PEER-TO-PEER PERSONAL HEALTH RECORD

A Thesis
Submitted to the Faculty
of
Purdue University
by
William Connor Horne

In Partial Fulfillment of the
Requirements for the Degree
of
Master of Science Electrical and Computer Engineering

August 2019
Purdue University
Indianapolis, Indiana
THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

Dr. Zina Ben Miled, Chair
Department of Electrical and Computer Engineering

Dr. Lauren Christopher
Department of Electrical and Computer Engineering

Dr. Maher Rizkalla
Department of Electrical and Computer Engineering

Approved by:
Dr. Brian King
Head of the Graduate Program
To Kate and Cullane.
ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Zina Ben Miled. Her advice and support have been invaluable. I would also like to thank my committee members Dr. Lauren Christopher and Dr. Maher Rizkalla. The previous work by Zachary King was valuable to the success of this thesis. The IUPUI ECE and CS departments are appreciated for helping me develop my knowledge. Special thanks to the Naval Research Laboratory and Dr. Judson Hervey for supporting my graduate education. I would also like to thank all the students who have been my friends for these past two years.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>ABBREVIATIONS</td>
<td>xii</td>
</tr>
<tr>
<td>GLOSSARY</td>
<td>xv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>xvi</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Health Records</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Proposed Framework</td>
<td>3</td>
</tr>
<tr>
<td>2 RELATED WORK</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Personal Health Record</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Peer-to-Peer Networks</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 Distributed Transactions</td>
<td>19</td>
</tr>
<tr>
<td>2.2.2 Blockchain</td>
<td>21</td>
</tr>
<tr>
<td>2.2.3 Applications of P2P Networks to Health Records</td>
<td>22</td>
</tr>
<tr>
<td>2.3 Application Layer Protocol</td>
<td>24</td>
</tr>
<tr>
<td>2.4 EHR Datamining</td>
<td>25</td>
</tr>
<tr>
<td>2.4.1 Predictive Models and Bayesian Statistics</td>
<td>27</td>
</tr>
<tr>
<td>2.4.2 Bayesian Applications to EHR Datamining</td>
<td>28</td>
</tr>
<tr>
<td>2.4.3 EHR Datasets</td>
<td>31</td>
</tr>
<tr>
<td>3 ARCHITECTURE</td>
<td>33</td>
</tr>
<tr>
<td>3.1 Network Actions</td>
<td>36</td>
</tr>
<tr>
<td>3.2 Transactions</td>
<td>42</td>
</tr>
<tr>
<td>3.2.1 Authorization and Operation</td>
<td>43</td>
</tr>
<tr>
<td>3.2.2 Protocol</td>
<td>45</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>3.3</td>
<td>Hypertension Prediction Service</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Data Set</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Preprocessing</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Algorithm</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Training and Testing</td>
</tr>
<tr>
<td>4</td>
<td>IMPLEMENTATION</td>
</tr>
<tr>
<td>4.1</td>
<td>Index Server</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Database</td>
</tr>
<tr>
<td>4.1.2</td>
<td>API and Handlers</td>
</tr>
<tr>
<td>4.2</td>
<td>Peer Client</td>
</tr>
<tr>
<td>4.2.1</td>
<td>FHIR Client</td>
</tr>
<tr>
<td>4.2.2</td>
<td>P2P Subsystem</td>
</tr>
<tr>
<td>4.2.3</td>
<td>TCP Library</td>
</tr>
<tr>
<td>4.3</td>
<td>Service Client</td>
</tr>
<tr>
<td>4.4</td>
<td>Network</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Registration</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Login</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Resource Lookup</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Posting Documents</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Requesting and Approving Transactions</td>
</tr>
<tr>
<td>4.5</td>
<td>Transaction</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Get</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Push</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Service</td>
</tr>
<tr>
<td>4.6</td>
<td>Hypertension Prediction Service</td>
</tr>
<tr>
<td>4.6.1</td>
<td>Naive Bayes Model</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Performance</td>
</tr>
<tr>
<td>5</td>
<td>CONCLUSION</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>50</td>
</tr>
<tr>
<td>3.2</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>99</td>
</tr>
<tr>
<td>B.1</td>
<td>121</td>
</tr>
</tbody>
</table>

