International Journal of Software Engineering and Knowledge Engineering


World Scientific
www.worldscientific.com

# A RAID-BASED SECURE AND FAULT-TOLERANT MODEL FOR CLOUD INFORMATION STORAGE*

DANIEL FITCH† and HAIPING XU‡

*Computer and Information Science Department, University of Massachusetts Dartmouth*
*North Dartmouth, MA 02747, USA*
†*daniel.fitch@umassd.edu*
‡*hxu@umassd.edu*

Received (15 November 2012)
Revised (24 February 2013)
Accepted (3 April 2013)

Cloud computing allows for access to ubiquitous data storage and powerful computing resources through the use of web services. There are major concerns, however, with data security, reliability, and availability in the cloud. In this paper, we address these concerns by introducing a novel security mechanism for secure and fault-tolerant cloud information storage. The information storage model follows the RAID (Redundant Array of Independent Disks) concept by considering cloud service providers as independent virtual disk drives. As such, the model utilizes multiple cloud service providers as a cloud cluster for information storage, and a service directory for management of the cloud clusters including service query, key management, and cluster restoration. Our approach not only supports maintaining the confidentiality of the stored data, but also ensures that the failure or compromise of an individual cloud provider in a cloud cluster will not result in a compromise of the overall data set. To ensure a correct design, we present a formal model of the security mechanism using hierarchical colored Petri nets (HCPN), and verify some key properties of the model using model checking techniques.

*Keywords*: Cloud computing; information storage; data security; fault tolerance; RAID; colored Petri nets; formal modeling and verification; model checking.

## 1. Introduction

As the Internet continues to evolve, service-oriented systems are becoming more widely adopted by large companies and government into their computing platforms. Cloud computing extends this concept, allowing for access to ubiquitous data storage and powerful computing resources by the general public through the use of web services. Although there is a large push towards cloud computing, there is a lack of work in the field of data security, ownership and privacy in cloud computing. A survey conducted by the US Government Accountability Office (GAO) states that "22 of 24 major federal agencies reported that they were either concerned or very concerned about the potential information security risks associated with cloud computing" [1]. Due to the infancy of cloud computing, there are not many standards or best practices in terms of securing data in the cloud. In addition, there is a diverse group of companies emerging as potential

* Corresponding author: Dr. Haiping Xu, Associate Professor, Computer and Information Science Department, University of Massachusetts Dartmouth, Email: hxu@umassd.edu.

cloud service providers, some with large financial backings such as Amazon and Microsoft, and some startup businesses attempting to capitalize on the increasing popularity of cloud computing. For those smaller entities especially, there are no guarantees that they have the resources available to provide successful services and follow best practices for securing their data centers. Examples of such best practices include creating secure facilities with environmental and burglar alarms, properly vetting employees that work closely with data stored in the cloud, and following software and hardware manufacturer's suggestions and best practice configurations. In addition, the startup companies may not have the financial or business backing required to thrive in the market. There have already been instances where smaller companies have failed to provide reliable cloud computing services. For instance, "The Linkup," an online cloud storage provider, shut down unexpectedly on August 8, 2008 after it announced that it had potentially lost 45% of customers' data residing on their servers [2]. They advised customers to retrieve the data that still existed on their shares and if they were missing any data, there would be no guarantee that it could be retrieved. Even the largest providers have not been exempted from such issues surrounding cloud computing. For example, Gmail, Google's cloud-based e-mail service, experienced a major outage in 2009, which prevented users worldwide from accessing their e-mails for approximately two and a half hours [3]. In addition, services are changing frequently as product offerings are developed and discontinued. These changes would leave users of the service scrambling to find alternatives, being forced to take a migration path etched by the provider, or being stuck using a service that was no longer being developed, refined, and patched. For example, Microsoft Windows Live Spaces, a cloud-based online blogging site, was discontinued at the end of March 2011 [4]. Users were forced to either migrate their blogs to Wordpress.com or download the content from their blogs onto their own computer. Microsoft "shifted their strategy," causing 30 million active users to find an alternative or take down their blogging sites. Note that in an enterprise environment, the unresolved issues of widespread outages, loss of data, and service offering changes that cloud computing is currently facing would be considered unacceptable. If the corporate world is to adopt cloud computing, there must be a reasonable guarantee that the stored data is secure, stable, and available in the cloud.

Furthermore, personal information (PI), such as credit card information, that falls under Payment Card Industry (PCI) data security standards legislation [5], or medical records that fall under the Health Insurance Portability and Accountability Act (HIPAA) [6], are especially at question regarding if and how exactly these pieces of data can be stored and managed utilizing cloud computing. Although there are some attempts to address HIPAA compliance in the cloud [7], there exist no widely accepted best practices or clear recommendations as to how such data can be stored in the cloud. Currently, users are advised to seek their own legal counsel on this matter since the providers are offering no liability for misguiding or incorrect advice. In addition, legislative acts, such as HIPAA, were developed with traditional network architectures in mind, with regulations regarding employee training and firewall configurations amongst other components. In a

cloud environment, all or at least most of such implementation details are hidden from cloud consumers, so consumers do not have direct influence or authority over the compliance details. Though related procedures and requests can be specified during contract creation time between the cloud provider and the consumer, this requires complex negotiation and audit procedures that the cloud provider or consumer may not be equipped for or willing to follow. There are efforts in the industry to certify certain cloud providers as being compliant with legislative mandates for handling PI, but no established practice can be found at this time.

In this paper, we focus on information storage, redundancy, and privacy issues in cloud computing. We extend our previous work [8] by developing a security model that addresses these issues of cloud computing, easing concerns of legislatures and enterprise of storing data in the cloud. Our approach aims for securing critical data with a reasonable size that is under privacy and confidentiality regulations. We attempt to create an agile and resilient solution in the cloud, so that if a provider became unresponsive, fell below a certain service level, or discontinued providing a service, the storage array could seamlessly transfer its data to another provider. As a result, the stored information will still be widely available to its consumers with the transition process being transparent to the consumers. In addition, we attempt to secure the data so that if the information stored at a particular storage provider were to be compromised or leaked, the overall data stored in the cloud would not be revealed. As a part of our security strategy, encryption keys are utilized in our approach. Furthermore, we attempt to extend the level of security so that, under normal operating procedures, discovery of the encryption keys used to encrypt the data in the cloud would not be sufficient to compromise the data set.

The rest of the paper is organized as follows. Section 2 discusses previous work related to our research. Section 3 presents a motivating example and an architectural design of our secure and fault-tolerant cloud information storage model, and introduces the major operations such as *read*, *write* and *restore*. Section 4 presents a formal colored Petri net model for a cloud information storage system using the proposed architecture, followed by the analysis of some key properties of the model in Section 5. Section 6 concludes the paper and mentions future work.

## 2. Related Work

Although cloud computing is still in its infancy, there has been a considerable amount of work on distributed data security and federation for distributed data, to which this work is closely related. Goodrich *et al.* explored efficient authentication mechanisms for web services with unknown providers [9]. In their approach, they utilized a third party notary service that could ensure users the trustworthiness of the service providers. Weaver studied the topic of exploring data security through the use of web services [10]. In his approach, he placed authentication and authorization services between the clients and the web services that they were trying to access in order to authorize users. He explored the issues of federation and authentication related to web services, which could be extended towards securing cloud computing. Santos *et al.* provided efforts to establish a secure and

trusted cloud computing environment [11]. They, however, assumed that providers could prevent physical attacks to their servers, which might not be true in the case of a poorly vetted employee or a poorly designed facility. Hwang and Li used data coloring and software watermarking techniques to protect shared data objects and massively distributed software modules [12]. Their approach supports multi-way authentications, enables single sign-on in the cloud, and enforces access control for sensitive data in both public and private clouds. Liu *et al.* further proposed a data coloring method based on cloud watermarking to recognize and ensure mutual reputation between data owners and storage service providers [13]. The proposed approach can guarantee a user's embedded social reputation identification. Although the data coloring and watermarking mechanisms can be useful for cloud security, it is not clear how such approaches perform better than traditional encryption methods. Different from the above approaches, in our method, we address the data security, fault tolerance, and privacy issues in cloud computing by developing a novel cloud information storage model. Our proposed model utilizes a cloud cluster with multiple cloud providers to leverage the benefits of storing data in the cloud while minimizing the risks associated with storing data in an offsite, insecure, unregulated and possibly noncompliant atmosphere.

Fault tolerance as it pertains to cloud computing has also gained momentum, as work related to producing fault-tolerant applications, computation, and storage in the cloud emerges. Deng *et al.* explored methods to create fault-tolerant computation in the cloud [14]. They examined the operation of matrix multiplication, a basis operation to solving multiple scientific issues, and offered techniques to outsource the computation of such operation to cloud providers while improving fault tolerance and reliability. However, they assumed that the reliability of a cloud provider was a discrete value that could be ascertained from previous knowledge and experience of the provider. This may not be always true, as even the most reputable cloud providers may have periods of unavailability and outages. Zhao *et al.* focused on the fault tolerance of cloud-based applications, and developed a software system called Low Latency Fault Tolerance (LLFT) middleware for distributed applications that could be deployed in a cloud computing or data center environment [15]. Their work only focused, however, on the application processes in the cloud, giving no guidance for data retention once it resides in the cloud. Cachin *et al.* provided general guidance for users of cloud service offerings on how they can protect themselves from fault and malicious behavior with regards to cloud storage [16]. They pointed out issues in trusting the cloud, including loss of privacy in data stored, the potential to lose data, and loss of access to the stored data due to either misdefined policy for cloud storage or software malfunction from the cloud provider. Different from the above approaches, we follow the RAID (Redundant Array of Independent Disks, originally known as Redundant Array of Inexpensive Disks) concept [17] by considering cloud providers as independent virtual disk drives; therefore, our approach is inherently a fault-tolerant data storage solution.

