On the design and analysis of protocols for Personal Health Record storage on Personal Data Server devices

Kirill Belyaev, Wuliang Sun, Indrakshi Ray*, Indrajit Ray

Department of Computer Science, Colorado State University, Fort Collins, USA

HIGHLIGHTS

• The architecture of Personal Data Server overlay is presented.
• The data is replicated in user-controlled secure portable tokens.
• Properties of the protocol are demonstrated using UML and Alloy.

ABSTRACT

The electronic Personal Health Records (PHRs) such as medical history, lab reports, and insurance are stored in systems such as Microsoft Health Vault where a medical care provider or a patient is responsible for uploading and managing the health information. Storing PHRs in such a manner prohibits the patients from having complete control over their data and also may make the PHR system the target of security attacks. Towards this end, we propose a new architecture, namely Personal Data Server overlay, where the data is stored on a set of Secure Portable Tokens (SPTs) that are under the control of individual users. SPTs are cheap, portable, and secure devices that combine the computing power and tamper-resistant properties of the smart cards and the storage capacity of NAND flash memory chips and they can act as a Personal Data Server (PDS).

We need formal assurance of data availability when information is stored in PDS overlays. Thus, data must be replicated at multiple PDSs. We propose a data replication protocol that ensures that the PHRs for each user have replicas in the PDS overlay. It is crucial to ensure correctness of the data replication protocol. Consequently, we formalize the protocol using the Unified Modeling Language (UML) and specify a number of desirable properties. We need to provide formal assurance of these properties in an automated manner. We demonstrate how the UML model can be transformed into Alloy using the UML-to-Alloy transformations. This obviates the need for the protocol designer to know Alloy. The analysis uncovers a significant error in the protocol. Uncovering such errors help refine the protocol and ensures its correctness before deployment.

1. Introduction

In the digital age, there is a plethora of personal information that individuals need to store and access. Consider the storage of Personal Health Records (PHR) as a motivating example. A PHR system can store diverse information related to public health, such as, medical history and lab reports, insurance, consent forms, and other relevant information. The use of PHRs can improve the health care of individuals and communities in both developed and developing countries [1] and there are over 200 different PHR systems in the USA [2].

We have various types of PHR systems, some of which are cloud based such as Microsoft HealthVault [3], where the user is responsible for uploading and managing her health information. Storing sensitive personal data, such as health data, on current cloud systems, such as Microsoft HealthVault [3], improves upon the systems supported by providers in that the users are responsible for managing their own data. However, such a solution, has several problems, including the potential for security and privacy breaches, that we outline below. First, service downtime and unavailability of Internet connection may limit an individual from accessing her data stored on the cloud. Clearly, this may
be unacceptable to many users who expect their data to be available all the time. Second, such servers storing PHRs of a large number of users are also target for attackers who have much to gain by compromising the privacy of health data. Note that, attackers can be insiders as well, such as dishonest or disgruntled employees of the organization. Third, the individuals must trust these organizations for adequate protection of the data against security and privacy breaches. Typically, users are reluctant to place trust on these organizations [4]. Fourth, even though an individual trusts an organization and is willing to comply with its security and privacy policies, such policies are subject to change with time. Moreover, security and privacy policies will almost certainly be changed if there is an acquisition or merger of companies. It is not clear how such a change will impact the existing records of an individual user and whether such a change will respect the original preferences of the user. Fifth, the servers storing data of a very large number of users must be kept online, which requires enormous power consumption. Consequently, we propose an alternative approach for storing PHR data.

Our approach builds upon the use of a new emerging technology of cheap ($10–20 range), portable (carry it in your pocket/purse) and secure devices that combine the computing power and strong, tamper-resistant security of smart cards and the ever increasing storage capacity of NAND flash memory chips. We call such devices Secure Portable Tokens (or SPTs in short). Examples of such SPTs can be found in wireless secure dongles, and smart USB jump drives with embedded chips. A very similar technology can be found in the SIM cards of mobile telephones and smart phones.

Individuals own SPTs and store their personal data on the SPT, forming what we refer to as a Personal Data Server (PDS). The PDS alternative to existing PHR systems exploits the computing power of 32 bit RISC processors and strong, tamper-resistant security of smart cards.

1.1. Our contributions

Individuals have their PHRs on their own PDSs. For reasons of availability, the individuals may wish to replicate their PHRs on devices belonging to other individuals. The users have total control over what data they wish to replicate, where they would like to replicate, and may also change these decisions at any point of time. Thus, a user may replicate his data to one or more PDSs belonging to other trustworthy individuals. Each PDS, under the control of an individual user, may be online or offline depending on the user's discretion. To facilitate communication between two PDSs, we employ the services of an online broker. The broker simply provides a temporary storage service to which a PDS can upload data (publish) and from which another PDS can download data (subscribe). Replica propagation, in light of the fact that PDSs may be offline, poses a challenge. We have devised replica control protocols to address this issue. Eventual consistency property is assured provided the PDS holding the replica comes online after the update operation.

Queries may be posed on the PHR by individuals or organizations. Query processing poses similar challenges because a PDS containing the required PHR may be offline. In such a case, we demonstrate how the query response can be constructed from the PDS holding the original PHR or from the other PDSs containing the replica PHR. We try to minimize the response time for query processing, by trying to answer the query using the original or replica PDSs, whichever comes online first. We also guarantee the data freshness property that ensures that the response to the query is always accurate, even if some replicas may have not been online all the time.

We need to provide assurance that the system's behavior is as intended. Towards this end, we focus on the replication protocol because it forms the basis of data availability and reliability. We formalize the protocol using the Unified Modeling Language (UML) [5] notation and the Object Constraint Language (OCL) [6]. Our experience suggests that protocols that constrain how system elements are replicated can be conveniently expressed in class models augmented with OCL invariants and operation specifications. Structural aspects of these protocols can be expressed in terms of classes and relationships among classes, while functionality can be represented by operations associated with OCL pre- and post-conditions. We chose UML and OCL as they are the de-facto specification language used in the software industry and it is easy to use and understand. However, limited tool set is available for automated analysis of UML models.

We further demonstrate how Alloy [7,8] can be used for the automated verification of the replication protocol. Alloy is a modeling language capable of expressing complex structural constraints and behavior. It has been used to specify network protocols [9–12] and security polices [13–15]. It has very good tool support in the form of the Alloy Analyzer that translates an Alloy specification into a Boolean formula that is evaluated by embedded SAT-solvers. However, few practitioners have the background needed to develop good Alloy models. We provide an approach by which the UML model can be converted into an Alloy model using UML-to-Alloy transformations described in [16]. This obviates the need for protocol verifiers to understand the Alloy specification language.

The approach described in the paper first derives a UML class model from the data replication protocol, and then uses the UML-to-Alloy transformations to generate the Alloy model. The correct behavior of the model is formalized through a set of constraints. The states satisfying these constraints are referred to as valid, and the other prohibited states are called invalid. The generated Alloy model allows a protocol verifier to check whether a system can move from a valid to an invalid state as result of a sequence of operation calls. If analysis uncovers such a sequence of operation calls then the verifier uses the trace information output by the analysis to help find the source of the errors in the UML class model. The analysis was able to uncover a significant error in the original data replication protocol.

We would like to emphasize at this point that the security and privacy issues related to PDS and/or the broker paradigm is not a contribution of this work. The hardware based security services provided by the SPT include protection against side channel attacks. We assume that the broker services are honest but curious. The security protocols needed by the PDS to interact with the broker have been discussed in our previous works [17,18].

