Applying Software Patterns to Address Interoperability Challenges in Blockchain-based Healthcare Apps

Peng Zhang, Vanderbilt University, Nashville, TN
Jules White, Vanderbilt University, Nashville, TN
Douglas C. Schmidt, Vanderbilt University, Nashville, TN
Gunther Lenz, Varian Medical Systems, Palo Alto, CA

Since the inception of Bitcoin technology, its underlying data structure—the blockchain—has garnered much attention due to properties such as decentralization, transparency, and immutability. These properties make blockchains suitable for apps that require disintermediation through trustless exchange, consistent and incorruptible transaction records, and operational models beyond cryptocurrency. In particular, blockchain and its smart contract capabilities have the potential to address healthcare interoperability issues, such as enabling effective interactions between users and medical applications, delivering patient data securely to a variety of organizations and devices, and improving the overall efficiency of medical practice workflow. Despite the interest in using blockchain technology for healthcare interoperability, however, little information is available on the concrete architectural styles and patterns for applying blockchain technology to healthcare apps. This paper provides an initial step in filling this gap by showing: (1) the features and implementation challenges in healthcare interoperability, (2) an end-to-end case study of a blockchain-based healthcare app that we are developing, and (3) how applying foundational software patterns can help address common interoperability challenges faced by blockchain-based healthcare apps.

Categories and Subject Descriptors: H.5.2 [Software Patterns] Blockchain-based healthcare apps

1. INTRODUCTION

Over the past several years blockchain technology has attracted interest from computer scientists and domain experts in various industries, including finance, real estate, healthcare, and transactive energy. This interest initially stemmed from the popularity of Bitcoin [Nakamoto 2012] and the Bitcoin platform, which is a cryptographic currency framework that was the first application of blockchain. Blockchains possess certain properties, such as decentralization, transparency, and immutability, that have allowed Bitcoin to become a viable platform for "trustless" transactions, which can occur directly between any parties without the intervention of a centralized intermediary.

Another blockchain platform, Ethereum, extended the capabilities of the Bitcoin blockchain by adding support for "smart contracts." Smart contracts are computer programs that directly control exchanges or redistributions of digital assets between two or more parties according to certain rules or agreements established between involved parties. Ethereum's programmable smart contracts enable the development of decentralized apps (DApps) [Johnston et al. 2014], which are autonomously operated services cryptographically stored on the blockchain that enable direct interaction between end users and providers.

This paper focuses on a previously unexplored topic related to blockchains, namely, the application of software patterns to modularize and facilitate extensibility of blockchain-based apps that focus on addressing the interoperability challenges in healthcare. In the healthcare context, interoperability is defined as the ability for different information technology systems and software apps to communicate, exchange data, and use the information that has been exchanged. The high (and growing) cost of healthcare in many countries motivates our focus on applying...
blockchain technologies to help bridge the gap in communication and information exchange [DeSalvo and Galvez 2015].

The remainder of this paper is organized as follows: Section 2 gives an overview of the blockchain technology and the Ethereum framework—an open-source blockchain implementation; Section 3 outlines key challenges regarding healthcare interoperability faced by blockchain-based apps; Section 4 provides an end-to-end case study of a blockchain-based healthcare app we are developing and describes four blockchain-related versions of familiar software patterns applied to address interoperability requirements as well as key concerns when realizing these patterns; Section 5 compares our work with existing work on potential pros and cons of a health blockchain; and Section 6 presents concluding remarks and summarizes our future work on applying blockchain technologies in the healthcare domain.

2. OVERVIEW OF BLOCKCHAIN AND ITS ROLE IN HEALTHCARE APPS

This section gives a general overview of blockchains and the open-source Ethereum implementation that provides additional support for smart contracts, which are computer protocols that enable different types of decentralized apps beyond cryptocurrencies. The general blockchain overview is followed by a discussion of Solidity, which is the programming language used to write Ethereum smart contracts.