Finally, there has been a considerable amount of work associated with formal modeling of service-oriented systems. As formal methods provide a strong backbone for

validation of a correct design, it has been used extensively in modeling service-oriented systems and is beginning to gain momentum in its application to cloud computing. Gu and Luo proposed an autonomic Quality of Service (QoS) control that can be adapted to cloud computing providers using Controlled Stochastic Petri Nets (COSTPN), a derivative of the traditional Petri nets [18]. They proposed a mechanism that could automatically manage grid services and resources, and also designed a COSTPN model to produce its performance characteristics. Xu *et al.* introduced a formal XML firewall security model for service-oriented systems using colored Petri nets [19]. The formal security model supports user authentication and Role-Based Access Control (RBAC) according to policy rules that can be updated dynamically. Accorsi and Lowis developed ComCert, which is a Petri net model for compliant business processes of cloud computing providers [20]. Requirements such as regulations HIPAA and the Sarbanes-Oxley Act were drawn upon to create compliant business practice models, and the system was able to detect and generate reports of policy violations. A major weakness of the above proposed approaches is their limited capability to support only a subset of the regulations or policy rules. In contrast, we provide a Petri net based formal model that attempts to secure the data being stored so that even in the case when a provider neglects to adhere to a particular regulation, the overall data being stored in the cloud can still be kept secure. Furthermore, in our approach, we separate the security mechanism and policy rules to allow different applications to use the same mechanism with different policies. In this paper, we focus on describing and modeling the security mechanism, and envision specification of policy rules for specific applications as our future work.

## 3. Secure and Fault-Tolerant Cloud-Based Information Storage

### 3.1. *A Motivating Example*

Consider a scenario where a medical company wishes to have all medical records of its patients available to its trusted partners, as well as to doctors who may be off site. First, the medical data is required to be highly fault tolerant, as losing patient records is not an option. Secondly, the data must be secure, as the company has an obligation to its patients to protect their personal information. Thirdly, the medical records must be guaranteed to be available, as it may become a matter of life or death if the data cannot be accessed quickly. The company realizes that storing the data on site would require a complex setup to make the data widely available to its central location and branch offices, with a large cost to purchase servers and storage devices. Furthermore, storing the data on site also requires a robust mechanism to ensure that the data is redundant and available in case of disaster, as well as a scalable infrastructure in case of growth. The company is attracted by the benefits of cloud computing, namely the availability of the data over the Internet for its remote offices and doctors, not having to invest a large amount of money to establish the infrastructure, the scalability, and the promise of resiliency and redundancy. Therefore, the company wishes to explore the option of using cloud computing for information storage and archiving of its data. It is, however, very concerned with moving

its data into the cloud since losing physical control of its data would be of high risk. Although the company can choose reputable cloud providers to host its data, there is no way to vet individual employees who are hired by the cloud provider to prevent insider attacks, whereas the medical company is required to do full background checks and audits on employees who are allowed to handle its data. The company is also concerned with the physical locations of its data. With a cloud provider, the medical company does not even know where its data resides in the cloud, let alone what safeguards are at place at the physical facility. The company has also seen through the media the amount of damage that can be caused by its data being compromised by a third party. It must assume that by storing its data in the cloud, it can be compromised, so it needs to ensure that the cloud providers and their employees absolutely do not have access to the underlying information. The company is finally concerned with the availability of its data, as although it sees cloud computing as mostly reliable, it needs to make sure that its data is available and that there are no extended length of outages. When the medical company is treating a patient, for example, it is critical to know if the patient is allergic to any medication. If this information became unavailable, there may be dire consequences. If the company chooses to build the data center by itself, it would take into account all of the above concerns. In the cloud, however, the environment is ever-changing with providers having the ability to decide critical implementation details, where data resides, and can at whim discontinue or radically change a service offering. Given the current state of cloud computing, the healthcare provider would have some very serious and legitimate concerns that need to be addressed.

In order to mitigate the major concerns that the medical company faces, we design a reliable, fault-tolerant, and secure architecture for cloud information storage. Our approach can assist in resolving a major issue that resides in cloud computing, namely how to securely store personal or sensitive information in the cloud. Thus, our approach can mitigate concerns from companies that are trying to adopt cloud computing and regulators as well as the general public who are concerned for the security and confidentiality of the stored information in the cloud.

### 3.2. *An Architectural Design*

Our proposed information storage model for cloud computing can be deployed on multiple cloud service providers. As shown in Fig. 1, the model consists of users, a *Service Directory* (*SD*), and groups of cloud storage providers. The users are cloud clients who wish to store and access data in the cloud. A user can first interact with the *SD*, which acts as a coordinator for cloud information storage. The *SD* first sets up a *Cloud Provider Cluster* (*CPC*) with multiple cloud service providers and assign it to the user. It also stores information regarding the addresses of all service providers in the *CPC*. Once the client obtains the address information of the cloud providers from the *SD*, it interacts with the *CPC*, namely a collection of available service providers that can store and send data using predefined protocols. Each set of data, composed of the user's personal or sensitive information is split into multiple pieces using a predefined security

mechanism, and are stored into the *CPC* after they are encrypted. The cloud providers in the *CPC* have no knowledge of which cluster they belong to as well as which are the other members in the *CPC* that they are servicing. This means the *CPC* are *virtual* clusters in the cloud, which are generated and exclusively managed by the *SD*. Furthermore, the *SD* has the capability to restore data when needed. When a service provider, say $\alpha$, in a *CPC* fails, the *SD* can automatically restore the data using a predefined restoration algorithm, and replace $\alpha$ with another cloud service provider $\beta$ from a collection of cloud providers called *Cloud Provider Pool* (*CPP*). Note that before the replacement happens, the service provider $\beta$ in *CPP* did not provide any services to the user; therefore, it will not charge for usage according to the elasticity feature of storage services in cloud computing.

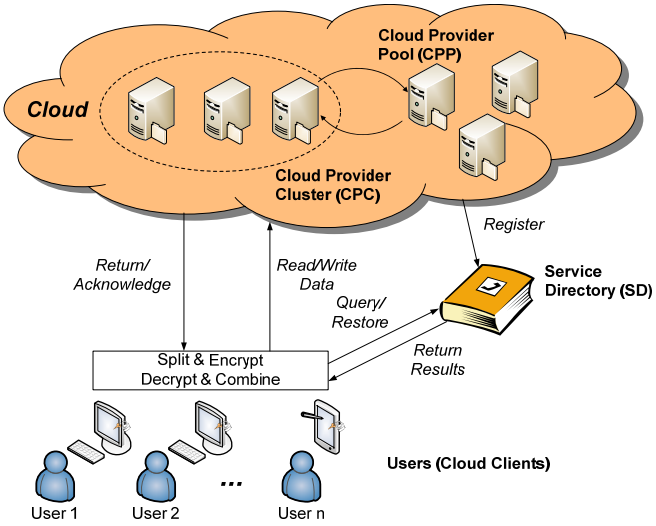


Fig. 1. An architectural design of cloud information storage systems

The security mechanism defined in our model consists of two levels. In the first level, the information to be stored is split into multiple pieces using the RAID 5 technique with distributed parity [21] so that if a provider fails, the data stored collectively in the cluster would be recoverable. Note that the RAID 5 technique uses block-level striping with distributed parity in a cluster of disk drives [22]. Due to data redundancy, when a disk drive fails, any subsequent reads can be calculated from the distributed parity, and the data in the failed drive can be restored. In our approach, we consider each cloud provider in a *CPC* as a *virtual* disk drive; thus, our information storage model is fault tolerant upon the failure of any cloud provider in the *CPC*, and the missing piece of data can be recovered from the distributed parity blocks stored with the other cloud providers. Another advantage of our approach is, due to the distribution of data over multiple cloud providers in a *CPC*, no cloud provider is able to calculate the original data because it has no knowledge of who are the other members in the *CPC*.

The second level of our security mechanism provides another layer of security in our approach, where encryption plays an important role. To ensure that providers do not have access to the underlying data that is being stored, symmetric key encryption can be used. A symmetric key is an encryption key that is used for both encryption and decryption of data and should be kept secret from all entities that do not have access to the data. Prior to sending a data block *DB* out to a provider in a *CPC*, a user *U* first encrypts *DB* using a symmetric key into $DB' = E(k_{SYM\text{-}CPC}, DB)$, where $k_{SYM\text{-}CPC}$ is the symmetric key associated with the *CPC*. The data block remains encrypted while in the cloud and is only decrypted by the user upon retrieval of the data into $DB = D(k_{SYM\text{-}CPC}, DB')$. Note that cloud providers are never given access to these keys so that the data stored on their services are not readily compromised. In addition to symmetric key encryption, there needs to be some mechanism in place where the different entities of the storage model could positively identify themselves to one another, so that impersonation attacks could not take place. In order to fill this need, asymmetric key encryption is used in this model. Cloud users and cloud service providers are all assigned asymmetric keys, individual to each entity. Asymmetric key encryption consists of a pair of keys, namely a public key and a private key. The public key is widely published and available, whereas the private key should be kept secret. In our application, we utilize asymmetric key encryption to provide a digital signature. A message digest *MD* of a data block is generated by the sender and encrypted using the sender's private key, $k_{PRIV\text{-}S}$, into $MD' = E(k_{PRIV\text{-}S}, MD)$. The recipient receives the data block along with the encrypted signature and decrypts the signature using the sender's public key into $MD = D(k_{PUB\text{-}S}, MD')$. The recipient then verifies the data block and its message digest to ensure that the data block was sent by the claimed sender and was not compromised in transmission. The private key is issued by the *SD* and stored on the individual client's machine, whereas the public key is published in the *SD*, which can be retrieved by any users or service providers. Once the encryption is done, the data blocks being stored are distributed to the cloud providers in a RAID 5 fashion with distributed parity so that if a particular provider were to fail or be compromised, the data stored collectively in the *CPC* can be recovered. On the other hand, when a *read* operation is performed, all pieces of information need to be decrypted after being retrieved from the cloud providers in the *CPC*, and then they are combined into the original information.

