P2HR, A PERSONALIZED CONDITION-DRIVEN PERSON HEALTH RECORD

A Thesis

Submitted to the Faculty

of

Purdue University

by

Zachary King

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electric and Computer Engineering

August 2017

Purdue University

Indianapolis, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Zina Ben Miled

Department of Electrical and Computer Engineering

Dr. Brian King

Department of Electrical and Computer Engineering

Dr. Dongsoo Kim

Department of Electrical and Computer Engineering

Approved by:

Dr. Brian King

Head of the Graduate Program

This thesis is dedicated to my family, specifically my parents Brian and Sue King who have supported me through my education.

ACKNOWLEDGMENTS

I would like to acknowledge my thesis advisor Dr. Zina Ben Miled and the other members of my thesis committee Dr. Brian King and Dr. Dongsoo Kim. I would also like to thank Dr. Titius Schleyer and Dr. Latifat Oyekola for their help and guidance. Finally, I would like to recognize the team support of Kyle Haas and the members of the Data Driven Knowledge Discovery and Management lab.

TABLE OF CONTENTS

Page

LIST OF FIGURES vi

LIST OF ALGORITHMS vii

ABSTRACT viii

1 INTRODUCTION 1

1.1 Background 1

1.2 Personal Health Records 2

1.3 Proposed Framework 6

2 SYSTEM DESIGN 8

2.1 Communication Model 11

2.2 Data Model 15

3 SYSTEM IMPLEMENTATION 27

3.1 Data Model 27

3.2 Network Processes 38

3.2.1 Index Server 39

3.2.2 Peer 45

4 CONCLUSION 53

REFERENCES 55

LIST OF FIGURES

Figure Page

1.1 Hypertension 5
2.1 P2HR Architecture 8
2.2 Three Tier Architecture 9
2.3 Hypertension Condition-Based 9
2.4 Hypertension 10
2.5 Communication Model 12
2.6 Sub-Network Initialization 14
2.7 Information Exchange 15
2.8 Example of Collections in the PHR 17
2.9 CDA Components 18
2.10 CDA Observation 19
2.11 CDA Performer 21
2.12 CDA Participant 22
2.13 CDA Product 23
2.14 CDA JSON Format for Chest X-Ray 24
2.15 Two Body Weight Readings 25
3.1 Data Flow 28
3.2 XML Vital Sign 29
3.3 CDA Observation 31
3.4 Data Structure for Vital Sign in Golang 32
3.5 Diabetes Data Structure 35
3.6 Network Processes 38
3.7 Message Data Structure 40
3.8 Node Data Structure 40

LIST OF ALGORITHMS

Algorithm Page

3.1 Converting XML to JSON 29
3.2 Push Event-Based Health Record into MongoDB 33
3.3 Inserting Multiple JSON Files 33
3.4 Extracting Event-based Data from MongoDB 34
3.5 Converting Event-Based Data to Condition-Based 35
3.6 Extracting Condition-Based Data from MongoDB 36
3.7 Updating Condition-Based Data into MongoDB 36
3.8 Inserting Condition-based Heatlh Records into MongoDB 37
3.9 Index Server Initial Receive 41
3.10 Send Function 42
3.11 Receive Function 42
3.12 Sub-network Initialize Request 43
3.13 Sub-network Initialize Response 44
3.14 Server Information Exchange 46
3.15 Check Sub-Network 47
3.16 Peer Response 48
3.17 Response to Sub-Network Initialization 49
3.18 Update Sub-Network 50
3.19 Peer Information Exchange 52

ABSTRACT

King, Zachary. MSECE., Purdue University, August 2017. P2HR, A Personalized Condition-Driven Person Health Record. Major Professor: Zina Ben Miled.

Health IT has recently seen a significant progress with the nationwide migration of several hospitals from legacy patient records to standardized Electronic Health Record (EHR) and the establishment of various Health Information Exchanges that facilitate access to patient health data across multiple networks. While this progress is a major enabler of improved health care services, it is unable to deliver the continuum of the patient's current and historical health data needed by emerging trends in medicine. Fields such as precision and preventive medicine require longitudinal health data in addition to complementary data such as social, demographic and family history.

This thesis introduces a person health record (PHR) which overcomes the above gap through a personalized framework that organizes health data according to the patients disease condition. The proposed personalized person health record (P2HR) represents a departure from the standardized one-size-fits-all model of currently available PHRs. It also relies on a hybrid peer-to-peer model to facilitate patient provider communication. One of the core challenges of the proposed framework is the mapping between the event-based data model used by current EHRs and PHRs and the proposed condition-based data model. Effectively mapping symptoms and measurements to disease conditions is challenging given that each symptom or measurement may be associated with multiple disease conditions. To alleviate these problems the proposed framework allows users and their health care providers to establish the relationships between events and disease conditions on a case-by-case basis. This organization provides both the patient and the provider with a better view of each disease condition and its progression.

1. INTRODUCTION

1.1 Background

As patient-provider interactions become increasingly specialized through advances in the medical field, the expectation of a centralized medical record becomes increasingly unattainable. Settling for a scattered record, however, may lead to gaps in the patient's medical history rendering a holistic approach to medical treatment also unattainable. Evidence for a preferred holistic approach or at least an approach that is based on a wide health information spectrum can be found in several cases. In [1], it was found that a number of coronary artery bypass patients develop depression, placing these patients at the intersection of two diverse fields of medicine by todays practice. Similarly, the Alzheimer Association has found evidence that links Type 2 diabetes to Alzheimer disease and the Center for Disease Control established that Obesity can lead to countless number of health problems [2].

The above examples suggest that the involvement of both the patient and the provider in maintaining personal health records is the only viable and practical solution for efficient health care delivery. Indeed, person health records (PHR) can provide an efficient means through which patients can interact with health care providers in various fields as well as share health information with them. This information can be

- Extracted from Electronic Health Record (EHR) systems, from different health institutions and different health networks. EHRs are becoming ubiquitous in hospitals and other medical services facilities. Recent studies indicate that 96% of non-Federal acute care hospitals have adopted certified EHRs systems by 2015 [3]. In addition, initiatives such as Blue Button+ [4] and FHIR [5] allow patients to electronically access their own health information from various health providers such as health plans, pharmacies and hospitals.

- Augmented through patient self-captured information from applications and devices (e.g., fitness, nutrition, and health monitoring devices). In recent years, we have seen an emergence of home health care devices including step counters, blood glucose monitors and heart rate monitors. Such devices can be aligned with the proposed P2HR system to enable patients to manage the devices and store the data that these devices generate.

- Shared with health providers when and as needed. This is an important step that completes the feedback loop in patient managed health care. Recently, the office of the National Coordinator for Health Information Technology conducted a pilot through the National Association for Trusted Exchange (NATE) [6] to demonstrate the potential and current gaps in the digital communication between the patient and the health care provider. This experiment highlighted the need for the development of efficient bidirectional communication mechanisms between PHRs and EHRs.

As defined in [7], a PHR is an application that allows people to access and manage their lifelong health information and make this information available to health care providers as needed. Our vision of a PHR system takes into account not only the need for patient managed health record, but also the future focus on precision medicine and personalized preventive medicine. The proposed Personalized Person Health Record (P2HR) aligns with these future trends since each health record is personalized to the context and disease conditions of the patient.

1.2 Personal Health Records

Currently available PHR systems support one or more of the following functionalities [8]:

- Information Collection: This functionality is concerned with health data storage. The data of interest can be entered into the system directly by the user or extracted from an EHR using a patient portal.

- Information Management: This functionality is related to the update of the health information by the patient. There are currently many fitness mobile applications and devices that can be connected to a PHR where the generated data can be stored over time. This functionality can also be extended to storing notes of daily diet, workout details, or a wellness diary.

- Information Sharing and Exchange: This functionality covers patient-provider exchanges of health information. It allows users to directly connect with their health care providers and receive up-to-date health data.

Examples of PHR systems include the Dossia Health Manager [9] which places an emphasis on information sharing and exchange through its real-time news feed. MeTree [10] and Health Heritage [11, 12] both focus on including family health history. The aim of MeTree is to aid health care providers with decision making while that of Health Heritage is to match family history with current research in order to identify effective preventive strategies. HealthVault [13] focuses on the ease of access to information and self-management through interoperability support with numerous external applications including health and fitness devices. It also permits the aggregation of multiple family member's accounts. Google Health [14, 15], which has been discontinued in 2011, emphasized an efficient environment for data collection and storage. One of the most functional systems available is PatientsLikeMe [16]. It is dedicated to visualtizing data based on condition as well as allowing patient-to-patient communication. PatientsLikeMe is technically not a PHR, but a social network dedicated to connecting patients with similar disease conditions in order to share their experiences.

Supporting a holistic approach to medical treatment by using a PHR has several advantages, however it also has several challenges. One of its major challenges is the potential for mainstream adoption. In [17], the low adoption rate was found to be attributed in part to the lack of personalization in currently available one-size-fits-all PHR systems. Mainstream adoption is critical because PHR systems inherently

place the burden of the management and ownership of the health information upon the patient. Therefore, in order for this type of system to be accessible to all, it must be flexible and support the interoperability with many EHRs.

In order to facilitate the interoperability among health information systems, the U.S. National Library of Medicine (NLM) created the Unified Medical Language System (UMLS) which integrates several standards including HL7, LOINC, and RxNORM into a unified Metathesaurus [18, 19]. UMLS includes two additional components. The first is a semantic network which categorizes the entities in the Metathesaurus and establishes the relationships among them and the second component is a specialist lexicon which can correctly interpret user-entry errors [18, 19]. This overarching unified metadata standard is consistent with the current unification trend in the health sector. A different standard, the Consolidated Clinical Document Architecture, which aims at unifying the data model used by various health information systems is also being promoted [20]. Both of these standards are essential to the interoperability of health information systems. From the perspective of the proposed P2HR, the use of these standards by EHRs entail that one unique interface is sufficient to support the exchanges between the proposed P2HR and the multitude of available EHRs.

In order to achieve mainstream adoption, it is imperative that the PHR makes the patient's health record understandable to an average user with potentially limited medical knowledge. Currently most patient records retrieved from EHRs use the Clinical Document Architecture (CDA) as the standard for exchanging health records. CDA was developed by the Health Level Seven International (HL7) and is currently the ANSI approved national standard format for exchanging clinical documents [21]. Specifically, CDA is based on the HL7 Reference Information Model (RIM). An example of the CDA format is shown in Figure 1.1.

CDA does not address how the documents are exchanged even though HL7 includes the HL7 messaging component which is an interoperable specification for exchanging health records [5]. HL7 is currently developing a new specification Fast

```
<code
   code="8480-6"
   codeSystem="2.16.840.1.113883"
   codeSystemName="LOINC"
   displayName="Intravascular Systolic"/>
<text><reference
   value="#vit6"/></text>
<statusCode
   code="completed"/>
<effectiveTime
   value="20120806"/>
<value
   xsi:type="PQ"
   value="145"
   unit="mm[Hg}"/>
<interpretationCode
   code="N"
   codeSystem="2.16.840.1.113883.5.83"/>
```

Fig. 1.1. Hypertension

Healthcare Interoperability Resources (FHIR). FHIR will most likely replace HL7 messaging. The goal of FHIR is to "simplify and accelerate HL7 adoption by being easily consumable but robust, and by using open Internet standards where possible" [5]. FHIR uses a RESTful protocol [5]. One advantage that FHIR has over its predecessor HL7 messaging v3 is its applicability to mobile device. Moreover, compared to CDA, it is more modular and uses JSON objects making it easier for the data to be accessed from any programming language.

Blue Button+ proceeded FHIR. It is an online patient portal that gives patients access to their health records [4]. It returns health records in a machine readable standard XML file [4]. This XML file is partitioned into multiple sections including vital signs, encounters, immunizations, medications, etc. This information is organized in exactly the same way as in the original EHR system. Blue Button+ stores the information using the CDA standard. Figure 1.1 [4] is an example of an extracted Blue Button file corresponding to the vital signs partition. The information in the XML file references multiple coding standards including SNOMED-CT, RxNorm, and LOINC.