### Table 3.1
Selected Condition Features

### Table 3.2
Selected Consolidated Features

### Table 4.1
Classifier Performance with Panels 17, 18, and 19

### Table B.1
Feature Vector
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Types of Network Architectures</td>
<td>16</td>
</tr>
<tr>
<td>2.1 Types of Network Architectures (continued.)</td>
<td>17</td>
</tr>
<tr>
<td>2.2 MEPS Panel Design</td>
<td>32</td>
</tr>
<tr>
<td>3.1 System Architecture</td>
<td>34</td>
</tr>
<tr>
<td>3.2 Client Document</td>
<td>35</td>
</tr>
<tr>
<td>3.3 Registration</td>
<td>36</td>
</tr>
<tr>
<td>3.4 Heartbeat Process</td>
<td>37</td>
</tr>
<tr>
<td>3.5 Search for Network Resources</td>
<td>38</td>
</tr>
<tr>
<td>3.6 Fetch FHIR Documents from Portal</td>
<td>39</td>
</tr>
<tr>
<td>3.7 Record Registration</td>
<td>40</td>
</tr>
<tr>
<td>3.8 Registration and Approval of Service</td>
<td>41</td>
</tr>
<tr>
<td>3.9 Network Transaction</td>
<td>42</td>
</tr>
<tr>
<td>3.10 Transaction Request Data</td>
<td>43</td>
</tr>
<tr>
<td>3.11 Transaction Request and Approval</td>
<td>43</td>
</tr>
<tr>
<td>3.12 Transaction Data</td>
<td>44</td>
</tr>
<tr>
<td>3.13 Update Data</td>
<td>44</td>
</tr>
<tr>
<td>3.14 Get Protocol</td>
<td>46</td>
</tr>
<tr>
<td>3.15 Push Protocol</td>
<td>47</td>
</tr>
<tr>
<td>3.16 Service Protocol</td>
<td>48</td>
</tr>
<tr>
<td>3.17 MEPS Data Preprocessing Pipeline</td>
<td>54</td>
</tr>
<tr>
<td>3.18 Feature Vector</td>
<td>56</td>
</tr>
<tr>
<td>4.1 Index Server Classes</td>
<td>59</td>
</tr>
<tr>
<td>4.2 Database Collection</td>
<td>60</td>
</tr>
<tr>
<td>4.3 REST Access</td>
<td>61</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>4.4 Peer Client Subsystems</td>
<td>62</td>
</tr>
<tr>
<td>4.5 Peer Client Collections</td>
<td>63</td>
</tr>
<tr>
<td>4.6 Network and WUI Classes</td>
<td>65</td>
</tr>
<tr>
<td>4.7 WUI Paths</td>
<td>66</td>
</tr>
<tr>
<td>4.8 Registration</td>
<td>66</td>
</tr>
<tr>
<td>4.9 Login</td>
<td>66</td>
</tr>
<tr>
<td>4.10 Record</td>
<td>67</td>
</tr>
<tr>
<td>4.11 Account Page</td>
<td>67</td>
</tr>
<tr>
<td>4.12 Network Documents and Transaction Request</td>
<td>69</td>
</tr>
<tr>
<td>4.13 Transaction Page</td>
<td>70</td>
</tr>
<tr>
<td>4.14 Peer Records Table</td>
<td>70</td>
</tr>
<tr>
<td>4.15 FHIR Query</td>
<td>71</td>
</tr>
<tr>
<td>4.16 Peer-to-Peer Classes</td>
<td>72</td>
</tr>
<tr>
<td>4.17 Job Classes</td>
<td>74</td>
</tr>
<tr>
<td>4.18 TCP library</td>
<td>76</td>
</tr>
<tr>
<td>4.19 Service Client Subsystems</td>
<td>79</td>
</tr>
<tr>
<td>4.20 Transaction Process</td>
<td>89</td>
</tr>
<tr>
<td>4.21 Hypertension Prediction Classes</td>
<td>96</td>
</tr>
<tr>
<td>C.1 Welcome Page</td>
<td>124</td>
</tr>
<tr>
<td>C.2 Patient Registration</td>
<td>125</td>
</tr>
<tr>
<td>C.3 Patient Login</td>
<td>125</td>
</tr>
<tr>
<td>C.4 Network Page</td>
<td>126</td>
</tr>
<tr>
<td>C.5 Portal Selection</td>
<td>126</td>
</tr>
<tr>
<td>C.6 Downloading Records from a Registered Portal</td>
<td>127</td>
</tr>
<tr>
<td>C.7 Viewing Records</td>
<td>128</td>
</tr>
<tr>
<td>C.8 FHIR Data for a Single Record and Posting to Network</td>
<td>129</td>
</tr>
<tr>
<td>C.9 Provider: Network Records</td>
<td>130</td>
</tr>
<tr>
<td>C.10 Provider: Selecting Patient Record for a Transaction</td>
<td>130</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>C.11</td>
<td>Provider: Sending Request for Get Transaction</td>
</tr>
<tr>
<td>C.12</td>
<td>Patient/Provider: Transaction Waiting for Approval</td>
</tr>
<tr>
<td>C.13</td>
<td>Patient: Approving Transaction</td>
</tr>
<tr>
<td>C.14</td>
<td>Provider/Patient: Transaction Complete</td>
</tr>
<tr>
<td>C.15</td>
<td>Provider: Viewing Peer Records</td>
</tr>
<tr>
<td>C.16</td>
<td>Provider: Transferred Peer Record</td>
</tr>
<tr>
<td>C.17</td>
<td>Patient Filling out Survey for Service</td>
</tr>
<tr>
<td>C.18</td>
<td>Survey as a Record</td>
</tr>
<tr>
<td>C.19</td>
<td>Survey Posted to Network</td>
</tr>
<tr>
<td>C.20</td>
<td>Selecting Service Transaction</td>
</tr>
<tr>
<td>C.21</td>
<td>Requesting Service Transaction</td>
</tr>
<tr>
<td>C.22</td>
<td>Service Transaction Finished</td>
</tr>
<tr>
<td>C.23</td>
<td>Hypertension Service Result</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>ALS</td>
<td>Amyotrophic Lateral Sclerosis</td>
</tr>
<tr>
<td>ARHQ</td>
<td>Agency for Healthcare Research and Quality</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application-specific Integrated Circuit</td>
</tr>
<tr>
<td>BFT</td>
<td>Byzantine Fault Tolerance</td>
</tr>
<tr>
<td>BMI</td>
<td>Body Mass Index</td>
</tr>
<tr>
<td>BNP</td>
<td>Bayesian Non-Parametric</td>
</tr>
<tr>
<td>BSON</td>
<td>Binary JSON</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer-aided Diagnosis</td>
</tr>
<tr>
<td>CEHRT</td>
<td>Certified Electronic Health Record Technology</td>
</tr>
<tr>
<td>CMS</td>
<td>Centers for Medicare and Medicaid Services</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DBN</td>
<td>Dynamic Bayesian Network</td>
</tr>
<tr>
<td>DHT</td>
<td>Distributed Hash Table</td>
</tr>
<tr>
<td>EM</td>
<td>Expectation Maximization</td>
</tr>
<tr>
<td>EVM</td>
<td>Ethereum Virtual Machine</td>
</tr>
<tr>
<td>FHIR</td>
<td>Fast Healthcare Interopability Resources</td>
</tr>
<tr>
<td>FN</td>
<td>False Negative</td>
</tr>
<tr>
<td>FP</td>
<td>False Positive</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GPI</td>
<td>Generic Product Identifier</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HHS</td>
<td>Health and Human Services</td>
</tr>
<tr>
<td>HIE</td>
<td>Health Information Exchange</td>
</tr>
<tr>
<td>HIPPA</td>
<td>Health Insurance Portability and Accountability Act</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>HITECH</td>
<td>Health Information Technology for Economic and Clinical Health Act</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>ICD-10</td>
<td>International Classification of Diseases Revision 10</td>
</tr>
<tr>
<td>ICD-10-CM</td>
<td>International Classification of Diseases Revision 10 Clinical Modification</td>
</tr>
<tr>
<td>ICD-9</td>
<td>International Classification of Diseases Revision 9</td>
</tr>
<tr>
<td>ICD-9-CM</td>
<td>International Classification of Diseases Revision 9 Clinical Modification</td>
</tr>
<tr>
<td>ICU</td>
<td>Intensive Care Unit</td>
</tr>
<tr>
<td>IHIE</td>
<td>Indiana Health Information Exchange</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>LSTM</td>
<td>Long Short-Term Memory</td>
</tr>
<tr>
<td>MAP</td>
<td>Maximum a Posteriori</td>
</tr>
<tr>
<td>MEPS</td>
<td>Medical Expenditure Panel Survey</td>
</tr>
<tr>
<td>MR</td>
<td>Medical Record</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Transaction</td>
</tr>
<tr>
<td>ONC</td>
<td>Office of National Coordinator for Health Information Technology</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>PHR</td>
<td>Personal Health Record</td>
</tr>
<tr>
<td>POW</td>
<td>Proof-of-Work</td>
</tr>
<tr>
<td>PPV</td>
<td>Positive Predictive Value</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RFI</td>
<td>Request for Information</td>
</tr>
<tr>
<td>RMSE</td>
<td>Root Mean Squared Error</td>
</tr>
<tr>
<td>RTT</td>
<td>Round-Trip-Time</td>
</tr>
<tr>
<td>RV</td>
<td>Random Variable</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>SDA</td>
<td>Stack Denoising Autoencoder</td>
</tr>
<tr>
<td>SEER</td>
<td>Surveillance, Epidemiology, and End Results Program</td>
</tr>
<tr>
<td>SHA2</td>
<td>Secure Hash Algorithm 2</td>
</tr>
<tr>
<td>SHA3</td>
<td>Secure Hash Algorithm 3</td>
</tr>
<tr>
<td>SMOTE</td>
<td>Synthetic Minority Over-sampling Technique</td>
</tr>
<tr>
<td>SMR</td>
<td>State Machine Replication</td>
</tr>
<tr>
<td>SNAT</td>
<td>Symmetric Network Address Translation</td>
</tr>
<tr>
<td>SSP</td>
<td>SAS Transport File</td>
</tr>
<tr>
<td>STUN</td>
<td>Session Traversal Utilities for NAT</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TN</td>
<td>True Negative</td>
</tr>
<tr>
<td>TP</td>
<td>True Positive</td>
</tr>
<tr>
<td>TURN</td>
<td>Traversal Using Relay around NAT</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>ULMS</td>
<td>Unified Medical Language System</td>
</tr>
<tr>
<td>VA</td>
<td>Department of Veteran Affairs</td>
</tr>
<tr>
<td>WEKA</td>
<td>Waikato Environment for Knowledge Analysis</td>
</tr>
<tr>
<td>WUI</td>
<td>Web User Interface</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
</tbody>
</table>
GLOSSARY