The rest of the paper is organized as follows. In Section 2, we first present the architecture of the SPT and its security properties. We then follow it with a design of the PDS and how security and privacy of PHR data is ensured by a combination of hardware security of SPT and anonymous protocols built into a PDS. In Section 3, we discuss how the information in PDS can be replicated. In Section 4, we provide details on the replication protocol. In Section 5, we illustrate our query processing approach. In Section 6, we prove the feasibility of our approach by demonstrating the prototype and providing experimental results. In Section 7, we provide detailed analysis of the properties of the proposed suite of PDS protocols. In Section 8, we provide the formal specification approach, and in Section 9 we describe the results of the formal analysis. In Section 10 we discuss related work, and in Section 11 we conclude the paper and provide our plans for future work.
2. SPT architecture and its impact on PHR security

For the reader to appreciate the security features that PDS inherits by virtue of it being built on the SPT and to give an idea of privacy protection offered to PHR data by the PDS, we begin with a brief discussion on the architecture of an SPT, the security threat model for PDS and the corresponding PDS design. Fig. 1 gives a schematic overview of an SPT hosting the PDS.

At the heart of the SPT is the Secure Tamper Proof Processor the major components of which are a (typically) 32 bit RISC CPU, a cryptographic module that is implemented in hardware, a RAM and a NOR Flash based read-only memory. All these modules are built on the same die. The SPT operating system and all related application software (the PDS software in our case) are programmed into the NOR memory during the manufacturing process. The basic software (operating system, database engine and PDS generic tools) can be certified according to the Common Criteria, making software attacks also highly difficult. This is the biggest advantage of using an SPT for building the PDS: once the PDS software is programmed into the SPT, it cannot be tampered with any more. No malware can infect the device. Thus, we can provide strong guarantees that the PDS is performing exactly those operations that it is supposed to perform (assuming of course that the PDS has been designed properly).

The NAND Flash memory can be accessed either directly or via the SPT processor. When it is accessed via the SPT processor, all data stored on the memory are in encrypted form with the encryption being performed by the cryptographic module within the SPT processor. Any key that is used for the encryption is stored within the SPT itself and is thus protected. If a piece of encrypted data is accessed directly, bypassing the SPT processor, the data will just be a random blob.

Since the SPT is tamper-proof, it cannot be physically probed to reveal keys. The only way then that the encrypted data can be decrypted is by launching a side-channel attack against the cryptographic module. Such an attack will be typically based on monitoring power consumption during execution of cryptographic functions or the computation time, and can potentially reveal the stored key. However, this attack is not easy to launch, since it requires an attacker to physically access the SPT. Nonetheless, such a side channel attack can be thwarted by using suitable key masking techniques [19,20]. The AES algorithm, which we use, coverts a secret key into several intermediate round keys during the encryption process. In our case, we store our secret key in the form of such intermediate keys and use random variables called masks to shield these intermediate round keys using simple XOR operations.

The SPT lacks a power source, an internal clock and any I/O device. For each of these functionalities it needs to rely on a host device. When the SPT is connected to a host I/O system, the host can potentially act as a conduit of attack vectors to the SPT. To protect against these attacks, we first note that any piece of data that the SPT externalizes is either encrypted by a key that is protected inside the SPT, or is provided in a read-only manner by software within the SPT. In our case, only the PHR data owner can get read-only versions of her data from her SPT. Any data that the SPT receives from the outside world need to be certified as non malicious and need to be encrypted with the key of the receiving SPT.

Even if any data that is externalized by an SPT is encrypted, the identity of the users downloading and uploading this data must be obfuscated. Indeed, linking attacks based on spying the identity of users in communications (even network addresses) could lead to disclosure of sensitive information (e.g., the data sent by an SPT to a cancer hospital may reveal nature of disease). We briefly discuss the privacy preserving communication protocol employed in Section 2.1.1 since this is not a property of the SPT but is provided by the PDS software. We would like to emphasize that addressing the security and privacy challenges of the PDS is not the major contribution of this work, but has been addressed separately in other works [17,21].

2.1. PDS overview

Our core idea is to embed a full-fledged secure PHR data server and related software components within the tamper proof hardware, and store the PHR data in encrypted manner in the accompanying NAND flash storage. The application software as well as the user’s PHR data will be protected by strong hardware-based security. No malware can damage the software since it is built into the hardware. Moreover, the whole system is tailor made for PHR information, and hence will require very little administration from the user.

The central building block of the PDS, namely the SPT, exhibits very strong hardware constraints, namely, (i) the availability of very limited RAM in the SPT, (ii) the lack of an on-chip clock for the CPU to use, and (iii) the lack of communication facility with the outside world. Traditional database techniques related to storage, indexing, query and transaction processing need to be fully revisited to build an engine with acceptable performance. The interested reader is directed to our earlier work in designing such an embedded database engine for more details related to addressing the challenges arising out of constraints (i) and (ii) [17].

Fig. 2 provides a schematic overview of the major components of the PDS engine. The PDS subsystem comprises of the components enclosed within the red and the green boundaries in the figure. The components enclosed by the red boundary provide the functionality discussed in this paper. The data generated/used by the PDS is stored in the encrypted NAND Flash and is logically shown as part of the PDS. The PDS engine is built into the ROM of the SPT. Fig. 2 also shows the rest of the SPT to indicate that the PDS relies on some of the functionalities provided by the SPT operating system.

For the user to access the PHR data, or for the PDS to connect to different service providers to update the PHR data, the SPT would need to be attached to a host device such as a smart phone or a computer. In such cases, the host device provides only an interface to the external world. For example, the host can run a proxy to provide the interface to the Internet and the PDS software will have a tiny client (running as part of the PDS interface in Fig. 1).
that will interact with the proxy on the host. All computations, communication, security policy enforcement, data storage and retrieval are still carried out by the application embedded in the SPT hardware. The host proxy will only get an encrypted blob of data to be transmitted to the proper destination. Security policies can be set or overridden only by the user. However, the policies are still enforced by the hardware.

A group of such PDSs form a personal data server eco-system. In order to build such an eco-system, various research challenges must be addressed. However, we scope this paper and provide a simple baseline architecture and implementation for the system. For everyday use, a PDS, storing a single user’s PHRs, need not be connected continuously to the Internet. An individual can get all his PHR information from his own PDS even though he is not connected to the Internet by running a host client such as a web browser. However, PDSs may need to communicate with the external world to receive or send medical information to other entities, such as, healthcare providers or sharing their information with trusted friends. We do not assume that when a PDS seeks to send (or receive) data from the external world the corresponding end point of communication would be available and connected to the network. As a result, such communication is performed in an asynchronous manner facilitated through an online broker that serves as a temporary storage for data exchange. The broker is implemented using the publish–subscribe model. We require the broker to be online all the time and serve as a temporary storage for data exchange. The broker does not store the PHRs of the individuals; rather it stores only encrypted blobs of data that are externalized by a PDS. Thus, the broker storage is significantly smaller than that required to store the PHR data of all the users. The owner has complete control over what data he pushes to the broker for sharing with external entities and he also decides when such data is deleted from the broker.

We discuss the broker architecture in details and also the protocols that are needed to communicate with the external entities as relevant to the operation of the PDS.

2.1.1. PHR privacy considerations

Earlier, in Section 2 we indicated that privacy of sensitive information can be compromised by launching linking attacks on the identity of users in communication messages to and from a PDS. For example, if an encrypted blob of data is externalized to the broker by a PDS and is later downloaded by a cancer hospital, it can be reasonably inferred that the data is related to a cancer patient.

To protect against such attacks, we employ an anonymizing protocol such as TOR [22,23] to support all communication between a PDS and the broker. We assume that if a PDS is intentionally sharing its data with another entity, there is no need to protect the privacy of the PDS owner from that entity. Thus there is a need to utilize a proper authentication protocol. To balance privacy and accountability, researchers have proposed many protocols such as k-times anonymous authentication [24], anonymous authentication and credential systems [25–27], group signature schemes [28], and anonymous e-cash [29] and fair-exchange protocols [30]. We use the k-times anonymous authentication protocol for authentication as and when needed.