In a survey, it was found that 87% of non Veteran Affairs health care providers found the information that the veterans obtained using Blue Button was helpful or very helpful when treating their patient [22].

Blue Button, the previous version of Blue Button+, is used by Health vault to enable users to upload health records that are in CDA format into the PHR. However, FHIR is not currently used by any PHR as it is still under development. One of the main gaps in the currently available standards is that they do not allow for user generated information to be exchanged. Future extensions to these standards are likely to address this gap.

1.3 Proposed Framework

The above example PHRs have attempted to gain mainstream adoption through content delivery rather than functionality. Indeed, each differentiates itself by the content it presents instead of how it is presented. Furthermore, all currently available PHRs are event-driven (e.g., encounters, lab results, medications, etc.). The proposed framework organizes health information based on disease conditions (e.g., hypertension, diabetes, asthma, etc.). We believe that this new approach facilitates personalization and encourages mainstream adoption. For example, if a patient is prescribed medication for arthritis and high cholesterol by a general practitioner, these medical conditions are distinct and this distinction should be reflected in the presentation of the information as it relates to each condition. Under current event-driven PHRs, this information is collectively associated with a single event (i.e., encounter with health provider) rather than being differentiated according to the respective conditions. Organizing records based on condition rather than events can make health information more readily accessible to both the patient and the health care providers including pharmacists, physicians, nurses, as well as home health care specialists.

Chapter 2 of this thesis describes the design of the proposed PHR. Chapter 3 presents the implementation of the system and Chapter 4 summarizes the main contributions of this thesis and presents direction for future work.

2. SYSTEM DESIGN

This chapter introduces the key components as well as the communication model and data model of the proposed system. Figure 2.1 shows the basic architecture of the proposed system (P2HR).

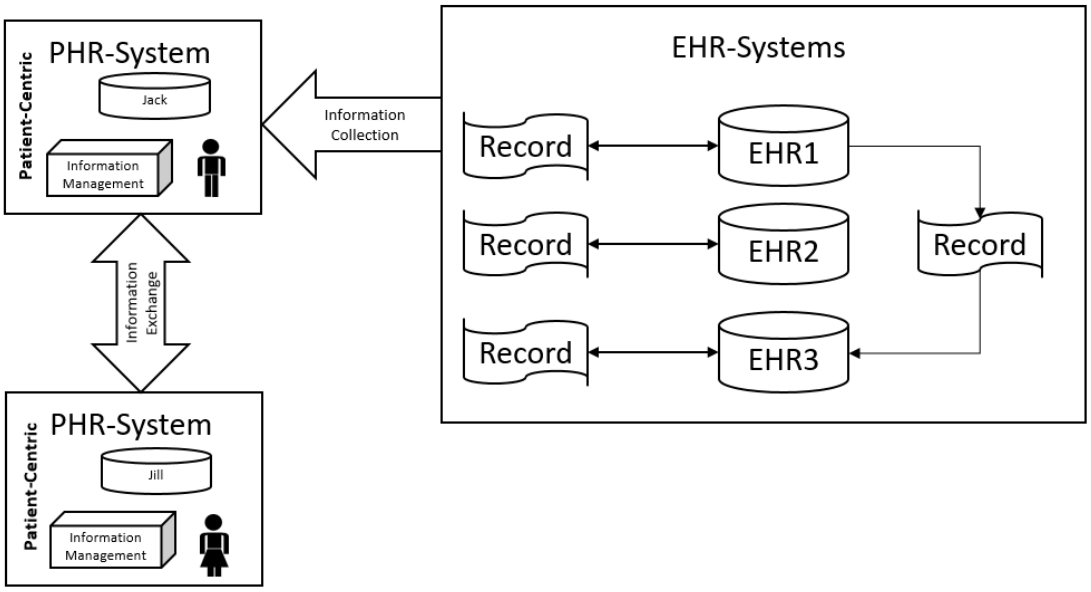

Fig. 2.1. P2HR Architecture

P2HR has a three-tier software architecture [23] (as illustrated in Figure 2.2). The interface of the system allows the users to access their health information as well as contain functionalities found in most social networks. These functionalities include inserting health records, connecting with other users, posting and sharing content, and joining groups. The system combines the functionalities of both traditional PHRs and common social networks in order to promote health awareness and engage users.

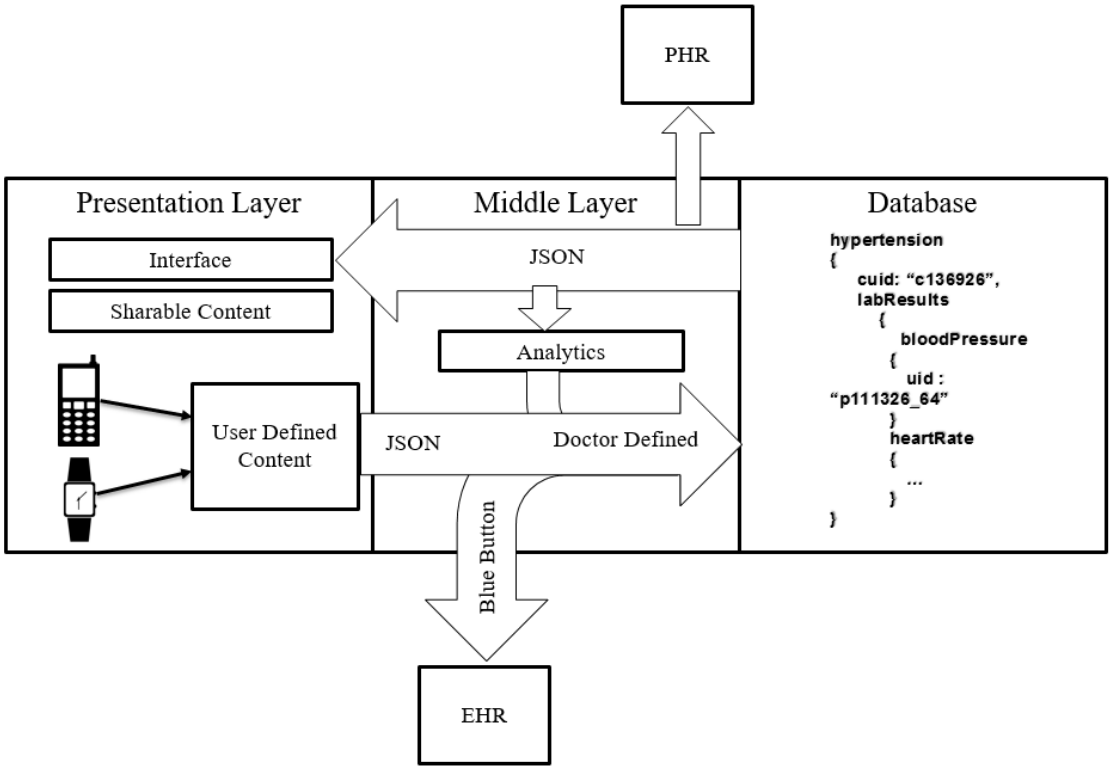

Fig. 2.2. Three Tier Architecture

```
hypertension
{
   cuid: "c136926",
   labResults
      {
         bloodPressure
         {
           uid : "p111326_64"
         }
         heartRate
         {
           ...
         }
}
```

Fig. 2.3. Hypertension Condition-Based

```
<code
   code="8480-6"
   codeSystem="2.16.840.1.113883"
   codeSystemName="LOINC"
   displayName="Intravascular Systolic"/>
<text><reference
   value="#vit6"/></text>
<statusCode
   code="completed"/>
<effectiveTime
   value="20120806"/>
<value
   xsi:type="PQ"
   value="145"
   unit="mm[Hg}"/>
<interpretationCode
   code="N"
   codeSystem="2.16.840.1.113883.5.83"/>
```

Fig. 2.4. Hypertension

The middle tier manages information flow between the presentation tier and the data management tier. This tier handles the mapping from the event-based XML files of the EHRs to the condition-based JSON files. In addition, JSON files retreived from other health care providers or external devices are also mapped into the condition-based data model as shown in Figure 2.2. Currently, the mapping is done manually by the user. Future work will investigate automating this process. The mapping begins when the user adds a new condition. The interface will then prompt the user to establish the relationships between the extracted files and the given condition. The middle tier is also responsible for connecting a given person's P2HR to the P2HRs of other individuals belonging to the person's sub-network. The sub-network is defined by the user and identifies the other peers from the general network that the user would like to communicate with.

The third tier is the data management tier. This tier is responsible for storing and retrieving information from the back-end NoSQL database. Each document in the database includes information related to a given disease condition. An exam-

ple of a hypertension document is shown in Figure 2.3 which is converted from the hypertension xml file shown in Figure 2.4.

Information is passed between the three tiers (i.e., Presentation Tier, Middle Tier and Data Management Tier) by using a JSON string format. The P2HR data model presented several challenges related to data extraction, mapping and management. These challenges and proposed solutions are discussed in the next sections.

2.1 Communication Model

The proposed framework is based on a distributed architecture which consists of an index server and several nodes. This architecture is shown in Figure 2.5. The index server collects information from the peers, such as IP address, user identifiers and user connections. It then distributes this information to other nodes in the network as needed. The index server has two main functions. It acts as a lookup table and controls access authorization. The peers also have two main functions. The first is to store and distribute data and the second function is to send requests to the index server.

There are two widely used communication models: the centralized client-server model and the distributed peer-to-peer (P2P) model. In the client-server model, the server handles requests from all the clients placing the burden of computing power and data management on the server [24]. This centralized access and data management is a key advantage in the client-server model. The disadvantages include a single point of failure and the increasing cost associated with processing a large number of requests from the clients.

The distributed communication model consists of nodes that cooperate to service each other's requests. Typically, all the nodes in a peer-to-peer (P2P) network have the same privileges and roles. In [25], the authors argue that the major advantages of P2P networks are improved scalability, low cost infrastructure and improved resource

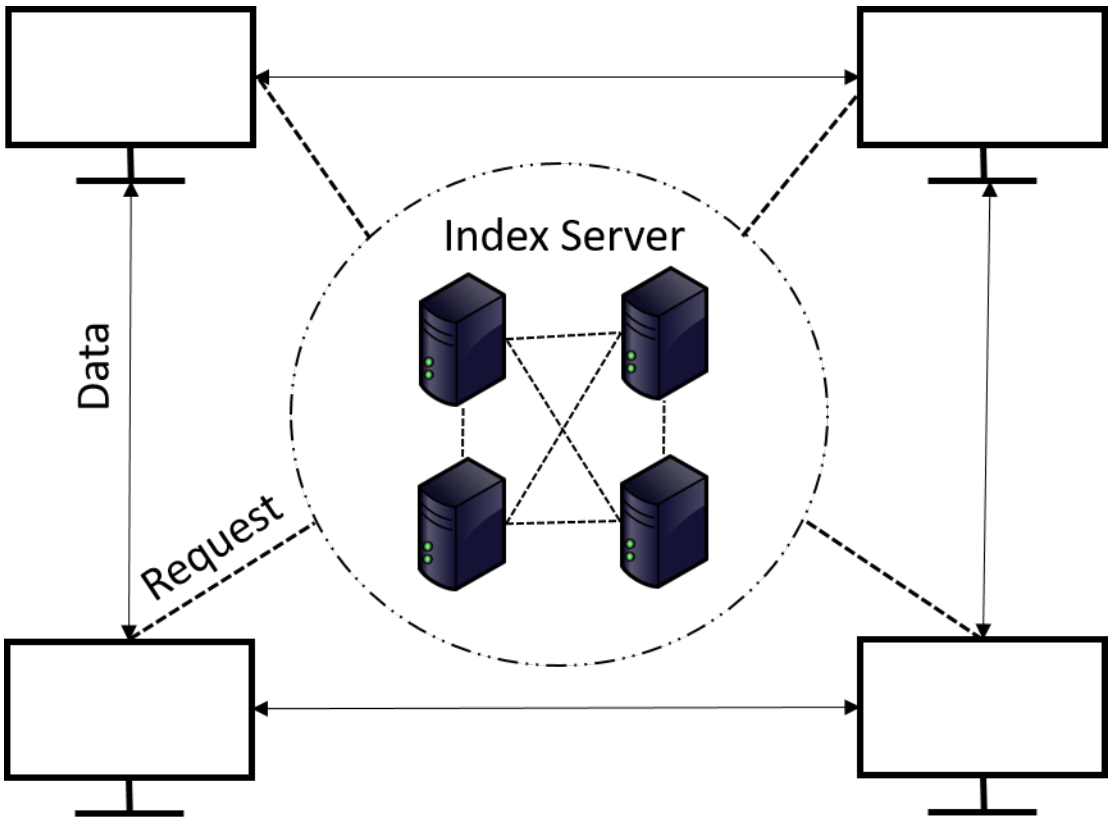

Fig. 2.5. Communication Model

aggregation. There are three variations of P2P networks: Unstructured Network, Structured Network, and Hybrid Network.