Begnin fault a distributed fault such as message loss or a peer crash
Byzantine fault a distributed fault such as a malicious or faulty peer
posterior probability of an event given evidence
prior probability of an event using previous information
consensus agreement on a single value among distributed peers
features input data dimensions for a machine learning model
index server the central server used to progress and ensure transactions
job a work unit to carry out a transaction
liveness the distributed system does not halt in progress
panel a survey period in MEPS
patient person that is the recipient of healthcare services
peers a program that is participating in a peer to peer network, acts as both a client and server
provider person or organization that provides healthcare services
quorum the number of peers necessary to agree for consensus
record a health record in the network
requester the peer requesting a transaction
safety only correct actions are taken in a distributed network
target the peer that is the target of the transaction request
third-parties agent that needs access to the healthcare information
ABSTRACT


Patients and providers need to exchange medical records. Electronic Health Records and Health Information Exchanges leave a patient’s health record fragmented and controlled by the provider. This thesis proposes a Peer-to-Peer Personal Health Record network that can be extended with third-party services. This design enables patient control of health records and the tracing of exchanges. Additionally as a demonstration of the functionality of a potential third-party, a Hypertension Predictor is developed using MEPS data and deployed as a service in the proposed framework.
1. INTRODUCTION

Patients and healthcare providers need to exchange electronic health records. This exchange is necessary for patients to inform providers of their health and for multiple health care providers to collaboratively provide the care needed by each patient. Traditionally, health record exchanges have primarily focused on provider-to-provider exchanges and with a common strategy being the use of a data warehouse called a Heath Information Exchange (HIE). However, these strategies and the associated HIE often do not involve the patient. Exchanges are limited to a given region or target only large institutions. Therefore, patients that seek health services from small health providers or move from one region to another will have a fragmented health record.

To overcome the above issues, this thesis proposes a Peer-to-Peer (P2P) Personal Health Record (PHR) that enables the exchange of patient data between patients and health providers. The system consists of a transaction-based model with patients authorizing access to their records. The network logs each transaction, enabling the trace of accesses to each record. Providers can post new records thereby updating the patient’s PHR. Moreover, both patients and providers can access third party network services to perform operations on these records. The advantages of the proposed PHR include:

- patient control of his/her health records
- a common platform for the exchange of health data that can offer a continuum of medical history for each patient
- ease of access to historical data for new health providers added to the system
- ability to keep the health record for each patient up-to-date and accessible in real time to all concerned
- a distributed risk of compromised data
In addition, an efficient representation of the health record will help the patient make informed decisions about their health. In order to act upon the information in the PHR, the proposed system is augmented with a prediction service. The prediction service uses a Bayesian Network to predict a patient’s risk of developing Hypertension based on a submitted health record. This predictor serves as an example of the types of services that can enhance the utility of the proposed PHR.

1.1 Health Records

A patient’s electronic health record documents the health of a patient and their interactions with healthcare providers. Though implementations vary, the FHIR ontology [11] contains a generalization of the information captured in health records. It details roles, workflows, and financial information that documents the patient’s interaction with a provider. The provider records the observations, symptoms, and conditions for each patient. Diagnoses in FHIR are standardized according to various ontologies including ICD-10 [165], the International Classification of Diseases used for medical diagnosis. In summary, the FHIR ontology includes definitions for:

1. Individuals and their roles in the system (e.g., patient, provider, related, etc.)
2. Entities such as organizations, locations, or devices
3. Workflows such as tasks or appointments
4. Management of the encounter between a patient and a health provider
5. Clinical information such as observations, conditions, and medications
6. Financial information including billing

Several initiatives have attempted to overcome the fragmentation of patient health records across multiple institutions. One such initiative is the provider’s use of a Health Information Exchange (HIE) [59]. HIE are data warehouses that aggregate health records from different providers. Providers can then pull the aggregated records from the HIE to update their internal information. However, while HIE collect information from large providers, they are often unable to reach out to smaller providers.
Moreover, HIE are generally limited to a certain region and unable to aggregate health records from outside their hub of providers.

In 1996, the United States government passed HIPAA, which mandated a patient’s right to access their own health record [156]. To fulfill this requirement, some providers have created online portals for patients to view their records. Patients can access their records but the service is ineffective. Providers control and define the representation of the records limiting the service’s usefulness to the patient [92, 170]. Records from multiple portals are often incompatible with each other, making it hard to build a comprehensive view of a patient’s health. An alternative to this approach of provider controlled medical records is a Personal Health Record.

A Personal Health Record (PHR) [109] is an inversion of the provider-centric approach to health records. The patient stores and maintains the health record and the provider contacts the patient for record access. This patient-centric approach engages the patient and allows him or her to choose services from providers. Moreover, the PHR provides a platform for customized services such as appointment notifications, research related to a patient’s specific conditions, treatment options, and a connection to social networks for information sharing [50].

