, for each file, three nodes to store it will be informed.

3. After receiving the acknowledgments from the corresponding nodes, the meta server responds to the client and sends it only the location of the node closest to the requested tenant, hiding the locations of other replication nodes.
4. The client writes to the location, and then the writing data will be pipelined to the next location recursively.
5. When the pipeline writing is finished, the client receives a response. Finally, the meta server obtains a confirmation from the client.

Since we follow the write-once-read-many access model, if the writing file does not exist, the meta server assigns new locations (i.e., allocates new blocks) and creates such a file. Otherwise, the meta server abandons the request. Here, a “location” is a pair (*nid*, *bid*), where *nid* denotes a node identifier and *bid* denotes a block identifier. Each node may have many blocks, and such a location uniquely determines block data stored in a node.

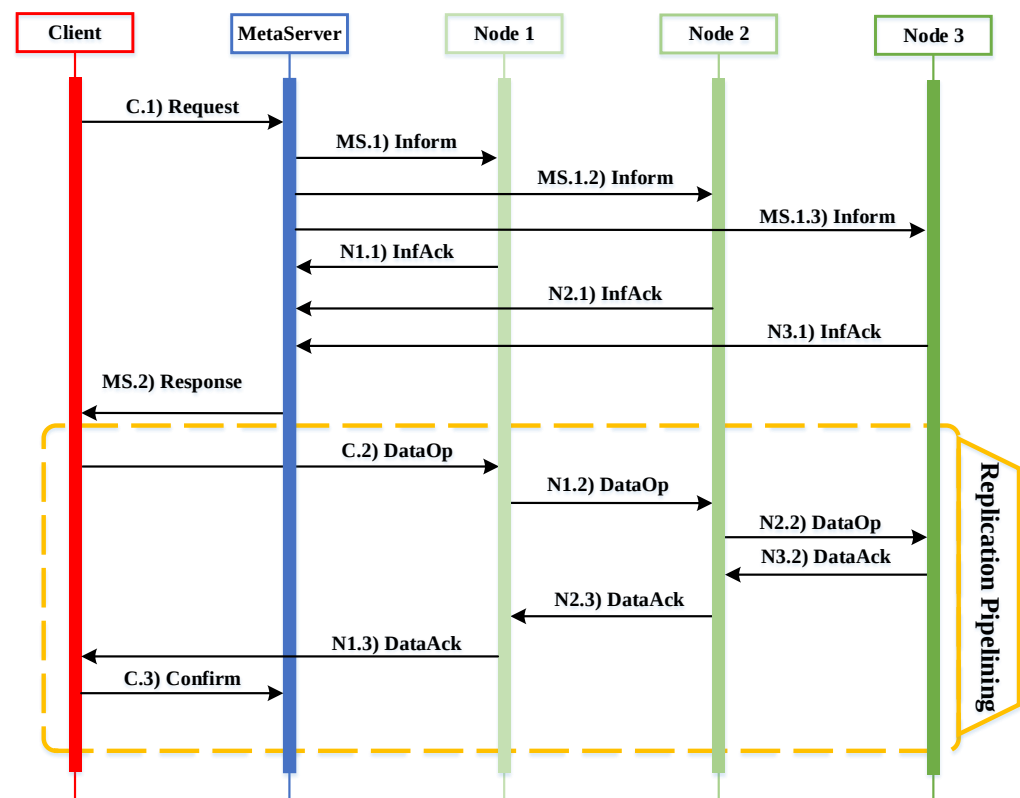


Figure 4. Writing process.

5. Modeling MSCFS-RP Using CPNs

CPNs offer a powerful tool for modeling the information flow within cloud storage systems. Different colored tokens can be used to represent different types of information, such as data blocks, requests, and responses. The flow and transformation of information between clients, meta servers, and clusters can be clearly illustrated. Combining information theory concepts, places and transitions in the CPNs simulate fault tolerance mechanisms, with a specific focus on redundant storage.

5.1. MSCFS-RP Model

As shown in Figure 5, components and their interactions are visually represented through places, transitions, and tokens. Places are represented by circles and denote states within the system. Transitions are represented by rectangles and indicate events or changes in the system. Tokens are often within places and signify the presence of an entity or the fulfillment of a condition. The high hierarchical model of such an MSCFS-RP mainly involves three top modules, i.e., Client, MetaServer, and Cluster. A tenant can read or write

data to the cluster through packages. A meta server (or a so-called namenode) provides query services to clients, making them know which location to read or write, and manages necessary meta information. All the data are stored in nodes (or so-called datanodes) of a cluster. The place Network is used to simply abstract the network environment. Each object sends packages to Network and takes away packages from Network according to its needs.

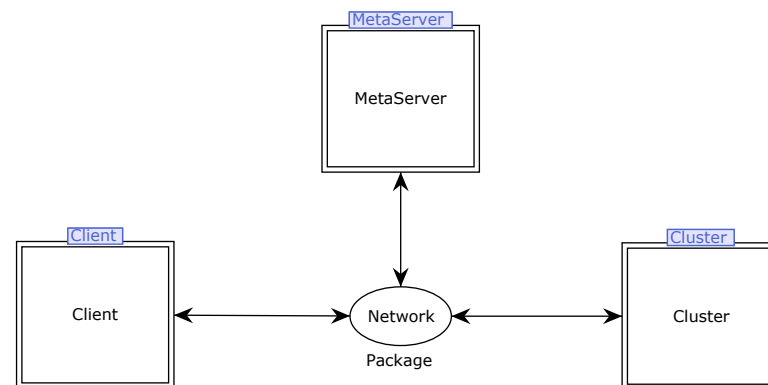


Figure 5. MSCFS-RP model.

A request may be sent by a human user or a process in a system (called a tenant here) from the client. Each request has a unique identifier, e.g., ‘T1’. There are only two types of requests, i.e., reading and writing. A reading request means to read a file from the cluster, while a writing request means to write certain data as a file to the cluster.

5.2. Client Model

As shown in Figure 6, the client initiates a request for reading or writing a file. This is represented by a token in the Request place, indicating the start of the process. The client sends the request to the meta server. The meta server processes the request and sends back the file location information. The client awaits the response from the meta server. This is shown by the Awaited place, which holds a token representing the pending request. Upon receiving the response, the client processes the file location information. This is symbolized by the File Location place, which might contain tokens representing the file location details. The client then sends a reading or writing package to the appropriate storage node via the network. This action is represented by the Read Send or Write Send transitions, which consume tokens from the File Location and Writing Data place and produce tokens in the Network place. The storage node processes the request and sends back the result. This is shown by the Result Confirm transition, which moves tokens from the Network place to the Result place, which now contains the result of the operation. The process ends with the client acknowledging the result, which might be represented by a token CF(tid), which signifies the end of the transaction.

The places Request, FileLocation, WritingData, and Result are used to hold reading or writing requests, file location information, data to write, and the final reading or writing result, respectively. In Figure 6, there are two requests, identified as ‘T1’ and ‘T2’. One of them is a reading request, and the other is a writing request. The requests are assumed to be sent from the location coordinates (1, 26) and (68, 97), respectively. For the writing request, the data ‘XXX’ is to be written to the cluster according to the marking of place WritingData.

5.3. Meta Server Model

In Figure 7, the basic process of a meta server is modeled. There are five places concerning the node information, registered file identifier, writing file identifier, file location, and reading or writing record. The place NodeInfo stores the information of each node, including the node identifier, node state, node coordinates, and available block identifiers. For example, (‘Node11’, AVAILABLE, (15, 26), [‘Block1’, ‘Block2’, ‘Block3’]) indicates that

Node11 is an available storage node located at the coordinates (15, 26) and can provide three blocks. The color sets of NInfo and NInfoList are as follows.

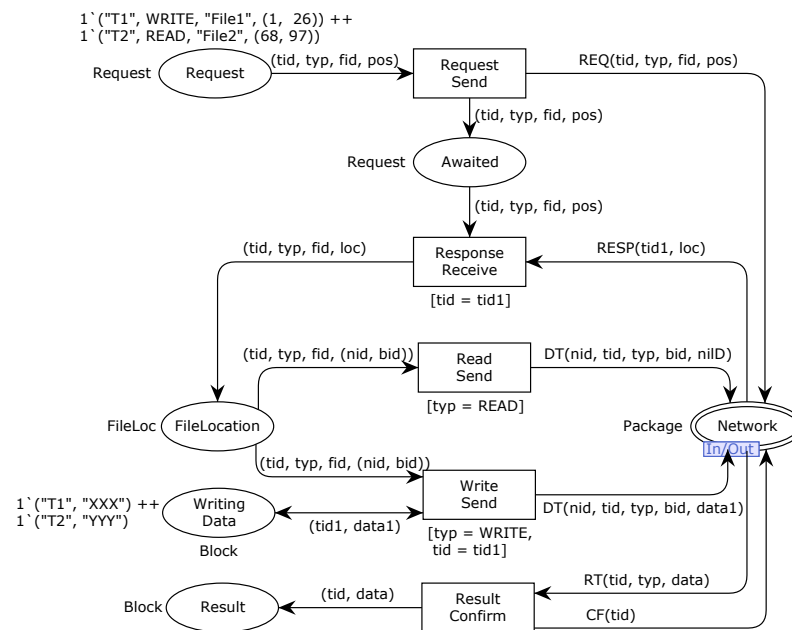


Figure 6. Client model.

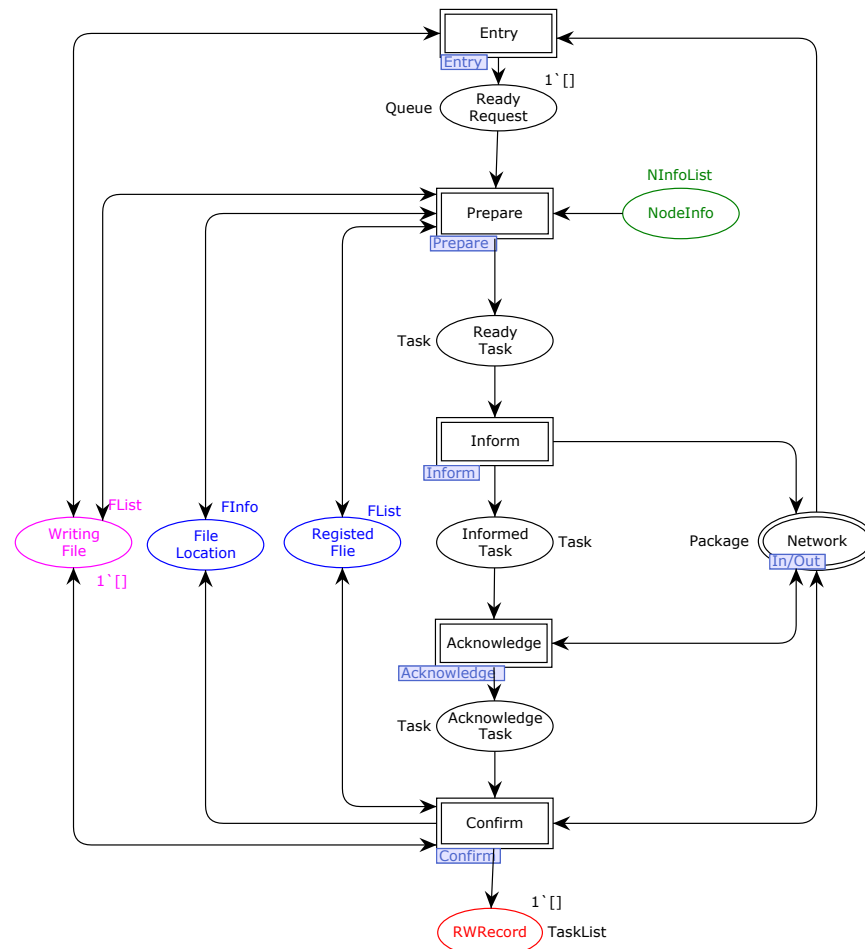


Figure 7. Meta server model.

```
colset NInfo = product NID * NState * Coordinate * BIDList
colset NInfoList = list NInfo
```