We also assume the existence of 4 different protocols that execute over the anonymizing network that allow a PDS to undertake anonymous publish–subscribe with the broker, which we assume to be honest but curious. These are:

1. **Anonymous upload–tag generation**—The “upload–tag” is a cryptographically generated piece of data that is created collaboratively between two entities that want to exchange information anonymously via the publish–subscribe services of the broker. An upload–tag is attached to every piece of encrypted data that is being externalized by a PDS whenever it is published (or uploaded, hence the name) to the broker over the anonymizing network. The property of the upload–tag is such that the broker or any other attacker will not be able to trace the data to its publisher using information in the upload–tag; however, the intended recipient of the data (the subscriber, that is) would recognize the upload–tag as being one that it had created (in collaboration with the publisher). As a result, utilizing the upload–tag the receiver would be able to access the encrypted BLOB of data.

2. **Anonymous upload**—This protocol involves a publisher attaching an upload–tag (which is unique to a publisher–subscriber pair) and a per data unique “delete-tag” to an encrypted piece of data and then transmitting it to the broker over the anonymizing network. The delete-tag prevents malicious delete of data by an attacker. A publisher of data (or its representative) can anonymously request the broker to remove any of the data that the publisher had uploaded to the broker earlier. Since all such requests arrive over anonymous channels and we do not want the broker to be able to trace a request back to its originator or the data to be deleted to its publisher, there is a possibility that an entity who has not published the data to be deleted (or that entity’s authorized representative) sends a deletion request to the broker. To ensure that data is not maliciously deleted by
such a request, we utilize the delete-tag which is also cryptographically generated by the publisher. However, the publisher generates this tag unilaterally. Necessary checksums are appended to the transmitted data to ensure integrity. The broker utilizes the upload-tag to retrieve the corresponding BLOB of data.  

3. Anonymous retrieve—This protocol involves a subscriber utilizing an upload-tag that it had created earlier in collaboration with a publisher to request an encrypted BLOB of data from the broker over the anonymizing network. The broker utilizes the upload-tag to retrieve the corresponding BLOB of data.  

4. Anonymous delete—To anonymously request the broker to delete its data, the original publisher provides a pre-hash of the delete-tag corresponding to the data. The broker computes the hash of this information. If it matches the delete-tag corresponding to the data then the broker deletes the data.

We had reported on these protocols earlier in a different work. We omit the details here since it is not among our contributions for this work. The interested reader is referred to [18] for further details. For the rest of the paper, we assume that all protocols are carried over the anonymizing network and utilizes these four protocols as and when needed.

3. PDS replication architecture

In order to make personal information reliable and available, we propose to replicate the information over various SPTs, referred to as PDS nodes, as shown in Fig. 3. Each PDS node is identified through a unique identifier, which we refer to as Pdslid, and is owned by one user. Each PDS can take on at most two roles: owner PDS and replica PDS. A PDS acts as the owner PDS when it is dealing with the PHR record of the person who owns the PDS. A PDS acts in the role of replica PDS when it is involved with the PHR record of a person who is not the owner of that PDS. Note that, all PDSs possess the role of the owner PDS. However, only a select set of PDSs, may be assigned the role of replica PDS. We assume that a replica PDS contains the PHR of at most one user who is not the owner. However, an owner PDS may choose to replicate its data to multiple replica PDSs.

The PDSs communicate with each other via the broker using the publish–subscribe mechanism. The PDSs are responsible for pushing/pulling the information stored on the broker. Although the broker can have various types of architectures, we assume a centralized architecture in this paper. Thus, we have a network of PDS nodes that connect to a broker. The broker stores various types of information that are needed to communicate with the PDSs. The architecture of the broker is shown in Fig. 3. Following is a description of the various tables stored at the broker site.

[Mappings Table:] The schema of this table is given by mappings(Pdslid, Rpdslid). The mappings table stores information about owner to replica mappings, defined by the pair (Pdslid, Rpdslid), where Pdslid is the identifier of the owner PDS and Rpdslid denotes the identifier of the replica PDS.

[Updates Table:] The updates table stores updates made by the owner PDS that must be propagated to its replicas. The schema of this table is given by updates(Pdslid, DataPacket). The owner PDS is identified by Pdslid and the update that must be propagated is the DataPacket, which is represented as a BLOB holding Java class object that encapsulates the data and methods of the corresponding type. Each DataPacket has a sequence number associated with it that identifies the unique PHR received from the owner PDS.

[Queries Table:] The queries table stores recent queries on the PHR data that must be answered. The schema of this table is given by queries(Pdslid, QPdslid, QId, QueryPacket) where Pdslid is the PDS issuing the query, QPdslid is the PDS which is being queried, QId is the unique query identifier, and QueryPacket is the BLOB (Java class object) representing the actual query. In order to speed up query processing, this table is indexed by QId and Pdslid.

[Results Table:] The result table stores the query responses that must be downloaded by the query issuer. The schema of this table is given by results(Pdslid, QId, ResultPacket) where Pdslid is the query issuer, QId is the query identifier, and ResultPacket is the BLOB representing query results. In order to speed up query processing, this table is indexed by QId and Pdslid.

In addition, the broker also stores two main memory hash maps of the following information.

[Replica Updates:] This is defined by the pair (RPsdlid, seqNo) and it records the sequence number associated with the latest update, denoted by seqNo, that is received by the replica PDS with identifier RPsdlid.
4. Replication protocol

A data replication protocol has been proposed in [21] to ensure that the PHR stored in a PDS can be always available and reliable. In this protocol, the owner PDS chooses the set of PDSs which are responsible for hosting its replicas. It does this by sending a replica set message to the broker. The broker updates the mapping table that records the owner to replica mappings. The owner can change this set at any point of time.

The new inserts into the PHR data are propagated through the broker which acts as a temporary storage. The owner PDS sends a data update message to the broker. The broker on receiving this message stores it in the updates table until it receives instructions from the owner PDS to delete it. In addition to the new PHR record, the data update message also contains a monotonically increasing sequence number, seqNo. The broker changes the memory hashmap owner updates to record the latest sequence number, seqNo, received from the owner PDS, PdsId.

The broker hosts the new PHR records until it receives the delete request message from the owner PDS. The owner PDS executes a Replication Watchdog that keeps track of replica freshness by sending periodic replica seqNo request to the broker in order to get the latest sequence number of the PHRs received by its replicas. It compares these sequence numbers with that of the last PHR it has sent; if a quorum of replicas has received the latest PHRs, then it sends a delete request message to the broker. The broker on receiving such a message from the owner PDS, deletes the PHRs belonging to the owner PDS from the updates table.

Each replica PDS executes a Replication Manager that is responsible for pulling the updates and keeping the replica that it hosts up-to-date. It does this by sending an updates request message to the broker. The broker retrieves the owner PDS corresponding to the replica and sends the corresponding messages stored in the updates table. If the sequence number, seqNo, associated with the response message is less than or equal to those updates it has already received, then the replica PDS does not insert this new data in its updates table. Note that, the broker sends new updates to replica only if there is a disparity in seqNo. The broker sends updates to the replicas only if the seqNo attribute in the owner PDS hash map table is different from the seqNo of the PDS replica stored in the replica hash map store. In such cases, the broker sends the records that are more recent than that given by the replica seqNo attribute. The replication protocol is depicted in Fig. 4.

Note that the broker does not take any active role in maintaining replica consistency, but merely acts as a temporary storage. Clearly, the duration of storage depends on how often the owner PDS and replica PDS connect to the broker.