To formalize the basic idea of our approach, we now provide the definitions of a few key concepts used in this paper.

**Definition 3.1.** *Cloud Provider Cluster* (*CPC*)
A *Cloud Provider Cluster* (*CPC*) $\Theta$ is a 3-tuple (*RNP*, *SAP, SIP*), where *RNP* is the required number of active cloud providers in the *CPC*, which must be no less than 3; *SAP* is a set of active cloud providers in the cluster; and *SIP* is a set of inactive cloud providers in the *CPC*, where $SAP \bigcap SIP = \phi$. If an element $\alpha$ in *SAP* becomes unavailable, it will be replaced by another element $\beta$ from *SIP* or from the cloud provider pool. In this case, $\alpha$ will be deposited into *SIP* for future usage; however, if it becomes unavailable for too long, it will be permanently removed from *SIP*.

**Definition 3.2.** *Cloud Provider Pool* (*CPP*)
The *Cloud Provider Pool* (*CPP*) Ω is a 2-tuple (*NRP*, *SRP*), where *NRP* is the number of registered cloud service providers, and *SRP* is a set of registered cloud service providers. For a *CPC* $\Theta$, $(\Theta.SAP \bigcup \Theta.SIP) \subset \Omega.SRP$, and for $\forall \alpha \in \Omega.SRP$, $\alpha$ may belong to more than one *CPC*.

**Definition 3.3** *Service Directory* (*SD*)
The *Service Directory* (*SD*) Φ is a 7-tuple (*SRP*, *SCPC*, *SCU, SSK, MSK, SPK*, *MPK*), where *SRP* is a set of registered cloud service providers, *SCPC* is a set of cloud provider clusters, *SCU* is a set of cloud users, *SSK* is a set of symmetric keys, *MSK* is a mapping function defined as $MSK : SCPC \mapsto SSK$, *SPK* is a set of public keys, and *MPK* is a mapping function defined as $MPK : (SRP \bigcup SCU) \mapsto SPK$. Note that the symmetric key assigned to a *CPC* is only available to a qualified user of the *CPC*, but it is kept invisible to members of the *CPC*.

**Definition 3.4.** *Cloud User* (*CU*)
A *Cloud User* (*CU*) is a 3-tuple (*UID*, *PRIKEY*, *CPC*), where *UID* is a user identifier, *PRIKEY* is a private key of the user, and *CPC* is a cloud provider cluster assigned to the user by the *SD*.

**Definition 3.5.** *Registered Cloud Provider* (*RCP*)
A *Registered Cloud Provider* (*RCP*) is a 2-tuple (*PID*, *PRIKEY*), where *PID* is a provider identifier, *PRIKEY* is a private key of the provider, and for any *RCP* $\alpha$, $\alpha \in \Omega.SRP$, where Ω is the *CPP*. Note that a *RCP* has no knowledge about which *CPC* it belongs to.

### 3.3. *RAID-Based Data Distribution and Integration*

In order to provide redundancy for the underlying data, we use an adaptation of the RAID 5 approach in our cloud storage model. In a traditional RAID 5 based system, data is distributed among multiple disk drives using a distributed parity-based mechanism. We extend this idea to multiple cloud providers in our cloud information storage model as illustrated in Fig. 2, which shows the RAID-based design of a *CPC* with 3 cloud providers. This design provides for resiliency and efficiency. If any cloud provider in a *CPC* becomes unavailable, the system will still be able to function due to data redundancy. Once the failed provider is replaced, the directory is able to restore the missing information by calculating the missing data blocks or parity blocks using the remaining ones, and recover the storage cluster back to a redundant state. Note that in our approach, data is stored at the block level. Let *n* be the number of providers per *CPC*, for every *n*-1 blocks of data, one block of parity is generated. For example, in Fig. 2, the *Parity Block* 1 stored with *Cloud Provider* 1 is calculated based on *Data Blocks* 1_1 and 1_2, which are stored with *Cloud Providers* 2 and 3, respectively. This parity block provides data redundancy, which allows the *SD* to recover missing data blocks so that users may continue to read data from the storage array even when one of the providers were unresponsive.

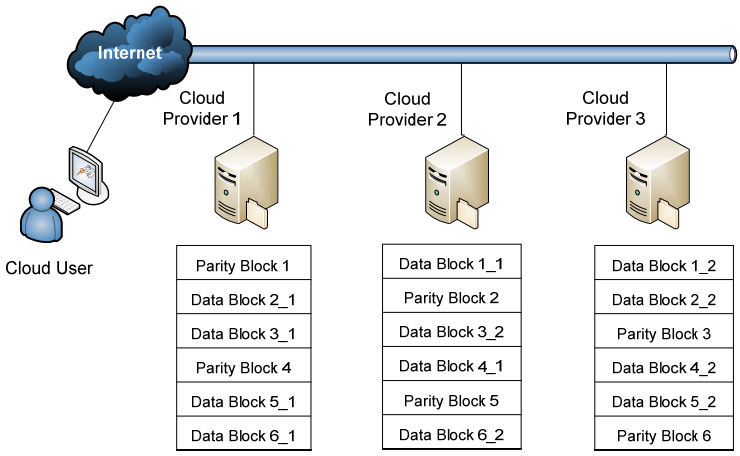


Fig. 2. RAID-based design of Cloud Provider Cluster (CPC)

Based on the RAID 5 technique, block-level striping is used and parity blocks are distributed evenly among the cloud providers of a *CPC*. As demonstrated in Fig. 2, there are 6 stripes of data with 12 data blocks and 6 parity blocks that are distributed evenly amongst 3 cloud providers in the *CPC*. Using this simple RAID 5 scheme, the number of data blocks and the number of stripes are calculated as in Eqs. (1) and (2).

$$n_{block} = \left\lceil \frac{n_{datasize}}{n_{blocksize}} \right\rceil \tag{1}$$

$$n_{stripe} = \left\lceil \frac{n_{block}}{n_{cloudprovider} - 1} \right\rceil \tag{2}$$

where $n_{block}$ is the total number of data blocks, $n_{datasize}$ is the total amount of data (in bytes), $n_{blocksize}$ is the predefined block size (in bytes), $n_{stripe}$ is the number of stripes, and $n_{cloudprovider}$ is the required number of active cloud providers in a *CPC*. Note that the function *ceiling*($x$) = $\lceil x \rceil$ is the smallest integer not less than $x$; thus, if $n_{datasize}$ is not a multiple of $n_{blocksize}$, the last block containing the remaining data will be filled up with random data. Similarly, if $n_{block}$ is not a multiple of ($n_{cloudprovider}$ - 1), the last stripe shall be filled up by data blocks containing random data. Once the data is divided into blocks of the established size, and further organized into stripes based on the number of active cloud providers in a *CPC*, parity blocks can be calculated using *XOR* operations (denoted as $\oplus$ ) as in Eq. (3).

$$P_i = B_{i\_1} \oplus B_{i\_2} \oplus \ldots \oplus B_{i\_N-1} \tag{3}$$

where $1 \le i \le n_{stripe}$, $N = n_{cloudprovider}$, and $P_i$ and $B_{i\_m}$ are the $i$-th parity block and the $m$-th data block in stripe $i$, respectively.

With the parity blocks, our approach is fault tolerant because when a cloud provider fails, restoring the data blocks and parity block become as simple as performing the *XOR*

operation on the other blocks. For any stripe *i*, the missing data block or parity block can be calculated using Eq. (4) or (3), respectively.

$$B_{i\_j} = B_{i\_1} \oplus ... \oplus B_{i\_j-1} \oplus B_{i\_j+1} \oplus ... \oplus B_{i\_N-1} \oplus P_i \quad (4)$$

Note that the data blocks and parity blocks may also be distributed according to a predefined algorithm that is only known by the users, but appears random to the cloud providers. This would allow for increased security, as it assists in preventing collusions from cloud providers, where multiple providers from the same *CPC* may attempt to combine their pieces of data to recreate the original information. Not knowing which data is the parity information makes reconstructing the data much more difficult if not impossible. The detailed implementation of such "random" data distribution for RAID is out of the scope of this paper, but is envisioned as a future, and more ambitious research direction.

### 3.4. *Major Operations for Cloud-Based Information Storage*

In order to establish a *CPC*, a cloud user must first communicate with the *SD* to join the service and request that a *CPC* be created. During this process, a trusted third party may perform any necessary background checks on the cloud user to ensure that the user should have access to the cloud. Once the request has been approved, the *SD* sends the user an approval message, followed by an assignment of a public/private key pair, which is used for digital fingerprinting, as discussed in Section 3.2. The private key is distributed to the user and the public key associated with the user is published in the *SD*. Then the *SD* fulfills the user's request by finding *n* suitable cloud providers from the pool of the registered ones, where *n* is the required number of providers in a *CPC*. For each registered cloud provider, the directory also maintains a public/private key pair for digital signature purpose. Thus, any messages originated from a user or a cloud provider must be signed and verified using the asymmetric key pair.