The place RegisteredFile records all the file identifiers that have been stored in the cluster, where the color set of RegisteredFile is just a list of file identifiers. The place WritingFile is used to record the file identifiers being written to ensure that each file can only be written once, where the color set of WritingFile is also a list of file identifiers. The place FileLocation indicates where a file is stored in the cluster by a location, i.e., a tuple consisting of a node identifier and a block identifier. For example, ("File2", [("Node21", "Block1"), ("Node23", "Block1"), ("Node12", "Block1")]) indicates that a file called "File2" has three replications and is stored in the above three locations. The color sets of Location, LocationList, and FInfo are as follows.

```
colset Location = product NID * BID
colset LocationList = list Location
colset FInfo = product FID * LocationList
```

After a reading or writing request has been finished and confirmed by the meta server, a record token will be put into the RWRecord place for the convenience of verification. In our model, packages are used to exchange information. If an object needs to communicate with others, it should put a package, which includes the target identifier and the related content, into the network. Correspondingly, the other object will check the package and receive it if the target identifier is consistent. The color set of Package is defined as follows.

```
colset Package = union REQ: Request + RESP: Response + INF: Inf + INFACK: InfAck +
DT: Dt + DTACK: DtAck + RT: Return + CF: TID;
colset Request = product TID * Operation * FID * Coordinate;
colset Response = product TID * Location;
colset Inf = product NID * TID * Last * Next;
colset InfAck = product TID * Location;
colset Dt = product NID * TID * Operation * FID * Data;
colset DtAck = product NID * TID * Next;
colset Return = product TID * Operation * Data;
colset Operation = with READ | WRITE;
colset Coordinate = product INT * INT;
colset Location = product NID * BID;
```

Here, the color sets TID, FID, NID, and BID denote identifiers of tasks, files, nodes, and blocks, respectively. The color set Data is just a color set that denotes storage data and is essentially a string. The Request and Response packages are used to deliver information between the client and the meta server. The Inf and InfAck packages are used to inform and acknowledge between the meta server and the cluster. The Dt and DtAck packages are used to send data operations and receive acknowledgments among the client and storage nodes in the cluster. The Return package is used to return results from corresponding storage nodes.

Finally, the Return package is used by the client to confirm the completion of the task with the meta server. There are two operation types, i.e., READ and WRITE. A coordinate is a two-tuple and records the position information to compute distances. A location is also a two-tuple consisting of a node identifier and a block identifier and uniquely determines where a file is stored. The color sets Last and Next denote the locations of the last node and the next node, respectively, of a current node in a replication pipeline. Since this paper focuses on the system business process and the replication pipelining mechanism, we simply assume that a block of a node is enough to store a file.

In Figure 7, there are five subtransitions, called Entry, Prepare, Inform, Acknowledge, and Confirm, respectively. A reading or writing request will go through all of them to fulfill a task. We introduce them one by one as follows.

5.3.1. Entry Module

As shown in Figure 8, Entry is responsible for controlling the occurrence of a task. First, according to the sequence of arrival, each request is enqueued and recorded. Then, the meta server picks the first request from the queue and checks whether the file concerning this request is being written. If so, the request has to wait until the corresponding writing file has finished. Otherwise, the request can move to the next stage. Note that our model is still highly concurrent, which means that any other reading or writing requests without writing conflicts can be processed concurrently.

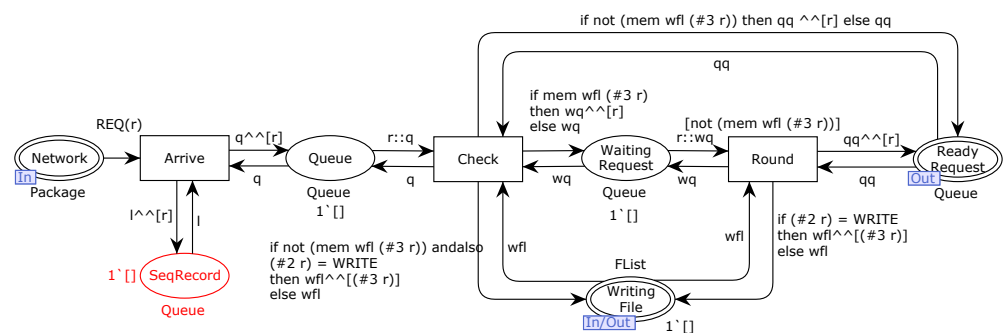


Figure 8. Entry module.

When a request arrives, it is first recorded in the SeqRecord place. This place records tokens that represent the sequence of incoming requests. Each request is then enqueued in the Queue place. This place acts as a buffer where incoming requests wait to be processed. Once recorded and enqueued, the requests are ready to be picked up by the meta server for processing. The meta server checks for conflicts, specifically, whether the file associated with the request is currently being written to. This check is represented by the transition that reads the writing file list `wfl` and the request `r`.

If the file is being written to (`mem wfl (#3 r)` is true), the request must wait. This is shown by the Round transition that does not modify the `wq` and `qq` but keeps the request in the Waiting Request place. If the file is not being written to and the request is to write (`(#2 r) = WRITE`), the file identifier is added to the `wfl`, indicating that the file is now being written to. If there is no conflict, the request is enqueued in the Ready Request place, which represents the queue of tasks that are ready to be executed without any conflicts. Once a task is completed, it is removed and the system updates the `wfl` (in the other module) to reflect that the file is no longer being written to. In Figure 8, “`^^`” is used to concatenate two lists and “`#`” retrieves an indexed record from a tuple.

5.3.2. Prepare Module

Prepare (Figure 9) allocates new available locations for writing requests and queries where files are stored in the cluster for reading requests. First, the file's existence is checked. For reading, if the file to read does not exist, the request token will be moved to the place ReadNotExist. For writing, if the file to write is already in the cluster, the request token will be moved to the place WriteMoreThanOnce to obey the "Writing Once" principle, and the file identifier will also be removed from WritingFile. Specifically, when all of the replication nodes concerning the locations of a file fail, the reading task will be moved to the place ReadAllFault.

Concretely, when a reading or writing request arrives, it is first registered in the system, represented by the Registered File place. The module checks whether the file for the request exists in the cluster. This check is performed by the CheckFile Exist transition. For read request handling, if the file does not exist (not (mem fl fid) andalso typ = READ), the request is moved to the Read NotExist place, indicating that the file is not available for reading. For write request handling, if the file already exists in the cluster and the request is a writing request (mem fl fid andalso typ = WRITE), the request is moved to the WriteMoreThanOnce

place to adhere to the “Writing Once” principle. The file identifier is also removed from the Writing File place to prevent multiple writes. The wfl list is updated, ensuring that the system maintains an accurate record of files being written to.

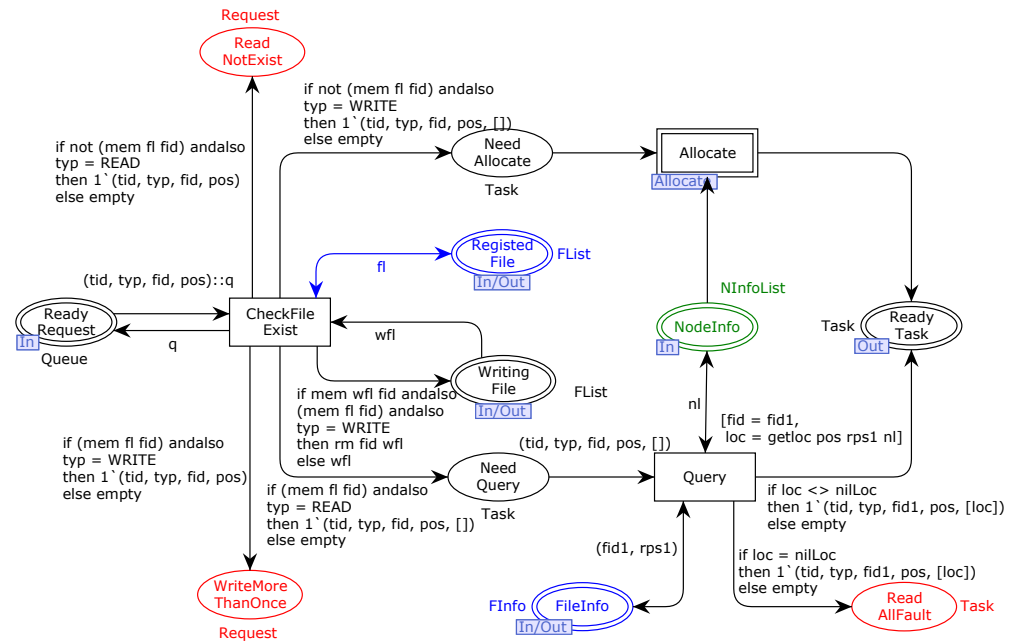


Figure 9. Prepare module.

For new write requests where the file does not exist (not (mem fl fid) andalso typ = WRITE), the module allocates a new location for the file, represented by the transition that produces a token in the Need Allocate place, which is then processed by the Allocate submodule. For read requests where the file exists, the module queries the location of the file in the cluster. This is represented by moving the request to the Need Query place. The module queries (the Query transition) the node information for the available locations, represented by the NodeInfo place. The file information is updated based on the type of request, either read or write, and the availability of the file, represented by the FileInfo place.

For special case handling, if all replication nodes for a file fail, the reading task is moved to the ReadAllFault place, indicating that all copies of the file are unavailable. Once the file's existence is checked and the necessary allocations or queries are performed, the task is moved to the Ready Task place, indicating that it is ready for the next processes.

Specifically, as a submodule, Allocate (Figure 10) selects a location for each replication of a file. The parameter *rNum*, concerning the initial marking of the ReplicationNum place, is used to declare the number of replications of a file. For the convenience of explanation, *rNum* is set to three in most cases in this paper. However, this parameter can be set to any value less than the storage capacity. The allocation strategy used here requires that each file should be read from or written to the closest node, and each replication should be assigned to a different node.

The Allocate submodule checks for available nodes in the Node Available place, which contains information about the nodes that can be used for replication. The Need Allocate place holds tasks that are ready to be allocated a new location for file replication. The Allocate transition selects a location for each replication of a file. The *select* function is used to choose the location based on the allocation strategy. The submodule counts the number of successful replications. If the count (*cnt*) is greater than 0, the system continues to allocate new locations. For each successful allocation, *rps* is updated with the new location (*newloc*). This is shown by the transition that produces a token in the Need Allocate place with the updated replication. After each successful allocation, the counter (*cnt*) is decremented. If the counter reaches zero, no further allocations are needed. The submodule has successfully

allocated locations for all the required replications of the file. If the counter is still greater than zero after the allocation, the system continues to allocate new locations until the counter reaches zero.

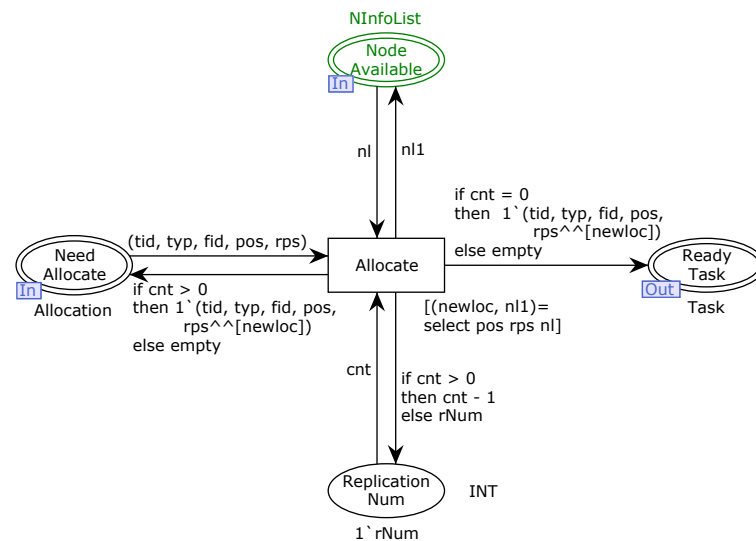


Figure 10. Allocate submodule.