Unstructured P2P networks do not have a central entity in the network and all peers are the same. In this type of network, data will travel through other peers to reach its destination. Examples of unstructured P2P networks include the file sharing applications Gnutella [26] and FreeNet [27]. The advantage of this network is that its scalability is not limited by a central management node. However, as stated in [25], this increased scalability is at the expense of a reduced level of information validity since any node is allowed to participate in the network.

Structured P2P networks organize the nodes in a tree structure where leaf nodes have to rely on their parent nodes in the tree to communicate with other nodes. Bittorrent [28] is an example of a structured P2P network. One advantage of structured

P2P networks over unstructured networks is the ability to trace messages back to their original sender since the nodes are organized according to a specific tree. The disadvantage to this network; is the same as the unstructured network, that being a reduced level of information validity.

The Hybrid P2P network is the third type of distributed communication model. This model is a combination of the client-server and the P2P architectures. In the Hybrid P2P model an index server is used as a centralized lookup table for the nodes in the network. Napster [29] is an example of a hybrid P2P network. It uses a centralized cluster of index servers that maintain indexes for the information in the network. Each peer is assigned one of the index servers in the cluster. When a given peer wants to exchange data with another peer, it will request the IP address of the target peer from the assigned index server. The two peers can then exchange data directly [29].

The hybrid P2P network offers additional security features compared to the previous two types of P2P networks. Indeed, the index server can be used to authenticate each peer. However, this advantage comes at the expense of introducing a single point of failure in the network which may be mitigated through a cluster of index servers as used in Napster.

The proposed system is based on the hybrid P2P communication model and has two core processes: sub-network initialization (Figure 2.6) and information exchange (Figure 2.7). To create a sub-network, each user has to connect with a target peer. The sub-network initialization starts with peer A finding peer B using some unique identifier (i.e. email address, name, etc.) then requesting B to join his/her sub-network. In order to execute this process, peer A sends a request message to the index server. This request message contains peer B's unique identifier. The index server uses this identifier to lookup peer B in its local database. The index server will also issue an outstanding sub-network request to peer B which can be either accepted or declined. If the request is accepted by peer B, the index server will update each of the peer's sub-network and send the updated sub-network information to both peers.

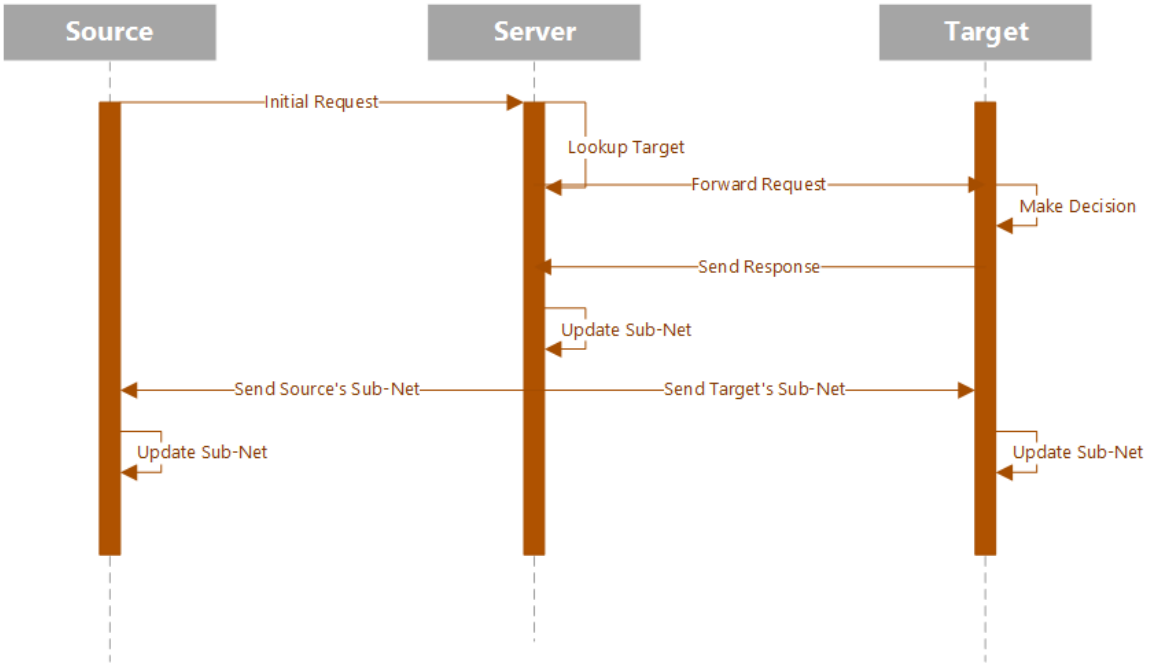

Fig. 2.6. Sub-Network Initialization

The second process is the information exchange process and consists of sharing and exchanging health information between peers. The process starts with peer A requesting communication with peer B through the index server following the same steps used in the previous process. When the index server receive this communication request, it will first use the information in the message to check whether or not peer B is already a member of peer A‘s sub-network and if peer B is active. If both of these conditions are true, the index server notifies peer B of the request and sends the IP address of peer B to peer A. In the future, this step can be used to incorporate security features ensuring that each peer is communicating with trusted and known peers. Once peer A has the IP address of peer B, it can establish a TCP connection and exchange data with the target peer.

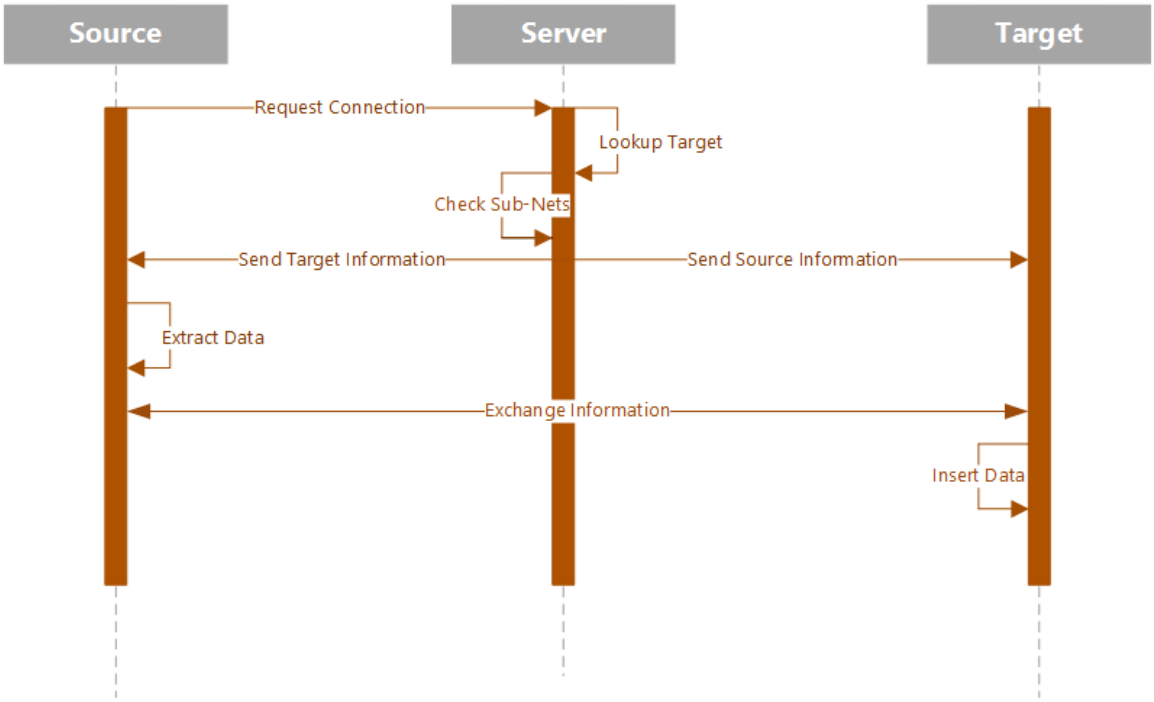

Fig. 2.7. Information Exchange

2.2 Data Model

The data model defines the structure of the data being exchanged and how this data is stored. Traditionally, data is stored in relational databases for most commonly used information systems. These relational databases consist of a collection of tables. A table in a relational database is made up of rows and columns where a row can be seen as an object (i.e., person, medicine, etc.) and a column is an attribute of the object (i.e., height, dosage, etc.). Relational databases are suitable for structured data.

The emergence of semi-structured and unstructured data led to a new data model: NoSQL. NoSQL databases can be classified into different categories: document-based, key-value, graph, tabular, etc. [30]. In this thesis, a document-based NoSQL database is used. Data is stored in collections of documents. A collection is the counterpart

of a table and a document is the counterpart of a row in a relational database. NoSQL databases led to improved scalability and improved performance compared to relational databases because they can accommodate unstructured data and avoid the strict consistency rules imposed by relational databases [31].

The document-based database used is MongoDB [32]. MongoDB differs considerably from a relational database. For instance, there are no primary or foreign keys. Instead, documents are connected using an objectID which is a unique identifier to a document that is populated whenever a document is inserted into the database.

The data management tier in the peer application contains databases for each person represented in the user's sub-network (as shown in Figure 2.8). Each database is identified by the user's uid and is split into two sections event-based and condition-based data. The documents that are event-based correspond to a single event and are stored in the collection associated with the event type. The left hand side of Figure 2.8 shows the collections Encounters and Vital Signs. For the condition-based data, there is a single collection for each person. Whenever information is exchanged between peers, the recipient will store the senders' data locally in his/her own database. The documents in the collection will each refer to the sender's disease condition. The condition document contains multiple arrays, one for each event type (i.e. Encounter, Vital Sign, etc.). The arrays are tagged with the standard code that connects it to the instances related to the condition. These codes come from the event-based data which is defined when extracted from the EHR. In traditional EHRs and PHRs, the event-based data is stored in a relational database and is structured according to each event (e.g., encounter, lab result, medication) where each event is associated with a table and the rows in the table are instances of the event. In the proposed system, the event are also stored based on the event type as in the traditional PHR. However, the proposed system allows for an event to be referenced by multiple conditions.