2.1 Overview of Blockchain

A blockchain is a decentralized computing architecture that maintains a growing list of ordered transactions grouped into blocks that are continually reconciled to keep information up-to-date, as shown in Figure 1. Only one block can be added to the blockchain at a time. Each block is mathematically verified (using cryptography) to ensure that it follows in sequence from the previous block, thereby maintaining consensus across the entire decentralized network. The verification process is called “mining” or Proof of Work [Nakamoto 2012], which allows network nodes (also called “miners”) to compete to have their block be the next one added to the blockchain by solving a computationally expensive puzzle. The winner then announces the solution to the entire network to gain a mining reward paid via cryptocurrency.

The blockchain verification process combines game theory, cryptography, and incentive engineering to ensure that the network reaches consensus regarding each block in the blockchain and that no tampering occurs in the transaction history. All transaction records are kept in the blockchain and are shared with all network nodes. This decentralized process ensures properties of transparency, incorruptibility, and robustness (since there is no single point of failure).

In the Bitcoin application, blockchain serves as a public ledger for all transactions of cryptocurrency in bitcoins to promote trustless financial exchanges between individual users, securing all their interactions with cryptography. The Bitcoin blockchain has limitations, however, when supporting different types of applications involving contracts, equity, or other information, such as crowdfunding, identity management, and democratic voting registry [Buterin 2014]. To address the needs for a more flexible framework, Ethereum was created as an alternative blockchain, giving users a generalized trustless platform that can run smart contracts.
The Ethereum blockchain is a distributed state transition system, where state consists of accounts and state transitions are direct transfers of value and information between accounts. Two types of accounts exist in Ethereum: (1) *externally owned accounts* (EOAs), which are controlled via private keys and only store Ethereum's native value-token "ether" and (2) *smart contract accounts* (SCAs) that are associated with contract code and can be triggered by transactions or function calls from other contracts [Buterin 2014].

To protect the blockchain from malicious attacks and abuse (such as distributed denial of service attacks in the network or hostile infinite loops in smart contract code), Ethereum also enforces a payment protocol, whereby a fee (in terms of "gas" in the Ethereum lexicon) is charged for memory storage and each computational step that is executed in a contract or transaction. These fees are collected by miners who verify, execute, propagate transactions, and then group them into blocks.

As in the Bitcoin network, the mining rewards provide an economic incentive for users to devote powerful hardware and electricity to the public Ethereum network. In addition, transactions have a gas limit field to specify the maximum amount of gas that the sender is willing to pay for. If gas used during transaction execution exceeds this limit, computation is stopped, but the sender still has to pay for the performed computation. This protocol also protects senders from running completely out of funds.

2.2 Overview of Solidity

Ethereum smart contracts can be built in a Turing-complete programming language, called Solidity [Foundation 2015b]. This contract language is compiled by the Ethereum Virtual Machine (EVM), which enables the Ethereum blockchain to become a platform for creating decentralized apps (DApps) that provide promising solutions to healthcare interoperability challenges. Solidity has an object-oriented flavor and is intended primarily for writing contracts in Ethereum.

A "class" in Solidity is realized through a "contract," which is a prototype of an object that lives on the blockchain. Just like an object-oriented class can be instantiated into a concrete object at runtime, a contract may be instantiated into a concrete SCA by a transaction or a function call from another contract. At instantiation, a contract is given a uniquely identifying address similar to a reference or pointer in C/C++-like languages, with which it can then be called.

Contracts may also contain persistent state variables that can be used as data storage. Although one contract can be instantiated into many SCAs, it should be treated as a singleton to avoid storage overhead. After a contract is created, its associated SCA address is typically stored at some place (e.g., a configuration file or a database) and used by an app to access its internal states and functions.

Solidity supports multiple inheritance and polymorphism. When a contract inherits from one or more other contracts, a single contract is created by copying all the base contracts code into the created contract instance. Abstract contracts in Solidity allow the definition of declaration headers without concrete implementations. Although these abstract contracts cannot be compiled into a SCA they can be used as base contracts. Due to Solidity contracts' similarity to C++/Java classes, many foundational software patterns can be applied to smart contracts to address various design challenges, as described next.