5. Query processing

We have two types of queries: global queries and individual queries. Global queries involve PHRs of a group of people. Consequently, the response of the query resides in several PDSs. Individual queries involve the PHR of one user, so the answer can be obtained from the owner PDS or any of its replicas. In this paper, we focus on individual queries only. We have an embedded relational database engine residing on the PDS that can process queries. The schema of our PHR table is as follows:

```
(hospitalName, patientName, medicine, date, doctorName, diagnosis, prescription)
```

Examples of typical queries supported by our system are given below.

- **[Q1:]** SELECT COUNT(*) FROM ehr
- **[Q2:]** SELECT date FROM ehr WHERE diagnosis = "flu" AND prescription = "aspirin"

In the following, we use the term `query requester` (or simply `requester`) and `query responder` (or simply `responder`) to refer to the different roles involved in query processing. Note that, we do not focus on access control issues, but assume that the requester has permissions to query the PHR of some data owner. `Responder` may be the owner PDS or the replica PDS which has the PHR corresponding to the query.

We describe query processing in Fig. 5. `Requester` executes a `jobber` process that sends the query message to the broker (shown as Step 1 in Fig. 5). The query is temporarily stored in the `queries table`. The response to the query is stored by the broker in the `results table`. The `jobber` process in the `requester` sends query result request message (shown in Step 5 of Fig. 5) to get the response of the query. After receiving the responses from the broker (shown in Step 6 of Fig. 5), the `jobber` sends a delete query message to the broker to delete the query identified by `QId` and `PdsId` (not shown in Fig. 5). If the query result is resent to the requester, it signals that the `delete query message` has been lost. In such a case, the `jobber` sends the `delete query message` once again.
**ARTICLE IN PRESS**

K. Belyaev et al. / Future Generation Computer Systems [ ](#future-generation-computer-systems-%5B%5D) [ ](#future-generation-computer-systems-%5B%5D) [ ](#future-generation-computer-systems-%5B%5D)

**Protocols for Personal Health Record storage on Personal Data Server devices, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.05.027**

---

Responder has a query manager that sends the query request message (shown in Step 2 of Fig. 5) periodically to the broker to check whether there are any outstanding queries on the PHRs residing in it. If there are outstanding queries, the broker forwards them to the responder (shown in Step 3 of Fig. 5). The responder constructs a query result message (shown in Step 4 of Fig. 5) and sends it back to the broker.

### 6. Prototype implementation and experimental results

The PDS model (both the broker and the PDS itself) has been implemented using Java Standard Edition (Java SE) using SQLite Java driver to work with the embedded database. We used MySQL for broker backend storage of the PDS storage tables. We deployed the PDS model prototype on the PlanetLab [31] testbed. Fig. 6 shows the location of the different PDS nodes and their Round Trip Time (RTT) delays in milliseconds from PDS location to the broker network and back to the PDS. The broker infrastructure has been set up in a geographical Internet location that is different from the location of the PDS nodes. The backend database server and Broker process run on separate machines within the same network so that JDBC queries would not be affected by the network losses or delays.

The PDS nodes have been deployed in United Kingdom, Poland, US (at Colorado State), Germany, Sweden, Portugal and Slovenia. No particular choice for location has been made except that the locations listed have been observed to possess relative stability of the machines uptime on PlanetLab global network. Most of the queries have been issued from US (Colorado State) location and tested with replicas or target PDSs answering the queries from the above listed geographical internet locations.

As we see from the plots in Figs. 7 and 8 different geographical locations incur different QoS values for the replication and query processing adding the RTT delays to the constant sleep periods of Replication Manager and Query Manager threads. Therefore replication and query processing delays have been defined as the sum of RTT from the source to target and back to carry the protocol messages plus the sum of sleep periods for Replication and Query Managers correspondingly at both source and target PDS nodes. Since both threads are expected to pull requests with a reasonable sleep period to spare the broker computational resources (and network bandwidth) we could define the service as having the eventual consistency property when data is replicated and queried at unpredictable time intervals but with guaranteed consistency properties provided that PDS nodes are able to go online.

Also the plots do not reflect the delays that would be introduced by broker serving requests from a large number of pulling nodes (more than a thousand) and sending JDBC queries to backend database with highly populated tables (introducing local query processing delays on the backend DB side). This would introduce additional service delays and is a subject for separate study with highly loaded broker instances. We did not provide the separate plot for the Replica Query Manager since the QoS would be identical to the Query Manager because both threads operate on the local embedded database entries (replicated database and the PDS own database respectively).

Note that, the tests of query processing do not introduce inconsistencies in query results when several replicas are answering queries, even if some of them go offline. This is possible because each replica PDS first downloads the updates before responding to any query. Thus, data freshness is ensured by our protocol.

Sample queries that we executed are given below.

- [Q1:] SELECT * FROM ehr WHERE medicine = "aspirin" AND diagnosis = "flue"

- [Q2:] SELECT * FROM ehr WHERE medicine = "aspirin" AND diagnosis = "flue" AND hospitalName = "PVH"

- [Q3:] SELECT COUNT(*) from ehr

- [Q4:] SELECT COUNT(*) from ehr

The measured CPU and memory load on the Broker Java process has been observed to be relatively negligible with a testing PDS set (up to 12 nodes going online) ranging from 1% to 2% of CPU utilization. We anticipate that highly loaded Broker instance may be able to service a large number of pulling PDS subscriber nodes provided that JVM does correct CPU cores scheduling with underlying OS support.

Overall the developed system prototype has been deployed successfully with replication and query processing performed in the environment where PDS nodes are located in geographically dispersed locations and queries are issued on data distributed across the Internet.

---

**Fig. 6.** PDS nodes distribution on PlanetLab Testbed.

**Fig. 7.** PDS replication QoS on PlanetLab Testbed.

**Fig. 8.** PDS query processing QoS on PlanetLab Testbed.
7. Discussion on architectural properties

Based on the previous work and conducted evaluation we outline a number of properties of personal record storage using the proposed approach and point to potential extensions and improvements of the current design.

Replication properties: The PDS model assumes that individuals have their PHRs on their own PDS devices. For reasons of availability, they may wish to replicate their PHRs on devices belonging to other individuals. The users have total control over what data they wish to replicate, where they would like to replicate, and may also change these decisions at any point of time. Thus, a user may replicate his data to one or more PDS devices belonging to other trustworthy individuals. There is an original socio-economic assumption that replicas are chosen out of the reliable peers (such as family members or close friends) that will be responsible to keep the replica online periodically to ensure the adequate synchronization with the owner PDS [21]. From this standpoint the eventual consistency of personal records is guaranteed provided the PDS holding the replica comes online after the update. However, the property although supported by the replication protocol itself is guaranteed only with the requirement of periodic network connectivity of the PDS devices.

The PDS model assumes that only a select set of PDS devices may be assigned the role of a replica PDS. We assume that a replica PDS contains the PHR of at most one user who is not the owner. However, an owner PDS may choose to replicate its data to multiple replica PDS devices. Therefore a replica PDS cannot make copies of other people’s PHRs and use them at a later stage because the decision to become a replica is granted by a PDS owner only once and that decision is made solely on the inter-personal basis. In the event the owner decides to revoke replication privilege from the device that was chosen to be a replica, the underlying PDS implementation precludes the storage of replicated updates that belong to other PDS devices unless the original replicated content is removed from the device via a hardware tamper-resistant reset operation.

Properties of query processing: In PDS model, we not only ensure eventual consistency of replicated data but also ensure that queries issued by the user get accurate results. The utility of developed query processing protocol lies in the ability to perform relational database queries on a set of embedded database engines running in the remote PDS devices thus opening its usage for potentially wider application domains. Currently our research efforts have been focused on querying the individual PDS databases of a single user where the answer can be obtained from the owner PDS or any of its replicas. Note that the protocol for query processing allows the delegation of query privileges to any set of PDS devices that may have very different roles. For instance a PDS owner may grant or revoke implicit query privilege for its database to a health care practitioner as well as his best friend or a family member at any time. That is ensured through the mapping mechanism of the designed suite of PDS protocols. Therefore a single health care practitioner is able to query the PHR records of different patients provided that they explicitly allow that to happen through their individual PDS devices. A sample use case is depicted in Fig. 9. Note that although PDS replicas could store owner’s PHR data, that does not implicitly allow them to perform queries unless that privilege is explicitly granted by the owner PDS.