Once initialized, three major operations, namely *read*, *write*, and *restore*, can be performed for cloud information storage. These operations are now described as follows.

**Read Operation.** The *read* operation is initiated by a user to request information from the *CPC* for a particular file. In order to read from the *CPC*, the user first sends a request to the *SD* for the providers' locations. Once the *SD* verifies the digital fingerprint of the user, it sends the user the current addresses of the providers that make up the *CPC* assigned to the user. The user then sends concurrent requests signed with its private key to each of the providers in the *CPC*. Once a provider verifies the signed message, the provider sends the requested data back to the user along with a signed response. The user allows a reasonable amount of time for the providers to respond prior to proceeding. The user then verifies the providers' messages against their public keys, and decrypts the content of the data by utilizing the symmetric key. Once the user receives all pieces of data, it is able to reconstruct the file, which completes the *read* operation. It is interesting to note, however, that due to the fact that the cluster is using a *RAID* 5 based approach, once data has been received from *n*-1 providers, the user is able to read the data due to

data redundancy. This provides a needed level of availability for the requested information. However, the user typically waits for the last response for a reasonable amount of time prior to attempting to calculate the missing information utilizing the parity, as the user needs to make a determination whether the provider is down or not. If the last provider is considered to be down, the user completes its *read* and sends a *restore* request to the *SD* for the last provider's data to be restored to another provider. Although *read* request can continue to complete, it is important that the cluster restores the data from the failed provider to another provider quickly, as while the provider is down, the *CPC* has lost its redundancy. Though it is not likely to happen, if another cloud provider in the *CPC* was to go down prior to the completion of the restoration, the data might be permanently lost.

**Write Operation.** The *write* operation can be used to write information to a *CPC* for a particular file. When the *write* operation is initiated, the user first sends a request, signed with its private key, to the *SD* asking for the addresses of the providers in the *CPC*. If the user is verified and has access to the *CPC*, the *SD* responds with the provider locations. Then, the user calculates the distribution utilizing standard parity distribution, as discussed in Section 3.3, forms the requests, and concurrently sends them to each of the providers. Once each piece of messages is verified and stored at the provider side, each provider sends an acknowledgement message back to the user, signed with its private key. If the user receives responses from all of the providers in a reasonable amount of time, it considers the *write* operation successful; otherwise, it sends a *restore* request to the *SD* to restore data on a possibly failed cloud provider. Once the missing data and the *CPC* are restored, the user can resubmit the uncompleted *write* request.

**Restore Operation.** Finally, in the case where a cloud provider falls below an expected QoS level, becomes unresponsive due to either failure or maintenance, or is deemed compromised, it is necessary to restore the information stored with the provider on a new provider. In a *restore* operation, the user initiates a request to the *SD* to replace a provider servicing the *CPC* with another. This would occur after a provider had not responded in a reasonable amount of time to a *read* or *write* operation or if a provider was sending back incorrectly signed messages. It is the *SD*'s responsibility to maintain a pool of registered providers and identify a suitable replacement when one is required. Once this happens, the *SD* notifies all other providers of the *CPC* for the restoration process. The *CPC* then temporarily becomes "*ReadOnly*," so that any duplicated *restore* request on the *CPC* can be ignored, and no *write* request is allowed to go through while the system is being restored. The *SD* calculates the missing blocks using the remaining data blocks and streams them to a new cloud provider. Once all of the information has been recovered, the *CPC* becomes ready again and can process additional requests.

## 4. Formal Modeling of a Cloud Information Storage System

In order to ensure a correct design of the security mechanism, we develop a formal model of a secure and fault-tolerant cloud information storage system, and verify some key

properties of the model. We adopt colored Petri net (CPN or CP-net) formalism because it is a well-founded process modeling technique that has formal semantics to allow specification, design, verification, and simulation of a complex concurrent software system [23]. A Petri net is a directed, connected, and bipartite graph, in which each node is either a place or a transition. In a Petri net model, tokens are used to specify information or conditions in the places, and a transition can fire when there is at least one token in every input place of the transition. CPN is an extension of ordinary Petri nets, which allows different values (represented by different colors) for the tokens. CPN has a formal syntax and semantics that leads to compact and operational models of very complex systems for modular design and analysis. The major advantage of developing a CPN model of the cloud information storage system is to provide a precise specification, and to ensure a correct design of the information storage model; therefore, design errors, such as a deadlock, can be avoided in the implemented system.

To make the model easy to comprehend, we utilize hierarchical CPN (HCPN), which allows using substitution transitions and input/output ports to represent a secondary Petri net in the hierarchy. In our design, we first provide the high-level model with its key components. Then we utilize HCPN to refine each component into a more complete Petri net. Since the architecture we proposed is most suitable for storing personal or confidential data, in the following sections, we present the HCPN model with an example of medical record online storage system, which consists of a *SD*, a *CPC* with three cloud providers, and two clients (cloud users), namely a patient and a doctor.

### 4.1. *High-Level Petri Net Model*

The HCPN model can be devloped using CPN Tools [24]. In Fig. 3, we present a high-level model that defines the key components, namely the *Doctor*, the *Patient*, the *Cloud*, and the *Directory*, as well as the communications among the components. The key components are defined as substitution transitions, denoted as double rectangles in Fig. 3. The purpose of the communications among the patient, the doctor, and the cloud is to transfer and access the patient's medical record. The directory serves as the *SD*, which acts as a data coordinator between the users and the cloud.

To simulate the cloud providers that are selected as clutser providers (i.e., members of a *CPC*) as well as the data being transferred between the users and the cloud providers, a `PROV` and a `MEDRECORD` colored token type are defined using the ML functional language integrated in CPN Tools as follows:

```
colset PROV = record
  prID: STRING *
  ready: BOOL *
  mrec: MEDRECORD;
```

```
colset MEDRECORD = record
  recID:STRING *
  data: STRING;
```

where `prID` is a provider ID, `ready` is a flag of the provider indicating whether the provider is functioning or failed, `mrec` is a medical record, `recID` is a record ID, and `data` is the medical data stored in the record.

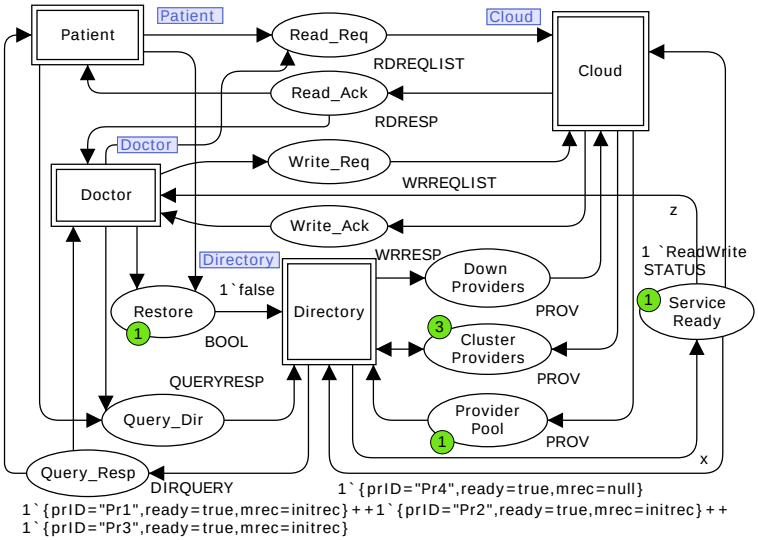


Fig. 3. High-level CPN model of the medical record cloud storage system

The directory is responsible for initializing the cloud providers in a cloud cluster assigned to a user, replying queries from a user for providers' addresses, and processing *restore* request upon the failure of a cloud provider in the cloud cluster. As shown in Fig. 3, the cloud cluster or *CPC* (denoted as the place "*Cluster Providers*") is initialized with three providers `"Pr1"`, `"Pr2"` and `"Pr3"` of type `PROV`, each of which is initialized with `initrec` that contains a blank medical record. Furthermore, the place "*Provider Pool*" is initialized with one spare cloud provider `"Pr4"`, which can be used to replace a failed cloud provider in the cloud cluster when needed. A read request (`RDREQ`) and a write request (`WRREQ`) to a cloud provider can be defined as colored tokens as follows:

```
colset RDREQ = record
  clID:STRING *
  recID:STRING *
  prID:STRING;
```

```
colset WRREQ = record
   clID:STRING *
   mrec:MEDRECORD *
   prID:STRING;
```

where `clID` is a client ID. Note that in Fig. 3, `RDREQLIST` and `WRREQLIST` are defined as a list of *read* requests, and a list of *write* requests, repectively. Thus, our model allows accessing multiple pieces of information *concurrently* from the cloud providers participating in a cloud cluster. After a *read* (*write*) request has been processed, a *read* (*write*) response will be returned to the user, simulated as a token of type `RDRESP` (`WRRESP`) being deposited in place "*Read_Ack*" ("*Write_Ack*"). The colored token types `RDRESP` and `WRRESP` are defined as follows:

```
colset RDRESP = record
  clID:STRING *
  prID:STRING *
  mrec:MEDRECORD *
  success:BOOL;
```

```
colset WRRESP = record
  clID:STRING *
  prID:STRING *
  mrec:MEDRECORD *
  success:BOOL;
```

where the flag `success` indicates if a *read* request or a *write* request is successful or failed. In case a *read* or *write* request fails (i.e., a cloud provider is *down*), the user will change the token in place "*Restore*" from *false* to *true*, and notify the directory to start the restoration process for the cloud cluster.