5.3.3. Inform Module

After the Prepare process, the meta server needs to inform the corresponding nodes, as shown in Figure 11. The Ready Task place holds tasks that are ready to inform the nodes. For reading requests, the Read Inform transition is activated. The task token, which includes the read request details (tid, typ, fid, pos, [loc]), is identified. For writing requests, the Write Inform transition is fired.

Based on the type of request (READ or WRITE), the Inform module creates an inform package that includes the node identifier, the task identifier, and location details. The inform package is then sent out through the network to the corresponding nodes. This is represented by the Network place, indicating the outgoing communication. Once the inform package is sent, the task is moved to the Informed Task place. After the inform package is sent and the task is moved to the Informed Task place, the node can now proceed with the execution of the task as informed by the meta server.

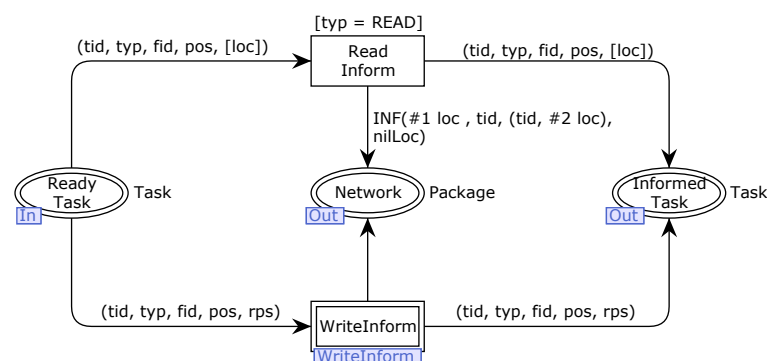


Figure 11. Inform module.

For reading, only one node needs to be informed, while for writing, all the nodes for replication should be informed, as shown in Figure 12. The Write Inform submodule identifies writing tasks that are ready to initiate the inform process. These tasks are represented by tokens in the Ready Task place. For the first node in the replication list, the system prepares an inform package. This is represented by the First Inform transition,

which checks if the replication pipelining sequence (*rps*) is not empty. Subsequent nodes in the replication pipelining sequence are informed in the middle stage.

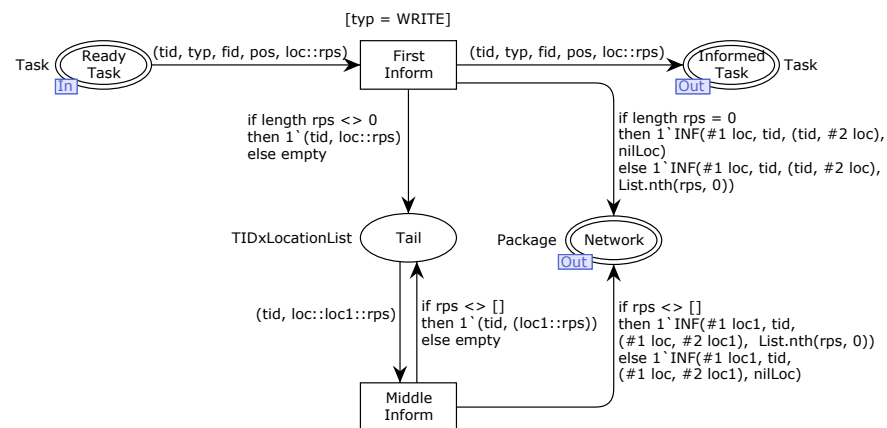


Figure 12. Write Inform submodule.

The Middle Inform transition handles this by checking the length of the *rps* and preparing inform packages accordingly. If the *rps* is not empty, an inform package is prepared with `List.nth(rps, 0)`. If the *rps* is empty, an inform package is prepared with `nilLoc`. The inform packages are sent to their respective nodes via the network. The process continues until all nodes in the replication pipelining sequence have been informed. Once an inform package is sent, the task is updated to reflect that the node has been informed. This is shown by the transition that moves the task from the Ready Task place to the Informed Task place.

5.3.4. Acknowledge Module

Accordingly, Acknowledge deals with acknowledgments sent from the storage nodes in Figure 13. The Acknowledge module identifies tasks that have informed and are awaiting acknowledgments. These tasks are represented by tokens in the Informed Task place. When a storage node completes its task (either reading or writing), it sends an acknowledgment back to the system. For reading tasks, the INFACK package includes the task ID (*tid1*) and the location (*loc1*). The `RESP(tid, loc)` package represents the module's response for a read acknowledgment. For writing tasks, the acknowledgment (INFACK) includes similar information.

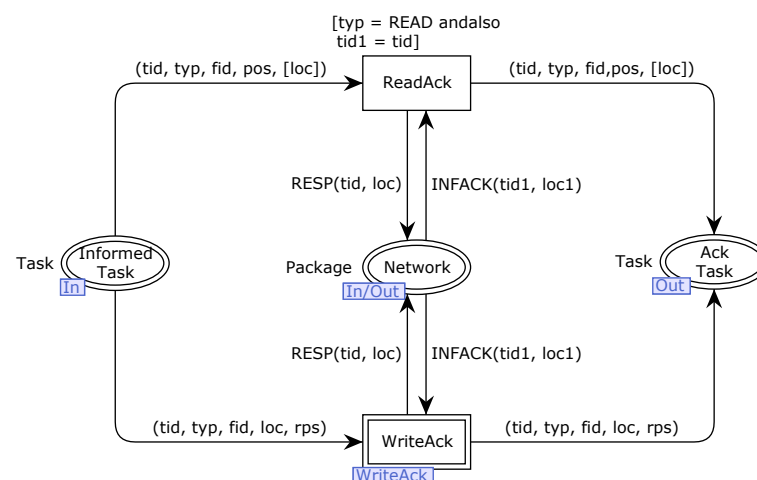


Figure 13. Acknowledge module.

The ReadAck and WriteAck transitions deal with a reading or writing acknowledgment. The task moves from the Informed Task place to the Ack Task place, indicating that the acknowledgment has been received and the task is now considered complete from the storage node's perspective. In cases where multiple storage nodes are involved (e.g., in writing operations with replication), the system continues to process acknowledgments until all nodes have responded.

Similar to Inform, only one acknowledgment is needed for reading, while acknowledgments from all replication nodes should be collected, as shown in Figure 14. The Write Acknowledge submodule identifies writing tasks that have been informed to the storage nodes and are now awaiting acknowledgments. The tasks move to the Waiting Ack place, indicating that they are in the stage where they are waiting for acknowledgments from the nodes, and a token is put in the Process Ack place for further processes. When a storage node completes the writing operation, it sends an acknowledgment back to the submodule. This is represented by the Network place and indicates the incoming acknowledgment package. Upon receipt of an acknowledgment, the Ack transition checks the task ID (*tid*) and the replication pipelining sequence (*rps*).

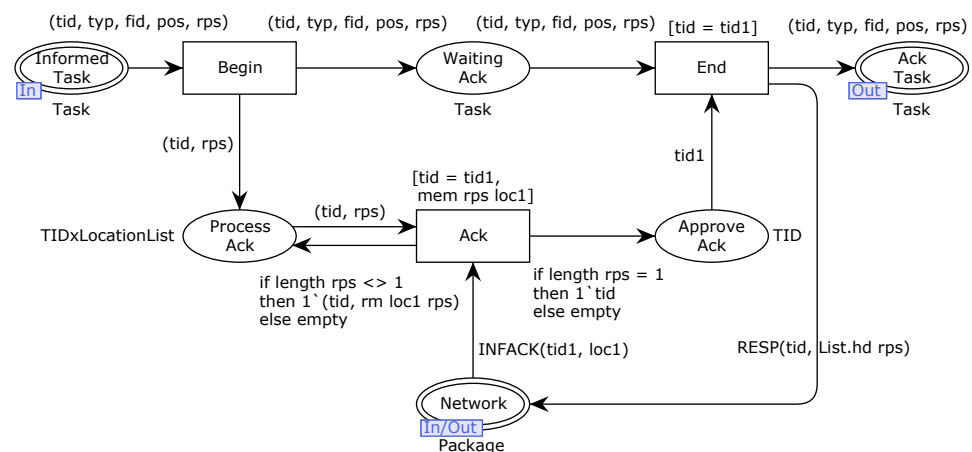


Figure 14. Write Acknowledge submodule.

If an acknowledgment has been successfully processed and the location is no longer needed in the replication pipelining sequence, the Ack transition removes it. This is shown by the transition that outputs a new token to the Process Ack place. If the replication pipelining sequence (*rps*) contains only one location, i.e., the acknowledgment received is the last one expected, the task moves to the Approve Ack place. After triggering the End transition, the submodule sends a response (RESP) back to the storage node that sent the acknowledgment, confirming receipt. Once all acknowledgments have been received and processed, the task is considered complete, and it moves from the Waiting Ack Task place to the Ack Task place, indicating successful completion of the writing operation from the perspective of acknowledgment collection.

5.3.5. Confirm Module

Finally, after the client has read or written successfully, the meta server will confirm this task and record it in RWRecord. Specifically, for writing, the writing file identifier will be removed from WritingFile, and the file identifier and the location information will be updated to RegisteredFile and FileInfo, respectively (Figure 15).

Concretely, the Confirm module identifies tasks that have been completed, either through reading or writing. These tasks are represented by tokens in the Ack Task place. The client sends a confirmation (CF(*tid*)) to the meta server, indicating that these tasks have been completed from the perspective of Client. The Read Confirm transition checks if the task type is READ and matches the task ID. It is similar for writing tasks. For writing tasks, if the file identifier (*fid1*) is found in the writing file list (Writing File), it is removed, indicating

that the file is no longer being written to. The file identifier and location information are updated in the Registered File and FileInfo places. This ensures that the system's record of file locations is accurate and up-to-date. The Registered File place may be sorted or organized to reflect the changes made during the writing operation.

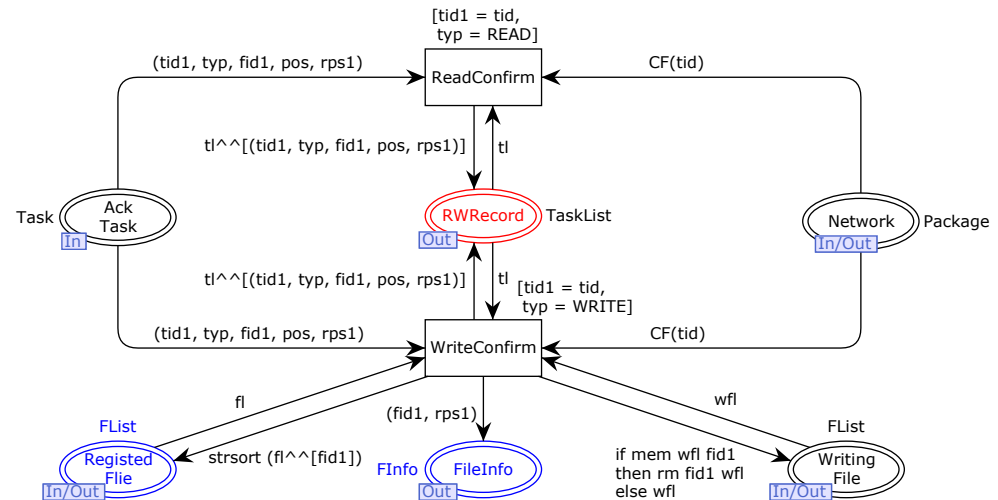


Figure 15. Confirm module.