Note that PDS devices in our model that form a semi-connected overlay do not have to mutually authenticate with each other (e.g. SPT to Broker, and SPT to SPT, Broker to HealthCare provider or owner, and vice versa). The developed set of protocols incorporates the authentication properties through the notion of mapping the PdsIds that are globally routable overlay addresses used as the identity keys for various aspects of the protocols operation.

Properties of broker layer: Although the broker layer can have various types of architectures, we assume a centralized architecture in this paper that is depicted in Fig. 3. However the different types of architectures do not affect the design and implementation of the introduced suite of replication and query processing protocols. That is ensured by the virtue of our hybrid model that leverages peer-to-peer and publish–subscribe paradigms with the semi-disconnected property of individual PDS devices. The developed suite of protocols ensures the record storage and querying with the fact that the PDS devices could be disconnected from the network and be offline at least for a certain period of time. By the virtue of the publish–subscribe paradigm the broker plays a middleware role in our publish–subscribe interaction where participants are decoupled in space, flow and time and do not require persistent connectivity to the broker instance [32].

Note that, the broker does not take any active role in maintaining replica consistency, but merely acts as a passive temporary storage with the least amount of involved computation efforts. In this regard we adhere to the principle of smart endpoints and dumb network adapted by the Internet design principles. Clearly, the duration of storage depends on how often the owner PDS and replica PDS connect to the broker. The active role that is played by the PDS devices is illustrated by the presence of a Replication WatchDog thread on the owner PDS that keeps track of replica freshness by requesting replica sequence numbers through the broker. If a quorum of replicas is established, a Replication WatchDog can decide to request the deletion of records from the broker storage.

In theory, broker can be equipped with ability to decide when to delete the records in the updates table instead of relying on endpoint WatchDogs that consume a percentage of network traffic. Updates could also be pushed to any of the replica nodes by the more complicated mechanisms that take advantage of the datacenter infrastructure such as load-balancing layer. However, our design model does not rely on the advanced broker layer capabilities since we adhere to the principle of delegating total control of the sensitive record management to the endpoint PDS devices. Moreover, the key original motivation behind the PDS paradigm is to provide the ability to deploy extremely low cost PDS support infrastructure in the emerging markets such as developing countries where datacenter costs have to be reduced to the absolute minimum. Therefore the least expensive broker layer has to be supported with minimal computational overheads to support cheap off-the-shelf server components. Since devices and their replicas have to store all personal records, the backend broker storage temporarily stores records for a short duration after which
the records are periodically removed—this minimizes the storage costs at the broker end.

In large PDS installations, it is possible that a single broker cannot cope with the requests from the various PDS devices. In such cases, the deployment has to be partitioned into a broker mesh. Various types of designs could be considered for the placement and interconnection of individual broker nodes [33]. Since the broker nodes are provisioned in the datacenter, different strategies to ensure operational deployment are possible including horizontal and vertical scaling of the broker nodes, load-balancing of PDS requests, computation-dispatching and automatic provisioning.

One of the possible designs is presented in Fig. 10 where a set of load balancers is deployed in front of the broker mesh to carry out the selective routing of PDS requests to the nearest broker according to geographical prefix that could be incorporated into the PdsId identifier that serves as an overlay routing address for individual PDS devices.

Note that the persistence layer that is responsible for the storage and management of the database tables involved in the PDS broker layer is also subject to various partitioning strategies. For instance a single database table could be vertically or horizontally partitioned across a set of database nodes to ensure consistency guarantees. Similar strategies could be applied to the hash maps that store sequence numbers in main memory. In the event of the node failure those entries are lost and additional resiliency strategies such as distributed shared memory mechanisms have to be designed to ensure the availability [34].

However, the stated enhancements to the centralized architecture are the subject of orthogonal research oriented towards the traditional desire to provide CAP theorem guarantees for modern distributed systems [35] and therefore are out of scope for the analysis of PDS protocols described in the current paper.

Properties of implementation: Since the experimental implementation is done in Java Standard Edition, the communication packets involved in the operation of PDS protocols are implemented in the form of Java class objects—specifically Plain Old Java Objects (POJO). Such a design decision has been made for ease of development and inherent ability of Java to serialize objects during transport. The choice of POJO allows to utilize the ability to get and set the various operational variables of the protocols suite without reliance on external data representation such as XML. Although the communication using XML messages is platform-independent and allows effective inter-application data exchange [32], such a choice is computation-intensive and may not be suitable for resource-constrained hardware such as PDS devices.

Computational limitations and necessity to minimize network traffic overheads have also precluded the realization of PDS protocols on top of existing HTTP/HTTPS protocol implementation [36] during the initial prototype phase.

We utilize the ability of some relational databases such as MySQL to store POJO objects in the form of BLOB directly in the database tables. If both the broker and communicating PDS devices are compatible with respect to Java standard and version support (which is the case for the conducted experiments), then the protocols will not require a computation intensive transformation of POJO into alternative IPC formats. Thus, a broker instance implemented in Java is able to store and process the PDS POJO communication packets directly in memory on the fly without CPU intensive transformation routines.

Applicability of the developed protocols: The developed suite of PDS protocols could potentially be employed in other application domains such as personal information stores and areas related to document flow where personal office documents, invoices, billing statements and checks could be stored in the embedded relational database without exposing the sensitive private information into the public cloud domain [37]. The replication and retrieval of such documents across a plethora of personal devices could be realized by using the developed suite of PDS protocols.

8. PDS protocol analysis

The protocols, so developed, need to be analyzed to ensure their correct behavior. Manual analysis is tedious and error-prone. Towards this end, we demonstrate how Alloy [8] can be used for automated analysis. Alloy is a textual modeling language based on first-order relational logic. An Alloy model consists of signature, fields, facts and predicates. Each field belongs to a signature and represents a relation between two or more signatures. Facts are statements that define constraints on the elements of the model. Predicates are parameterized constraints that can be invoked from within facts or other predicates.

Alloy provides a trace mechanism that associates the transitions triggered by operation invocations with states defined by signatures. Alloy defines an ordering type that can cast a set of states into a sequence of states. A trace fact defines states that are reachable through the invocation of sequence of operations.

We developed [16] a class model analysis approach that uses the Alloy Analyzer [7] to find scenarios (sequences of operation invocations) that start in valid states (states that satisfy the invariants in the class model) and end in invalid states (states that satisfy the negation of the invariants). The analysis uses the operation specifications to determine the effects operations have on the state. If analysis uncovers a sequence of operation calls that moves the system from a valid state to an invalid state, then the designer uses the trace information provided by the analysis to determine how the operation specifications should be changed to avoid this scenario. The approach uses UML-to-Alloy transformation to shield the designer from the back-end use of the Alloy language and analyzer.

The approach in [16] builds upon our previous work on the Scenario-based UML Design Analysis (ScCUDA) approach [38–41]. A designer uses ScCUDA to check whether a specific functional scenario is supported by a design class model in which operations are specified using the OCL. In ScCUDA, the property to be verified is expressed as a specific sequence of state transitions (a functional scenario). The approach described in [16] goes further in that the property to be verified is expressed in terms of valid and invalid states, and analysis attempts to uncover scenarios that start in a specified valid state and end in a specified invalid state. In summary, ScCUDA is used to answer the question “Is the given scenario supported by the UML class model?”, while the approach
Fig. 11. UML-to-alloy transformation overview.