3. HEALTHCARE INTEROPERABILITY CHALLENGES FACED BY BLOCKCHAIN-BASED APPS

Many research and engineering ideas have been proposed to apply blockchains to healthcare and implementation attempts are underway [Ekblaw et al. 2016; Peterson et al. 2016; Porru et al. 2017; Bartoletti and Pompianu 2017]. Few published studies, however, have addressed software design considerations needed to implement blockchain-based healthcare apps effectively. While it is crucial to understand the fundamental properties that blockchains possess, it is also important to apply solid software engineering practices when programming a blockchain to minimize software maintenance effort via code modularity and extensibility, especially in the fast-growing and highly demanding healthcare domain. This section summarizes key interoperability challenges in healthcare faced...
by blockchain-based apps, focusing on four aspects: app evolvability, costly storage on blockchain, healthcare information privacy, and scalability.

3.1 Evolvability Challenge: Maintaining Evolvability While Minimizing Integration Complexity

Many apps are written with the assumption that data is easy to change. In a blockchain app, however, data is immutable and difficult to modify en masse. A critical design consideration when building blockchain apps for healthcare is thus ensuring that the data written into blockchain contracts are designed to facilitate evolution where needed. Although evolution must be supported, healthcare data must often be accessible from a variety of deployed systems that cannot easily be changed over time. App evolvability should therefore be designed in a way that is loosely coupled and minimizes the usability impact of evolution on the clients that interact with data in the blockchain. Section 4.2.1 shows how the Abstract Factory pattern can be applied in Ethereum contracts to facilitate evolution, while minimizing the impact on dependent healthcare app clients.

3.2 Storage Challenge: Minimizing Data Storage Requirements

Healthcare apps can serve thousands or millions of participants, which may incur enormous overhead when all the data is stored in a blockchain—particularly if data normalization and denormalization techniques are not carefully considered. Not only is it costly to store large amounts of data in the blockchain, but data access operations may fail if the cost exceeds the Ethereum gas limit discussed in Section 2.1. An important design consideration for blockchain-based healthcare apps is thus to maximize data sharing while ensuring sufficient flexibility to manage individual health concerns. Section 4.2.2 shows how the Flyweight pattern can be applied to help ensure that common intrinsic data is shared across Ethereum contracts while still allowing extrinsic data to vary in contracts specific to individuals.

3.3 Privacy Challenge: Balancing Interoperability with Privacy Concerns

The US Office of the National Coordination for Health Information Technology (ONC) has outlined basic technical requirements for achieving interoperability [ONC 2014]. These requirements include identifiability and authentication of all participants, ubiquitous and secure infrastructure to store and exchange data, authorization and access control of data sources, and the ability to handle data sources of various structures. Blockchain technologies are emerging as promising and cost-effective means to meet some of these requirements due to their inherent design properties, such as secure cryptography and a resilient peer-to-peer network. Likewise, blockchain-based apps can benefit the healthcare domain via their properties of asset sharing, audit trails of data access, and pseudonymity for user information protection, which are essential for solving interoperability issues in healthcare.

Although there are substantial potential benefits to interoperability and availability of storing patient data in the blockchain, there are also significant risks due to the transparency of blockchain. In particular, even when encryption is applied it is still possible that the current encryption techniques may be broken in the future or that vulnerabilities in the encryption algorithms implementations used may lead to private information potentially being decryptable in the future. Section 4.2.3 shows how the Proxy pattern can be applied to aid in facilitating interoperability through the blockchain while keeping sensitive patient data from being directly encoded in the blockchain.

3.4 Scalability Challenge: Tracking Relevant Health Changes Scalably Across Large Patient Populations

Communication gaps and information sharing challenges are serious impediments to healthcare innovation and the quality of patient care. Providers, hospitals, insurance companies, and departments within health organizations experience disconnectedness caused by delayed or lack of information flow. Patients are commonly cared for by various sources, such as private clinics, regional urgent care centers, and enterprise hospitals. A provider may have hundreds or more patients whose associated contracts must be tracked. Section 8 shows how the
Publisher-Subscriber pattern can be applied to scalably detect when changes of relevance occur in an individual’s Ethereum contracts.