Finally, the task is removed from the Ack Task place, indicating that all steps of the process have been successfully executed. Regardless of the task type, the meta server records the task details in RWRecord, which may be used for monitoring, logging, or further processing. This includes the task ID, the task type, the file identifier, the position, and the replication pipelining sequence.

5.4. Cluster Model

The cluster consists of a series of storage nodes. In Figure 16, there are a total of twelve storage nodes. In the cluster model, each storage node, distinguished by a unique identifier such as N11 or N21, serves as a basic unit that is tasked with the critical operations of reading and writing data. These nodes are interconnected through a network place that enables data transfer and communication. The nodes manage data blocks, denoted by b11, b12, ..., b43, with each block being associated with and stored on a specific node. This setup allows for node-to-node communication, facilitating operations such as data replication, distribution, and retrieval.

The architecture of clusters incorporates redundancy to ensure data availability and to provide fault tolerance, so that if a node fails, the network's connectivity allows other nodes to assume its responsibilities, thereby ensuring continuous operation. This redundancy, along with the distributed nature of the cluster, supports parallel processing and load balancing, which in turn enhances the system's overall performance and reliability.

5.4.1. Storage Node Module

As shown in Figure 17, once the INF package is processed by the node, which includes the destination node ID (*des*), the task ID (*tid*), and the information for the replication pipelining sequence (*last*, *next*), the node responds by sending an acknowledgment (INFACK(*tid*, (*nid*, #2 last))) back to the network, represented by the Network place. Each storage node within the cluster is distinguished by a unique identifier stored in the NodeID place for node identification. Upon receiving the INF package, a replication pipelining information token is placed into the Guard place, which contains essential details about the replication process, such as the last and next node identifiers in the replication pipelining sequence. The core functions of a node are reading and writing, which are modeled as two transitions,

i.e., the Read and Write transitions. Data blocks are managed within the node's storage, with the Storage place illustrating the flow of blocks.

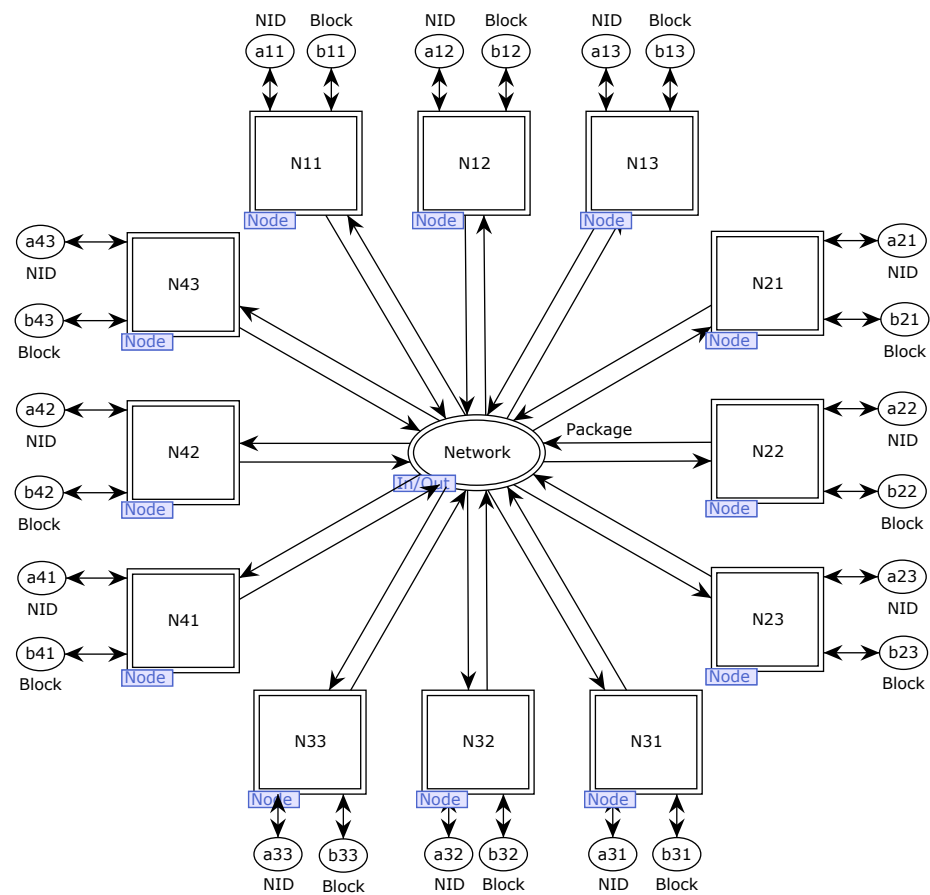


Figure 16. Cluster model.

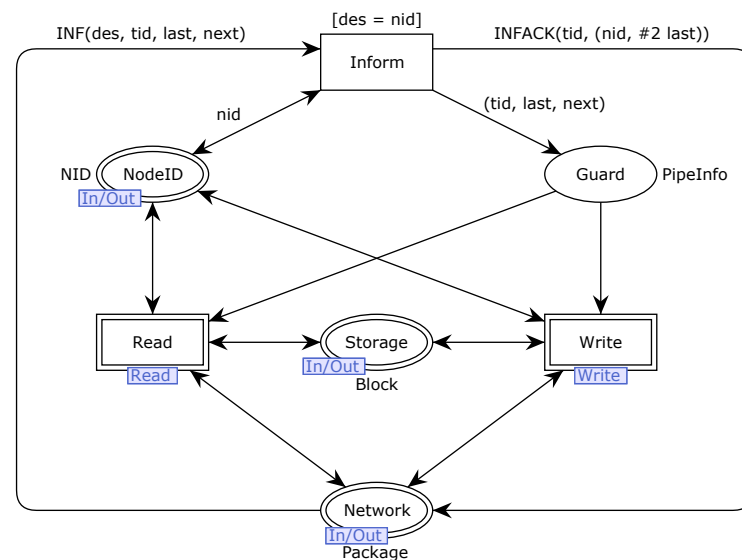


Figure 17. Storage node module.

For example, given a replication pipelining sequence $Node1 \leftrightarrow Node2 \leftrightarrow Node3$, $Node2$ should receive a data operation from $Node1$ and send a data operation to $Node3$. Correspondingly, $Node2$ receives an acknowledgment from $Node3$ and sends an acknowledgment to $Node1$. So for $Node2$, the last node ID is $Node1$ and the next node ID is $Node3$.

5.4.2. Reading from a Node

Reading from a storage node is simple: just select the block data in the storage according to the Dt package and return the result to the client (Figure 18). The process begins with identifying the node that will perform the reading operation, according to the NodeID place. The Guard place holds the information about the replication process. The Read transition is fired when receiving a reading request. This request includes the destination node ID (des), the task ID (tid), the operation type (typ, which is READ in this case), and the block ID (bid).

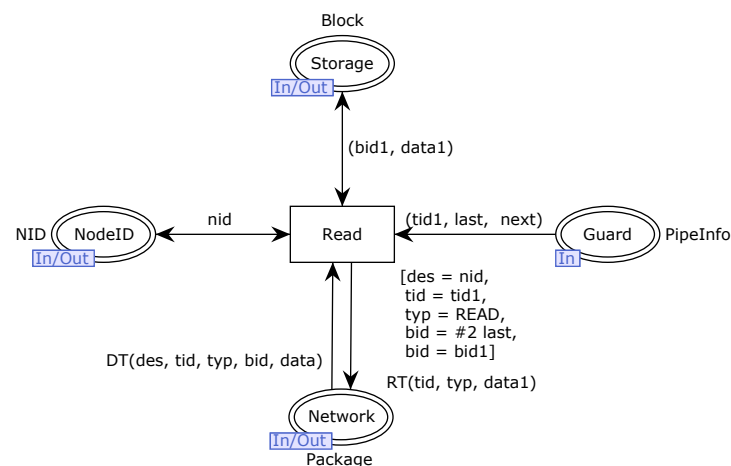


Figure 18. Reading from a node.

The storage node selects the block (bid1) and retrieves the block data (data1) from its storage based on the information provided in the Dt package. Once the block data are retrieved, they are prepared to be sent over the Network place. The node sends an RT package back to the client, which includes the task ID, the operation type, and the retrieved data.

5.4.3. Writing to a Node

Writing is a little complex due to the replication pipelining mechanism (Figure 19). The writing process also begins by identifying the node responsible for the writing operation according to the NodeID place. The Guard place provides the last and next node ID for the replication pipelining sequence. The Write transition is fired upon receiving a Dt package. This package contains the destination node ID (des), the task ID (tid), the operation type (typ, which is WRITE), and the block ID (bid).

If the current node is not the first in the replication pipelining sequence, the sequence writes the data to storage, prepares to forward the operation to the next node using a DT package, and sends an acknowledgment to the last node using a DTACK package. The AckWait place records information for checking acknowledgments to ensure all nodes in the replication pipelining sequence have successfully written the data. If the current node is first in the replication pipelining sequence and has received an acknowledgment, the sequence returns the result to the client using an RT package with a success message. For the last node in the replication pipelining sequence, the sequence writes the data to storage and sends an acknowledgment to the previous node without forwarding a Dt package to any other node.

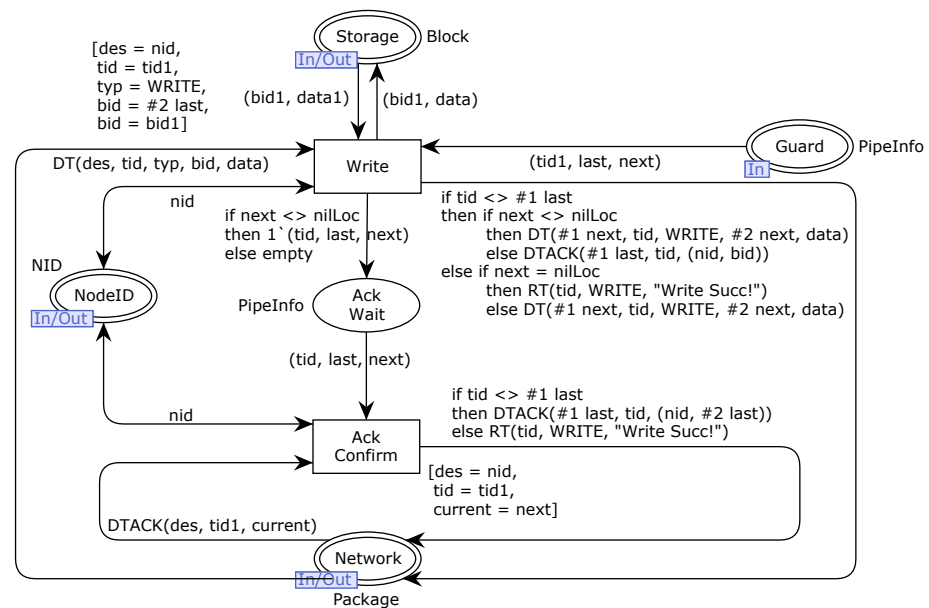


Figure 19. Writing to a node.

6. Formal Analysis of MSCFS-RP

In order to ensure the validity and trustworthiness of our model, we analyze some important properties. CPNs can be used to verify information-theory-related properties, such as data consistency and reliability, through state space analyses or model checking techniques. The formal analysis of MSCFS-RP includes two parts. The first part is about state space statistics that are automatically generated by CPN Tools. The second part involves the verification of properties.

6.1. State Space Statistics

CPN Tools provides a powerful state space analysis capability. Given a colored Petri net, the corresponding state space report can be generated automatically and includes basic statistics, liveness properties, home properties, fairness properties, and boundedness properties. We give eight representative scenarios and generate their state space statistics with the help of CPN Tools, as shown in Table 2.

Table 2. State space statistics.