The health records in the proposed PHR are initially stored in the event-based collection until the patient or doctor associates the specific event with a condition. This is accomplished through an interface that allows the patient or doctor to drag and

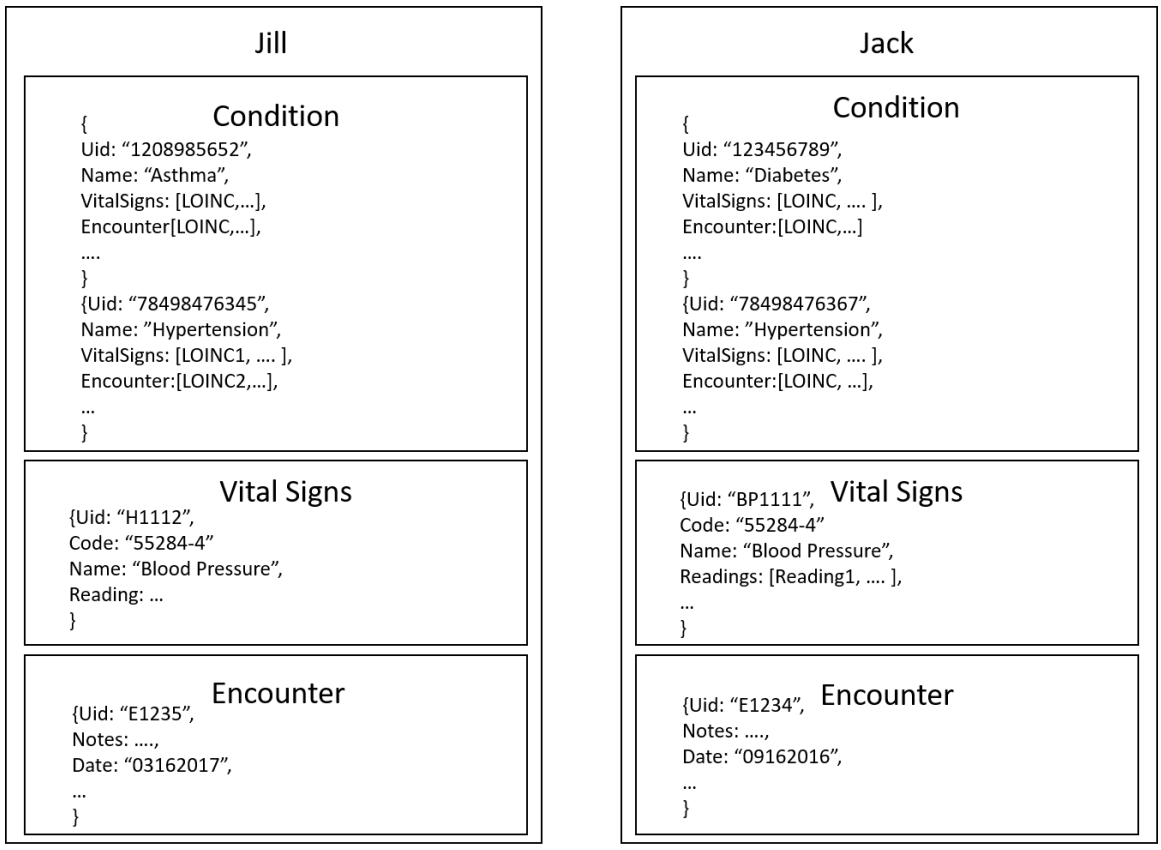

Fig. 2.8. Example of Collections in the PHR

drop events to the corresponding condition. Future work will investigate automating this step.

The records that the user uploads are extracted from an EHR. When a health record is retrieved from an EHR, it is split up by event (e.g., Encounter, Procedure, Lab Result, etc.). Currently, there are 15 components represented in the system where each component corresponds to an equivalent component in the Blue Button XML file (Figure 2.9). For each of these components a data structure is created and populated with the extracted data which is obtained by parsing the XML file. The structure of a Blue Button document is in the CDA format as shown in Figure 2.9.

```
<Clinical Document>
 <Component>
  <title>ALLERGIES, ADVERSE REACTIONS, ALERTS</title>
</Component>
 <Component>
  <title>Encounters</title>
</Component>
 <Component>
  <title>IMMUNIZATIONS</title>
</Component>
 <Component>
  <title>Medications</title>
</Component>
 <Component>
  <title>Care Plan</title>
 </Component>
 <Component>
  <title>Discharge Medications</title>
 </Component>
 <Component>
  <title>Reason for Referral</title>
 </Component>
 <Component>
  <title>Problem List</title>
 </Component>
 <Component>
  <title>Procedures</title>
 </Component>
 <Component>
  <title>Functional and Cognitive Status</title>
 </Component>
 <Component>
  <title>Lab Results</title>
 </Component>
 <Component>
  <title>Social History</title>
 </Component>
 <Component>
  <title>Vital Signs</title>
 </Component>
 <Component>
  <title>Discharge Instructions</title>
 </Component>
</Clinical Document>
```

Fig. 2.9. CDA Components

The CDA document can be split up into two main sections: the header and the body. The header of CDA document is the first of the 15 components and includes the information surrounding the patients, the encounters, the providers, and the authenticity of the document [33]. All of this data is static and will rarely change. The information in the header will not have any relationship with any condition. Therefore, it will not be represented in the condition-based collection of the proposed system. This information is important when exchanging health data with EHRs as these EHRs use it to distinguish between patients. For this reason, the header maintains its format in the associated collection of the proposed system.

The body of the CDA document is composed of 14 components. The structure of the CDA document and its components are shown in Figure 2.9. Each component in Figure 2.9 is also split up into the same two sections: the header and the body.

```
<procedure
 classCode="PROC"
 moodCode="EVN">
 <templateId
  root="2.16.840.1.113883.10.20.22.4.14"/>
 <id
  extension="123456789"
  root="2.16.840.1.113883.19"/>
 <code
  code="168731009"
  codeSystem="2.16.840.1.113883.6.96"
  displayName="Chest X-Ray"
  codeSystemName="SNOMED-CT">
  <originalText>
   <reference value="#Proc2"/>
  </originalText>
 </code>
 <statusCode
  code="completed"/>
 <effectiveTime
  value="20120807"/>
 <priorityCode
  code="CR"
  codeSystem="2.16.840.1.113883.5.7"
  codeSystemName="ActPriority"
  displayName="Callback results"/>
 <methodCode
  nullFlavor="UNK"/>
 <targetSiteCode
  code="82094008"
  codeSystem="2.16.840.1.113883.6.96"
  codeSystemName="SNOMED CT"
  displayName="Lower Respiratory Tract Structure"/>
```

Fig. 2.10. CDA Observation

The header stores the information concerning the respective component. It mainly includes the standard coding system that is used to reference the component. The body of the component stores the values related to an event as well as the information surrounding the event. The structure of an event is shown in Figure 2.10. The event contains the coding standard used to identify the event. Figure 2.10 is for a Chest X Ray and the standard used, in this case, is SNOMED-CT which is stored in the "code" element. This portion of the body will be called the observation and is

included in all components. For the procedure component it is called the procedure, but in most other components it is labeled as observation. The observation will also store measurements if necessary. For instance the observation portion for Body Weight will contain a weight and the unit that represents the weight (pounds, grams, stones, etc.). The observation portion will also contain more information depending on the component. Figure 2.11 shows a portion from the Procedure component which includes information about the performer or the doctor that performed the procedure. Figure 2.12 shows another portion from the Procedure component which contains information about the health care provider. The rest of the components have similar structure and include the previously mentioned segments or other segments. For instance, some observations include an additional segment called Product. This segment stores information on a given product used by the patient and it is related to the Medications component. An example of this segment is shown in Figure 2.13. All of the example CDA documents are from the Blue Buttons implementation guide [4].

Inserting the CDA documents into the MongoDB database begins by converting the documents to JSON. The JSON format of the chest x-ray observation is shown in Figure 2.16. Each observation will be in an array, where each member of the array will be translated to a document. Once the data is converted to JSON, a collection is constructed for each component and each observation becomes a document in the corresponding collection. The structure of the observation will remain the same in the database. The headers of each component are uniform across all patients as they are used to distinguish between components within the document. For this reason the headers can be discarded as the values can be added to an XML file when the health records are exchanged.

The condition-based collection of the data management tier is structured differently from the event-based collections. A user's application may store multiple databases, one for each person (e.g., the individual, a parent, or a child) each of these databases contain the condition-based collection of represented person as well as the event-based data that is represented in the condition-based collection.

```
<performer>
 <assignedEntity>
  <id
   root="2.16.840.1.113883.19.5“
   extension="1234"/>
  <addr>
   <streetAddressLine>1002 Healthcare Dr
   </streetAddressLine>
    <city>Portland</city>
    <state>OR</state>
    <postalCode>97266</postalCode>
    <country>US</country>
  </addr>
  <telecom
   use="WP"
   value="(555)555-555-1234"/>
  <representedOrganization>
   <id root="2.16.840.1.113883.19.5"/>
   <name>Community Health and Hospitals</name>
   <telecom nullFlavor="UNK"/>
   <addr nullFlavor="UNK"/>
  </representedOrganization>
 </assignedEntity>
</performer>
```

Fig. 2.11. CDA Performer

Each user (i.e., person) is associated with a unique identifier (*uid*). Furthermore, each document in a condition-based collection is specific to one disease condition. It is referenced by a unique identifier that corresponds to the document type (*cuid*). The combination of *uid* and *cuid* are used by other collections and documents to reference any document. For instance the *uid* identifier can be used to connect two family members. The structure of a document in P2HR contains categories similar to those found in Blue Button+ (e.g., lab results, medications, vital signs).

Instead of storing the actual data in the condition-based collection of the database, a different approach is used. The condition-based data is populated with references to the corresponding events that are located in the event-based collections of the database. This is shown in Figure 2.10. The process uses the corresponding code to reference the events that are related to the condition. The coding standard used

```
<participant
 typeCode="LOC">
 <participantRole
  classCode="SDLOC">
 <templateId
  root="2.16.840.1.113883.10.20.22.4.32"/>
 <code
  code="1160-1"
  codeSystem="2.16.840.1.113883.6.259"
  codeSystemName="HealthcareServiceLocation"
  displayName="Urgent Care Center"/>
 <addr>
  <streetAddressLine>1002 Healthcare Dr</streetAddressLine>
  <city>Portland</city>
  <state>OR</state>
  <postalCode>97266</postalCode>
  <country>US</country>
 </addr>
 <telecom
  nullFlavor="UNK"/>
 <playingEntity
  classCode="PLC">
 <name>Community Health and Hospitals</name>
 </playingEntity>
 </participantRole>
</participant>
```

Fig. 2.12. CDA Participant

depends on the component as each component may use a different standard. For instance, the medications component uses RxNORM. The coding system used by the component can be found in the code element of the original CDA document from Figure 2.14 under the key codeSystemName. This approach is possible because each code is unique and will be present in all instances of an event. There are three main reason why the data is only stored in the event-based collections.

- The event may be related to several conditions: For example, a patient's vital signs are associated with nearly all conditions. In order to avoid replication by all conditions, the event is stored in a single location and then referenced.

```
<product>
 <manufacturedProduct
  classCode="MANU">
 <templateId
  root="2.16.840.1.113883.10.20.22.4.23"/>
 <id
  root="2a620155-9d11-439e-92b3-5d9815ff4ee8"/>
 <manufacturedMaterial>
  <code
   code="573621"
   codeSystem="2.16.840.1.113883.6.88"
   displayName="Albuterol 0.09 MG/ACTUAT inhalant solution">
   <originalText><reference value="#Med1"/></originalText>
   <translation
    code="573621"
    displayName="Albuterol 0.09 MG/ACTUAT inhalant solution"
    codeSystem="2.16.840.1.113883.6.88"
    codeSystemName="RxNorm"/>
  </code>
  </manufacturedMaterial>
  <manufacturerOrganization>
   <name>Medication Factory Inc.</name>
 </manufacturerOrganization>
 </manufacturedProduct>
</product>
```

Fig. 2.13. CDA Product

- The event may not relate to any conditions: For example, personal information like phone number, address or name are not related to a condition.

- Reduces the complexity of the translation between the two different data models: Since the condition-based data model is unique to the proposed system, every time a user exchanges records with an EHR the record will have to be converted back to the original event-based data model. The proposed cross-referencing mechanism avoids the need for a translation during each exchange.