### 4.2. *Petri Net Model for the Directory Component*

We now refine the *Directory* component (i.e., the *Directory* substitution transition in Fig. 3) into a CPN model as shown in Fig. 4.

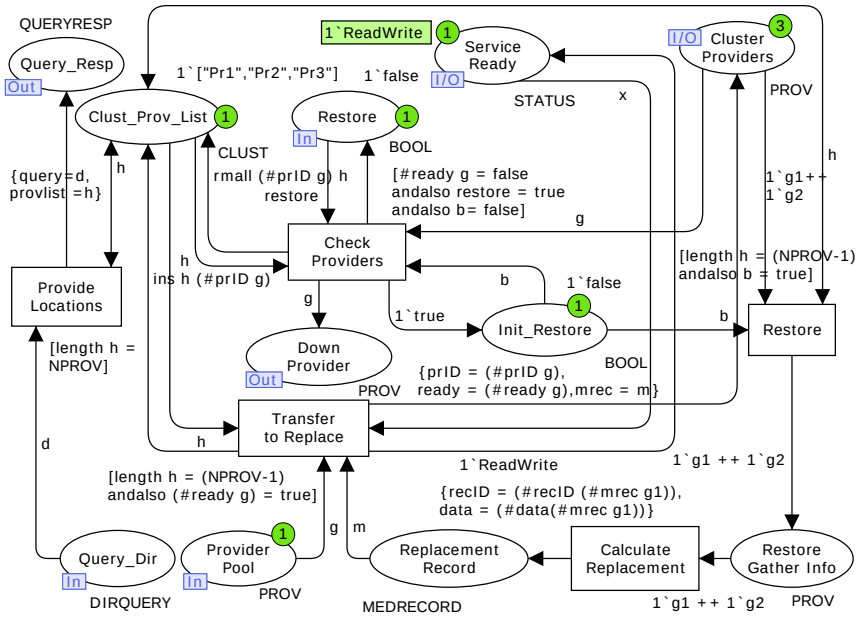


Fig. 4. CPN model for the Directory component

In Fig. 4, the place "*Clust_Prov_List*" is initialized with a list of providers `["Pr1","Pr2","Pr3"]` due to the initial setting of the cloud cluster in place "*Cluster Providers*." When a patient client or a doctor client starts querying the directory for the addresses of the providers in its assigned cloud cluster, a query token will be placed by the client into place "*Query_Dir*." This enables the transition "*Provider Locations*." When it fires, it creates a token of `QUERYRESP` type in place "*Query_Resp*," which attaches the provider information stored in place "*Clus_Prov_List*." Since the place "*Query_Resp*" is an input port of the clients, the token becomes available to the client for further processing. Note that to simplify our CPN model, the provider information only consists of the provider IDs rather than the providers' actual endpoint addresses. Therefore, a service invocation to a cloud provider is simulated by matching the cloud provider's ID rather than calling the service at its endpoint address. On the other hand, if the "*Restore*" place contains a `true` token due to an access error experienced by a user, the "*Check Providers*" transition becomes enabled as long as the directory is not

currently restoring the cloud cluster (denoted by a `false` token in place "*Init_Restore*") and there is a failed provider, whose `ready` flag is set to `false`, in place "*Cluster Providers*." Once the transition fires, it places a `true` token into the "*Init_Restore*" place, signifying that a restoration process should take place. The firing also removes the failed provider from the "*Clust_Prov_List*" place and transfers the provider from the "*Cluster Providers*" place to the "*Down Provider*" place. When the restoration process starts, the "*Restore*" transition fires, and deposits a copy of the remaining two providers into the "*Restore Gather Info*" place. This enables the "*Calculate Replacement*" transition, and its firing simulates the calculation of the missing piece of data based on the distributed parity information, and results in the restored medical record being placed in the "*Replacement Record*" place. Note that for simplicity, the detailed procedure of the parity calculation is not modeled in Fig. 4.

### 4.3. *Petri Net Model for the Patient and Doctor Clients*

A patient client should have the permission to read its own medical record. As shown in Fig. 5, a patient first requests the addresses of the cloud providers in the cloud cluster assigned to her, which is modeled by placing a `true` token in the "*Query Directory*" place. With this token as well as the client ID of the patient in the "*ClientID*" place, the "*Init_Query*" transition can fire, and the firing results in a `DIRQUERY` token to be placed in the "*Query_Dir*" output port. When a response from the directory is put into the "*Query_Resp*" input port, the providers' address information becomes available. This enables the "*Extract Providers*" transition, and the firing of the transition places a `CLUST` token in the "*Provider Locations*" place. The `CLUST` token type is defined as a list of providers as follows:

```
colset CLUST = list STRING with 0..3;
```

where the `with` clause specifies the minimum and maximum length of the list, and each item in the list contains the address of a provider (represented by its provider ID as a string for simplicity) that can be used by the client to communicate with the provider. To model a *read* operation, a token `"P1.rec"` is initialized in the "*Data to Read*" place, which is a record ID representing patient `P1`'s medical record. The firing of the "*Init_Read*" transition starts the *read* process, and places the record ID along with the provider information into the "*Read Information*" place.

Note that in this model, we assume that there is only one record for each patient that can be matched with the medical data stored with the providers. Now the "*Construct Read Req*" transition can fire once for each provider in the provider list, and creates a token of type `RDREQLIST` in the "*Read Request*" place, such that the multiple *read* requests in the list can be made concurrently to the cloud providers in the cloud cluster. This makes the associated providers in place "*Read Information*" being removed and enables the "*Start Read*" transition. When it fires, it transmits the `RDREQLIST` token to the "*Read_Req*" place, which is an input port to the cloud cluster. After the requests have been processed by the cloud providers, multiple tokens of type `RDRESP` will be deposited

in place "*Read_Ack*." If a `RDRESP` token contains a `success` flag with a `true` value, it indicates that the *read* request has been completed successfully by the corresponding cloud provider. In this case, the piece of medical record is extracted from the token and placed in the file store after being decrypted. Once all pieces of the medical record are successfully decrypted, the "*Combine Data*" transition becomes enabled and can fire. The firing simulates the process of generating the original medical record by recombining the data slices retrieved from the cloud providers. If one of the providers returns a token with the `success` flag set to `false`, a *read* failure occurs for the cloud provider. In this case, the "*Read Fail*" transition becomes enabled. Once it fires, it changes the token in place "*Restore*" from `false` to `true`, signifying the directory to initiate a "restore" operation.

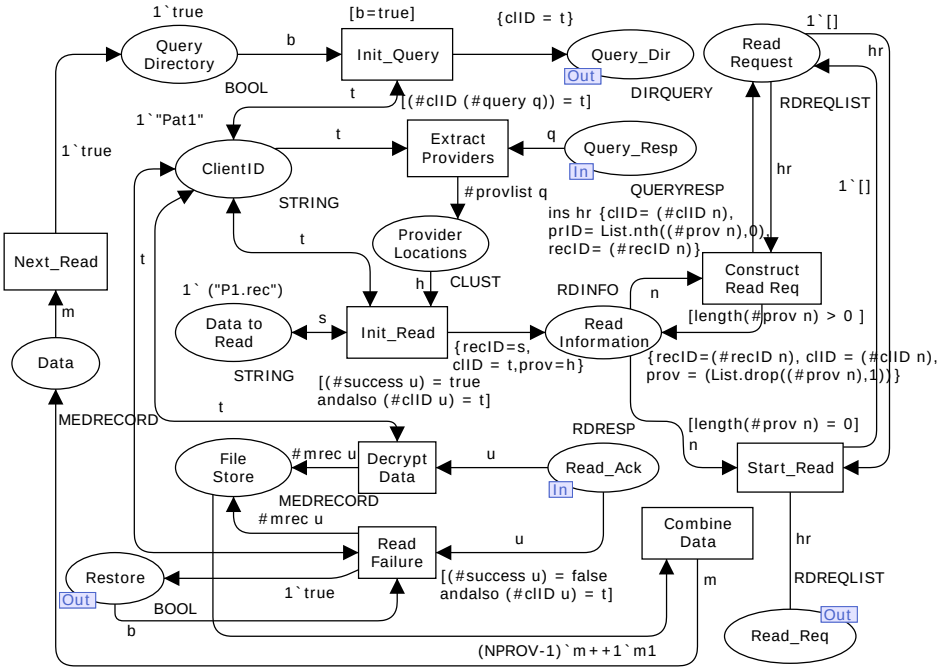


Fig. 5. CPN model for the patient client

The CPN model for the doctor client is similar to the one for the patient client, but a doctor client should also have the privilege to write data into the cloud. The CPN model for the doctor client that replaces the `Doctor` substitution transition of the high-level model is illustrated in Fig. 6. Before the doctor client makes a *write* request, the "*Data to Write*" place is initialized with the medical data that the client wishes to write to the cloud cluster. For simulation purpose, we set the data to be written as `"This is Pat1's medical record!"`. Similar to a *read* request, once the "*Init_Write*" transition fires, it places the information to be written along with the provider information into the "*Write Information*" place. Once the required information is ready, the transition

"*Split_Encrypt_Data*" will fire once for each provider in the provider list, which splits the data using the RAID 5 techniques and encrypts the data using a symmetric key.