Researchers have estimated that a move to PHRs in the United States will result in a benefit of $11-19 billion annually [74, 144]. However, despite this economic benefit, PHR’s have experienced low uptake. This is attributed to a lack of patient awareness about PHRs, provider misgivings, and issues in interoperability [88, 147].

1.2 Proposed Framework

This thesis proposes a Peer-to-Peer (P2P) network for the aggregation and exchange of patient health records. The network consists of:

- A P2P network of users that is maintained by an index server
- A client that the patient or provider can use to interact with the network
- A protocol for exchanging records between the members of the health network
• A callable network service that implements a Bayesian model to predict a patient’s risk for Hypertension, exemplifying how services can be added to the proposed platform.

The proposed PHR system avoids the central governance and the overhead associated with maintaining an HIE. Indeed, in the proposed system, the provider can directly contact the patient. Moreover, the patient has control over his or her data and can selectively authorize access to this data. Finally, by leveraging the underlying P2P communication network, third party providers can offer additional health services to the patient such as the prediction of a patient’s risk to develop a condition from their health record.

The remainder of this thesis is organized as follows. The second chapter summarizes the current state of health record systems, peer-to-peer architectures, distributed transactions, EHR datamining, and Bayesian approaches. The third chapter describes the design of the proposed system, including system architecture, client features, and the dataset and algorithm used for predicting a patient’s risk of developing Hypertension. Chapter four presents the system implementation. Chapter five concludes with an analysis of the proposed system, its limitations, and potential extensions.
2. RELATED WORK

Patients, providers, and third parties are roles in the healthcare system. A patient is an individual who is seeking, receiving, or received medical care from a provider. A healthcare provider is the direct giver of care, usually a hospital, doctor’s practice, or medical institution. This exchange of service to improve the patient’s health is healthcare. A third party is an agent that needs access to information on the exchange of care between the patient and the provider but is not a direct beneficiary. Common third parties include, but are not limited to: referred specialist, researchers, government agencies, insurers, and family of the patient. The interactions between the patient and the provider are documented in the patient’s health record.

The stakeholders in healthcare, the patients, providers, and third-parties, have differing requirements that a health record needs to fulfill. A patient needs unrestricted access to their health record. Indeed, a patient may need to transfer this information to a new provider. A provider has similar requirements with a greater emphasis on managing the patient’s EHR, interacting with various payment outlets, providing a legal record, and tracking resources. Third party requirements are more diverse. For instance, a diagnostic service might need partial access to lab results and payment information in order to perform the service on behalf of the provider. A research group or government organization may want to analyze and track statistics for a population of patients. A patient’s descendants may want to preserve a family member’s record to document medical history. Thus, an EHR must balance and flexibly handle each party’s needs.

According to [120], a health record contains the following information:

- Administrative and billing data
- Patient demographics
- Progress notes
- Vital signs
- Medical history
- Diagnosis
- Medications
- Immunizations
- Allergies
- Radiology images
- Lab and test results

The patient medical information in EHRs conforms to various standards. The current diagnosis standard is the World Health Organization’s ICD-10 medical classification list [165]. This specification standardizes the naming of diseases, symptoms, and medical terms. Older records use the earlier ICD-9 standard [164]. Gaps in the standard have caused countries, medical groups, and EHR software developers to create extensions such ICD-10-CM [27]. Other organizations have developed ontology’s such as SNOMED-CT [154] and the Unified Medical Language System (ULMS) [86,157], to encode the semantic meaning of the medical terminology. Some encodings are proprietary such as the Generic Product Identifier (GPI) [162], which codes prescription drugs and groups.

Hospitals and care providers have been digitizing health records for the past three decades [16]. The resulting electronic health records (EHR) consist of the patient’s history, encounters with providers, and the patient medical record (MR) (e.g., procedure, lab reports, prescriptions, etc.). Several countries require providers to use EHRs [16] or heavily incentivize their adoption [16, 45]. In the United States, EHRs are certified by the Centers for Medicare and Medicaid Services (CMS) and Office of the National Coordinator for Health Information Technology (ONC) through the Certified Electronic Health Record Technology (CEHRT) certification [58,119]. Moreover, CMS and ONC promote the adoption of a certified EHR through Medicare and Medicaid funding programs. EHR developers have created various electronic health-
care solutions, often tailored to individual providers. Example commercial EHR systems include: EPIC [43], Cerner [29], and Meditech [98], which together comprise 66% of the market [149]. These systems adopt different representations of electronic health records. Therefore, transferring records between these EHRs requires using protocols such as HL7 [63] or the more recent FHIR [11]. To support these protocols, vendors must implement a mapping of the EHR’s internal representation to the protocol’s representation. Developing this mapping may be costly and tedious [52].

The expense and vendor lock-in associated with commercial EHR’s has created interest in open EHR standards. The openEHR foundation proposed an open specification of templates and archetypes to implement an EHR, rather than purchase a proprietary solution [124]. Several implementations have been deployed [123], but none in the United States. Another standard is the ISO-13606, Health informatics - Electronic health record communication [69]. A benefit of these open specifications is the ease of development of novel solutions that are customized to providers or specific fields [37]. However, these standards leave implementation details unanswered, often leading to wildly varying solutions [49]. Moreover, the implementation of these standards requires software expertise which is tangential to the core business of healthcare providers.

To facilitate the exchange of records between providers, States and providers have created Health Information Exchanges (HIE) [59]. HIE are third party brokers that aggregate health records for various regional providers, handle the conversion of the various records to a common data representation, and transfer records to requesting third parties. HIE fulfill three types of transfer [161]:

- Direct Exchange: Securely send and receive patient information between providers.
- Query Exchange: Find information related to an unknown patient.
- Consumer Mediated Exchange: Intake records directly from the patient.
The ONC has incentivized the development of HIEs with grants from the Health Information Technology for Economic and Clinical Health act (HITECH) [2,59]. As of 2017, 69% of hospitals had agreements with an HIE [71] but this number is misleading. The coverage is patchwork, with access to only members of the HIE network resulting in providers needing multiple HIE agreements to service patients. Furthermore, it is difficult to transfer records across multiple HIEs and the capabilities of the exchanges are often inadequate, making it difficult for providers to integrate HIEs with their local EHR [117]. In Indiana, the Indiana Health Information Exchange (IHIE) [68] encompasses most of the major providers in the state but has yet to include rural health providers. In Illinois, established regional exchanges prevented the success of a statewide exchange [139]. At the national level, the ONC has sponsored the creation of eHealth Exchange [42] to facilitate interstate transfers. As of 2019, 75% of US hospitals and 59 HIEs had agreed to participate in eHealth Exchange. When these mechanisms are absent, the patient needs to facilitate the transfer of records between providers.