Fig. 11 shows an overview of the UML-to-Alloy transformation. The dotted area includes the front-end activities and models. The front-end models are the only models that a designer needs to manipulate directly. The designer is responsible for (1) providing a UML class model with operation specifications, and (2) specifying the property-to-verify using the OCL [6]. The property-to-verify is expressed in terms of an invariant and its negation: The invariant characterizes the form of valid source states, and its negation characterizes target invalid states. Object configurations representing software states are called snapshots in this paper.

The back-end activities use three transformations (indicated in Fig. 11). Transformation 1 transforms the UML class model to a model, called a snapshot transition model, that specifies valid snapshot transitions, where a transition describes the effect of an operation on a state. The UML-to-snapshot transition model transformation defined in the ScUDA approach [38] is used for this purpose. Transformation 2 converts the snapshot transition model to an Alloy model. The property-to-verify is transformed to an Alloy predicate, referred to as the verification predicate, that is added to the Alloy model generated from the snapshot model. The resulting Alloy model is fed into the Alloy Analyzer and the verification predicate is evaluated. The Alloy Analyzer is used to determine if there exists an operation invocation sequence that starts from a specified valid snapshot and ends in a specified invalid snapshot. If the Analyzer finds a sequence then Transformation 3 is needed to convert the Alloy instance model of the sequence to a UML object model describing the sequence. The designer can examine the result to determine where the error lies in the model.

The formal specification approach described in the paper involves two steps. First, it derives a UML class model with operation specifications from the data replication protocol. Note that the properties defined on the protocol are specified using the OCL invariants, and the functionalities for the data replication are expressed using the operation specifications in the form of OCL pre- and post-conditions. Second, it uses the UML-to-Alloy transformation to generate an Alloy model from the UML class model with the OCL operation specifications.

In the remainder of this section we illustrate the formal specification approach in detail.

8.1. Deriving a class model

Fig. 12 shows a UML class model for the PDS overlay. In the class model, broker can be associated with multiple pdss, while a pd can communicate with only one broker at any given time. BrokerPDS is the association between Broker and PDS. Given a broker, broker.BrokerPDS returns the set of the pdss associated with the broker. Class Broker has four attributes: MappingsTable, UpdatesTable, ReplicaUpdates and OwnerUpdates, where each corresponds to a table or hash map.

Note that by PHR (phr) we mean Personal Health Records as a set of records stored in the PDS embedded database storage. A pd can belong to only one user, and a user can have only one phr. A pd can contain one or two phrs. If a pd is an owner pd, it contains only the phr of the user it belongs to. If a pd is a replica pd, it also contains the phr of a user it does not belong to.

Note that PDSs are associated with themselves by the OwnerReplica association. Given an owner pd, owner.pds.OwnerReplica returns a set of replica pdss associated with the owner pd.

In this paper, we use OCL to specify the operations defined on the PDS overlay class model. For example, addReplica is an operation defined in the context of the Broker class, and it takes as input an owner pd and a replica pd, and associates the replica pdss with the owner pd. The addReplica operation is specified using the OCL as below:

```java
// Add a replica pd for an owner pd
defects Broker::addReplica(owner:PDS, replica:PDS)
// Precondition: the replica pd has not been associated with the owner pd
// Postcondition: the replica pd has been associated with the
```
Note that owner.OwnerReplica returns a set of replica pdss associated with the owner pds. Both excludes() and includes() are the built-in operations in the OCL. The excludes operation returns true if the replica pds is not in the set of the replica pdss, while the includes operation returns true if the replica pds is in the set of the replica pdss.

To ensure that the PHR stored in a pds is always available and reliable, one of the properties for the data replication protocol would be each pds must have at least one replica pds. Such properties can be specified using the OCL invariants, and an example of an OCL invariant for the protocol is given below:

// Each pds must have at least one replica pds.
Context PDS inv MinNumReplicas::
    self.OwnerReplica → includes(replica)

Note that notEmpty() is the built-in operation in the OCL, and it returns true if the collection that calls the operation has more than one element.

8.2. Generating an alloy model

As shown in Fig. 11, the UML-to-Alloy transformation involves the class model-to-snapshot transition model transformation and the snapshot transition model-to-Alloy model transformation. In the remainder of this section we describe these two transformations in Sections 8.2.1 and 8.2.2 respectively.

8.2.1. Generating a snapshot transition model from a class model

Software behavior can be represented as a sequence of state transitions, where each transition is triggered by an operation invocation. Yu et al. [38–40] proposed a scenario-based static analysis approach, called ScUDA, that allows a developer to check whether a particular sequence of state transitions is supported by a class model in which operations are specified in the OCL. In ScUDA, a design class model with operation specifications is transformed to a static model of behavior, called a snapshot transition model. A snapshot represents a system object configuration at a particular time. A key concept in a snapshot transition model is the snapshot transition. A snapshot transition describes the behavior of an operation in terms of how system state changes after the invoked operation has completed its task. It consists of a before state, an after state, and the operation invocation that triggers the transition. An operation invocation is described by the operation name and the parameter values used in the invocation.

Fig. 13 shows an example of a snapshot transition model generated from the class model (including the addReplica operation) given in Fig. 12. Two classes, Snapshot and Transition, are added into the snapshot transition model. The instances of class Snapshot, are snapshots, and the instances of class Transition are transitions that each relates a before snapshot with an after snapshot. A snapshot consists of linked instances of classes in a class model (i.e., an object configuration).

Operations defined on the class model are transformed to specializations of class Transition in the snapshot transition model. For example, operation addReplica(owner: PDS, replica: PDS) is transformed to a specialization of class Transition (e.g., AddReplica). Its parameters (e.g., owner and replica) are transformed into references (shown as attributes) in the Transition specialization. Moreover, if a parameter has a class type, it is transformed into two references. For example, the parameter owner in the operation is transformed into ownerPre: PDS and ownerPost: PDS, one of which (ownerPre) specifies the parameter’s state before the execution of the operation and the other (ownerPost) specifies the parameter’s state after the execution of the operation. Also, two references (e.g., bPre : Broker and bPost : Broker) pointing to before and after states of the object on which the operation is called are generated and placed in the specialized Transition class representing the operation.

Operation specifications are transformed into transition invariants that precisely specify the before and after snapshots that are associated with Transition instances. For example, the operation specification for addReplica is transformed to the following transition invariant on the class AddReplica:

Context AddReplica inv:

// Generated from precondition
before.brokers → includes(bPre) and
before.pdss → includes(ownerPre) and
before.pdss → includes(replicaPre) and
ownerPost.OwnerReplica → excludes(replicaPre) and
// Generated from postcondition
after.brokers → includes(bPost) and
after.pdss → includes(ownerPost) and
after.pdss → includes(replicaPost) and
ownerPost.OwnerReplica → includes(replicaPost) and
// Unchanged parts of object configuration
after.pdss → excluding(ownerPost) → excluding(replicaPost) =
before.pdss → excluding(ownerPre) → excludingAll(replicaPre)

Note that the excluding() an OCL operation that takes as input an object, and returns a collection containing all elements of the operation handler (e.g., after.pdss) minus the object (e.g., ownerPost).

8.2.2. Generating an alloy model from a snapshot transition model

A snapshot transition model is transformed to an Alloy model using the following algorithm described in [16]. Each class that is part of the Snapshot class in the snapshot transition model is transformed to a signature in Alloy. Fig. 14 shows the Alloy signatures that are generated from the classes in Fig. 13. For example, class User is transformed to a signature sig Role(). If a class has attributes, its attributes are transformed to fields of the signature corresponding to the class (e.g., PDS and Broker).