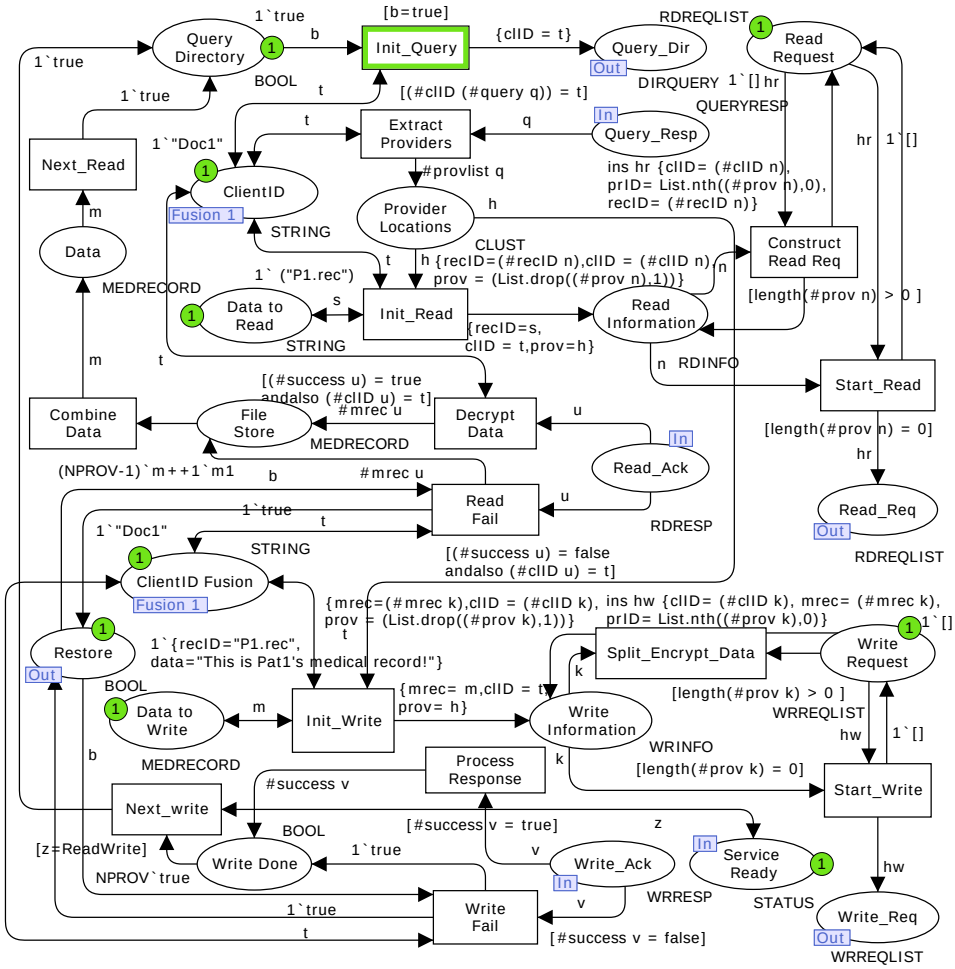


Fig. 6. CPN model for the doctor client

Once the token of type `WRREQLIST` in "*Write Request*" place contains the correct number of *write* requests, the "*Start_Write*" transition becomes enabled. When it fires, it transmits the `WRREQLIST` token to the "*Write_Req*" place, which is an input port to the cloud cluster. After the cloud providers process the *write* requests, each of them will place a token of type `WRRESP` in place "*Write_Ack*." The `WRRESP` token contains the information of the provider where the acknowledgment is coming from, the *write* request, and a `success` flag. Similar to a *read* response, the `success` flag represents whether or not the "write" operation was successfully performed by the corresponding cloud provider. If the flag is set to `true`, the "*Process Response*" transition may fire, and when it fires, it places a `Boolean` token into the "*Write Done*" place. On the other hand, if the

`success` flag of any response is `false`, the "*Write Fail*" transition becomes enabled. When it fires, it changes the token in place "*Restore*" from `false` to `true`, signifying the directory to initiate a "restore" operation. In either case, the "*Next Write*" transition may fire, either to start a new "write" operation or rewrite the previous medical information.

### 4.4. *Petri Net Model for the Cloud Component*

Finally, we refine the *Cloud* substitution transition of the high-level model into a CPN model as shown in Fig. 7, where the cloud providers are represented as colored tokens of type `PROV`. The clouds can accept either "read" or "write" requests from the clients, namely the patient and the doctor. Upon receiving the requests, cloud providers invoke corresponding cloud services by matching their IDs in the cloud cluster, and return responses to the clients. In addition, the cloud providers in a cloud cluster are also responsible for providing their data to the directory on demand in a case that a restoration process is initiated when a *read* or *write* request fails due to the failure of a cloud provider in the cloud cluster. In this model, the "*Cluster Providers*" place is shared with the directory, where the `PROV` tokens in the place represent the providers selected to constitute the cloud cluster. In addition, the "*Provider Pool*," "*Down Providers*," and "*Service Ready*" places are also shared places in the directory model. The "*Provider Pool*" acts as a holding place for available providers identified by the directory. The "*Down Providers*" place contains the providers that are down and deemed needing replacement. Finally, the "*Service Ready*" place acts as an input place to the cloud for simulation purposes only. In our model, we only consider a maximum of one cloud provider going down at a time. This is a reasonable assumption because cloud providers should be somewhat reliable. In order to satisfy this constraint in the model, the "*Provider Down*" place is connected to the "*Provider_Down*" transition, which allows the "*Provider_Down*" transition to fire once.

When a client makes a "read" request, a `RDREQLIST` token with a list of `RDREQ` requests will be deposited into place "*Read_Req*." This enables the "*Read_Start*" transition as long as the "*RW_Control*" place contains a unit token, which ensures "read" and "write" actions are mutual exclusive. When the "*Read_Start*" transition fires, it splits the `RDREQLIST` token into singular `RDREQ` tokens, places them into place "*Read_Start*," and removes the unit token from the "*RW_Control*" place. The "*Read_File*" transition then examines each of the `RDREQ` tokens, matches it with its respective cloud provider, and fires as long as the `success` flag of the corresponding `PROV` token in place "*Cluster Provider*" is `true`. The following ML transition guard code accomplishes this task:

```
[(#recID r) = (#recID (#mrec g)) andalso
 (#prID r) = (#prID g) andalso (#ready g)=true]
```

where `g` represents a cloud provider that is a member of the cloud cluster and `r` represents a "read" request. The guard selects the correct provider by comparing the provider ID in the request `(#prID r)` with that of a provider from the cluster `(#prID g)`, matches the medical record ID, and makes sure that the cloud provider is functioning (i.e., its `ready` flag is set to `true`). If all conditions are met, the transition can fire, and

the firing of the transition creates a `RDRESP` token and deposits it into the "*Read_Resp*" place. On the other hand, if a "read" request fails due to the corresponding provider being not ready (i.e., its `ready` flag is set to `false`), the "*Read_Fail*" transition can fire, and its firing sends a `RDRESP` token with a blank medical record and a `success` flag set to `false` to the "*Read_Resp*" place. Once all three tokens are in the "*Read_Resp*" place, the "*Read_Resp*" transition may fire. The firing of the transition returns a unit token to the "*RW_Control*" place and places the `RDRESP` tokens into the "*Read_Ack*" port, available for the clients to digest.

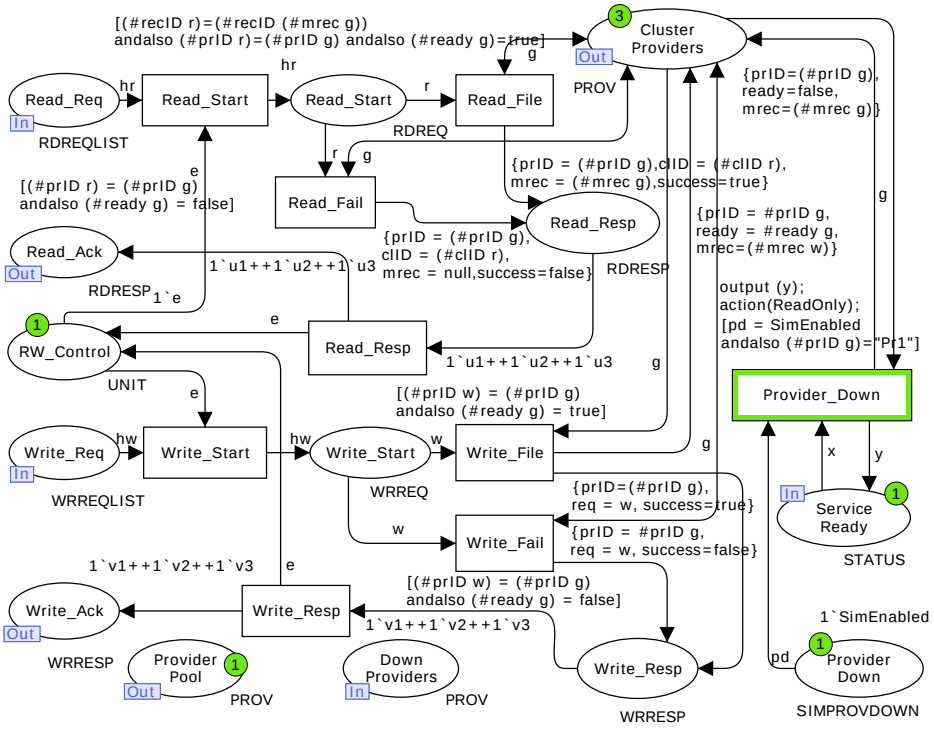


Fig. 7. CPN model for the cloud component

A "write" request follows an almost identical path through the model. When the doctor client places a `WRREQLIST` token into the "*Write_Req*" port, the "*Write_Start*" transition becomes enabled, and the firing of the transition places the individual `WRREQ` tokens into place "*Write_Start*." With the tokens in this place, the "*Write_File*" transition can fire as long as the `ready` flag of some `PROV` token in place "*Cluster Providers*" is `true`. The firing of the transition replaces the medical record stored in the `PROV` token with the replacement record, and also constructs a `WRRESP` token and places it in the "*Write_Resp*" place. On the other hand, if the `ready` flag of a provider is set to `false`, the "*Write_Fail*" transition may fire. In this case, the medical record is not altered, and a `WRRESP` token with the `success` flag set to `false` will be deposited in place "*Write_Resp*." Once all three `WRRESP` tokens are in the "*Write_Resp*" place, the

"*Write_Resp*" transition can fire, and its firing returns a unit token back to the "*RW_Control*" place and deposits the `WRRESP` tokens in the "*Write_Ack*" place, being available for the client to process.