4. APPLYING SOFTWARE PATTERNS TO BLOCKCHAIN-BASED HEALTH APPS

This section presents the structure and functionality of the DApp for Smart Health (DASH) we are developing to explore the efficacy of applying blockchain technology to the healthcare domain. It then describes how foundational software patterns [Gamma 1995; Buschmann et al. 2007] can be applied in blockchain-based health apps (such as DASH) to address the interoperability challenges described in Section 3.

4.1 Structure and Functionality of DASH

DASH provides a web-based portal for patients to access and update their medical records, as well as submit prescription requests. Likewise, the app allows providers to review patient data and fulfill prescription requests based on permissions given by patients. Figure 2 shows the structure and workflow of DASH.

DASH is implemented on an Ethereum test blockchain, with a FHIR (Fast Healthcare Interoperability Resources) [Bender and Sartipi 2013] schema as the standard data format to store patient data. Using the smart contract support in Ethereum, a Patient Registry contract is created to store a mapping between unique patient identifiers and their associated Patient Account contract addresses. Each Patient Account contract also contains a list of health providers that are granted read/write access to the patient’s medical records. DASH provides basic access services to two types of users: patients and providers.

4.2 Applying Software Patterns to Improve the Design of DASH

In the remainder of this section, we use pattern form to show how software pattern motivations and opposing forces are manifest in blockchain-based healthcare apps. We also explain how these patterns can be realized in these apps and the forces that they resolve in the blockchain platform. In particular, we focus on four design patterns—Abstract Factory, Flyweight, Proxy, and Publisher-Subscriber—that we applied to DASH to address the healthcare interoperability challenges in Section 3. Figure 3 shows how these patterns are applied in the DASH app to address the following design challenges:

1There are other patterns relevant in this domain, which will be the subject of our future work.
Applying Software Patterns to Address Interoperability Challenges in Blockchain-based Healthcare Apps — Page 6

---

Abstract Factory assists with user account creations based on user types.

Flyweight ensures accounts on the blockchain are unique and maximizes commonality sharing.

Flyweight ensures accounts on the blockchain are unique and maximizes commonality sharing.

Publisher-Subscriber notifies collaborating users when events of interest occur on the blockchain.

Each of these patterns is discussed further below.

4.2.1 Applying Abstract Factory to Address Evolvability Challenges.

**Context** Any blockchain-based healthcare app that handles entity creation and/or management.

**Problem** How to facilitate evolvability and ease of integration when new entities of different functions are introduced.

**Motivation** The incorruptibility property of blockchain ensures that interfaces (e.g., methods and states) of already instantiated contracts cannot be modified or upgraded. Each new version of a smart contract must be created as a new contract object on the blockchain and distributed among all the network nodes so it can be executed on demand. It is therefore important to design a contract to modularize code and minimize changes to its interface over time.

For example, if a blockchain is used to enable interactions between different departments in a hierarchical health organization without a coherent design, many decisions must be made by clients based on the specific department or sub-department type encountered (a large number of if-else statements). As new departments are introduced, the decision-making code will likely have to change more than once with each obsolete version of contract discarded. The desired model should minimize contract re-creation.

**Solution** By applying the Abstract Factory pattern in smart contracts, apps like DASH can shift the responsibility for providing entity/account creation and management services (in the above example case, for the departmental hierarchies in a health organization) to a “factory” object (the factory itself is a contract instance). Each factory object can create contracts for a group of departments or subdivisions that are related or always interact, making it easier to instantiate another factory object when new interactions take place. Figure 4 shows the basic structure of this pattern in the context of DASH.

As shown in Figure 4, a “super” contract Account could be used to define some common logic used by all types of user accounts, e.g., pharmacies, physicians’ offices, insurance companies. Such common logic may include patient data access, payment handling, and prescription lookup. At creation time, the Account contract may not be able to anticipate all user contract types, but by applying Abstract Factory future derived contracts can be created.
at runtime. Specific and concrete implementations of the logic functions can be deferred to individual derived entity contracts.