The Snapshot class is transformed to a Snapshot signature containing fields that specify the object configuration within a snapshot. Two groups of fields in the Snapshot signature are used to specify object configurations: fields defining a set of objects...
module PDS
open util/ordering[Snapshot] as SnapshotSequence

sig User() sig PHR()

sig PDS{ PdsId: one Int } sig Broker{ ... }

abstract sig ID()
one sig ID_addReplica, ID_phrUpdate, ..., ID_Null extends ID{}

sig Snapshot{
  operID: one ID,
  // Objects
  users: set User,
  phrs: set PHR,
  ...
  // Links
  UserPHR: User one->one PHR,
  ...
} // Linked objects must exist in the snapshot
UserPHR = UserPHR :> phrs & users <: UserPHR
...

Fig. 14. Alloy signatures generated from the classes in the snapshot transition model in Fig. 13.

(e.g. users: set User), and fields defining links between objects
(e.g. UserPHR : User one->one Role). Linked objects in a snapshot
must be in the domain defined by the Snapshot signature. This
constraint is expressed as a fact associated with the Snapshot
signature. For example, the fact UserPHR = UserPHR :> phrs & users <: UserPHR specifies that linked objects either belong to phrs or
users. The Snapshot signature also includes a field, operID, that
is used to identify the operation that causes a transition to the
snapshot when the snapshot is part of a sequence of transitions.

Each Transition specialization in the snapshot transition model
is transformed to a predicate in Alloy. Fig. 15 shows an Alloy
predicate that is generated from a Transition specialization (i.e.,
addReplica) in Fig. 13. If a Transition specialization has attributes,
its attributes are transformed to parameters of the predicate. Two
more parameters, before and after with the type Snapshot, are
added to each predicate to represent the system states before and
after the transition. OCL invariants associated with each Transition
specialization in the snapshot transition model are transformed
into the body of the predicate corresponding to the Transition
specialization using UML2Alloy [42–44]. Objects and links that are
not changed during the transition are explicitly specified in the
predicate. An equality that identifies the operation causing the
transition (e.g. after.operID = ID_addReplica) is also included in
each predicate.

Alloy provides a trace mechanism that associates the transitions
triggered by operation invocations with states (i.e., snapshots)
defined by signatures. The trace mechanism uses an ordering
type that casts a set of snapshots into a sequence of snapshots
(e.g., open util/ordering[Snapshot] as SnapshotSequence in Fig. 14).
The trace mechanism also uses an Alloy fact to associate transitions
between two consecutive snapshots with operation invocations.
An example of a trace fact for the Alloy model is given in Fig. 16.
This fact specifies that the snapshots supported by the Alloy model
are reachable through the invocation of sequence of operations
(e.g., addReplica and_phrUpdate).

9. Formal analysis results

After the snapshot transition model is transformed to an Alloy
model, the property-to-verify provided by the user is transformed
to an Alloy predicate, referred to as the verification predicate.

pred addReplica[diss before, after: Snapshot, ownerPre, ownerPost: PDS, replicaPre, replicaPost: PDS][
  after.operID = ID_addReplica
  // Precondition
  ownerPre in before.pdss
  replicaPre in before.pdss
  replicaPre not in ownerPre.(before.OwnerReplica)
  // Postcondition
  ownerPost in after.pdss
  replicaPost in after.pdss
  replicaPost in ownerPost.(after.OwnerReplica)
  // Unchanged objects
  before.users = after.users
  before.pdss = after.pdss
  ...
  // Unchanged links
  before.BrokerPDS = after.BrokerPDS
  before.OwnerReplica - ownerPre<(before.OwnerReplica)
  -(before.OwnerReplica) > replicaPre =
  after.OwnerReplica - ownerPost<(after.OwnerReplica)
  -(after.OwnerReplica) > replicaPost
  ...
]

Fig. 15. Alloy predicate generated from the transition specialization class in the
snapshot transition model in Fig. 13.

fact traces {
  all before: Snapshot - SnapshotSequence/last |
  let after = SnapshotSequence/next[before] |
  some owner: PDS | some phr: PHR | some replica: PDS|
    addReplica[before, after, owner, owner, replica, replica||phrUpdate[before, after, owner, owner, phr, phr]]|

  ...
}

Fig. 16. An example of an alloy trace fact.

pred checkMinimumReplicas{
  let first = SnapshotSequence/first | first.operID = ID_Null
  all owner1: first.pdss| owner1.(first.OwnerReplica) != none
  let last = SnapshotSequence/last | some owner2: last.pdss | owner2.(last.OwnerReplica) = none
}

Fig. 17. An example of a verification predicate.

The property-to-verify is expressed in terms of an invariant and
its negation: The invariant characterizes the valid source
snapshots, and its negation characterizes target invalid snapshots.
The verification predicate is added to the Alloy model generated
from the snapshot transition model. The resulting Alloy model
is fed into the Alloy Analyzer and the verification predicate is
evaluated. This predicate will check whether an invalid snapshot
is reachable from a valid snapshot through operation executions.
An example of a property-to-verify for the PDS class model
would be the MinimumReplicas invariant and its negation. Fig. 17
shows an example of a verification predicate generated from the
MinimumReplicas invariant and its negation. The Alloy Analyzer
uses the verification predicate to query whether there exists an
operation execution path from a valid snapshot, where each pds
has at least one replica, to an invalid snapshot, where there exists
one pds that has no replicas. The Alloy Analyzer returns an instance
that satisfies the predicate if one path is found.

In this case study, the Analyzer did find a path. Figs. 18 and
19 show a sequence of snapshots produced by the Analyzer for
the verification predicate. The snapshot in Fig. 18 (referred to as the first snapshot) shows that the initial state includes one broker, three users (i.e., User0, User1, User2), three phrs (i.e., PHR0, PHR1, PHR2) and three pdss (i.e., PDS0, PDS1, PDS2). Each user owns a pds that stores the user’s phr. For example, User0 owns PHR0, and PHR0 is stored in PDS2. In addition, each pds has a replica pds that stores the same phr as the pds does. For example, PDS1 is a replica pds of PDS2, and PDS1 stores PHR0 that belongs to User0. Note that since this is the start state, operID has the value ID_Null.

The snapshot in Fig. 19 (referred to as the second snapshot) shows that the addReplica operation has been invoked because the value of operID is ID_addReplica, indicating that the addReplica operation caused the transition from the first snapshot to the second snapshot. Before the execution of the addReplica operation, PDS2 was a replica of PDS0. In this transition, the addReplica operation added PDS2 to a list of replica pdss that belong to PDS1. After the operation execution, PDS1 has two replicas, PDS0 and PDS2. However, since the protocol requires a replica pds to contain the phr of at most one user who is not the owner (see Section 3), PDS2 was no longer a replica of PDS0 after the transition. At this point, PDS0 has no replicas, indicating that the execution of the addReplica operation makes the snapshot invalid.

The analysis results suggest that the operations specification for addReplica may need to be strengthened since it allowed a transition from a valid snapshot to an invalid one. To improve the model, we strengthened the precondition of the operation by adding a clause that allows a pds to be set as a replica only if the pds is not a replica of any pdss before the operation execution. When the PDS model with this modified operation specification is analyzed at the back-end by the Alloy Analyzer, no instance satisfying the verification predicate is found within a reasonably bounded scope.

10. Related work

10.1. Data management

PDS suggests a radically different way of considering the management of personal data. The idea of using SPTs for storing personal data was advocated in an earlier work [17] where we presented some high level ideas. Subsequently, we presented the PDS architecture and our original query processing and replication protocol [21] with respect to health care applications. The current work improves upon the earlier work by showing how to perform a formal analysis. Our analysis does reveal an error in the original data replication protocol and thereby helps to refine the protocol.