The ONC has expressed interest in making EHRs interoperable [118]. CMS has created the Promoting Interoperability Program to incentivize providers [28]. Several EHR vendors and HIE have agreed to increase interoperability under Carequality [24], an effort and standard to ease the difficulty of transfer and conversion. HL7 proposed the Fast Healthcare Interoperability Resources (FHIR) [11,18] as a standard to transfer records between providers. FHIR has two major components, a generalized ontology to represent the various aspects of an EHR and a Representational State Transfer (REST) protocol to exchange the information. Application developers can target the standard to create services based on the ontology and interact with servers that adhere to the REST representation. HL7 has continued development of services using FHIR under the Argonaut project [64]. One service is SMART [95], a patient approved access framework for third parties using OAuth2 [36]. Carequality has recently supported FHIR [150] and it is likely that FHIR will become the industry standard for EHR transfer.
Public sector development has focused on enhancing the Blue Button program. Blue Button is a CMS and US Department of Veterans Affairs (VA) initiative that started in 2010 [104] to enable patients to download their records. Blue Button supports access to Tricare, the VA, and Medicare but has also garnered support from the private sector. Over the years, CMS has updated Blue Button several times to reflect new EHR developments. Originally, downloaded records were text files. An update called Blue Button Plus, includes an option to export the record as HL7 CDA XML and supports online push/pull requests [20,104]. A supporting Department of Health and Human Services (HHS) project called Blue Button Connector [1] provided an online website to help patients search for accessible records. The connector later grew to include a service lookup component for developers. The connector was discontinued in May 2017. The development of FHIR generated a 2015 HHS pilot project to support the standard [61,142]. CMS continued this work and released Blue Button 2.0 in 2018 [26]. Blue Button 2.0 supports FHIR v3.0.0 and allows developers to register clients for the service. Approved clients can access Tricare, VA, and Medicare Part A, B, and D records.

All the above initiatives are institution-centric and in support of provider record ownership. The ownership and control of Health Records is controversial. A 2014 study [22,140,152] of primary care patients who had been given granular control of their records found that 43% of patients enabled some restriction on their record. Primary care providers could override patient wishes, with 10 overrides of restricted records out of 126 overrides. In surveying the patients, 93% agreed with restricting part of their record and 95% with restricting persons from accessing the record. The corresponding responses for providers were lower at 54% and 42%, respectively. Follow-up responses supported this patient/provider disparity.

In the United States, patients are guaranteed access to their records under HIPAA [156]. To fulfill this requirement, hospitals have begun offering online portals, sometimes called tethered PHR, for patients to extract and view their records [77]. In a 2017 review of patient portals [170], the United States Government Accountabil-
ity Office (GAO) interviewed providers who reported that 91-92% of their patients were offered electronic access to their health information. However, these numbers are not supported by other studies. For example, Lyes [92] reported that 33% to 45% of patients recalled being directly offered this access. Other studies in 2016 and 2018 [13, 127] reported offer access rates of 34 to 60.3%\(^1\). Peacock and Anthony [13, 127] found that uptake was heavily influenced by the provider’s explicit offer and that both men and minorities were offered access at lower rates despite both groups expressing the same level of interest as their peers. The GAO also interviewed patients about their portal experiences and found that patients were frustrated by incomplete or incorrect records and poor user interfaces. Patients expressed interest in aggregating records from multiple portals and viewing their longitudinal records.

A portal has limited usefulness to a patient and is oriented towards fulfilling regulatory requirements. Indeed, the provider still controls and defines the representation of the health records. Portal records downloaded from multiple providers often have incompatible formats, preventing the aggregation of these records. If a portal is not in place, it can take months to access this health data. Access can also be costly to the patient, and is often non-compliant with the patient’s wishes [92].

\(^1\)The 60.3% access offers includes access offers from insurers.
2.1 Personal Health Record

A Personal Health Record (PHR) makes the patient, rather than the provider, the custodian of his or her health information. A PHR stores and manages the health records from various providers, creating a single access point for the patient. Though the definition of a PHR has been historically ill-defined [116], a CMS Request for Information (RFI) suggested that a PHR has the following attributes [109]:

- owned and controlled by the patient
- contains lifetime health information
- stores health information from all healthcare providers for each patient
- accessible from any place at any time
- private and secure
- transparent, where the patient can track access to his or her health data by others
- facilitate the exchange of health information between health providers

A PHR resolves several issues inherent to provider-centric approaches. A single access point to a patient creates a hub for providers. This resolves the conformance issues between various EHRs and provides a communication bus between providers. The provider solutions of HIE and EHR systems consist of distributed architectures making it difficult to achieve a single consistent view of the patient’s records. Moreover, EHR and HIE have difficulties in of correctly aggregating records from external sources as patient records lack a primary key [44]. To correctly identify patient records, providers must resort to heuristic and probabilistic methods with error rates around 15-38% [44]. A PHR eliminates this issue by making the PHR the master view of the patient’s health records.

For patients, a PHR can provide access to curated health information, enable the easy tracking of chronic conditions, and facilitate communicate with providers [147]. Services and home instruments can connect and update the health record. Moreover, the records, transfers, and communications between various providers is transparent
to the patient. The patient can control access to their records, dictate the terms of access, and trace accesses by providers. This can provide a patient evidence of HIPAA violations. For caregivers, a PHR provides a framework to help manage the health of the patient [147].

Providers and third parties can benefit from PHRs. PHRs can help providers engage with patients to improve outcomes, transfer health records between different EHR systems, and get real time feedback from patients [147]. It also provides a common access point, eliminating the need to rely on HIE [147]. Moreover, PHR are theorized to provide economic benefit. A 2008 economic projection study [74,144] of PHR adoption in the United States found the economic benefits depend on the PHR used. Portal solutions with each provider implementing their own and with no interoperability were estimated to be an annual net negative of $29 billion with an initial cost of $130 billion. Third party and interoperable PHRs were projected to provide $11-19 billion in benefit with initial cost between $3.7-21 billion. The study’s authors noted that third party PHRs were highly scalable and that interoperability was key to reducing cost and providing benefit.

There are several challenges to PHR adoption: PHRs are difficult to implement and deploy [88,145,147]; Patients are often unaware of existing PHR systems and may be concerned that their records would be insecure [88,145,147]; Not all patients want or need a proactive role in their health management [115,145]; and Providers have concerns about workflow integration, patient interpretation, PHR validity, liability, and incentive [88,115,147]. In addition, independent PHRs limit institutional control of health data which is counter to the business model of these institutions [147]. The bearer of cost is also unresolved [147]. Recently, several high-profile PHR systems failed including Google Health [89], Microsoft Health Vault [101], and Dossia [40]. All three of these programs were targeted at the sponsoring company’s employees in hopes of decreasing the company’s healthcare cost. In light of these concerns and issues, it is important that both patients and providers see immediate benefits to
using a PHR. PHR and PHR-like systems are beneficial to patients with long-term conditions but have had issues with cost and accessibility [145].