The proposed data model is also efficient because it simplifies the queries used to retrieve relevant information from P2HR. For example, for a patient with diabetes the patient's body weight will be associated with the diabetes disease condition. This reading would be collected many times throughout the patient's lifetime and a new

```
{
 "entry": {
  "-typeCode": "DRIV",
  “observation": {
   "-moodCode": "EVN",
   "-classCode": "PROC",
   "templateId": { "-root": "2.16.840.1.113883.10.20.22.4.14" },
   "id": {
    "-root": "2.16.840.1.113883.19",
    "-extension": "123456789"
   },
   "code": {
    "-codeSystemName": "SNOMED-CT",
    "-displayName": "Chest X-Ray",
    "-codeSystem": "2.16.840.1.113883.6.96",
    "-code": "168731009",
    "originalText": {
     "reference": { "-value": "#Proc2" }
    }
   },
   "statusCode": { "-code": "completed" },
   "effectiveTime": { "-value": "20120807" },
   "priorityCode": {
    "-codeSystemName": "ActPriority",
    "-displayName": "Callback results",
    "-codeSystem": "2.16.840.1.113883.5.7",
    "-code": "CR"
   },
   "methodCode": { "-nullFlavor": "UNK" },
   "targetSiteCode": {
    "-codeSystemName": "SNOMED CT",
    "-displayName": "Lower Respiratory Tract Structure",
    "-codeSystem": "2.16.840.1.113883.6.96",
    "-code": "82094008"
   },
   "performer": {...},
   "participant": {...}
   }
  }
 }
}
```


Fig. 2.14. CDA JSON Format for Chest X-Ray

document is created in the Vital Signs collection for each reading (Figure 2.15). Each body weight reading will have a different objectID. Instead of storing all of these objectIDs in the condition-based collection, the system only stores the code for body

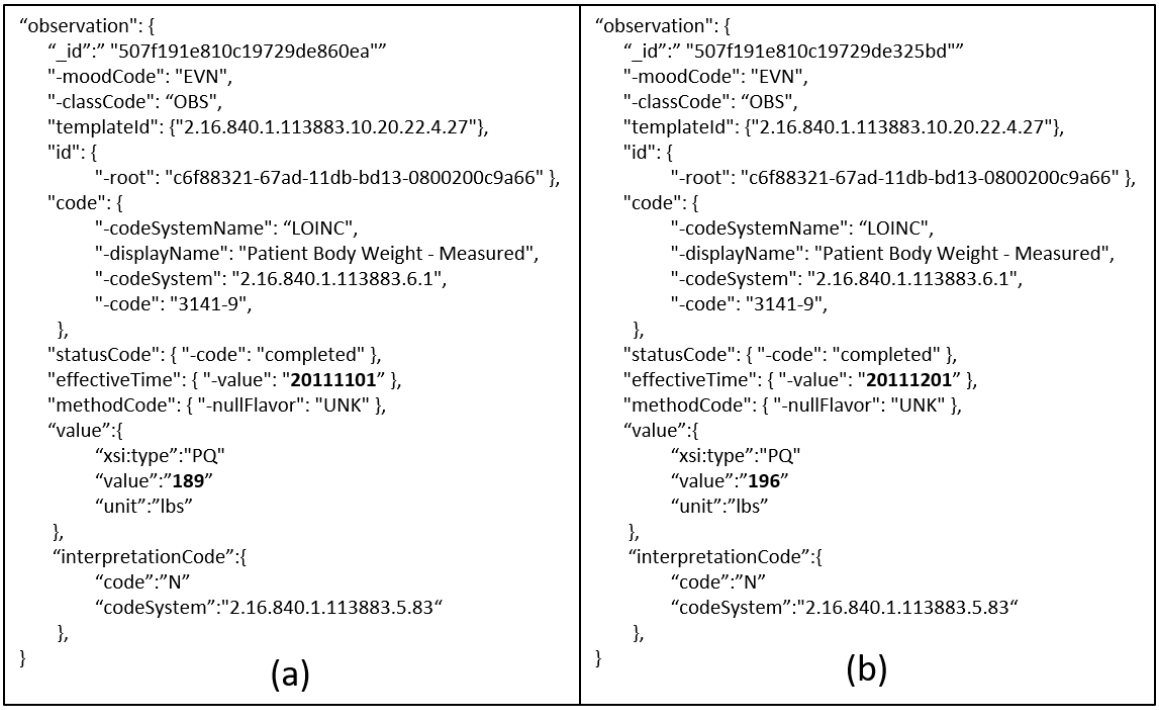

Fig. 2.15. Two Body Weight Readings

weight. A single query can then be used to retrieve all of the body weight readings. Figure 2.15 shows two different body weight readings from two separate days, each has a unique id and the same LOINC code. By using the LOINC code as a reference for the body weight in the diabetes document, the relationship between the two readings can be retained while still treating them as separate events. Another advantage to using the code of the event as a reference rather than the document objectID is that the coding standard is universal. Therefore, the relationships that a given user defines between his/her condition and events can be applied to another user's condition-based data. For instance, a doctor could define the relationship for one patient's condition and then apply those same relationships to the rest of his/her patients. The data will obviously be different but all of the events will use the same coding standard. The doctor will have to actively choose when this relationship can be used as a template as in few cases conditions can vary from one patient to another. The most common

example of this is medication since some medications can be used to treat multiple conditions. Finally the proposed approach can also help with the automated mapping of event-based data to the condition-based data model by using a learning algorithm to classify the events.

3. SYSTEM IMPLEMENTATION

This chapter describes the implementation of the proposed P2HR system. The implementation covers the data flow including how health records are received, processed, and stored. Particular emphasis is placed on the mapping from the event-based to the condition-based data model. The implementation of the underlying communication model for the proposed framework is also described in this chapter.

3.1 Data Model

Figure 3.1 displays the flow of the data within the P2HR system. Two types of data are handled by the system. The event-based data and the condition-based data. Both types of data are stored in a single database on a peers local machine. There can be multiple databases on a local machine, each containing the information associated with a single user. This can be useful for storing a relatives health information. However, it is mainly used by health care providers to store all of their patient's health records.

Data extracted from an EHR is either in XML or JSON format (Figure 3.1). Blue Button data is in XML and FHIR can be either XML or JSON. The Process of converting the event-based health records from an EHR to the proposed condition-based health records starts with extracting the data from the EHR. The extraction of information from the source EHRs is achieved through Blue Button+ [4] or FHIR [5]. As mentioned previously, Blue Button+ returns health records in a machine readable standard XML file [4] which is partitioned into multiple components including vital signs, encounters, immunizations, medications, etc. An example of Blue Button file is shown in Figure 3.2. The first indentation in the XML file is all the information related to the code of this particular event. In this case the LOINC standard is

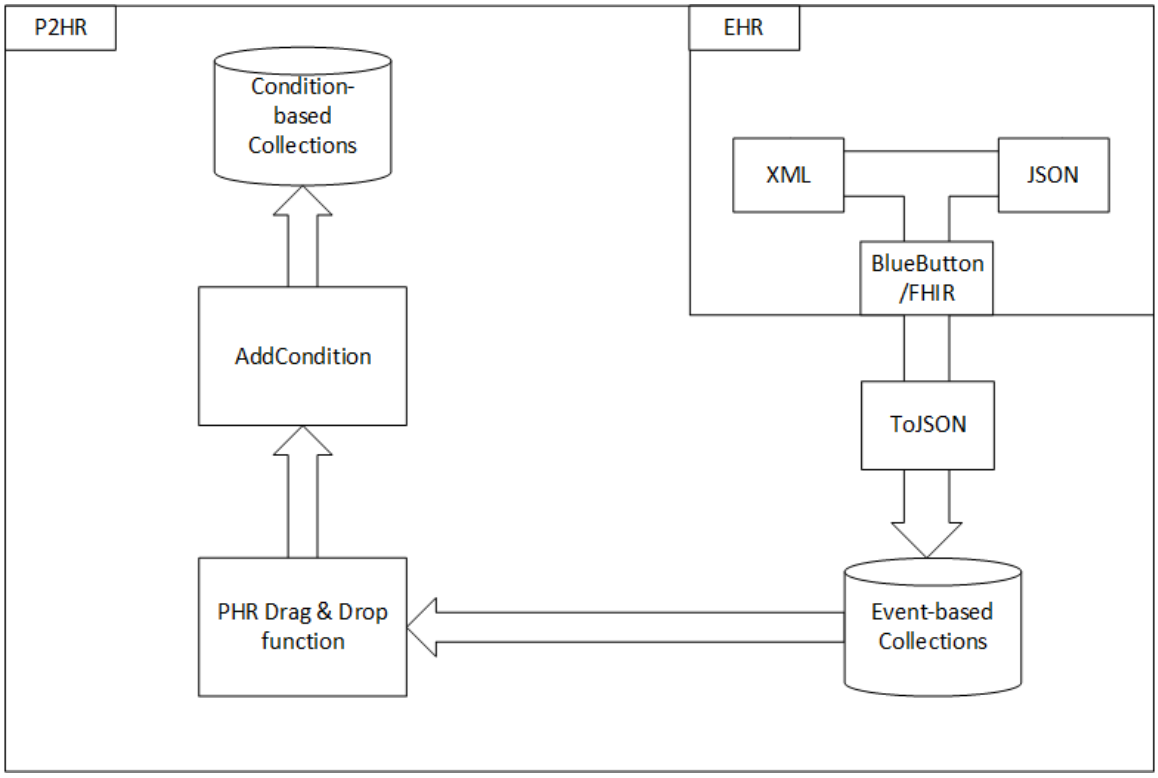

Fig. 3.1. Data Flow

used to define a code for Intravascular Systolic (Hypertension). The remainder of the XML file contains the measurement, unit, and date of the reading. The XML file is converted to JSON format so the record can be parsed and mapped to the proposed P2HR data model. This parsing is done in the middle tier using goxml2json [34], an open source Go Package. This package takes the XML input and returns the file in JSON format. However, in order to preserve bidirectional compatibility, the Blue Button XML file is not discarded. It is stored in the event-based collections of the database and its relation to the transformed data file is tracked. This facilitates the extraction and delivery of health information from the P2HR back to an EHR in the event-based format. Algorithm 3.1 shows the process of converting the XML file to JSON format.

```
<code
  code="8480-6"
  codeSystem="2.16.840.1.113883"
  codeSystemName="LOINC"
  displayName="Intravascular Systolic"/>
<text><reference
  value="#vit6"/></text>
<statusCode
  code="completed"/>
<effectiveTime
  value="20120806"/>
<value
  xsi:type="PQ"
  value="145"
  unit="mm[Hg}"/>
<interpretationCode
  code="N"
  codeSystem="2.16.840.1.113883.5.83"/>
```

Fig. 3.2. XML Vital Sign

```
Function readEMR(HR xmlfile)
1   json,err = xmltojson.convert(HR)
2   if err != nil then
3       return nil, err
    end

4   pushEvent(json)
5   return nil
```

Algorithm 3.1. Converting XML to JSON

In order to parse the incoming information, Golang structures were built for each component in the Blue Button+ XML file. Figure 3.3 shows an example P2HR JSON structure that corresponds to the XML example for Vital Signs from Figure 3.2. As mentioned previously, each component contains multiple entries. An entry

is an instance that is related to the corresponding component. Some instances are recorded many times over a patient's lifetime (e.g., blood pressure) and each instance is a unique entry in the component. Each of these instances are stored in their associated document within the respective collection. The structure to each document is unique to the component.

One of the difficulties of the conversion from XML to JSON is that XML can store multiple data types in an array, whereas in Golang this is not allowed. To circumvent this limitation, arrays are defined as JSON raw messages [35] with deferred decoding. These arrays of json.RawMessages are then unmarshaled as an array of interfaces. The latter process allows the mapping of array values to their corresponding variables. Building the Golang data structures internal to the proposed P2HR was straightforward, but time consuming. The process of building the golang structs was tedious because of the amount of detail that is present in each of the components extracted from the EHR. For instance, Blue Button has 15 components including the header. All the components have a similar structure, but they are not exactly the same. For example, Figure 3.2 is the structure for Vital Signs which is a relatively simple event when compared to a Procedure (Figure 3.3). The Procedure will have the same information as Vital Signs, excluding a value and unit as there is no measurement or reading for a procedure. In addition it also includes more information such as provider and participant.