SNo.	Requests		Nodes	Arcs	Secs	Status
	Request 1	Request 2				
S1	("T1", WRITE, "File1")	None	72	136	0	Full
S2	None	1'("T2", READ, "File2")	17	18	0	Full
S3	1'("T1", WRITE, "File1")	1'("T2", READ, "File2")	2112	6276	1	Full
S4	1'("T1", WRITE, "File1")	1'("T2", READ, "File1")	376	916	0	Full
S5	1'("T1", WRITE, "File1")	1'("T2", WRITE, "File3")	9794	37,488	8	Full
S6	1'("T1", WRITE, "File1")	1'("T2", WRITE, "File1")	566	1500	1	Full
S7	1'("T1", READ, "File2")	1'("T2", READ, "File4")	462	976	0	Full
S8	1'("T1", READ, "File2")	1'("T2", READ, "File2")	462	976	0	Full

In the initial marking, we assume that the replication number is set to three, and two files called “File2” and “File3” have already been stored in some locations. These eight scenarios include: S1—only one writing request, S2—only one reading request, S3—one writing request and one reading request about two different files, S4—one writing request and one reading request about the same file, S5—two writing requests about two different files, S6—two writing requests about the same file, S7—two reading requests about two different files, and S8—two reading requests about the same file. As we can see, writing commonly has more state nodes than reading, since the replication pipelining mechanism of writing needs more places, transitions, and arcs. Another thing is that the state space grows rapidly with increasing numbers of writing requests. Compared with writing the same file, writing different files leads to a larger state space since writing different files can occur concurrently. For reading the same file or different files, the state space is the same because there is no conflict, unlike in writing. The state space report of scenario S3 is shown in Table 3.

Table 3. Statespace report of scenario S3.

Statistics	Liveness Properties	
State Space	Dead Markings	
Nodes: 2112	[2109, 2110, 2111, 2112]	
Arcs: 6276	Dead Transition Instances	
Secs: 1	Entry'Round 1	
vStatus: Full	Node'Inform 10	
Scc Graph	...	
Nodes: 2112	Live Transition Instances	
Arcs: 6276	None	
Secs: 1		
Home Properties	Fairness Properties	
Home Markings	No infinite occurrence sequences.	
None		
Boundedness Properties		
Place	Upper	Lower
Client'Request 1	2	0
MetaServer'Ready_Request 1	2	0
...	...	...

In Table 3, there are 2112 nodes and 6276 arcs in the state space. The total time to calculate such a state space is 1 s, and the state space is full (compared with a partial state space). There are four dead markings. We list the differences between them in Table 4. The place SeqRecord is used to record the arriving sequence of requests, while the place RWRecord is used to record the confirmed writing or reading task. According to their markings, we know that the processing of a writing request and the processing of a reading request for different files do not affect each other, and all possible combinations of them can occur. There are some dead transition instances because not all transitions are fired in scenario S3. For example, the transition Entry'Round is only fired when a waiting request is resumed. In this scenario, the writing request and the reading request are about different files, so neither of them needs to wait, and the transition Entry'Round is dead. There is no home marking and no infinite occurrence sequence. For boundedness, an example is that the place Request of the page Client has a maximum of two tokens and a minimum of zero tokens since there are two requests (one for writing and one for reading), and they will be sent to the meta server.

Table 4. Four dead markings for scenario S3.

MNo.	Marking of Place SeqRecord	Marking of Place RWRecord
2109	[("T2", READ, "File2", (68, 97)), ("T1", WRITE, "File1", (1, 26))]	[("T2", READ, "File2", ...), ("T1", WRITE, "File1", ...)]
2110	[("T2", READ, "File2", (68, 97)), ("T1", WRITE, "File1", (1, 26))]	[("T1", WRITE, "File1", ...), ("T2", READ, "File2", ...)]
2111	[("T1", WRITE, "File1", (1, 26)), ("T2", READ, "File2", (68, 97))]	[("T2", READ, "File2", ...), ("T1", WRITE, "File1", ...)]
2112	[("T1", WRITE, "File1", (1, 26)), ("T2", READ, "File2", (68, 97))]	[("T1", WRITE, "File1", ...), ("T2", READ, "File2", ...)]

6.2. Property Specifications

In this section, several critical properties are verified for our model using CPN Tools. These properties are abstracted from functional requirements, and many of them refer to [7]. The illustrations of the nine properties are as follows. We first discuss properties P1–P7 (Table 5) and then analyze properties P8–P9 since their configurations are different. Note that these property specifications are described using ASK-CTL logic and ML within CPN Tools, integrating temporal logic with functional language. Further details regarding their syntax and semantics can be found in the referenced works [33,34].

- P1 **Deadlock freedom:** no infinite occurrence sequences in the state space;
- P2 **Writing correctness:** a file to write (if it is the first writing to this file) is finally written to the cluster;
- P3 **Reading correctness:** a file to read (if the file exists in the cluster) is finally received by the client;
- P4 **Mutual exclusion:** a file is being written by at most one writer at any time;
- P5 **Writing once:** a file can be written only once;
- P6 **Minimal distance:** a file is read from the closest storage node for a request;
- P7 **Replication consistency:** the replications should be the same after being confirmed by the meta server;
- P8 **Writing Robustness:** a file cannot be written to a storage node that breaks down.
- P9 **Reading Robustness:** a file can be read successfully when some of its storage nodes break down.

Table 5. Properties (P1–P7).

PNo.	Description	Formula
P1	Deadlock freedom	No formula. Check Fairness Properties in the state space report: No infinite occurrence sequences
P2	Writing correctness	val Formula1 = INV(POS(NF(" ", (IsAllWritten InitNode)))); eval_node Formula1 InitNode
P3	Reading correctness	val Formula2 = INV(POS(NF(" ", (IsAllRead InitNode)))); eval_node Formula2 InitNode
P4	Mutual exclusion	val Formula3 = INV(NF(" ", IsMutualExclusion)); eval_node Formula3 InitNode
P5	Writing once	val Formula5 = INV(NF(" ", IsWriteOnce)); eval_node Formula5 InitNode
P6	Minimal distance	val Formula6 = INV(NF(" ", IsAllMinimal)); eval_node Formula6 InitNode
P7	Replication consistency	val Formula7 = INV(POS(NF(" ", IsAllConsistent))); eval_node Formula7 InitNode

Here, the deadlock freedom indicates that there is no infinite loop in the state space of a CPN model. In the state space report automatically generated by CPN Tools, the Fairness Properties indicate this property. If “No infinite occurrence sequences” appears under the Fairness Properties item, we say deadlock freedom holds.

Writing correctness means that every first-time writing request has a corresponding storage result in the cluster. The main auxiliary functions are [Function 1](#) and [Function 2](#) (see more auxiliary functions in [Appendix A](#)). For each writing request in the initial state, the function `IsOneWritten(n, q)` is satisfied. Given an initial marking, if for all the state space nodes of an MSCFS-RP model it is possible from the initial state to reach a state where this holds, we say the model’s writing is correct. For a writing request (`tid, typ, fid`), check whether this file has been written. If so, the corresponding tokens should be in the place `WriteMore ThanOnce` and `File Location`, respectively. Otherwise, it should satisfy that (1) the place `FileLocation` has a token about the corresponding file (`fid, locs`), and (2) for the first location (`nid, bid`), namely, the location as a response to the client, the data of the block `bid` in the node `nid` should be the same as the data to write in the client.

Function 1 IsAllWritten

```

1: fun IsAllWritten init n =
2:   if List.null(GetWQueries init)
3:   then true
4:   else List.all (IsOneWritten n) (GetWQueries init)

```

The description of [Function 1](#) using mathematical formalism is illustrated as follows. Let `init` be the initial state and `n` be a node identifier. The function `IsAllWritten(init, n)` is defined.

$$\text{IsAllWritten}(\text{init}, n) = \begin{cases} \text{true} & \text{if } \text{GetWQueries}(\text{init}) \\ & \text{is empty} \\ \forall q \in \text{GetWQueries}(\text{init}), \text{IsOneWritten}(n, q) & \text{otherwise} \end{cases}$$

Here, `GetWQueries(init)` retrieves the list of write queries from the initial state. If this list is empty, the function returns `true`. Otherwise, it checks if the function `IsOneWritten(n, q)` returns `true` for all `q` in the list of write queries.

Function 2 IsOneWritten

```

1: fun IsOneWritten n q =
2:   let val (tid, WRITE, fid) = q in
3:     if List.exists (fn x => x = q) (Mark.Prepare'WriteMore_ThanOnce 1 n) andalso
4:       List.exists (fn x => (#1 x) = fid) (Mark.MetaServer'File_Location 1 n)
5:     then true
6:     else
7:       let val floc = GetFileLocation n fid in
8:         if isSome(floc)
9:         then
10:          let val (_, loc::_) = valOf(floc);
11:            val bdata = GetNodeBlock n loc;
12:            val wdata = GetWrittenData n tid in
13:              if isSome(bdata) andalso isSome(wdata)
14:              then #2 (valOf(bdata)) = #2 (valOf(wdata))
15:              else false
16:            end
17:          else false
18:        end
19:      end

```

The description of [Function 2](#) using mathematical formalism is illustrated as follows. Let n be a node identifier and $q = (tid, WRITE, fid)$ be a query tuple, where tid is a task ID, $WRITE$ is the operation type, and fid is a file ID. The function $IsOneWritten(n, q)$ returns true if q exists in the marking $PrepareWriteMore_ThanOnce(1, n)$ and the file ID fid exists in the marking $MetaServerFile_Location(1, n)$. If not, the function checks if the file location $floc$ is retrievable using $GetFileLocation(n, fid)$. If $floc$ is found, the function further checks if both block data $bdata$ from $GetNodeBlock(n, loc)$ and written data $wdata$ from $GetWrittenData(n, tid)$ are retrievable. If both data sets are present, the function compares the second elements of $bdata$ and $wdata$. If they match, the function returns true; otherwise, it returns false.

Similar to writing correctness, reading correctness means that for every reading request, a file to read (if it exists in the cluster) can be returned. There are also two main auxiliary functions, as shown in [Function 3](#) and [Function 4](#), for reading correctness. The only differences between reading correctness and writing correctness lies in that (1) reading requests instead of writing requests in a scenario are considered, (2) the file to read that does not exist is recorded in the place $ReadNotExist$, and (3) the data of the corresponding location are checked with the received result of the client in the place $Result$ but not the data in the place $WritingData$.

Function 3 IsAllRead

```

1: fun IsAllRead init n =
2:   if List.null(GetRQueries init)
3:   then true
4:   else List.all (IsOneRead n) (GetRQueries init)

```

Function 4 IsOneRead

```

1: fun IsOneRead n q =
2:   let val (tid, READ, fid) = q in
3:     if List.exists (fn x => x = q) (Mark.PrepareRead_NotExist 1 n)
4:     then true
5:     else
6:       let val floc = GetFileLocation n fid in
7:         if isSome(floc)
8:         then
9:           let val (_, loc::_) = valOf(floc) in
10:            val bdata = GetNodeBlock n loc;
11:            val rdata = GetResult n tid in
12:              if isSome(bdata) andalso isSome(rdata)
13:              then #2 (valOf(bdata)) = #2 (valOf(rdata))
14:              else false
15:            end
16:          else false
17:        end
18:      end

```

The description of [Function 3](#) using mathematical formalism is illustrated as follows. Let $init$ be the initial state and n be a node identifier. The function $IsAllRead(init, n)$ is defined.

$$IsAllRead(init, n) = \begin{cases} \text{true} & \text{if } GetRQueries(init) \\ & \text{is empty} \\ \forall q \in GetRQueries(init), IsOneRead(n, q) & \text{otherwise} \end{cases}$$

Here, `GetRQueries(init)` retrieves the list of read queries from the initial state `init`. If this list is empty, the function returns `true`. Otherwise, it checks if the function `IsOneRead(n, q)` returns `true` for all `q` in the list of read queries.