A restoration process can be simulated in the cloud model by setting the `SIMPROVDOWN` token in place "*Provider Down*" to `SimEnabled`. When the "*Provider_Down*" transition fires, it selects the provider `Pr1` from the place "*Cluster Providers*" and sets the `ready` flag of the provider to `false`. This step simulates the failure of a cloud provider in the cloud cluster. Furthermore, the firing of the transition also sets the `STATUS` token in place "*Service Ready*" to `ReadOnly`, which disables the "*Next_Write*" transition in the CPN model for the doctor patient. The doctor patient will be allowed to write again only after the `STATUS` token in place "*Service Ready*" is changed back to `ReadWrite`. Meanwhile, when a client experiences an access error to a cloud provider that is down, a restoration process will be initiated by the client. Communication with the directory for a "restore" operation is done through the shared port "*Cluster Providers*." This port, containing the `PROV` tokens of the providers who make up the cluster, allows the directory direct access to the `PROV` state when required. When the restoration process completes, the failed cloud provider in place "*Cluster Providers*" will be replaced by a new one taken from the "*Cluster Pool*."

## 5. Formal Analysis of the CPN-Based Model

In addition to providing an accurate model for our proposed security mechanisms for cloud information storage, building a formal design model also has the advantage of ensuring a correct design through state space analysis. Utilizing the CPN Tools [24], a formal analysis of the CPN model can be performed to verify if the model meets certain system requirements. Typically, the model we developed should be *live*, *bounded*, and *deadlock-free*. When we use the CPN Tools to calculate the state space and analyze its major behavioral properties, the CPN Tools produce the following results:

```
Statistics                    Liveness Properties
------------------------      --------------------------
  State Space                   Dead Markings
     Nodes:  53226                 33 [51458,51457,51316,
     Arcs:   195308                51315,39025,...]
     Secs:   201                Dead Transition Instances
     Status: Full                  None
  Scc Graph                     Live Transition Instances
     Nodes:  30300                 None
     Arcs:   144610
     Secs:   6
```

The analysis shows that the state space contains dead markings, thus the model we developed may contain a deadlock. By tracing the firing sequence for the deadlock states as we did in our previous work [19], we found a subtle design error. The error is due to the removal of the failed cloud provider from the place "*Cluster Provider*" in the CPN model for the *Directory* component, which occurs when the transition "*Check Providers*"

fires. However, some "read" request in place "*Read_Req*" of the CPN model for the *Cloud* component would require communicating with a removed cloud provider if the "read" request was created before the cloud provider fails. Since there is no matched cloud provider in the "*Cluster Provider*" place of the *Directory* model, the system may enter a deadlock state. An easy way to fix this problem is to keep the failed cloud provider in the "*Cluster Provider*" place. This would allow the "*Read_Fail*" transition to fire, and return a "read" error to the client. After we add a new arc from the transition "*Check Providers*" to the place "*Cluster Provider*" in the *Directory* model, the CPN Tools now produce the following results:

```
Statistics                  Liveness Properties
------------------------    --------------------------
  State Space                 Dead Markings
     Nodes:  69679               None
     Arcs:   298179           Dead Transition Instances
     Secs:   529                 None
     Status: Full             Live Transition Instances
  Scc Graph                      Cloud'Read_File 1
     Nodes:  44449               Cloud'Read_Resp 1
     Arcs:   242856              Cloud'Read_Start 1
     Secs:   11                  ...


Boundedness Properties
-------------------------------------------------------
  Place                          Upper      Lower
  High_Level'Cluster_Providers     4          3
  High_Level'Down_Providers        1          0
  High_Level'Provider_Pool         1          0
  High_Level'Read_Ack              6          0
  High_Level'Read_Req              2          0
  High_Level'Restore               1          1
  High_Level'Service_Ready         1          1
  High_Level'Write_Ack             3          0
  High_Level'Write_Req             1          0
  ...
```

The analysis shows that our modified net model is *deadlock-free*, and all transitions except those related to the restoration process are *live*. Note that in our simulation, we allow the "*Provider_Down*" transition in the *Cloud* model to fire only once. The analysis results also show that our net model is *bounded*. We notice that the upper bound of the place "*Cluster_Providers*" in the high-level model is 4 rather than 3. This is because the failed provider can now be kept in the cloud cluster after the directory restores the cloud cluster by adding a replacement cloud provider.

In addition, as demonstrated in our previous work [25], model checking techniques can be adopted to verify behavioral properties of our colored Petri net model, such as security and fault tolerance related properties. CPN Tools facilitate analysis of state spaces by means of CTL-like temporal logic, called ASK-CTL, for formulating queries about states and state changes. For example, the formula $F1$ listed in Table 1 defines an

ASK-CTL temporal formula `canWrite = POS(NF("", Info_In_Cloud))`. In this formula, `POS`, as a state formula, is true if it is possible, from the current state, to reach a state where the argument `NF("", Info_In_Cloud)` is true. Note that `NF` is a node function, where `Info_In_Cloud` is defined as follows:

```
fun Info_In_Cloud n = ((Mark.Cloud'Cluster_Providers 1 n) ==
  1`{prID="Pr1",ready=true,mrec={recID="P1.rec",data="This is
  Pat1's medical record!"}}++
  1`{prID="Pr2",ready=true,mrec={recID="P1.rec",data="This is
  Pat1's medical record!"}}++
  1`{prID="Pr3",ready=true,mrec={recID="P1.rec",data="This is
  Pat1's medical record!"}});
```

Table 1. Model checking results for the revised Petri net model

| Formula | ASK-CTL Temporal Formula | Result |
|---|---|---|
| F1 | `val canWrite = POS(NF("", Info_In_Cloud));`<br>`val writeOK = OR(OR(NOT(NF("", Write_Info)),`<br>`NOT(NF("", Provider_Ready))), canWrite);`<br>`val myASKCTLformula = INV(writeOK);`<br>`eval_node myASKCTLformula InitNode` | True |
| F2 | `val canRead = POS(NF("", DataRead));`<br>`val readOK = OR(OR(NOT(NF("", Read_Info)),`<br>`NOT(NF("", Info_In_Cloud))), canRead);`<br>`val myASKCTLformula = INV(readOK);`<br>`eval_node myASKCTLformula InitNode` | True |
| F3 | `val canRestore = POS(NF("", Pr1_Replaced));`<br>`val restoreOK = OR(NOT(NF("", Pr1_Failed)),`<br>`canRestore);`<br>`val myASKCTLformula = INV(restoreOK);`<br>`eval_node myASKCTLformula InitNode` | True |

It is easy to see that the function `Info_In_Cloud` returns true if and only if providers `Pr1`, `Pr2`, and `Pr3` are all ready and contain the stored medical record of patient `Pat1`. We further define two functions `Write_Info` and `Provider_Ready` as follows:

```
fun Write_Info n = ((Mark.Doctor'Write_Information 1 n) ==
  1`{mrec={recID="P1.rec",data="This is Pat1's medical
  record!"},clID="Doc1",prov=["Pr1","Pr2","Pr3"]});
```

```
fun Provider_Ready n = ((Mark.Cloud'Cluster_Providers 1 n) ==
  1`{prID="Pr1",ready=true,mrec={recID="P1.rec",data=""}}++
  1`{prID="Pr2",ready=true,mrec={recID="P1.rec",data=""}}++
  1`{prID="Pr3",ready=true,mrec={recID="P1.rec",data=""}});
```

The function `Write_Info` returns true if the doctor has the patient record `P1.rec` to be stored with cloud providers `Pr1`, `Pr2`, and `Pr3`, and the function `Provider_Ready` returns true if all providers `Pr1`, `Pr2`, and `Pr3` are ready for storing patient record `P1.rec`. We are now interested in knowing whether at any time (state), the security model satisfies the following property for writing information into the cloud: `(Provider_Ready ∧ Write_Info → canWrite)`, which is equivalent to

((¬Provider_Ready ∨ ¬Write_Info) ∨ canWrite). This property is specified by the temporal formula writeOK = OR(OR(NOT(NF("", Write_Info)), NOT(NF("", Provider_Ready))), canWrite). Since this property must be satisfied at any time, the formula INV(writeOK) must be true for our Petri net models, where INV(writeOK) is true if from the current state, the argument writeOK is true for all reachable states. The model checking result shows that the Petri net model satisfies this property, which means information can be successfully stored with the cloud providers in the cloud cluster.

Similarly, as shown in formula *F*2 in Table 1, we can specify the security requirement in terms of reading information from the cloud as (Read_Info ∧ Info_In_Cloud → canRead), where canRead is defined as an ASK-CTL formula POS(NF("", DataRead)), and the functions Read_Info and DataRead are defined as follows:

```
fun Read_Info n = ((Mark.Patient'Read_Information 1 n) ==
  1`{recID="P1.rec",clID="Pat1",prov=["Pr1","Pr2","Pr3"]});
```

```
fun DataRead n = ((Mark.Patient'Data 1 n) ==
  1`{recID="P1.rec",data="This is Pat1's medical record!"});
```

From Table 1, we can see that the model checking result shows that the Petri net model satisfies this property; thus, information stored with the cloud cluster can be successfully retrieved by a patient client.