Extracting health records from EHRs using FHIR follows the same procedure as Blue Button. However, unlike Blue Button, FHIR gives the user the option of extracting the records in XML format or JSON format. If the user uploads an XML FHIR document, then the conversion to JSON is performed. However, if the data is in JSON format then Algorithm 3.1 can be skipped. The main difference between FHIR and Blue Button is that FHIR is more modular. There are more components associated with FHIR and Golang structs have to be developed for these additional components in order to properly parse the health records.

```
<procedure
 classCode="PROC"
 moodCode="EVN">
 <templateId
  root="2.16.840.1.113883.10.20.22.4.14"/>
 <id
  extension="123456789"
  root="2.16.840.1.113883.19"/>
 <code
  code="168731009"
  codeSystem="2.16.840.1.113883.6.96"
  displayName="Chest X-Ray"
  codeSystemName="SNOMED-CT">
  <originalText>
   <reference value="#Proc2"/>
  </originalText>
 </code>
 <statusCode
  code="completed"/>
 <effectiveTime
  value="20120807"/>
 <priorityCode
  code="CR"
  codeSystem="2.16.840.1.113883.5.7"
  codeSystemName="ActPriority"
  displayName="Callback results"/>
 <methodCode
  nullFlavor="UNK"/>
 <targetSiteCode
  code="82094008"
  codeSystem="2.16.840.1.113883.6.96"
  codeSystemName="SNOMED CT"
  displayName="Lower Respiratory Tract Structure"/>
```

Fig. 3.3. CDA Observation

Once parsing is completed, Algorithm 3.2 is called in order to upload the records into the MongoDB database, the back-end of P2HR. MGO [36] is used to import and export documents into/from MongoDB. It is an open source driver that allows access to the database through a port. Functions in MGO are available to match the Mongo server functions query, insert, update, etc.

Algorithm 3.2 begins by connecting to the MongoDB database using mgo.Dial (localhost). The MongoDB server must be running in the background. The *for*

```
type Code struct{
        Code string
        CodeSystem string
        CodeSystemName string
        DisplayName string
}
type Text struct{
        Value string
}
type StatusCode struct{
        Code string
}
type EffectiveTIme struct{
        Value string
}
type Value struct{
        Xsi:type string
        Value string
        Unit string
}
type InterpretationCode struct{
        Code string
        CodeSystem string
}
```

Fig. 3.4. Data Structure for Vital Sign in Golang

loop iterates through each event type (i.e., Vital Sign, medication, lab results). The structure corresponding data is shown in Figure 3.4. Each event contains an array of structures where each instance in the array is a specific event. Finally, Algorithm 3.3 is invoked. It takes the array of event instances and the collection *coll* as variables. The *Insert* function inserts each instance of the event into the corresponding collection.

The process of converting event-based data is currently done by the user/patient or their health care provider. The interface of the P2HR system allows users to add new conditions then drag and drop instances of events to the most appropriate condition.

In order to display the data, it must first be extracted from the database. The process of extracting the event-based data is described in Algorithm 3.4. This algorithm begins by connecting to the MongoDB server. In the *for loop*, each collection

```
Function pushEvent(push jsonfile)
1   session,err = mgo.Dial(localhost)
2   if err != nil then
3       return nil, err
    end

4   for event in push do
5       d = session.DB(event-based).C(event.Name)
6       Insert(event,d)
    end

7   return nil
```

Algorithm 3.2. Push Event-Based Health Record into MongoDB

```
Function Insert(event []jsonfile, coll Collection)
1   for instance in event do
2       coll.Insert(instance)
    end

3   return nil
```

Algorithm 3.3. Inserting Multiple JSON Files

is accessed and then a query is executed to retrieve every event since there can be multiple occurrences of the same event. For example, a vital sign like blood pressure is measured often and each measurement is stored in a document. However, all associated documents have the same code. The drag and drop function would be time consuming and redundant if the user had to execute it for every individual blood pressure reading. Therefore, using the code from the coding standard allows the user to establish a relationship between a condition and all of the results. Using

```
Function pullEvent()
1   session,err = mgo.Dial(localhost)
2   if err != nil then
3       return nil, err
    end

4   for event in database do
5       d = session.DB(event-based).C(event)
6       err = d.Find().One(&result)
7       if err != nil then
8           return nil, err
        end

9       toHTML(event,result)
    end

10  return nil
```

Algorithm 3.4. Extracting Event-based Data from MongoDB

this approach, a single query can be used to return all related events to the given code and these events are then inserted into the *result* array and used as a parameter for the function *toHTML()*. The *toHTML()* function takes the event and all of the documents associated with the event as variables. The *toHTML()* will then display all of these events allowing the user to establish the relationships between events and conditions.

When a user inserts an event into a condition using the drag and drop function of the system, Algorithm 3.5 is called. The data that is fed into Algorithm 3.5 and includes the condition name and the information surrounding the event, specifically

```
Function convertEvent(push jsonfile)
1   condition = pullCond(cond)
2   update = append(condition.cond, insert)
3   pushCondition(update)
4   return nil
```

Algorithm 3.5. Converting Event-Based Data to Condition-Based

the code, coding standard and event type (Medication, Lab Result ...). This data is retrieved from the event-based collections in the database. Using these variables, the function extracts the document that relates to the corresponding condition from the condition-based collection in the database as shown in Algorithm 3.6. This algorithm is similar to Algorithm 3.4, but accesses the condition-based collection. The condition name is used as the query variable in the *Find* function. The *Find* function will subsequently return a single document similar to Figure 3.4. Algorithm 3.5 resumes when the function updates the condition by appending the specific event code to the corresponding event array.

```
type Diabetes struct{
        Medication[
                LOINC1,
                LOINC2,...]
        LabResults[
                LOINC1,
                LOINC2,...]
        ...
}
```

Fig. 3.5. Diabetes Data Structure

The final step takes the updated condition document and uploads it into the condition-based collection. This process is similar to uploading the event-based data as shown in Algorithm 3.7. The algorithm connects to the MongoDB server as in Algorithm 3.3. However, it will access the user‘s condition-based collection. The

```
Function pullCond(cond string)
1   session,err = mgo.Dial(localhost)
2   if err != nil then
3   |   return nil, err
    end

4   d = session.DB(Condition-based).C(me)
5   err = d.Find(bson.M"Condition":cond).One(&result)
6   if err != nil then
7   |   return nil, err
    end

8   return result
```

Algorithm 3.6. Extracting Condition-Based Data from MongoDB

```
Function pushCondition(push jsonfile, collection string)
1   session,err = mgo.Dial(localhost)
2   if err != nil then
3   |   return nil, err
    end

4   d = session.DB(Condition-based).C(collection)
5   d.Update(bson.M"Condition":push.Condition, push)
6   return nil
```

Algorithm 3.7. Updating Condition-Based Data into MongoDB

other difference is that it does not insert the document so as not to duplicate records. The algorithm calls the *Update* function which takes a query and the JSON file that will replace the extracted record. The *Update* function is built into the MGO

driver. The query is the same as Algorithm 3.6 and the JSON file that will replace the extracted record is the new updated document. It deletes the document found through the query and inserts the new file. If the query does not return a document then the update function works the same way as an insert.

```
Function peerExchange(push []jsonfile, user string)
1   session,err = mgo.Dial(localhost)
2   if err != nil then
3       return nil, err
    end

4   for x in push do
5       pushCondition(x,user)
    end

6   return nil
```

Algorithm 3.8. Inserting Condition-based Heatlh Records into MongoDB

The above algorithm is also called when P2HR users share their health records. The condition-based data that a peer might receive will already be in the correct format, as it must be coming from another P2HR. The data received will consist of multiple collections. In order to store these collections into MongoDB, Algorithm 3.8 is called which will in turn call Algorithm 3.7 multiple times, once for each document received. Algorithm 3.7 is called with the variables *doc* and *user*, where *doc* is the document that needs to be uploaded and *user* is the identifier of the person who owns the documents. In the case where a person is uploading his/her own health record, *user* will be the *username* of the owner of the PHR. When the data is uploaded for another person in the sub-network, *user* is this person's *username*. The *user* variable is important because the documents need to be placed into a different database based on this variable. As mentioned previously, the *update* function in Algorithm 3.7 will

still work even if the document does not already exist. If the query returns nil then the *update* function will work as an insert function.

3.2 Network Processes

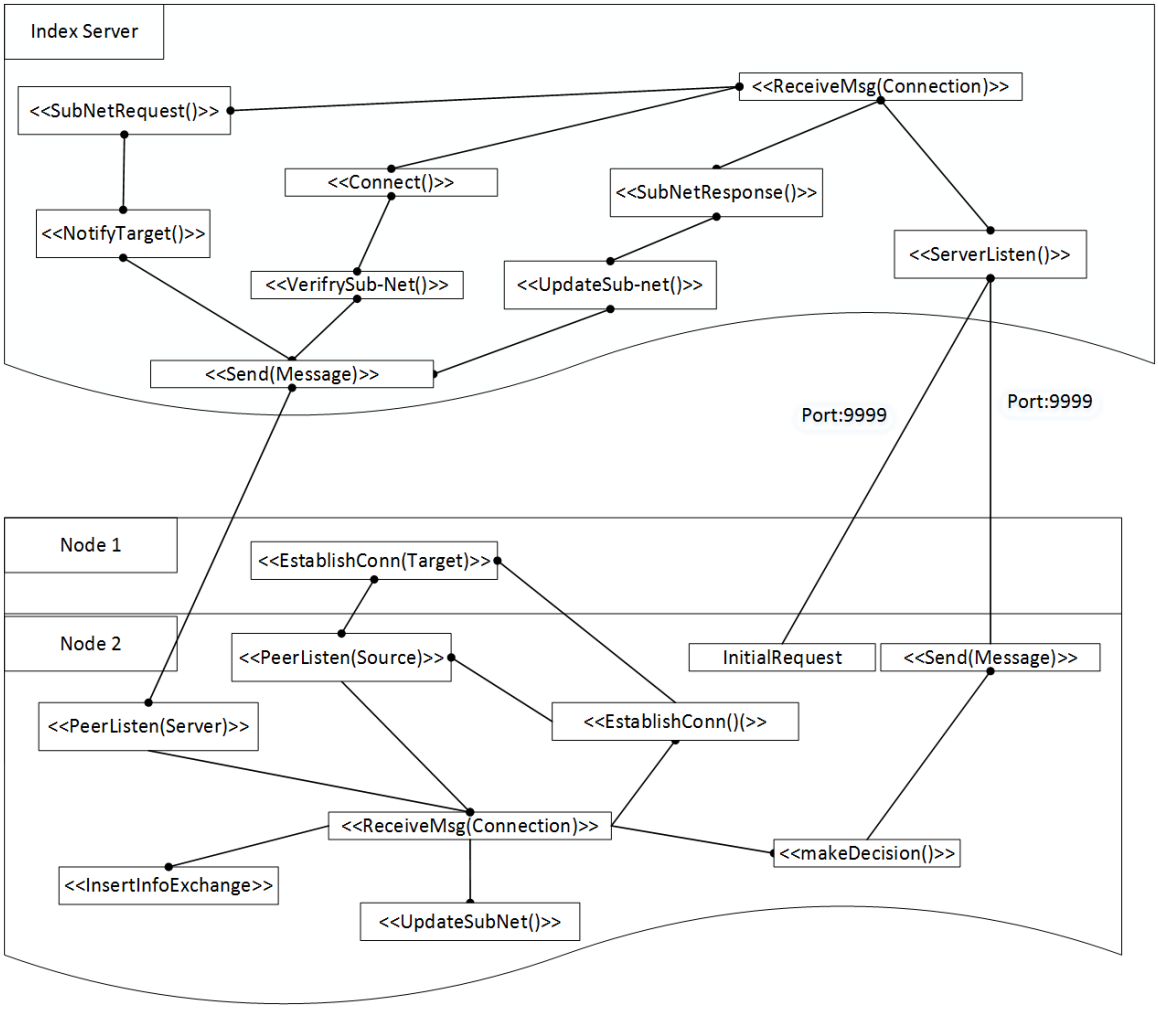

Fig. 3.6. Network Processes

There are two types of network processes involved in the proposed system: sub-network initialization and information exchange. Sub-network initialization is the process by which two peers join each other's sub-networks. Information exchange is

the process by which two or more peers that belong to each other's sub-networks send and receive information. Figure 3.6 shows the flow of function calls concerning the network processes and this section describes the underlying functions as executed by the index server and the peers.