Our work bears some similarity to peer-to-peer (P2P) data management research. Commercial peer-to-peer systems (such as Gnutella and Kazaa) are rather limited when considered from the perspective of database functionality. These systems provide only file level sharing with no sophisticated content-based search/query facilities. They are developed as single-application systems that focus on performing one task, and it is not straightforward to extend them for other applications/functions [45]. Typical applications that can take advantage of P2P systems are probably lightweight and involve some sort of cooperation [46]. One significant difference with P2P systems is that our query-processing is very close to conventional databases. We can express most of the relational queries on any node unlike P2P systems [47–49]. This is effectively due to the use of embedded database engine in the PDS node that provides the underlying support for relational queries.

The need for database functionality embedded in various forms of lightweight computing devices is giving rise to personal folders on chips, networks of sensors and mobile computing devices [50]. Sensor networks gathering weather, pollution or traffic information have motivated several recent works [51].
where a need arises to enable distributed query processing over a sensor network [52]. We do not focus on distributed query processing in this work, but instead on how data can be replicated across multiple nodes and query processed even when some nodes are not available.

PDS topology infrastructure uses the same concept of the overlay network—a similar infrastructure for all P2P systems, which is built on top of a physical network. PDS overlay uses unique 128 byte (and expandable) PdsId identifiers to distinguish nodes in the network from each other during query processing and replication operations. PDS overlay topology could be compared to the P2P hybrid networks that are commonly known as super-peer systems where some peers are responsible for controlling a set of other peers in their domain [46]. Examples of super-peer networks include Edutella [53] and JXTA [54]. PDS nodes are the nodes that interconnect with each other using a publish–subscribe model through broker network [55] that is somewhat different from the existing super-peer systems with better routing efficiency than that of peer-to-peer DHT alternative [46]. Thus, the PDS model borrows from P2P and publish–subscribe models; however, instead of relying heavily on DHT as is done in P2P systems, we use the broker network of supporting servers to carry out the efficient logical routing between the PDS nodes.

Certain applications, such as, agenda management, bulletin boards, cooperative auction management, and reservation management, require the data to be current. Replica consistency is particularly important in such applications. Supporting data freshness in P2P systems is difficult as there is considerable dynamism in the peers joining and leaving the system. In P2P systems the problem is sometimes addressed by using data versioning [56] where each replica has a version number that is increased after each update. Data freshness in PDS being used for PHRs is extremely important since the action taken by individuals/organizations depends on the accuracy of the health records. In PDS model, we not only ensure eventual consistency of replicated data but also ensure that queries issued by the user get accurate results.

PDS model also has some commonalities with the Local Relational Model (LRM) [47]. LRM assumes that the peers hold relational databases, and each peer knows a set of peers with which it can exchange data and services. The data residing in each peer may have semantic dependencies with that stored in the others. The authors propose coordination formulas that help resolve the semantic heterogeneity of the data stored at different peers. We do not consider semantic heterogeneity in this paper at all. We focus on how PHRs can be stored and replicated in low powered devices which are under the control of individual users.

10.2. Formal verification

Formal languages, such as OCL, Alloy, and Z, have been proposed to analyze UML models. As OCL becomes part of UML, several OCL-based model analysis tools have appeared. The ModelRun tool [57] allows interactive verification of OCL properties and can load the UML model from the files created by other tools such as Rose 2000. However, neither OCL operation evaluation nor pre/postconditions verification can be performed by ModelRun. The OCL [58] provides model validation against methodology, profile or target implementation language rules expressed in OCL. However, above OCL tools cannot validate whether an object model of the system conforms to OCL constraints defined in the class model of the system. USE [59–61] provides a snapshot that shows an object configuration of the specified system at a particular point of time. Both structural and behavioral models can be checked and evaluated against OCL constraints given by a user in USE. However, a user must manually simulate the system behavior to analyze a scenario.

Model checking techniques, such as NuSMV [62], have been proposed to test whether a model satisfies a given specification. However, these techniques face the state-space explosion problems. Also, these approaches require users to have a background in temporal logic.

Other formal languages, such as Alloy [8,63,7], have been proposed as a light-weight formal modeling analyzer. Alloy allows users to generate instances of invariants, animate the execution of operations and check user-specified properties by exploring a search space. Most importantly, Alloy provides a dynamic analysis mechanism that can automatically generate a sequence of system states associated with the random execution of operations. However, Alloy is not an object oriented language and requires users to have background in relation logic.

Anastasakis [43] and Bordbar [44] proposed a model-based analysis technique, UML2Alloy, for the automated transformation of UML class diagrams with OCL constraints to Alloy code. Sun [64] proposed a transformation approach to bridge the gap between EMF-based modeling tools and the USE tool, in order to take advantage of different tools for rigorous model analysis. Unlike our approach, their approach focuses on static analysis of a UML model. Alloy has been used to check network protocols [9–12] and access control polices [13–15]. Zave et al. [9–11] used Alloy to analyze a variety of network protocols (e.g., Chord protocol). Arye et al. [12] proposed an Alloy model for analyzing the stability properties of the Border Gateway Protocol (BGP). Schaaf et al. [13] used Alloy to formally analyze role-based access control policies. Samuel et al. [14] proposed a framework for specifying and verifying generalized spatio-temporal role-based access control model using Alloy. Toachchoodee et al. [15] used Alloy to analyze a spatio-temporal access control model that supports delegation. However, all these works require the users to have a good background in relational logic and Alloy.

11. Conclusion and future work

Personal Data Server (PDS) suggests a radically different way of considering the management of personal data. In this work, we investigated how PHRs of individuals can be stored on PDSs such that individuals have more control over their data. The data of an individual is stored on the owner’s PDS and it is replicated on the PDS of one or more individuals who are trusted by the owner. We proposed an architecture for PDS overlay and discussed how the data can be replicated and simple queries processed. Our replication algorithm guarantees eventual consistency provided the PDS holding the replica comes online after the update. Our query processing ensures that query response time is minimized and the latest updates are reflected in the query response. We built a prototype and performed experiments using the PlanetLab to demonstrate the feasibility of our approach.

Formal analysis is needed to ensure the correct behavior of the protocols used in PDS overlays. Towards this end, we demonstrated how the data replication protocol can be formally specified using UML and OCL. We also illustrate how the specifications can be automatically verified for Alloy using our UML-to-Alloy tools. The Alloy specifications so generated can be automatically verified. Our analysis reveals that there was a flaw in the earlier proposed protocol.

A lot of work remains to be done. Our current implementation uses a centralized broker architecture. Such an architecture will not scale and may be a target of denial-of-service attacks. One future work involves how to develop a distributed broker architecture to improve reliability and availability. We plan on how to specify access control, retention, audit, and privacy policies for PHR applications and how to analyze the policies in real-time to check for conflicts. We plan to design a reference monitor in the
secure hardware of SPTS to enforce such policies. We also plan to address denial-of-service attacks in the context of PDSs.

We plan to produce an analysis framework that allows developers to formally explore how data replication protocols interact with other desired properties of the PDS system. We also plan to provide a formal specification for analyzing the query processing described in [2]. Specifically we are investigating how we can introduce the time concept into the Alloy model to check whether the response time is minimized and the latest updates are reflected in the query response.

Acknowledgments

This work was partially supported by the NSF under grant CCF-1018711, CISE-1619641, CISE-1540041, by the NIST under grant 70NANB15H264, and by a Colorado State University Quarterly VIPrvestment Fund Q1-15 grant.

References


Kirill Belyaev is a Ph.D. student at Colorado State University.

Wuliang Sun is a Software Engineer at Amazon. He obtained his Ph.D. from Colorado State University and an M.S. from Baylor University.

Indrakshi Ray is a Professor of Computer Science at Colorado State University. She obtained her Ph.D. from George Mason University.

Indrajit Ray is a Professor of Computer Science at Colorado State University. He obtained his Ph.D. from George Mason University.