CMS conducted a Personal Health Record Pilot called MYPHRSC for Medicare enrollees in South Carolina from 2009-2011 [115]. The PHR consisted of a web-based application from HealthTrio that provided users summaries of health information, medications, claims, calendars, and more. An outreach program contacted 100,000 Medicare beneficiaries in South Carolina and attempted to generate interest with demonstrations and adverts. The study conducted follow up surveys and interviews with beneficiaries. Overall, 4,114 beneficiaries registered and 3,041 used the PHR more than once. Pilot users expressed interest in the improved communication with providers, health summary information, and the printable wallet card with information on health, medications, and emergency contact. Even though the study focused on patients with chronic conditions, these users were not more likely to use the PHR. The lack of uptake among this group might have been caused by the short time period of the pilot and the lack of results tracking. Other cited issues were inaccurate claims data, usability issues, and lack of integration with beneficiaries’ workflows.

MyHealtheVet (MVH) [153] is a tethered PHR system operated by the VA which was launched in 2003 and is still in operation. Veterans can refill prescriptions, look at health records, search for health information, message the VA, and more. A 2011 usability survey of MVH [56] reported that patients were interested in the prescription refill feature and in printing their health information. However, there was negative feedback in regards to the search function and usability. MVH added the Blue Button feature in 2010. MyHealtheVet is not a true PHR. Patients must manually share their information downloaded through Blue Button with a third party. The third party is then responsible for correctly extracting the information.

Third-party applications have started to integrate with the updated Blue Button service. One application is iBlueButton, a PHR by Humetrix [65]. It is a mobile-based PHR that allows a patient to access and locally store their health records from government Blue Button providers. The application allows the patient to make
notes and check for drug conflicts. The application is limited as it strictly targets government Blue Button projects. Additionally, the application can only pull records and can’t post updates back to the provider.

Patient’s Like Me is an online social network for patients with chronic conditions [126]. Originally the program targeted patients with Amyotrophic lateral sclerosis, ALS, but has since grown to encompass more conditions. Users can share information on conditions and treatments [50,160]. However, users do not control the shared information and the information is subject to research datamining to fund the program. Several services can be added to a PHR in order to enhance its benefits to the patients. For example, a service that enables the prediction of risk factors for chronic disease. An example of such a service for Hypertension is demonstrated in this thesis. Another service that might be integrated into a PHR is facilitated provider-patient communication. For example, providers can monitor medication adherence and disease conditions. A 2011 study found that testing and screening reminders helped increase patient medication adherence [166]. A commercial application based on this idea is care.coach [23]. In care.coach, a provider can communicate with patients through an online 3D avatar thereby limiting the need for doctor’s visits.

2.2 Peer-to-Peer Networks

There are two different common types of network communication, Client-Server and Peer-to-Peer (P2P) (Fig. 2.1). In a Client-Server architecture (Fig. 2.1(a)), clients connect to a central server [78]. The central server controls access to information and clients exchange information by way of the central server. A Peer-to-Peer network architecture has clients called peers, who communicate directly with each other [78]. These peers must operate as both client and server in the network.

Network design and topology play key roles in the design of a P2P architecture. There are several models: centralized, pure, and hybrid. These designs have advantages and disadvantages.
A centralized P2P model (Fig. 2.1(b)) uses an index server (also called a central server) to provide resource lookup for the network [67]. Clients search the index server to find peer resources and then contact the target peer in order to exchange data directly. Benefits of this system are ease of deployment and fast resource lookup. Drawbacks include a reliance on the index sever and limited scalability.

A pure P2P network (Fig. 2.1(c)) has no index server and clients must query other peers to discover resources [67]. To search for a resource, the client either floods the network with a query [67] or accesses a distributed hash table (DHT) [67, 78] maintained by the peers which lists the resources and owner. The primary benefit of a pure P2P system is an equal relationship among the peers, with peers sharing costs and resources fairly. However, drawbacks include slow lookup and high network complexity. Pure P2P networks such as Guntella [134] had issues with network traffic until the development of DHTs.

The hybrid P2P model (Fig. 2.1(d)) selects some peers in the network to act as supernodes [67]. Other peers connect to the network through these supernodes. Supernodes route peer messages to other supernodes who then deliver the messages. This design has a faster lookup than a pure model and is more resilient than the centralized model. However, the hybrid P2P model is difficult to implement. Skype prior to 2016 [151] used a hybrid P2P model [67].
Fig. 2.1.: Types of Network Architectures
Fig. 2.1.: Types of Network Architectures (continued.)
There are three stages to P2P communication: accessing the network, connecting to peers, and searching for a resource. Each stage has challenges that the network design must overcome.

During the first stage when accessing the network, the peer has no initial knowledge of the network topology. A standard method to locate other peers is a signaling server. A signaling server has a predetermined location and holds a list of networked peers and their IPs. The client application queries this server for networked peers and attempts to connect to one of the peers using the peer’s IP. With IPV4, port forwarding, and Symmetric Network Address Translation (SNAT), it is not always possible to connect directly to another peer. TCP Hole punching [146] and STUN/TURN protocols [94] can overcome these issues. The communication between the peers is usually a custom application protocol built on top of TCP or UDP. Once connected to the network, the peer can search for resources in the network. The lifecycle of a search varies according to the network topology. In pure and hybrid P2P models, the query can flood the network or access the DHT maintained by peers. In a centralized model, the client sends the query to the index server. The index server has a list of resources in the network and can direct the client to the owner. The client can then connect to the target peer and exchange the resource directly or by routing the data through the network.