3.2.1 Index Server

Algorithm 3.9 describes the functions of the index server. The server is constantly running awaiting for a peer to send them a request. The structure of a message received by the server is shown in Figure 3.7. The most important part of the Message is the *MesType* which indicates the function the server will have to call to process the message. Currently there are three *MesTypes*:

- *MesType* 0 is the first step of the sub-network initialization process and corresponds to the *SubNetRequest* function. The *SubNetRequest* function triggers a local database search for the target peer. Subsequently, an invitation to join the source's sub-network is sent to the target peer.

- *MesType* 1 continues the sub-network initialization and corresponds to the *SubNetResponse* function. This function receives the response from the target peer. Depending on the response (i.e., accept or decline), the server will send a message to both peers with their updated sub-networks.

- *MesType* 2 indicates an information exchange request and corresponds to the *Connect* function. This function triggers a local database search in order to verify the target peer's membership to the source's sub-network. Depending on whether or not the target is a member of the source's sub-network, the server will send the communication information of each peer to their counterpart.

The message data type also contains the *uid* of the target peer and the source node information. The data structure associated with a node is shown in Figure 3.7. The Node stores the *name*, *uid*, and *IP* address of a peer. The other part of the message

is the *Utility* variable which is a json.RawMessage. The json.RawMessage data type allows the message to contain different data depending on the message type. This is used because the data being received has to be flexible so that different information can be stored in the variable and parsed according to the *MesType*. This will also be useful in the future to accommodate additional request types.

The *ServerListen* function is the initial function executed by the server and is shown in Algorithm 3.9. Since all network processes are managed by the server, this function executes continuously in order to receive requests from the peers. The first three functions called by the index server establish a connection and then parse the data into a message type. Based on the message type, the relevant function is invoked and the server continues to listen for requests.

```
type Message struct{
        MesType int
        Source Node
        Target_uid string
        Utility json.RawMessage
}
```

Fig. 3.7. Message Data Structure

```
type Node struct{
        uid string
        Ipaddr IP
        Name string
        sub-net []string
}
```

Fig. 3.8. Node Data Structure

Algorithms 3.10 and 3.11 describe the steps involved in sending and receiving messages. Algorithm 3.10 shows the send process. The underlying function takes two arguments: the Message that is to be sent and a string that contains the IP address of the target. The send function first dials the target node with the *Dial* function

```
Function ServerListen()
1   ln = net.Listen()
2   conn = ln.Accept()
3   Message = bufio.NewReader(conn).Read()
4   if Message.MesType == 0 then
5       SubNetRequest(Message)
    end

6   if Message.MesType == 1 then
7       connect(Message)
    end

8   if Message.MesType == 2 then
9       SubNetResponse(Message)
    end

10  return nil
```

Algorithm 3.9. Index Server Initial Receive

from the Golang library using the target node's IP address. The *Dial* function will establish a *TCP* connection with the target. The next step takes the Message and marshals it as a JSON file by using the *Marshal* function from the Golang library. Finally, the newly created JSON file is sent to the target node by invoking the *encode* function which uses the *conn* variable that was obtained using the *Dial* function. Again, the reason the message is marshaled is because custom data types cannot be sent and received in Golang.

The *Receive* function assumes that a connection with a source has already been established and accepted. The function takes the connection as an argument, it uses the connection to create an encoder the same way as the *Send* function. Then uses

```
Function Send(Message Message, target String)
1   conn, err = net.Dial("tcp", target)
2   if err != nil then
3   |   return nil, err
    end

4   j = json.Marashal(Message)
5   json.NewEncoder(conn).Encode(b)
6   return nil
```

Algorithm 3.10. Send Function

```
Function Receive(conn Connection)
1   j = new []bytes
2   result = new Message
3   json.NewEncoder(conn).Decode(&j)
4   json.Unmarashal(Message, &result)
5   return result
```

Algorithm 3.11. Receive Function

the *Decode* function to extract the message as an array of bytes. This is followed by the *Unmarshal* function which is the counterpart of the *Marshal* function mentioned previously. The *Unmarshal* function takes two arguments: an array of bytes and an empty structure that will house the new data. The final step consists of returning the *result* which is a *Message* struct that contains the data extracted from the JSON file that was received from the source node.

If the *MesType* is 0, then the source is trying to add a peer (target) to his/her sub-network. The *SubNetRequest* function is shown in Algorithm 3.12. The function begins by extracting the relevant information of the target peer from the index server's lookup database using the target's *uid* included in the message. The return value of

the *Find* function is of type *Node* and is populated with the data that satisfied the query. Using the IP address from this Node, the index server will send a message to the target peer. The message sent to the target is an exact duplicate of the message received from the source peer, except the *Source* will be the extracted *Node* and the *Target_uid* will be the original Source's *uid*. This is because the target node only needs the Source's *uid*. Once the message is sent, the server will resume the execution of the *ServerListen* function until a new message is received.

```
Function SubNetRequest(Msg Message)
1   session,err = mgo.Dial(localhost)
2   if err != nil then
3   |   return nil, err
    end

4   d = session.DB(Index).C(Nodes)
5   err = d.Find(bson.M"uid":Msg.Target_uid).One(&result)
6   if err != nil then
7   |   return nil, err
    end

8   Msg.Target_uid = Msg.Source.Uid
9   Msg.Source = result
10  Send(result.Ipaddr, Msg)
11  return nil
```

Algorithm 3.12. Sub-network Initialize Request

If the *MesType* is 1, then the sub-network initialization process continues and the *SubNetResponse* function is called (Algorithm 3.13). The message will contain a boolean value which is *true* if the target accepted the invitation and *false* otherwise. The boolean value will be contained in the *Utility* variable. The message will originally

be in a Json.RawMessage format until the *MesType* is known to be 1. The *Utility* variable will then be parsed as a boolean value using the the *unmarshal* function. If the response is true, the index server will send each node's updated sub-network to the source and target nodes, respectively. This is performed by calling the *update* function. The update function takes two *uids* and finds the sub-network of the first *uid* and adds the second *uid* to it then returns the updated Node. The message will be sent to each peer and will contain the peer's updated sub-network consisting of an array of *uids*. If the request is denied, the index server will notify the source node.

```
Function SubNetResponse(Msg Message)
1   var result bool
2   json.Unmarashal(Message.Utility, &result)
3   if result then
4       Send(Msg.Source.Ipaddr,Update(Msg.Target_uid))
5       Send(Msg.Ipaddr,Update(Msg.Source.Uid))
    end

6   else
7       Message.MesType =4
8       Send(Msg.Ipaddr,result)
    end

9   return nil
```

Algorithm 3.13. Sub-network Initialize Response

If the *MesType* corresponds to an information exchange, then Algorithm 3.9 will invoke Algorithm 3.14 which includes the *connect* function. The *connect* function begins by invoking the VerifySubNet function which will return a boolean depending on whether or not the target peer is a member of the source's sub-network. This

function is shown in Algorithm 3.15. The *VerifySubNet* function works by accessing the server's local database then extracting and then querying the database for the *Node* of the *Source*. Then the *Source's* sub-net is iterated upon in order to see if the *Target_uid* is contained in the sub-net. If this is true, the index server will extract the information of the target, as in Algorithm 3.12. The algorithm will then send the source node's IP address to the target node and vice versa. The message that is sent will stay the same as it was received, except the *Node* in the message will be the *Node* corresponding to the peers counterpart (i.e., The Source receives the Targets *Node* and vise versa). If the target is not a member of the source's sub-network, then the index server will notify the source. The message will be a duplicate of the original request made by the source except with a *MesType* of 3.

3.2.2 Peer

This section describes the two network processes from the peer's prospective. Algorithm 3.16 begins with the peer listening for messages from the index serve. The peer will always be listening unless it is sending a request to the index server. The peer will only listen to the index server, unless it received notification that a member of their sub-network wishes to connect. The messages that the peers receive are slightly different from those that are received by the index server. The structure of a message remains the same as in Figure 3.7. However, a new message type is added. These messages are defined as follows:

- *MesType* 0: This message corresponds to a sub-network initialization message and will only be received by the target peer. It will trigger a decision on whether or not the target wants to accept the invitation to join the sub-network initiated by the source. The decision is then sent to the server.

- *MesType* 1: This message corresponds to the second part of the sub-network initialization function and will trigger an update of the user's local database if the target peer accepts the invitation to join.

```
Function connect(Msg Message)
1   ch = VerifySubNet(Msg.Target_uid, Msg.source)
2   if ch then
3       session,err = mgo.Dial(localhost)
4       if err != nil then
5           return nil, err
        end

6       d = session.DB(Index).C(Nodes)
7       err = d.Find(bson.M"uid":Msg.Target_uid).One(&result)
8       if err != nil then
9           return nil, err
        end

10      Send(result.Ipaddr,Msg)
11      Msg.Source = result
12      Send(Msg.Source.Ipaddr,Msg)
    end

13  else
14      Msg.MesType =4
15      Send(Msg.Source.Ipaddr)
    end

16  return nil
```

Algorithm 3.14. Server Information Exchange

```
Function VerifySubNet(target string, source string)
1   session,err = mgo.Dial(localhost)
2   if err != nil then
3       return nil, err
    end

4   d = session.DB(Index).C(Nodes)
5   err = d.Find(bson.M"uid":source).One(&result)
6   if err != nil then
7       return nil, err
    end

8   for x in result.sub-net do
9       if x == target then
10          return true
        end

    end

11  return false
12  return boolean
```

Algorithm 3.15. Check Sub-Network

- *MesType* 2: This message corresponds to the information exchange process. The peers process this message depending on whether they are the source or the target: The target will listen for the source IP address and the source will establish a TCP connection with the target.

```
Function PeerListen()
1   ln = net.Listen()
2   conn = ln.Accept()
3   Message = Receive(conn)
4   if Message.MesType == 0 then
5       makeDecision(Message)
    end

6   if Message.MesType == 1 then
7       update(Message)
    end

8   if Message.MesType == 2 then
9       establishConn(Message.Source)
    end

10  else
11      return
    end

12  return nil
```

Algorithm 3.16. Peer Response

- *MesType* 3: This message is used when any of the processes fail. Failure scenarios include a) when the target declines to join the source's sub-network, b) when the target is not a member of the source's sub-network or c) when the target is not active.

If the message received by the peer is of type 0 it means that the peer is the target of a sub-network initialization process. This message will trigger the *makeDecision* function shown in Algorithm 3.17. The function begins by prompting the peer to make a decision on whether or not it would like to join the source's sub-network. As mentioned on the server side of this process, the response will be in the form of a boolean value *true* if accepted and *false* otherwise. The next step is to alter the message before it can be sent back to the server. The *MesType* is changed to 1 which will notify the server that this is in response to a sub-network initialization request. The response boolean value is inserted into the *Utility* variable and the message is sent to the server.

```
Function makeDecision(Message Message)
1   Response = decide(Message.source)
2   Message.MesType = 1
3   Message.Utility = Response
4   send(Server, Message)
5   return nil
```

Algorithm 3.17. Response to Sub-Network Initialization

The *update* function is invoked if *MesType* is 1 as shown in Algorithm 3.18. This function is the same as the one used by the server when updating sub-networks. Every peer will have a collection in their database that contains personal information including the user's sub-network. The function connects to the peer's local database and uses the *Find* function to extract the personal document. The next step consists of updating the *sub-net* variable to the new sub-network received from the server in the message. Finally, the peer uses the MGO function *Update* to insert the resulting structure into their local database.