In our DASH example, the billing department within a hospital organization may have smaller subdivisions that assist different sets of patients depending on what insurance they carry. When the hospital decides to accept insurance plans from a new provider, the interaction between the hospital subdivision and patients with the new provider can simply be created by a factory instead of changing the existing code to include this new interaction.

**Consequences** Without using a factory contract, the client interested in creating department accounts for the entire organization must access each specific account creation factory and make many if-else decisions at runtime. This tight coupling is cumbersome for the client since it needs to know all implementation details to use each factory’s methods correctly. As shown in Figure 5, to create an account for a care provider and insurance departments, the client imports both the ProviderFactory and the InsuranceFactory and invokes the createAccount() and createOrganization() methods for each account creation. As an organization scales out and up with more departments and subdivisions, the client will be overwhelmed with these implementation details.

The *Abstract Factory* pattern introduces a weak coupling between the app and concrete contract implementations, facilitating future contract extensions that share some commonality and also supporting customizations to the concrete contracts. As new interactions are introduced, new contracts can be created to capture the new change, leaving existing contracts intact, thus avoiding contract deprecation. The downside of applying this pattern in the blockchain, however, is the extra cost incurred by an added layer of indirection in terms of storage for the interface contract and computation of instantiating the interface contract.

### 4.2.2 Applying Flyweight to Address Storage Challenge.

**Context** Any blockchain-based healthcare app that stores data on-chain.

**Problem** How to maximize data storage sharing on the blockchain to avoid cost overhead.

**Motivation.** To achieve blockchain’s transparency and immutability properties, all of the data and transaction records are maintained in the blockchain by replicating and distributing to every node in the network. It is important to limit the amount of data stored in the blockchain to avoid high cost of data storage and unattended data when it is no longer needed.
In a healthcare scenario, if a blockchain is being used to store patient billing data, there will be millions of patients stored in the blockchain. Moreover, each patient will require storage of their insurance information, which will include details, such as their ID#, insurance contact information, coverage details, and other aspects that the provider needs to bill for services. Capturing all this information for every patient can generate a large amount of data in the blockchain.

In the DASH app case, we would like to store detailed insurance and billing information in the blockchain. Since most patients are covered by one of a relatively small subset of insurers (in comparison to the total number of patients), there is a substantial amount of extrinsic, non-varying information that is common across patients, such as the policies on what procedures are covered by their insurance policy. Each insurance policy may cover 10,000s or 100,000s of patients and this information can potentially be reused and shared across patients. The details on what a particular policy covers are common across every patient with that policy. To bill for a service, however, this common extrinsic information must be combined with extrinsic information (such as the patient's ID#) that is unique to each patient.

A good design should maximize common data sharing to reduce data storage cost on the blockchain and still have the capability to provide complete data objects based on specific requests on demand.

**Solution** To resolve the tension between needing to store detailed information and desire to minimize data stored in the blockchain, we can combine the Flyweight pattern with a factory to create a "registry" model to externalize shared data in a common contract, i.e. registry, while allow varying intrinsic data to be stored in entity-specific contracts. The registry can also store references or addresses to entity-specific contracts and return combined extrinsic and intrinsic data on demand.

Figure 6 shows how we applied the Proxy pattern to DASH. By applying Flyweight to create a registry contract in DASH, shared insurance information is stored only once in the registry, avoiding an exorbitant amount of memory usage from saving repeated data in all patient accounts. Varying, patient-specific billing information is stored in corresponding patient-specific account contracts.

The registry in DASH also maintains a mapping between unique user identifiers and the referencing addresses to the each patient account contract to prevent account duplication. At account creation, only if no account with the specified patient identifier exists in the registry does it create a new account; otherwise the registry retrieves the address associated with the existing account contract. To retrieve complete insurance and billing information of a
Fig. 6. Applying the Flyweight Pattern to DASH

particular patient, the client need only invoke a method call from the registry contract with the patient identifier, and the registry can return the combined extrinsic and intrinsic data object.