P2P networks have several advantages over Client-Server architectures. P2P networks are scalable [78]. Compute, network, and storage cost are shared by the peers and not concentrated in a single server. Lastly, P2P networks are resilient against network failure [67]. Despite interest in P2P networks, the architecture has several issues that limit its adoption. SNAT requires a third-party server to facilitate routing of data between peers [94]. Moreover, peer-churn [78], where peers enter and leave the network, can lead to network degradation when too few peers are available. To counteract this, peers must be properly incentivized to remain in the network. Lastly, P2P file-sharing networks such as BitTorrent are heavy used to illegally share copyrighted works, which limits business support of P2P models [67].
2.2.1 Distributed Transactions

In a distributed network, transactions among peers takes the form of *consensus*. A distributed environment is unreliable, creating the need for a protocol that solves consensus in a bounded time \(^2\). The performance of a distributed system is measured by its *safety* and *liveness* \(^7\). Safety means that an incorrect action does not occur, and liveness refers to the system’s ability to progress in state. Fischer, Lynch, and Paterson \(^{48}\) proved that there is no protocol that satisfies both. The focus changes to providing a best effort protocol. Another issue with distributed systems is the reliability of the individual parts. Processes can lie or fail, an error called a *Byzantine* fault \(^3\). A Byzantine fault can be malicious or as a result of a failure to follow the protocol. In contrast, a *Benign* fault is an unintentional crash or message loss \(^{25}\). Ignoring the unreliability of the connection, Byzantine Fault Tolerance is possible but only if more than \(\frac{2}{3}\) of the peers are working correctly.

The simplest solution to consensus is to have a central server manage the state of the system \(^{93}\). The server is the global view of the system and acts as the synchronization between clients. Clients replicate the state of the server upon connecting and post updates to the server. The server checks the updates against its own internal copy and broadcast state updates periodically to the clients. This architecture is used in distributed simulations \(^{93,113}\). This design provides some benefits. Incorrect actions are not possible, and it does not assume client trust. Moreover, client failure does not block progress of the server’s state and thus the network. However, there are limitations with this centralized architecture. For instance, clients may need to repeat actions if they are out of date. Clients can keep up with the state of the system at the cost of a Round Trip Time (RTT) \(^{78}\). The central authority is a point of failure in the network and the design suffers from limited scalability. Modified P2P versions can remove this issue at the cost of safety \(^{113}\).

\(^3\)Consensus is commonly illustrated as the allegory of the two-general problem \(^{55}\).
\(^3\)This is in reference to the allegory of the Byzantine General Problem \(^{84}\).
The Two-Phase-Commit [55] protocol is a commonly used protocol for consensus in which clients post to a central authority.

- The authority asks if the clients want to commit
- Clients post their messages and wait
- Authority verifies the transaction
- Authority broadcasts the commit to the Clients
- Clients acknowledge the broadcast

This protocol is safe in most scenarios but is reliant on the central authority. If the authority fails the clients will never release their resources. The Three-Phase-Commit protocol builds on this protocol by adding a timeout to make the protocol nonblocking but with a loss of safety.

The Paxos algorithm [82] is a complex algorithm that guarantees safety and good liveness. The protocol uses periods called rounds, each with a ballot ID and 3 phases. On each round, a process or multiple processes can act as the leader. Each leader queries the other processes for their ballot ID. If the proposed leader has the highest ID then they are the leader for the round. The next phase has the new leader push a new value to the group. The group responds to the leader. If the leader receives a majority of the responses, called a quorum, it broadcasts to the group that the round is complete. Otherwise, the leader moves onto the next round. Paxos is noted for its cost and complexity to implement and variations have been proposed [83, 122]. Further it can only protect against Benign faults. A program that implements a variation of Paxos is Apache Zookeeper [66].

State Machine Replication (SMR) [81, 138] can provide protection against Byzantine Faults. A state machine is a computation model in which a machine executes actions to transform its current state. In SMR, each client runs a replica of the same state machine executing the same actions. When a byzantine failure occurs, as long as more than 3 healthy replicas of the faulted process exists [84], the system is fault tolerant. However, a naive implementation of SMR can fail to protect against Byzantine faults [25]. A Byzantine Fault Tolerant system needs a replication algorithm with
proper creation, coordination, and verification. The first implementation of such a system was the Practical Byzantine Fault Tolerant (BFT) algorithm [25].

2.2.2 Blockchain

Blockchain is another architecture designed to solve Byzantine Faults. Unlike BFT replication protocols, it achieves its resiliency though hashing. Blockchain uses strong cryptography and Proof-of-Work (POW) [15] to provide an append only ledger [99] consisting of blocks. The proof-of-work prevents Byzantine faults by making network peers find cryptography hashes before committing transactions. These blocks are linked together by these hashes, making rewrites of the transaction tree increasingly difficult. Blockchains differ from BFT replication protocols. Though they both provide protection against byzantine faults, Blockchains are open to new users while BFT systems need a prior verification. Blockchains are also more resistant to Sybil attacks, where an attacker directs a large number of hidden nodes to disrupt the system. High-profile Blockchain applications include the cryptocurrencies Bitcoin [111] and Ethereum [163].

Bitcoin [111] is a digital currency that uses a Blockchain to maintain a distributed record of transactions and owners of bitcoins. The goal of the application is to provide a decentralized, secure, and anonymous method for exchanging assets. Each owner is identified by a public key. The coin’s value is tied to the market’s demand. Units of bitcoins can be transferred from one client to another by issuing a transaction to the network. The owner signs the transaction request with their private key. The system verifies the owner using the public key. Transaction requests are pooled together into blocks to be verified and appended to the ledger. Nodes in the network called miners verify a block by finding a hash of variable strength, a proof-of-work. After a hash is found, the block is added to the chain and the miner is rewarded a bitcoin for the proof-of-work. Bitcoin implements a stack-based language to support dynamic
transactions [85]. To prevent transactions from not completing, the stack language
does not implement loops and is not Turing Complete \(^4\).

Ethereum [163] is another cryptocurrency aimed at providing verified electronic
contracts called smart contracts. It is similar in design to Bitcoin but has several
key differences. One of Bitcoin’s issues is the hashing race between miners. Over
time miners have upgraded from CPUs to GPUs to ASICs in order to increase their
hashing output. This has contracted the market of miners, putting the network at-
risk. To combat this issue, Ethereum’s block hashing algorithm was changed from
SHA2 to an ASIC-resistant version of SHA3 [10]. To implement electronic contracts,
Ethereum implements the Ethereum Virtual Machine (EVM) in place of Bitcoin’s
stack language. EVM is Turing Complete, allowing arbitrary programs to execute on
the network. To prevent programs from running indefinitely, each program is charged
the cost of its execution steps. The verification of valid EVM programs has been hard
to implement in practice [19].

Blockchain systems are difficult to evaluate. Cryptocurrencies such as Bitcoin
and Ethereum have noted issues with high-transaction cost [79], illegal activities [31],
and market-volatility [79, 129]. Furthermore, the word Blockchain is used to market
alternative products. Hyperledger [148] is a series of industry-sponsored projects that
are marketed as private blockchains. However, most of these projects are append-only
BFT systems. An append-only BFT does not use POW and users are verified.

2.2.3 Applications of P2P Networks to Health Records

Both peer-to-peer and blockchain applications have been previously proposed.