The *establishConn* function is triggered when *MesType* is 2 as shown in Algorithm 3.19. The *establishConn* function encompasses the entire information exchange process of the peers. This function is split into two components one that is executed by

```
Function update(Message Message)
1   session,err = mgo.Dial(localhost)
2   if err != nil then
3       return nil, err
    end

4   d = session.DB(Me).C(personal)
5   err = d.Find(bson.M"uid":Me).One(&result)
6   result.sub-net = Message.Source.Sub-net
7   err = d.Update(bson.M"uid":Me, result)
8   if err != nil then
9       return nil, err
    end

10  return nil
```

Algorithm 3.18. Update Sub-Network

the source and the other is executed by the target peer. The source is the peer that establishes the TCP connection. The source will begin the execution of the function by extracting the data related to the disease condition he/she wants to send to the target. This data could be the entire health record, a single condition, or specific document related to the condition. Using the condition document, the appropriate data is extracted. As previously mentioned, the condition document does not store any data values related to the condition. It only references the data by using the appropriate code. These codes are used to query the event-based collections for the actual data related to the condition. For each component, a query is executed to find all of the documents that contain the codes mentioned in the query. The query will return the *comp* variable which contains all of the documents associated with the

condition. The *comp* variable is then appended to an array and this array is included into the *Utility* variable of the Message which is sent to the target node.

The target node will execute the same *Listen* function that was executed in the *PeerListen* function, except that in this case the node will only accept communication from the source's IP address which was received in the message from the server. The restricted communication is a security measure so that the peers can only communicate with the server or with a verified peer. Once the target peer establishes the connection with the source, the *Receive* function is invoked and returns the message after it has been *Unmarshaled*. The target will use the source information to create a new database or access a previously existing database that is labeled using the *Uid* of the source. The target will then will access each collection and insert the related documents from the *Utility* array. Each variable in *Utility* is an array that corresponds to a component (i.e., Procedure, Encounters, Vital Signs, etc.) each of these arrays contains the documents that relate to the condition specified.

The final message type (*MesType* 3) will only be received by the original requesting node. This message type is only received if the source was trying to add a peer to their sub-network and the request was denied or if the target peer is inactive. This message can also be received if during the information exchange process the target peer is inactive or the target is not a member of the source's sub-network.

```
Function establishConn(Message Message)
1   if Message.Source = Me.IP then
2       session = mgo.Dial(localhost)
3       d = session.DB(Me).C(personal)
4       err = d.Find(bson.M"uid":Me).One(&result)
5       for each component in result do
6           d = session.DB(Me).C(component)
7           err = d.Find(bson.M"code":result.component).All(&comp)
8           Message.Utility = append(Message.Utility, comp)
        end

9       Send(Message, Message.Target)
    end

10  else
11      ln = net.Listen(Message.Source.IPaddress)
12      conn = ln.Accept()
13      Message = Recevie(conn)
14      session = mgo.Dial(localhost)
15      for each component in result do
16          d = session.DB(Message.Source.Uid).C(component)
17          err = d.Find(bson.M"code":result.component).All(&comp)
        end

    end

18  return nil
```

Algorithm 3.19. Peer Information Exchange

4. CONCLUSION

The main contributions of this research include the design and implementation of the network architecture, the data model and the mapping from the EHR's event-based data model to the proposed condition-based data model. Implementing the system did come with some challenges. For instance, there was a lack of available data to test the system. Besides the example CDA document from Blue Button's implementation guide there were few other sources of test data. That being said had the mapping been implemented using FHIR instead of Blue Button there would have been more available sources of test data as there are multiple repositories filled with FHIR data. The reason we did not implement the system using FHIR is because it was still under development and is less usable for the average patient. Another challenge when implementing the system was developing the data model. Originally the condition-based data stored the *objectIds* of the event documents related to the condition. However, when the system was completed it was found that querying the system for each *objectId* was not only complex but also inefficient. The goal of the project was to create a single system that contains all three of the functionalities described in [8]. These functionalities include information collection, information exchange, and information management. The resulting model allows the efficient review of data by health providers and promotes the sustained mainstream engagement of patients.

In the future we would like the P2HR+ (our expanded version of P2HR) to assess family medical records in greater depth, possibly finding relationships between relatives. We would look at conditions shared by relatives, and identify similarities. This functionality can help advance preemptive diagnosing.

We would also like to automate the process of mapping the traditional event-based data model to the proposed condition-based data model. One way to accomplish this is through an ontology that maps the events in the retrieved EHR record to their

associated condition. Our ontology will be a network of multiple ontologies each section of the ontology can focus on a specific portion of an EHRs health record (e.g., Medications, Encounters, Lab Results and etc.). The medical classifier ICD-10, which is included in UMLS, contains information on thousands of conditions, symptoms, and most importantly causes of diseases. We can use ICD-10 in the ontology to identify the condition by using the symptoms. An ontology is appropriate because new treatments, lab tests, and medications are continuously being introduced. Another approach to automating this process is by applying a learning algorithm to our system. The training set would be collected using the currently implemented drag and drop approach. A neural network could then be applied to the training set and newly created conditions could automatically be filled. Furthermore, these functionalities can be used to replace the currently implemented drag and drop approach that maps event-based to condition-based records.

Patient controlled health information is not a novel idea. It has been proposed in numerous formats and phases, however widespread acceptance of PHRs has yet to be obtained. Presentation and storage of information has remained constant throughout many of these PHR iterations. It is important to thoroughly compare various methods of PHRs before any adoption takes place. A condition-driven PHR architecture organizes information in a much clearer format for both the patient and doctor. This creates the beneficial features of increased patient-to-doctor communication and involvement, and efficient diagnosis of conditions. The medical landscape is rapidly growing. The healthcare community must continue to develop and consider new ideas and approaches in order to handle the growing demands and needs.

REFERENCES

REFERENCES

[1] R. M. Carney, J. A. Blumenthal, D. Catellier, K. E. Freedland, L. F. Berkman, L. L. Watkins, S. M. Czajkowski, J. Hayano, and A. S. Jaffe, "Depression as a risk factor for mortality after acute myocardial infarction," *The American journal of cardiology*, vol. 92, no. 11, pp. 1277–1281, 2003.

[2] J. Janson, T. Laedtke, J. E. Parisi, P. OBrien, R. C. Petersen, and P. C. Butler, "Increased risk of type 2 diabetes in Alzheimer disease," *Diabetes*, vol. 53, no. 2, pp. 474–481, 2004.

[3] J. Henry, Y. Pylypchuk, T. Searcy, and V. Patel, "Adoption of Electronic Health Record Systems among US Non-Federal Acute Care Hospitals: 2008-2015," *The Office of National Coordinator for Health Information Technology*, 2016.

[4] Blue Button+ implementation guide. [Online]. Available: http://bluebuttonplus.org/ (accessed 6/27/2017).

[5] D. Bender and K. Sartipi, "HL7 FHIR: An Agile and RESTful approach to healthcare information exchange," in *Computer-Based Medical Systems (CBMS), 2013 IEEE 26th International Symposium*. IEEE, 2013, pp. 326–331.

[6] The PHR Ignite Project. [Online]. Available: https://www.healthit.gov/policy-researchers-implementers/phr-ignite (accessed 6/27/2017).

[7] S. Kohler, "Connecting for Health: A Public-Private Collaborative," *Markle Foundation*, 2002.

[8] D. C. Kaelber, A. K. Jha, D. Johnston, B. Middleton, and D. W. Bates, "A research agenda for personal health records (PHRs)," *Journal of the American Medical Informatics Association*, vol. 15, no. 6, pp. 729–736, 2008.

[9] Dossia Health Manager. [Online]. Available: http://dossia.com/ (accessed 6/27/2017).

[10] L. A. Orlando, A. H. Buchanan, S. E. Hahn, C. A. Christianson, K. P. Powell, C. S. Skinner, B. Chesnut, C. Blach, B. Due, G. S. Ginsburg *et al.*, "Development and validation of a primary care-based family health history and decision support program (MeTree)," *NC Med J*, vol. 74, no. 4, pp. 287–296, 2013.

[11] Health Heritage. [Online]. Available: http://www.healthheritage.org/ (accessed 6/27/2017).

[12] W. Cohn, M. Ropka, S. Pelletier, J. Barrett, M. Kinzie, M. Harrison, Z. Liu, S. Miesfeldt, A. Tucker, B. Worrall *et al.*, "Health Heritage©, a web-based tool for the collection and assessment of family health history: initial user experience and analytic validity," *Public health genomics*, vol. 13, no. 7-8, pp. 477–491, 2010.

[13] "Health Vault." [Online]. Available: https://www.healthvault.com/us/en (accessed 6/27/2017).

[14] Google Health. [Online]. Available: https://www.google.com/intl/en_us/health/about/ (accessed 6/27/2017).

[15] S. Lohr, "Google offers personal health records on the web," *The New York Times*, 2008.

[16] J. Frost and M. Massagli, "Social uses of personal health information within PatientsLikeMe, an online patient community: what can happen when patients have access to one anothers data," *Journal of medical Internet research*, vol. 10, no. 3, p. e15, 2008.

[17] A 10-Year Vision to Achieve an Interoperable health IT Infrastructure. [Online]. Available: https://www.healthit.gov/sites/default/files/ONC10yearInteroperabilityConceptPaper.pdf (accessed 6/27/2017).

[18] Unified Medical Language System. [Online]. Available: https://www.nlm.nih.gov/research/umls/ (accessed 6/27/2017).

[19] O. Bodenreider, "The unified medical language system (UMLS): integrating biomedical terminology," *Nucleic acids research*, vol. 32, no. suppl 1, pp. D267–D270, 2004.

[20] Implementation Guide for CDA Release 2.0 Consolidated CDA Templates. [Online]. Available: http://www.hl7.org/implement/standards/product_brief.cfm?product_id=258#ImpGuides (accessed 6/27/2017).

[21] R. H. Dolin, L. Alschuler, S. Boyer, C. Beebe, F. M. Behlen, P. V. Biron *et al.*, "HL7 clinical document architecture, release 2," *Journal of the American Medical Informatics Association*, vol. 13, no. 1, pp. 30–39, 2006.

[22] C. Turvey, D. Klein, G. Fix, T. P. Hogan, S. Woods, S. R. Simon, M. Charlton, M. Vaughan-Sarrazin, D. M. Zulman, L. Dindo *et al.*, "Blue Button use by patients to access and share health record information using the Department of Veterans Affairs' online patient portal," *Journal of the American Medical Informatics Association*, vol. 21, no. 4, pp. 657–663, 2014.

[23] I. Carrión, J. L. Fern, C. Jayne, D. Palmer-Brown, A. Toval, J. M. Carrillo-de Gea *et al.*, "Evaluation and neuronal network-based classification of the PHRs privacy policies," in *System Science (HICSS), 2012 45th Hawaii International Conference*. IEEE, 2012, pp. 2840–2849.

[24] M. D. Hanson, "The Client/Server Architecture," *Server Management*, p. 3, 2000.

[25] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu, "Peer-to-peer computing," 2002.

[26] E. Adar and B. A. Huberman, "Free riding on Gnutella," *First monday*, vol. 5, no. 10, 2000.

[27] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Designing Privacy Enhancing Technologies*. Springer, 2001, pp. 46–66.

[28] K. Hwang, J. Dongarra, and G. C. Fox, *Distributed and cloud computing: from parallel processing to the internet of things*. Morgan Kaufmann, 2013.

[29] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "Measurement study of peer-to-peer file sharing systems," in *Electronic Imaging 2002*. International Society for Optics and Photonics, 2001, pp. 156–170.

[30] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," in *Pervasive computing and applications (ICPCA), 2011 6th international conference*. IEEE, 2011, pp. 363–366.

[31] R. Cattell, "Scalable SQL and NoSQL data stores," *Acm Sigmod Record*, vol. 39, no. 4, pp. 12–27, 2011.

[32] K. Banker, *MongoDB in action*. Manning Publications Co., 2011.

[33] R. H. Dolin, L. Alschuler, C. Beebe, P. V. Biron, S. L. Boyer, D. Essin, E. Kimber, T. Lincoln, and J. E. Mattison, "The HL7 clinical document architecture," *Journal of the American Medical Informatics Association*, vol. 8, no. 6, pp. 552–569, 2001.

[34] goxml2json. [Online]. Available: https://github.com/basgys/goxml2json (accessed 6/27/2017).

[35] The Go Programming Language. [Online]. Available: https://golang.org/ (accessed 6/27/2017).

[36] MGO Rich MongoDB driver for Go. [Online]. Available: https://labix.org/mgo (accessed 6/27/2017).