The description of [Function 4](#) using mathematical formalism is illustrated as follows. Let `n` be a node identifier and `q = (tid, READ, fid)` be a query tuple, where `tid` is a task ID, `READ` is the operation type, and `fid` is a file ID. The function `IsOneRead(n, q)` returns `true` if `q` exists in the marking `Prepare'Read_NotExist(1, n)`. If not, the function checks if the file location `floc` is retrievable using `GetFileLocation(n, fid)`. If `floc` is found, the function further checks if both block data `bdata` from `GetNodeBlock(n, loc)` and result data `rdata` from `GetResult(n, tid)` are retrievable. If both data sets are present, the function compares the second elements of `bdata` and `rdata`. If they match, the function returns `true`; otherwise, it returns `false`.

As shown in [Function 5](#), mutual exclusion emphasizes that a file cannot be written by more than one writer at the same time. We use a place called `WritingFile` to record the file identifier that is being written. Also, this place is used to control write–write or write–read conflicts, which mean that when a writer is writing a file, the other writer or reader of the same file has to wait until the writing is finished. As the following function shows, to check mutual exclusion, we only need to determine whether there are two tokens of the same file identifier in the place `WritingFile`.

Function 5 IsMutualExclusion

```
1: fun IsMutualExclusion n =
2:   not(duplicated (List.concat(Mark.Entry'Writing_File 1 n)))
```

The description of [Function 5](#) using mathematical formalism is illustrated as follows. Let `n` be a node identifier. The function `IsMutualExclusion(n)` returns `true` if there are no duplicate tokens in the concatenated list of markings from `Mark.Entry'Writing_File(1, n)`. This is formally expressed as below, where `duplicated(x)` is a function that checks if there are any duplicate elements in the list `x`:

$$\text{IsMutualExclusion}(n) = \neg(\text{duplicated}(\text{List.concat}(\text{Mark.Entry'Writing_File}(1, n))))$$

In [Function 6](#), “writing once” indicates that each file cannot be written more than once. As stated in [6], the write–once–read–many access model for files simplifies data consistency issues and enables high throughput data access. This property can be simply checked by judging whether there are two writing records of the same file in the place `RWRecord`.

Function 6 IsWriteOnce

```
1: fun IsWriteOnce n =
2:   not(duplicated (List.map (fn x => #3 x)
3:     (List.filter (fn x => #2 x = WRITE)
4:       (List.concat(Mark.Confirm'RWRecord 1 n)))))
```

The description of [Function 6](#) using mathematical formalism is illustrated as follows. Let `n` be a node identifier. The function `IsWriteOnce(n)` checks for duplicates in the list of file IDs (denoted as `#3 x`) that are associated with the `WRITE` operation (denoted as `#2 x`). Specifically, it first concatenates the marking `Confirm'RWRecord(1, n)`, filters this list to retain only those entries for which the second element is equal to `WRITE`, maps this filtered list to extract the third element (the file ID), and then checks for duplicates in this resulting list. The function returns `true` if there are no duplicates, which is formally expressed as:

$$\text{IsWriteOnce}(n) = \neg(\text{duplicated}(\text{List.map}(\lambda x \rightarrow \#3 x, \text{List.filter}(\lambda x \rightarrow \#2 x = \text{WRITE}, \text{List.concat}(\text{Mark.Confirm'RWRecord}(1, n)))))$$

As shown in [Function 7](#) and [Function 8](#), the minimal distance means that a file is read from the closest replication node. We use the function `mindist(pos, rnl)` to compute which replication node has the minimal distance. If the node is the one with the minimal distance, the function `IsOneMinimal(n, r)` returns true; otherwise it is false. For all the reading records in `RWRecord`, the minimal distance property should hold.

Function 7 `IsAllMinimal`

```

1: fun IsAllMinimal n =
2:   if List.null(getRRRecord n)
3:   then true
4:   else List.all (IsOneMinimal n) (getRRRecord n)

```

The description of [Function 7](#) using mathematical formalism is illustrated as follows. Let n be a node identifier. The function `IsAllMinimal(n)` is defined.

$$\text{IsAllMinimal}(n) = \begin{cases} \text{true} & \text{if } \text{getRRRecord}(n) \\ & \text{is empty} \\ \forall q \in \text{getRRRecord}(n), \text{IsOneMinimal}(n, q) & \text{otherwise} \end{cases}$$

Here, `getRRRecord(n)` retrieves the list of read records for node n . If this list is empty, the function returns true. Otherwise, it checks if the function `IsOneMinimal(n, q)` returns true for all q in the list of read records.

Function 8 `IsOneMinimal`

```

1: fun IsOneMinimal n r =
2:   let val (_, _, fid, pos, [(nid, _)]) = r;
3:       val nl = List.concat(Mark.MetaServer'NodeInfo 1 n);
4:       val floc = List.find (fn x => (#1 x) = fid)
5:         (Mark.MetaServer'File_Location 1 n) in
6:   if isSome(floc)
7:   then
8:     let val nids = List.map (fn (nid, _) => nid) (#2 (valOf(floc)));
9:         val rnl = List.filter (fn (nid, _, _, _) =>
10:           (List.exists (fn x => x = nid) nids)) nl;
11:         val ni = mindist pos rnl in
12:         nid = #1 ni
13:       end
14:   else false
15: end

```

The description of [Function 7](#) using mathematical formalism is illustrated as follows. Let n be a node identifier and $r = (_, _, \text{fid}, \text{pos}, [(\text{nid}, _)])$ be a tuple representing a record r , where fid denotes the file ID and pos denotes a position. The function `IsOneMinimal(n, r)` first retrieves a list nl by concatenating all node information from the marking `Mark.MetaServer'NodeInfo(1, n)`. It then searches for a file location floc in the marking `Mark.MetaServer'File_Location(1, n)`, where the first element matches fid . If floc exists, the function extracts a list nids containing all node identifiers associated with floc . It filters nl to retrieve rnl , which includes entries where the node identifier nid exists in nids . The function then computes ni using a function `mindist(pos, rnl)` to determine the closest node identifier to pos within rnl . If ni is found and its first element matches nid , the function returns true; otherwise, it returns false.

In [Function 9](#) and [Function 10](#), replication consistency means that all the replications of a file should be the same after its writing request is confirmed. First, as shown in the following function `IsOneConsistent(n, ar)`, for one writing request, all the data in the locations of a writing record should be the same. Then, for each writing request in the place

RWRecord, the replication consistency holds. Note that the replication consistency might be not satisfied during writing, but it holds after the meta server confirms this writing, and a new token will be generated to record this writing in RWRecord.

Function 9 IsAllConsistent

```

1: fun IsAllConsistent n =
2:   if List.null(getWRecord n)
3:   then true
4:   else List.all (IsOneConsistent n) (getWRecord n)

```

The description of [Function 9](#) using mathematical formalism is illustrated as follows. Let n denote a node identifier. The function $\text{IsAllConsistent}(n)$ is defined.

$$\text{IsAllConsistent}(n) = \begin{cases} \text{true} & \text{if } \text{getWRecord}(n) \\ & \text{is empty} \\ \forall r \in \text{getWRecord}(n), \text{IsOneConsistent}(n, r) & \text{otherwise} \end{cases}$$

Here, $\text{getWRecord}(n)$ retrieves the list of write records associated with node n . If $\text{getWRecord}(n)$ is empty, indicating no write records exist, the function returns true. Otherwise, it checks if the function $\text{IsOneConsistent}(n, ar)$ returns true for all r in the list $\text{getWRecord}(n)$.

Function 10 IsOneConsistent

```

1: fun IsOneConsistent n ar =
2:   let val (_, _, _, loc::rps) = ar;
3:   val b = GetNodeBlock n loc;
4:   val bs = GetNodeBlockList n rps in
5:   (isSome b) andalso List.all (fn b' => (isSome b') andalso
6:     (#2 (valOf(b)) = #2 (valOf(b')))) bs
7:   end

```

The description of [Function 10](#) using mathematical formalism is illustrated as follows. The function $\text{IsOneConsistent}(n, ar)$ is defined as follows: Let n denote a node identifier, and $ar = (_, _, _, loc::rps)$ represents a record tuple, where loc denotes a location and rps denotes the replication pipelining sequence. The function first retrieves the block b from $\text{GetNodeBlock}(n, loc)$ and the list bs of blocks from $\text{GetNodeBlockList}(n, rps)$. It returns true if and only if:

- b is retrievable (i.e., $\text{isSome}(b)$ is true);
- All blocks bs are retrievable (i.e., $\text{isSome}(b')$ for all $b' \in bs$);
- For all $b' \in bs$, the second elements of b and b' are equal ($\#2(\text{valOf}(b)) = \#2(\text{valOf}(b'))$).

If these conditions are satisfied, the function returns true; otherwise, it returns false.

6.3. Verification Process

In this section, we use CPN Tools to implement verification of the mentioned properties, ensuring the correctness and reliability of our MSCFS-RP model. CPN Tools facilitates this process through comprehensive simulation capabilities that allow us to observe the behavior of the model across diverse operational scenarios. The simulation results of our MSCFS-RP model provide a better understanding of the data flow and the interactions between clients, meta servers, and clusters. A simulation scenario in CPN Tools is shown in [Figure 20](#).

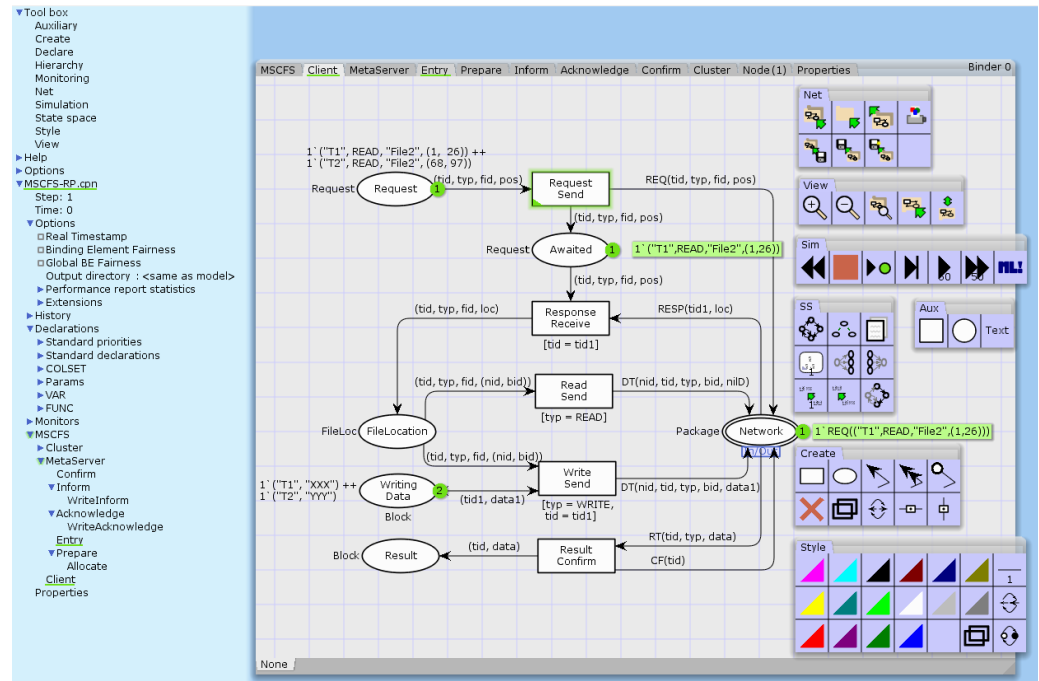


Figure 20. A simulation scenario in CPN Tools.