Consequences. Applying the Flyweight pattern can provide better management to the large object pool, such as user accounts in the example above. It minimizes redundancy in similar objects and, in the meantime, maximizes data and operation sharing. A further substantial advantage of Flyweight in the context of blockchains is that data is immutable once written. If insurance policy details are stored in each patient’s contract directly, the cost to change a policy detail will be immense, since it will require rewriting a huge number of impacted contracts. Flyweight helps minimize the cost of changes to the intrinsic state in blockchain apps.

Similar to the drawback of applying the Abstract Factory pattern, however, Flyweight incurs additional overhead to the client because of the extra layer of complexity. Factory operation that returns the flyweight contract may also take longer to execute because it becomes another transaction to verify and included in the blockchain before a valid address can be available. This computation delay, however, may still be outweighed by the efficiency in management provided by Flyweight.

4.2.3 Applying Proxy to Address Privacy Challenge.

Context Any blockchain-based healthcare app required to expose some data or meta data of sensitive information (such as patient health records) on the blockchain.

Problem How to maximize health data privacy on the blockchain while enabling health interoperability.

Motivation. A fundamental aspect of a blockchain is that all data stored in the blockchain is public, immutable, and verifiable. For financial transactions that are focused on proving that transfer of an asset occurred, these properties are critical. When the goal is to store data in the blockchain, however, it is important to understand how these properties will impact the use case.

For example, storing patient data in a blockchain can be problematic, since it requires the data to be public and immutable. Although data can be encrypted before being stored, should all patient data be publicly distributed in the blockchain to all other parties for verification? Even if encryption is used, it is possible that the encryption technique may be broken in the future or that bugs in the implementation of the encryption algorithms or protocols used may lead to the information potentially being decryptable in the future. The immutability, however, prevents owners of the data from removing the data from the blockchain if a security flaw is ever found. Many other scenarios, ranging from discovery of medical mistakes in the data to changing standards may necessitate the need to change the data over time.

In scenarios where the data may need to be changed, the public and immutable nature of the blockchain creates a fundamental tension that needs to be resolved. On the one hand, the healthcare providers would like the data to
be incorruptible so that it cannot be tampered with. At the same time, providers want the data changeable and private to protect patient privacy and account for possible errors. An interoperable healthcare blockchain app should protect patient privacy and also ensure data integrity.

**Solution** A DApp can implement the *Proxy* pattern in combination of a data retrieving service, such as Oracle [Foundation 2015a] to provide secure and private data services. The Oracle network is a third party that allows data sources outside the blockchain address space to be queried and retrieved by a contract on the blockchain while ensuring data retrieved is genuine and untampered. To reduce computational costs on-chain, a remote proxy can be created to provide some lightweight representation or placeholder for the real data object until the real object is required to be retrieved. Figure 7 shows how we applied the *Proxy* pattern to DASH. This figure shows how DASH uses a proxy contract to expose some simpler metadata of a patient and later refer to the heavyweight implementation on demand to obtain the real data object. Each read request and modification operation of the data store can be logged in an audit trail that is transparent to the entire blockchain network for verification against data corruption. In the case of proxified contract (with heavyweight implementation) being updated with new storage option (such as replacing an Oracle with some other data service), interface to the proxy contract can remain the same, encapsulating detailed implementation variations.

In addition, DASH also implements a protective proxy to control access to the original sensitive data object on the application server side. This proxy serves as an added layer of protection to prevent unauthorized users on the blockchain to change the states of a patient contract. DASH can also use a proxy to perform permission checking prior to forwarding the change request to the real data object’s contract methods.

**Consequences.** A proxy object can perform simple housekeeping or auditing tasks by storing some commonly used metadata in its internal states without having to perform heavy operations such as decrypting a file. It follows the same interface as the real object and can execute the original heavyweight function implementations on demand. It can also hide information about the real object from the client to protect patient data privacy.

Since *Proxy* introduces another layer of indirection, however, it could cause disparate behavior when the real object is accessed directly by some client while the surrogate is accessed by others. In addition, contracts in Solidity are instantiated by other contracts or transactions, meaning that the proxy can only reference the real contract with its address on the blockchain, which defeats the purpose of using a protection proxy to hide the real

![Diagram](image-url)
object from the client. Nevertheless, both remote and protective proxies can be used on the server side of the application that accesses these contracts.