Finally, as shown in formula *F*3 in Table 1, we specify a property related to fault-tolerance requirements as (Pr1_Failed → canRestore), where canRestore is defined as an ASK-CTL formula POS(NF("", Pr1_Replaced)), and the functions Pr1_Failed and Pr1_Replaced are defined as follows:

```
fun Pr1_Failed n = ((Mark.Cloud'Cluster_Providers 1 n) ==
  1`{prID="Pr1",ready=false,mrec={recID="P1.rec",data="This is
  Pat1's medical record!"}}++
  1`{prID="Pr2",ready=true,mrec={recID="P1.rec",data="This is
  Pat1's medical record!"}}++
  1`{prID="Pr3",ready=true,mrec={recID="P1.rec",data="This is
  Pat1's medical record!"}});
```

```
fun Pr1_Replaced n = ((Mark.Cloud'Cluster_Providers 1 n) ==
  1`{prID="Pr1",ready=false,mrec={recID="P1.rec",data="This is
  Pat1's medical record!"}}++
  1`{prID="Pr2",ready=true,mrec={recID="P1.rec",data="This is
  Pat1's medical record!"}}++
  1`{prID="Pr3",ready=true,mrec={recID="P1.rec",data="This is
  Pat1's medical record!"}}++
  1`{prID="Pr4",ready=true,mrec={recID="P1.rec",data="This is
  Pat1's medical record!"}});
```

From Table 1, we can see that the model checking result shows that the Petri net model satisfies this property; thus, when the cloud provider Pr1 fails, the cloud cluster as well as the stored information can be successfully restored by replacing Pr1 with Pr4.

## 6. Conclusions and Future Work

Cloud computing is quickly becoming a widely adopted Internet-based platform that allows for complex computational nodes and storage clusters, but with the tremendous difficulties and high costs associated with configuration and maintenance being hidden from the users. There are, however, major legitimate concerns from enterprises and sensitive data holders related to offsite storage of personal or mission-critical data. Studies show that given the current state of cloud computing, enterprises are very concerned with unresolved issues related to security, trust, and management in the cloud. For a majority of these enterprises, this is also the main reason why they have not yet adopted cloud computing into their infrastructure. In this paper, we introduced a secure and fault-tolerant cloud information storage model that takes into account the fact that cloud providers may experience outages, data breaches, and exploitations. We cope with these issues by developing a distributed cloud-based security mechanism following the RAID 5 concept. We then utilize hierarchical colored Petri nets to formally model and analyze our concurrent security model. The verification results show that the model we developed is *live*, *bounded* and *deadlock-free*, and satisfies major security and fault-tolerance related requirements.

For future work, we will consider adopting more advanced RAID techniques such as RAID 6 [26], to increase the fault-tolerant capability of our cloud information storage model. Such improvement would allow a cloud-based storage system to recover from up to two simultaneous cloud provider failures. We also plan to implement a prototype cloud information storage system with an improved distributed parity algorithm that may strengthen the security mechanism by preventing potential provider collusion to obtain information stored in a cloud cluster. In addition, we will develop a more sophisticated formal security model that allows more clients to access cloud clusters with shared cloud providers, and demonstrate how to cope with the state explosion problem using the net reduction approach [27]. Finally, we believe that we can further improve the cloud information storage model by allowing the service directory to autonomously detect failures, drops in QoS, and anomalies of the cloud providers, and react accordingly.

## References

1. GAO, Information Security: Additional Guidance Needed to Address Cloud Computing Concerns, *United States Government Accountability Office (GAO)*, October 6, 2011, retrieved on December 18, 2011 from http://www.gao.gov/new.items/d12130t.pdf
2. J. Brodkin, Loss of Customer Data Spurs Closure of Online Storage Service "The Linkup," *Network World*, August 11, 2008, retrieved on September 18, 2010 from http://www.network world.com/news/2008/ 081108-linkup-failure.html?hpg1=bn.
3. A. Cruz, Update on Today's Gmail Outage, *Google*, February 24, 2009, retrieved on September 20, 2010 from http://gmailblog. blogspot.com/2009/02/update-on-todays-gmail-outage.html
4. J. Mintz, Microsoft Dumps Windows Live Spaces for WordPress.com, *Huffington Post*, September 27, 2010, retrieved on April 25, 2011 from http://www.huffingtonpost.com/ 2010/09/27/microsoft-dumps-windows-l_n_741023.html.

5. SSC, PCI SSC Data Security Standards Overview, *Security Standards Council (SSC)*, October 1, 2008, retrieved on December 28, 2011 from https://www.pcisecuritystandards.org/security_standards/index.php
6. HHS, Understanding Health Information Privacy, *U.S. Department of Health & Human Services*, 2011, retrieved on December 28, 2011 from http://www.hhs.gov/ocr/privacy/hipaa/understanding/index.html
7. AWS, Creating HIPAA-Compliant Medical Data Applications with AWS, *Amazon Web Services (AWS)*, Amazon, April 2009, retrieved on August 22, 2010, from http://awsmedia.s3.amazonaws.com/AWS_ HIPAA _Whitepaper_Final.pdf.
8. D. Fitch and H. Xu, A Petri net model for secure and fault-tolerant cloud-based information storage, *Proc. of the 24th International Conference on Software Engineering and Knowledge Engineering (SEKE 2012)*, San Francisco Bay, CA, USA, July 1-3, 2012, pp. 333-339.
9. M. T. Goodrich, R. Tamassia, and D. Yao, Notarized federated ID management and authentication, *Journal of Computer Security* **16**(4) (2008) 399-418.
10. A. C. Weaver, Enforcing distributed data security via web services, *Proc. of the IEEE International Workshop on Factory Communication Systems (WFCS)*, Vienna, Austria, September 22-24, 2004, pp. 397-402.
11. N. Santos, K. Gummadi, and R. Rodrigues, Towards trusted cloud computing, *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud09),* San Diego, CA, June 2009.
12. K. Hwang and D. Li, Trusted cloud computing with secure resources and data coloring, *IEEE Internet Computing*, **14**(5) (2010) 14-22.
13. Y. Liu, Y. Ma, H. Zhang, D. Li, and G. Chen, A method for trust management in cloud computing: data coloring by cloud watermarking, *International Journal of Automation and Computing*, **8**(3) (2011) 280-285.
14. J. Deng, S. Huang, Y. Han, and J. Deng, Fault-tolerant and reliable computation in cloud computing, *Proc. of the GLOBECOM Workshops (GC Wkshps)*, Miami, FL, December 2010, pp. 1601-1605.
15. W. Zhao, P. M. Melliar-Smith, and L. E. Moser, Fault tolerance middleware for cloud computing, *Proc. of IEEE 3rd International Conference on Cloud Computing*, Miami, FL, July 5, 2010, pp. 67-74.
16. C. Cachin, I. Keidar, and A.Sharaer, Trusting the cloud, *ACM SIGACT News,* **40**(2) (2009) 81-86.
17. D. A. Patterson, G. Gibson, and R. H. Katz, A case for redundant arrays of inexpensive disks (RAID), *ACM SIGMOD Newsletter*, **17**(3) (1988) 109-116.
18. J. Gu and J. Luo, Modeling autonomic QoS control for grid service using Petri nets, *Proc. of Eighth International Conference on Grid and Cooperative Computing (GCC),* Lanzhou, China, Aug. 27-29, 2009, pp. 3-8.
19. H. Xu, M. Ayachit, and A. Reddyreddy, Formal modeling and analysis of XML firewall for service-oriented systems, *International Journal of Security and Networks (IJSN)*, **3**(3) (2008) 147-160.
20. R. Accorsi and L. Lowis, ComCert: automated certification of cloud-based business processes, *ERCIM News*, **83** (2010) 50-51.
21. A. Thomasian and J. Menon, RAID 5 performance with distributed sparing, *IEEE Trans. on Parallel and Distributed Systems*, **8**(6) (1997) 640-657.
22. D. A. Patterson, P. Chen, G. Gibson, and R. H. Katz, Introduction to redundant arrays of inexpensive disks (RAID), *Proc. of the COMPCPN Spring'89, Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers*, February 27 - March 3, 1989, San Francisco, CA , USA, pp. 112-117.

23. K. Jensen, *Colored Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. I: Basic Concepts*, EATCS Monographs on Theoretical Computer Science (New York Springer-Verlag, 1992).
24. A. V. Ratzer, L. Wells, H. M. Lasen, M. Laursen, J. F. Qvortrup, *et al.*, CPN Tools for editing, simulating and analyzing colored Petri nets, *Proc. of the 24th International Conference on Application and Theory of Petri Nets*, Eindhoven, Netherlands, June 2003, pp. 450-462.
25. H. Xu, L. Xing, and R. Robidoux, DRBD: dynamic reliability block diagrams for system reliability modeling, *International Journal of Computers and Applications (IJCA)*, **31**(2) (2009) 132-141.
26. M. Gilroy, J. Irvine, and R. C. Atkinson, RAID 6 hardware acceleration, *ACM Transactions in Embedded Computing Systems*, **10**(4) (2011) 1-17.
27. S. M. Shatz, S. Tu, T. Murata, and S. Duri, An application of Petri net reduction for Ada tasking deadlock analysis, *IEEE Trans. on Parallel and Distributed Systems*, **7**(12) (1996) 1307-1322.