In Section 6.2, nine critical properties are specified using auxiliary functions written in ASK-CTL. The ASK-CTL logic and model checker are implemented in ML [33]. Each property has a corresponding ML function and formula. These functions analyze the “Nodes” or “Arcs” of the state space to check if a property holds. In CPN Tools, the verification steps should be performed as follows [35]: (1) select the State Space Tool; (2) calculate a State_Space (including the SCC graph); (3) evaluate the following ML code: use (ogpath^“/ASKCTL/ASKCTLloader.sml”); (4) evaluate the ML functions (i.e., ASK-CTL formulas) of property specifications.

For example, “Writing correctness” is verified by checking if all writing requests result in the file being stored in the cluster. As shown in Figure 21, each auxiliary function is correctly defined and the result “it” is true. The verification process is repeated for each scenario. More details about how to use CPN Tools for verification can be found in [36]. We verified the above seven properties in the eight scenarios shown in Table 6. The results show that all the properties P1–P7 are satisfied in related scenarios.

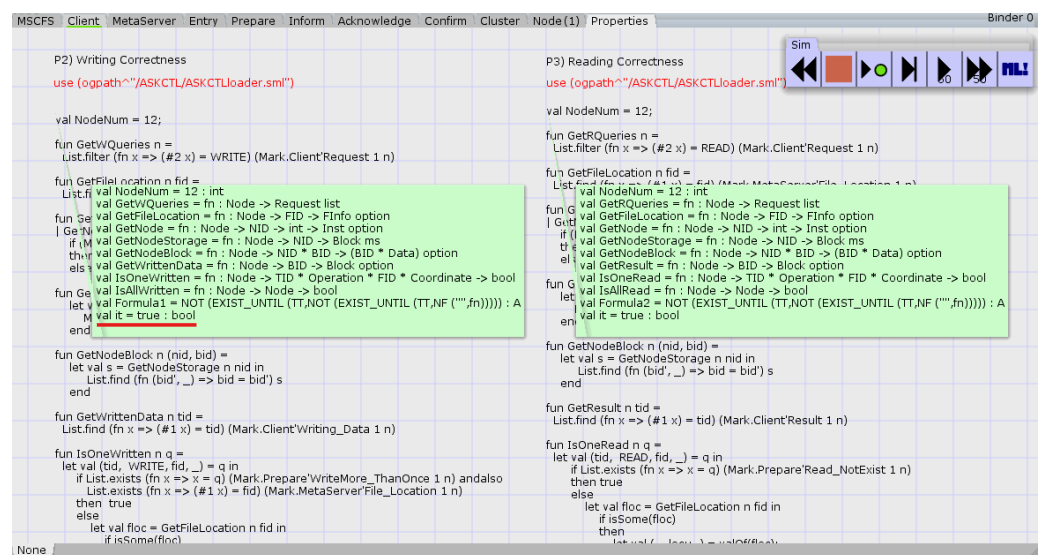


Figure 21. Verification results.

Table 6. Verification of properties (P1–P7) in different scenarios.

SNo.	Properties		SNo.	Properties	
	PNo.	Result		PNo.	Result
S1	P1	true	S2	P1	true
	P2	true		P3	true
	P7	true		P6	true
S3	P1	true	S4	P1	true
	P2	true		P2	true
	P3	true		P3	true
	P4	true		P4	true
	P5	true		P5	true
	P6	true		P6	true
	P7	true		P7	true
S5	P1	true	S6	P1	true
	P2	true		P2	true
	P3	true		P3	true
	P4	true		P4	true
	P5	true		P5	true
	P6	true		P6	true
	P7	true		P7	true
S7	P1	true	S8	P1	true
	P2	true		P2	true
	P3	true		P3	true
	P4	true		P4	true
	P5	true		P5	true
	P6	true		P6	true
	P7	true		P7	true

The property P8 is verified under scenario S1, as shown in Table 7. According to our previous simulations without node failure, the request (“T1”, WRITE, “File1”, (1, 26)) should result in “File1” being stored in “Node22”, “Node11”, and “Node31”. We assume some of these three nodes fail and check whether the writing concerning this request is still successful. As the results show, the property P8 still holds for all the cases. This is because our model will choose other nodes if the most suitable ones have faults.

The property P9 is verified under scenario S2, as shown in Table 8. In this scenario, a reader reads “File2”, which has three replications located in “Node21”, “Node12”, and “Node41”, respectively. We assume that any of the nodes can break down. The results show that if one or two replication nodes fail, the property P9 is still satisfied, i.e., the file can be read correctly, meaning that our model can tolerate node failure to some extent. When all the replication nodes fail, the file will be lost. The results in other reading scenarios are similar to this scenario.

The verification of the nine properties highlighted above demonstrates the significance of the MSCFS-RP model for reliability and trustworthiness. By rigorously verifying these critical properties, we ensure that the model operates as intended without inherent design flaws. Through formal analysis, MSCFS-RP can be evaluated against its properties, facilitating a comprehensive assessment within the broader context of similar systems.

Table 7. Writing robustness (P8).

No.	Node22 State	Node11 State	Node31 State	P2
1	A	A	A	true
2	F	A	A	true
3	A	F	A	true
4	A	A	F	true
5	F	F	A	true
6	F	A	F	true
7	A	F	F	true
8	F	F	F	true

“A” denotes “Available”; “F” denotes “Fault”.

Table 8. Reading robustness (P9).

No.	Node21 State	Node12 State	Node41 State	P3
1	A	A	A	true
2	F	A	A	true
3	A	F	A	true
4	A	A	F	true
5	F	F	A	true
6	F	A	F	true
7	A	F	F	true
8	F	F	F	false

“A” denotes “Available”; “F” denotes “Fault”.

There are some limitations of our method. First, as a common problem mentioned in [20], analysis of state spaces is always relative to a specific configuration (initial marking) of the system parameters. Second, when the value of parameters increases, the method will encounter the state space explosion problem like many model checking methods. The solution to the first problem might be selecting representative parameters or configurations and combining effective testing methods. The solution to the second problem might be abstracting and optimizing models according to the “separation of concerns” principle.

7. Conclusions

A master–slave cloud file system is one of the most representative information systems in cloud environments. Taking the replication pipelining mechanism into consideration, we use colored Petri nets to present an MSCFS-RP model with the main components, i.e., the client, meta server, and cluster. We also study some important properties of MSCFS-RP, including deadlock freedom, writing correctness, reading correctness, mutual exclusion, writing once, minimal distance, replication consistency, writing robustness, and reading robustness. The evaluation results of these properties demonstrate the validity and trustworthiness of the MSCFS-RP model.

Future work includes analyzing replication as well as fault tolerance strategies under such a model. Another research challenge involves developing a framework to quantify the information flow transferred between places in Petri nets. Specifically, a key focus will be on conducting comprehensive performance verification of the MSCFS-RP model. This will involve analyzing the system under various conditions, including network latencies, failure rates, and data loads, to assess its robustness and efficiency in diverse cloud environments. Understanding these factors is crucial for optimizing the performance and reliability of the MSCFS-RP model in practical applications.

Funding: This research was funded by the Natural Science Research Foundation of Jilin Province, China, grant number YDZJ202201ZYT5423; the Science and Technology Development Plan Project of Jilin Province, China, grant number 20240602108RC; the Fundamental Research Funds for the Central Universities, grant numbers 2412022QD040 and 2412022ZD018; and the Research Fund of the Guangxi Key Lab of Multi-source Information Mining & Security, grant number MIMS23-06.

Data Availability Statement: Data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The author declares no conflicts of interest.

Appendix A

To use the state space analysis tool of CPN Tools, the following steps are needed: (1) calculate the state space; (2) calculate the SCC graph; (3) generate the state space report; (4) load *use (ogpath^"/ASKCTL/ASKCTLloader.sml")*; (5) run the ML query functions.

Appendix A.1. Auxiliary Functions for Checking Writing Correctness (P2)

```

val NodeNum = 12;
fun GetWQueries n =
  List.filter (fn x => (#2 x) = WRITE) (Mark.Client'Request 1 n)
fun GetFileLocation n fid =
  List.find (fn x => (#1 x) = fid) (Mark.MetaServer'File_Location 1 n)
fun GetNode n nid 0 = NONE
  | GetNode n nid c =
    if (Mark.Node'NodeID c n) = [nid]
    then SOME c
    else GetNode n nid (c-1)
fun GetNodeStorage n nid =
  let val no = valOf(GetNode n nid NodeNum) in
    Mark.Node'Storage no n
  end
fun GetNodeBlock n (nid, bid) =
  let val s = GetNodeStorage n nid in
    List.find (fn (bid', _) => bid = bid') s
  end
fun GetWrittenData n tid =
  List.find (fn x => (#1 x) = tid) (Mark.Client'Writing_Data 1 n)
fun IsOneWritten n q =
  let val (tid, WRITE, fid, _) = q in
    if List.exists (fn x => x = q)
      (Mark.Prepare'WriteMore_ThanOnce 1 n) andalso
      List.exists (fn x => (#1 x) = fid)
      (Mark.MetaServer'File_Location 1 n)
    then true
    else let val floc = GetFileLocation n fid in
        if isSome(floc)
        then let val (_, loc::_) = valOf(floc);
            val bdata = GetNodeBlock n loc;
            val wdata = GetWrittenData n tid in
              if isSome(bdata) andalso isSome(wdata)
              then #2 (valOf(bdata)) = #2 (valOf(wdata))
              else false end
            else false end end
  end

```

```

fun IsAllWritten init n =
  if List.null(GetWQueries init)
  then true
  else List.all (IsOneWritten n) (GetWQueries init)
val Formula1 = INV(POS(NF("", (IsAllWritten InitNode))));
eval_node Formula1 InitNode

```

Appendix A.2. Auxiliary Functions for Checking Reading Correctness (P3)

```

val NodeNum = 12;
fun GetRQueries n =
  List.filter (fn x => (#2 x) = READ) (Mark.Client'Request 1 n)
fun GetFileLocation n fid =
  List.find (fn x => (#1 x) = fid) (Mark.MetaServer'File_Location 1 n)
fun GetNode n nid 0 = NONE
  | GetNode n nid c =
    if (Mark.Node'NodeID c n) = [nid]
    then SOME c
    else GetNode n nid (c−1)
fun GetNodeStorage n nid =
  let val no = valOf(GetNode n nid NodeNum) in
    Mark.Node'Storage no n end
fun GetNodeBlock n (nid, bid) =
  let val s = GetNodeStorage n nid in
    List.find (fn (bid', _) => bid = bid') s end
fun GetResult n tid =
  List.find (fn x => #1 x) = tid) (Mark.Client'Result 1 n)
fun IsOneRead n q =
  let val (tid, READ, fid, _) = q in
    if List.exists (fn x => x = q) (Mark.Prepare'Read_NotExist 1 n)
    then true
    else let val floc = GetFileLocation n fid in
      if isSome(floc)
      then let val (_, loc::_) = valOf(floc);
        val bdata = GetNodeBlock n loc;
        val rdata = GetResult n tid in
          if isSome(bdata) andalso isSome(rdata)
          then #2 (valOf(bdata)) = #2 (valOf(rdata))
          else false end
        else false end end
fun IsAllRead init n =
  if List.null(GetRQueries init)
  then true
  else List.all (IsOneRead n) (GetRQueries init)
val Formula2 = INV(POS(NF("", (IsAllRead InitNode))));
eval_node Formula2 InitNode

```

Appendix A.3. Auxiliary Functions for Checking Mutual Exclusion (P4)

```

fun duplicated [] = false
  | duplicated (x::xs) = (List.exists (fn y => x = y) xs) orelse (duplicated xs)
fun IsMutualExclusion n =
  not(duplicated (List.concat(Mark.Entry'Writing_File 1 n)))
val Formula3 = INV(NF("", IsMutualExclusion));
eval_node Formula3 InitNode