4.2.4 Applying Publisher-Subscriber to Address Scalability Challenge.

Context Any blockchain-based healthcare system with the need to track relevant health changes across large patient populations.

Problem How to handle large scales of communications occurred in the blockchain-based system.

Motivation. The Ethereum blockchain maintains a public record of contract creations and operation executions along with regular cryptocurrency transactions. The availability of information makes blockchain a more autonomous approach to improve the coordination of patient care across different teams (e.g., physicians, pharmacists, insurance agents, etc) who would normally communicate through various channels with a lot of manual effort, such as through telephoning or faxing. Due to the continually growing list of records on the blockchain, however, directly capturing any specific health-related topic of occurred events can incur significant transaction receipt lookups and topic filtering, which requires non-trivial computation and may result in delayed responses.

A good model of a blockchain-based health app should therefore support coordinated care. For instance, care should be seamless from the point when a patient self-reports of illness through a health DApp interface to the point when the patient receives prescriptions created by their primary care provider for their reported symptoms. In addition, the model should relay the relevant health information (such as clinical reports and follow-up procedure to and from the associated care provider offices) in a timely manner.

Solution Applying the Publisher-Subscriber pattern can assist blockchain-based healthcare apps in broadcasting the information only to providers that subscribe to events relating to patients they care for. It relieves the cumbersomeness of blindly filtering out which care provider should be notified of which patient's what activities as the large volumes of events take place. It also helps maintain a healthcare interoperable environment because providers across various organizations can participate. To avoid computation overhead on the blockchain, the actual processing of patient activities data can be done off-chain by the DApp server.

Figure 8 shows the structure of the Publisher-Subscriber pattern applied in DASH. Specifically, when the publisher sends an update, its subscribers only need to do a simple update to an internal state variable that records the publisher's address, which the DApp server actively monitors for change. When a change occurs, the responsibility for the heavy computation tasks of content filtering (e.g., retrieving the change activity from the publisher using the address) is delegated to the DApp server from the blockchain. The DApp server is context-

![Fig. 8. Applying the Publisher-Subscriber Pattern to DASH](image-url)
aware at this point because each subscriber has an associated contract address accessible by the server. The server can then perform content filtering based on subscribed topics and update the appropriate subscribers' contract states as needed.

In DASH, a patient's self-reports of sickness symptoms are subscribed to by their primary care provider and pharmacist. When this activity occurs, the patient contract sends a notification with the topic "Symptom Report" to the subscribed provider and pharmacist contracts by updating one of their state fields. When DASH server captures this event, it finds subscribers to the "Symptom Report" topic to be only the provider, so it in turn updates just the provider's "messages" state to send an internal message regarding this self-report.

DASH also applies the Publisher-Subscriber pattern directly to smart contracts instead of adding the extra step to delegate the server to perform content filtering because of the small topic and patient population size. In a production healthcare deployment with millions of patients and providers, it would be more cost-effective to pass the heavy computation to the server.

**Consequences.** The Publisher-Subscriber pattern is useful when a state change in one contract must be reflected in another contract without keeping the contracts tightly coupled. It also makes topic/content filtering more manageable when subscription relations are clearly defined in each participant's contract state. When DASH is enhanced with new topics or additional subscribers, only minimal changes to the contract states are needed, rather than major implementation changes, which avoids having to create new versions of contracts. In addition, the Publisher-Subscriber pattern enables communication across participants from various organizations, making it feasible to achieve healthcare interoperability.

The limitations of applying the Publisher-Subscriber pattern on the blockchain include (1) delays in updates received by subscribers due to the extra step of validation required by the blockchain infrastructure, (2) high computational costs associated with filtering very fine-grained topic subscriptions on-chain, and (3) high executional costs associated with notifying events. With a much faster mining process on the Ethereum blockchain compared to Bitcoin, each block can be mined and added to the blockchain roughly every 12 seconds in Ethereum.