```

Appendix A.4. Auxiliary Functions for Checking Writing Once (P5)

```

fun duplicated [] = false
  | duplicated (x::xs) = (List.exists (fn y => x = y) xs) orelse (duplicated xs)
fun IsWriteOnce n =
  not(duplicated (List.map (fn x => #3 x)
    (List.filter (fn x => #2 x = WRITE)
      (List.concat(Mark.Confirm'RWRecord 1 n))))))
val Formula4 = INV(NF("", IsWriteOnce));
eval_node Formula4 InitNode

```

Appendix A.5. Auxiliary Functions for Checking Minimal Distance (P6)

```

fun GetFileLocation n fid =
  List.find (fn x => (#1 x) = fid) (Mark.MetaServer'File_Location 1 n)
fun mindist (x, y) [] = raise Empty
  | mindist (x, y) [n] = n
  | mindist (x, y) ((n:NInfo)::ns) =
    let val n' = mindist (x, y) ns;
        val (x1, y1) = (#3 n);
        val (x2, y2) = (#3 n');
        val d1 = (x - x1) * (x - x1) + (y - y1) * (y - y1);
        val d2 = (x - x2) * (x - x2) + (y - y2) * (y - y2) in
        if d1 < d2
        then n else n' end
fun IsOneMinimal n r =
  let val (_, _, fid, pos, [(nid, _)]) = r;
      val nl = List.concat(Mark.MetaServer'NodeInfo 1 n);
      val floc = List.find (fn x => (#1 x) = fid)
        (Mark.MetaServer'File_Location 1 n) in
    if isSome(floc)
    then let val nids = List.map (fn (nid, _) => nid) (#2 (valOf(floc)));
        val rnl = List.filter (fn (nid, _, _) =>
          (List.exists (fn x => x = nid) nids)) nl;
        val ni = mindist pos rnl in
        nid = #1 ni end
    else false end
fun GetRRRecord n =
  List.filter (fn x => (#2 x) = READ)
    (List.concat(Mark.Confirm'RWRecord 1 n))
fun IsAllMinimal n =
  if List.null(GetRRRecord n)
  then true
  else List.all (IsOneMinimal n) (GetRRRecord n)
val Formula5 = INV(NF("", IsAllMinimal));
eval_node Formula5 InitNode

```

Appendix A.6. Auxiliary Functions for Checking Replication Consistency (P7)

```

val NodeNum = 12;
fun GetNode n nid 0 = NONE
  | GetNode n nid c =
    if (Mark.Node'NodeID c n) = [nid]
    then SOME c
    else GetNode n nid (c-1)

```

```

fun GetNodeStorage n nid =
  let val no = valOf(GetNode n nid NodeNum) in
    Mark.Node'Storage no n end
fun GetNodeBlock n (nid, bid) =
  let val s = GetNodeStorage n nid in
    List.find (fn (bid', _) => bid = bid') s end
fun GetNodeBlockList n [] = []
  | GetNodeBlockList n (loc::locs) =
    let val b = GetNodeBlock n loc in
      b::(GetNodeBlockList n locs) end
fun IsOneConsistent n ar =
  let val (_, _, _, loc::rps) = ar;
    val b = GetNodeBlock n loc;
    val bs = GetNodeBlockList n rps in
      (isSome b) andalso List.all (fn b' => (isSome b') andalso
        (#2 (valOf(b)) = #2 (valOf(b')))) bs end
fun GetWRecord n =
  List.filter (fn x => (#2 x) = WRITE)
    (List.concat(Mark.Confirm'RWRecord 1 n))
fun IsAllConsistent n =
  if List.null(GetWRecord n)
  then true
  List.all (IsOneConsistent n) (GetWRecord n)
val Formula6= INV(POS(NF("", IsAllConsistent)));
eval_node Formula6 InitNode

```

References

1. Mell, P.; Grance, T. The NIST definition of cloud computing. In *National Institute of Standards and Technology Special Publication 800-145*; U.S. Department of Commerce: Gaithersburg, MD, USA, 2011; pp. 1–7.
2. Li, S.; Liu, H.; Li, W.; Sun, W. An optimization framework for migrating and deploying multiclass enterprise applications into the cloud. *IEEE Trans. Serv. Comput.* **2023**, *16*, 941–956. [CrossRef]
3. Wang, L. Establishing enterprise information management platform using cloud storage technology under e-commerce environment. *Soft Comput.* **2024**, *28*, 3649–3665. [CrossRef]
4. Zhou, W.; Liu, L.; Zhang, P.; Lü, S.; Li, J. SDAC: A model for analysis of the execution semantics of data processing framework in cloud. *Comput. Lang. Syst. Struct.* **2018**, *54*, 406–426. [CrossRef]
5. He, Z.; Chen, Y.; Luo, Y.; Zhang, L.; Tang, Y. Revocable and traceable undeniable attribute-based encryption in cloud-enabled e-health systems. *Entropy* **2024**, *26*, 45. [CrossRef] [PubMed]
6. HDFS Architecture. Available online: <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html> (accessed on 12 June 2024).
7. Xie, W.; Zhu, H.; Wu, X.; Xiang, S.; Guo, J.; Vinh, P.C. Modeling and verifying HDFS using process algebra. *Mob. Netw. Appl.* **2017**, *22*, 318–331. [CrossRef]
8. Lahouij, A.; Hamel, L.; Graiet, M. Formal reconfiguration model for cloud resources. *Softw. Syst. Model.* **2023**, *22*, 225–245. [CrossRef]
9. Mammar, A.; Belguidoum, M.; Hiba, S.H. A formal approach for the correct deployment of cloud applications. *Sci. Comput. Program.* **2024**, *232*, 103048. [CrossRef]
10. Ölveczky, P.C. Design and validation of cloud storage systems using formal methods. In Proceedings of the International Conference on Topics in Theoretical Computer Science, Tehran, Iran, 12–14 September 2017; pp. 3–8.
11. Jensen, K.; Kristensen, L.M. Colored Petri nets: A graphical language for formal modeling and validation of concurrent systems. *Commun. ACM* **2015**, *28*, 61–70. [CrossRef]
12. Ali, N.; Punnekkat, S.; Rauf, A. Modeling and safety analysis for collaborative safety-critical systems using hierarchical colored Petri nets. *J. Syst. Softw.* **2024**, *210*, 111958. [CrossRef]
13. Yu, W.; Kong, J.; Ding, Z.; Zhai, X.; Li, Z.; Guo, Q. Modeling and analysis of ETC control system with colored Petri net and dynamic slicing. *ACM Trans. Embed. Comput. Syst.* **2024**, *23*, 14. [CrossRef]
14. Entezari-Maleki, R.; Etesami, S.E.; Ghorbani, N.; Leonel, S.; Ali, M. Modeling and evaluation of service composition in commercial multiclouds using timed colored Petri nets. *IEEE Trans. Syst. Man Cybern.* **2020**, *50*, 947–961. [CrossRef]

15. Liu, R.; Delgado-Frias, J.G.; Boyce, D.; Yi, Q.; Rahul, K. On-line firmware functional validation scheme using colored Petri net model. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 1532–1545. [\[CrossRef\]](#)
16. Wu, D.; Lu, D.; Tang, T. Qualitative and quantitative safety evaluation of train control systems (CTCS) with stochastic colored Petri nets. *IEEE Trans. Intell. Transp. Syst.* **2022**, *23*, 10223–10238. [\[CrossRef\]](#)
17. Attia, H.B.; Kahloul, L.; Benhazrallah, S.; Bourekkache, S. Using hierarchical timed coloured Petri nets in the formal study of TRBAC security policies. *Int. J. Inf. Secur.* **2020**, *19*, 163–187. [\[CrossRef\]](#)
18. Lages, D.; Borba, E.; Tavares, E.; Balieiro, A.; Souza, E. A CPN-based model for assessing energy consumption of IoT networks. *J. Supercomput.* **2023**, *79*, 12978–13000. [\[CrossRef\]](#)
19. Bal dit Sollier, L.; Ourghanlian, A.; Amari, S. Coloured Petri nets for temporal performance evaluation of distributed control systems—Application to a FIFO queue. *IEEE Robot. Autom. Lett.* **2022**, *7*, 11268–11274. [\[CrossRef\]](#)
20. Jensen, K.; Kristensen, L.M. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*, 1st ed.; Springer Publishing Company, Incorporated: Berlin/Heidelberg, Germany, 2009; pp. 1–384.
21. CPN Tools: A Tool for Editing, Simulating, and Analyzing Colored Petri Nets. Available online: <http://cpntools.org/> (accessed on 12 June 2024).
22. MSCFS-RP. Available online: <https://github.com/TURTING-BO/MSCFS-RP> (accessed on 12 June 2024).
23. Ölveczky, P.C. Design and validation of cloud storage systems using rewriting logic. In Proceedings of the 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, Romania, 4–7 September 2019; pp. 17–21.
24. Grov, J.; Ölveczky, P.C. Formal modeling and analysis of Google’s Megastore in Real-Time Maude. In *Specification, Algebra, and Software*; Springer Publishing Company, Incorporated: Berlin/Heidelberg, Germany, 2014; pp. 494–519.
25. Liu, S.; Muntasir, R.R.; Stephen, S.; Indranil, G.; José, M. Formal modeling and analysis of Cassandra in Maude. In Proceedings of the 16th International Conference on Formal Engineering Methods, Luxembourg, 3–5 November 2014; pp. 332–347.
26. Yin, J.; Zhu, H.; Vinh, P.C. Formalization and analysis of haystack architecture from process algebra perspective. *Mob. Netw. Appl.* **2020**, *25*, 1125–1139. [\[CrossRef\]](#)
27. Fang, Y.; Zhu, H.; Lu, G.; Xiao, L.; Xie, W. Modeling and verifying MooseFS in CSP. In Proceedings of the IEEE 42nd Annual Computer Software and Applications Conference, Tokyo, Japan, 23–27 July 2018; pp. 270–275.
28. Liu, Z.; Fan, G.; Yu, H.; Chen, L. Modelling and analysing the reliability for microservice-based cloud application based on predicate Petri net. *Expert Syst.* **2022**, *39*, e12924. [\[CrossRef\]](#)
29. Jammal, M.; Kanso, A.; Heidari, P.; Shami, A. Evaluating high availability-aware deployments using stochastic Petri net model and cloud scoring selection tool. *IEEE Trans. Serv. Comput.* **2021**, *14*, 141–154. [\[CrossRef\]](#)
30. Li, X.; Liu, Y.; Lin, Y.; Xiao, L.; Zio, E.; Kang, R. A generalized petri net-based modeling framework for service reliability evaluation and management of cloud data centers. *Reliab. Eng. Syst. Saf.* **2021**, *207*, 107381. [\[CrossRef\]](#)
31. Rizwan Ali, M.; Ahmad, F.; Hasanain Chaudary, M.; Ashfaq Khan, Z.; Alqahtani, M.A.; Saad Alqurni, J.; Ullah, Z.; Khan, W.U. Petri Net based modeling and analysis for improved resource utilization in cloud computing. *PeerJ Comput. Sci.* **2021**, *7*, e351. [\[CrossRef\]](#) [\[PubMed\]](#)
32. Zhou, W.; Dague, P.; Liu, L.; Ye, L.; Zaidi, F. A coloured Petri nets based attack tolerance framework. In Proceedings of the 27th Asia-Pacific Software Engineering Conference, Singapore, 1–4 December 2020; pp. 159–168.
33. Manual, A. Design/CPN ASK-CTL Manual. Available online: <https://cpntools.org/wp-content/uploads/2018/01/askctlmanual.pdf> (accessed on 10 July 2024).
34. Standard ML Family GitHub Project. Available online: <https://smlfamily.github.io/> (accessed on 10 July 2024).
35. CPN Tools: Temporal Logic for State Spaces. Available online: <https://cpntools.org/2018/01/15/temporal-logic-for-state-spaces/> (accessed on 10 July 2024).
36. CPN Tools: Getting Started. Available online: <https://cpntools.org/2018/01/16/getting-started/> (accessed on 10 July 2024).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.