The order of which transactions are to be executed is determined by the miners based on the transaction fees paid by the senders and how long transactions have been in the transaction pool. Transactions with the highest priority will be executed first and added to the blockchain sooner, which could cause delays in publishers sending out messages and subscribers receiving messages of their interest. Every computation occurred on the blockchain will be charged some amount of "gas", so the more intensive the computations are, the more costs to realize them.

If subscribers want to receive very fine-grained message topics, the amount of computation for filtering out the messages sent out by publishers may cost an exorbitant amount, resulting in either failure to publish messages due to gas shortage or unacceptable implementation costs. A workaround could be to have broader topics with fewer filter requirements on-chain and handle more detailed message filtering off the blockchain.

5. RELATED WORK

Although relatively few papers focus on realizing software patterns in blockchains, many papers relate to healthcare blockchain solutions and applying software design principles in this space. This section gives an overview of related research on (1) the challenges of applying blockchain-based technology in the healthcare space and innovative implementations of blockchain-based healthcare systems and (2) design principles and recommended practice for blockchain application implementations.

5.1 Challenges of healthcare blockchain and proposed solutions.

Ekblaw et al. [Ekblaw et al. 2016] proposed MedRec as an innovative, working healthcare blockchain implementation for handling EHRs, based on principles of existing blockchains and Ethereum smart contracts. The MedRec system uses database "Gatekeepers" for accessing a node's local database governed by permissions stored on the MedRec blockchain. Peterson et al. [Peterson et al. 2016] presented a healthcare blockchain with a single
centralized source of trust for sharing patient data, introducing "Proof of Interoperability" based on conformance to the FHIR protocol as a means to ensure network consensus.

5.2 Applying software design practice to blockchain.

Porru et al. [Porru et al. 2017] highlighted evident challenges in state-of-the-art blockchain-oriented software development by analyzing open-source software repositories and addressed future directions for developing blockchain-based software, focusing on macro-level design principles such as improving collaboration, integrating effective testing, and evaluations of adopting the most appropriate software architecture. Bartoletti et al. [Bartoletti and Pompianu 2017] surveyed the usage of smart contracts and identified nine common software patterns shared by the studied contracts, e.g., using "oracles" to interface between contracts and external services and creating "Polls" to vote on some question. These patterns summarize the most frequent solutions to handle some repeated scenarios.

6. CONCLUDING REMARKS

Blockchain and its smart contract capabilities provide a platform for creating decentralized apps that have the potential to improve healthcare interoperability. To best leverage this platform that is decentralized and transparent, several design concerns need to be addressed. These concerns include but are not limited to application system evolvability, storage requirements, patient data privacy protection, and application scalability across large number of users. In this paper we expanded on these design concern discussions and attempted to mitigate the challenges by applying existing software design patterns—Abstract Factory, Flyweight, Proxy, and Publisher-Subscriber—in our DApp for Smart Health (DASH) implementation as a case study.

Based on our experience developing the DASH case study we learned the following lessons:

—The public, immutable, and verifiable properties of the blockchain enable a more interoperable environment that is not easily achieved using traditional approaches that mostly rely on a centralized server or data storage. To best leverage these properties, concerns regarding system evolvability, storage costs, sensitive information privacy, and application scalability must be taken into account.

—Each time a smart contract is modified, a new contract object is created on the blockchain. Important design decisions must therefore be made in advance to better take advantage of smart contract support while avoiding the cost and storage overhead caused by contract interface change.

—Combining good software design practices with the unique properties of the blockchain will help create DApp models that are more modular, easier to integrate and maintain, and less susceptible to change.

Our future work will extend the app described in Sections 3 and 4 to delve into the challenges and pinpoint the most practical design process in designing a healthcare blockchain architecture. In addition, we will explore the potential application of other software patterns to handle various challenges, such as security, privacy, dependability, and performance. One approach is to conduct experiments to evaluate the efficacy of applying software patterns to this architecture compared to other alternative designs. We will also investigate extensions of the blockchain from a healthcare perspective, such as creating an alternative health chain that exclusively serves the healthcare sector.

REFERENCES