King [76] proposed a peer-to-peer PHR utilizing Blue Button Plus and FHIR.
The system uses a client-server architecture and a series of subnetworks. Peers can

\(^4\)A Turing complete language can approximate any Turing machine (an idealized computer), or
equivalently, create any algorithm that can run on a Turing machine [87]. A Turing complete
language does not comment on the difficulty of implementation or the computational cost of a target
program. A non-Turing complete language has limitations in what programs can be represented and
in the case of bitcoin, is purposefully chosen to prevent unsafe programs.
request access to a patient’s subnetwork. The architecture proposed in this thesis extends this previous work by providing full PHR functionality as well as integration with third-party service providers.

Roehrs [135] proposed a distributed PHR called OmniPHR. It uses the Chord algorithm to manage a series of encrypted blocks of health records over a set of peers. Each block is digitally signed. Each peer translates the records into open standards, distributes the blocks to peers, validates a peer’s blocks, and manages communication between peers. Roehrs tested the system’s network dynamics by increasing the number of nodes and “backbone” routers. He found that the Chord algorithm scaled with the number of peers and did not have increased latency. Roehr’s system is similar to the PHR proposed in this thesis. However, Roehrs is focused on the performance analysis of the Chord algorithm. The focus of this thesis is on the functionalities needed to facilitate interactions between patients and health providers.

Xia [167] proposes a cloud-based health record sharing system that uses a Blockchain style ledger to track transfers of health records. Peers upload health record data to a shared cloud repository. Peers request and grant access to records through a transaction. A node selects transactions and computes a proof-of-work hash. After the proof-of-work is completed, the request is granted. An exchange protocol is then used to access the shared repository. This system benefits little from the Blockchain solution. Another issue of this solution is that data is stored off chain which removes the benefit of a Merkle tree [100]. Indeed, the transactions and hashes of the data are increasingly secure, but the data itself is not. The participants of the network need to trust the hash of the data and cannot verify it until reading, at which point it could be corrupted.

Urovi [155] proposed a Peer-to-Peer framework to exchange records across HIEs and the use of semantics to enable dynamic queries. The study explores integrating the Swiss healthcare system without reliance on a central authority. The model has a few limitations. For example, a provider needs predefined knowledge on where target records are stored and changes to a record are not propagated to providers. Com-
munication is based on a tuple space architecture called TuCSoN [121], a derivative of Linda [51]. A tuple space is a distributed shared memory programming paradigm that creates a series of shared objects between parties called *tuples*. The TuCSoN triple store ontology was modeled in OWL. This allowed the distributed objects to be defined and resolved at runtime. The underlying P2P architecture is a pure model that uses TOM P2P [21], a Kademlia DHT, for resource discovery. Each node, called a community, contains a replica of the tuple primitives with agents reacting to tuple changes. The implementation consists of three basic agents: log, update, and search. For each community, a single coordinator called a Policy Tuple Center is used to mediate the tuple actions. The available actions included connect, subscribe, and search. The system could query the network for a resource and then use the network’s response to access the record. The agents could also respond to network requests. Three types of subscriptions were modeled: events, service changes, and policy changes. The implemented model was tested on Amazon cloud. The network’s search complexity was in line with theoretical estimates but experienced extra latency as more communities subscribed. Overall, the use of tuple space offers a flexible distributed programming model whose definitions can be injected at runtime. However, several issues have been overlooked. For example, consensus, security, and data representation of current health domain ontologies have not been addressed.

### 2.3 Application Layer Protocol

Peer-to-peer networks are application protocols built on top of the transport layer of the OSI model [78]. Programs exchange data in the application layer in a known format and exchange pattern. Example application layer protocols are HTTP [46, 78] and FTP [78, 131]. The underlying transport layer delivers packets between applications. There are two common methods for packet delivery in the transport layer, TCP and UDP.
TCP is a connection-oriented protocol between a client and a server [8, 78]. It provides error-corrected and in-order delivery of packets. TCP provides a guarantee of the delivery of a message. A connection under the TCP protocol is a safe and reliable method to transfer data, at the cost of increased overhead.

UDP is a connection-less protocol [78, 130]. The sender sends packets to the receiver in a best-effort approach. The protocol cannot guarantee the arrival of a packet. The application protocol must re-request missing packets. The receiver verifies the packets with a checksum. This makes UDP suitable for real-time applications where low latency is a priority. It is up to the application layer to provide any additional guarantees.

2.4 EHR Datamining

Predicting a patient’s future medical conditions is beneficial to patients, providers and the healthcare system. Patients can take proactive measures to prevent conditions, providers can recommend tests to perform, and the healthcare system can plan for future health needs. Identifying at-risk patients before conditions occur can lower the cost of medicine and saves lives. Current clinical research uses either randomized trials [32, 75] or cohort studies [32, 33, 96] to understand conditions and symptoms. This supports the approach of diagnosing patients by domain experts and Computer-aided Diagnosis (CAD) [39]. However, these methods are expensive, difficult to design, and suffer from selection bias [32, 75, 96].

Health providers have generated millions of patient records. Providers primarily use these records to treat patients and create workflows. This collection is valuable as it contains detailed health information for a population of patients rather than a small group. Researchers have proposed to use this wealth of data for the secondary purposes of clinical research, prediction systems, and clinical decision systems [34, 70, 108]. Present research into EHR datamining has had mixed results due to varying quality, encoding differences, and missing data [60, 159]. Researchers have proposed
the standardization and quality assessment of EHR data to overcome these issues [17]. Despite the problems with mining EHR repositories, researchers have conducted several large-scale studies [102]. They have used a variety of methods, such as Neural Network, Bayesian Statistics, Trees, and K-Nearest Neighbor [169]. However, medical institution control of the research data [90] makes evaluating these studies difficult.

Yadav [169] surveys the current research in EHR datamining. Applications of EHR datamining include: medical trajectories, cohort studies, risk prediction, intervention modeling, clinical guidelines, and detecting adverse events. Each of these applications have varying levels of implementation difficulties, with intervention modeling being notably difficult. EHRs contain three forms of health data with decreasing structure: structured, workflows, and clinical notes. Issues with EHR data include: difficulty in extraction, unobserved data, data fragmentation, bias, and irregular data.

Several recent papers propose utilizing Neural Networks to mine EHRs. Recent Neural Network approaches focus on many nested hidden layers of “neurons” to approximate complex functions. Given enough data, neural networks show a good performance on a variety of health modeling problems. Miotto et al. [102] and Trang et al. [128] approach EHR datamining from an unsupervised perspective with Neural Networks.

Miotto et al. [102] presents a large-scale study using Stack Denoising Autoencoders (SDA) and Random Forest to diagnose conditions such as diabetes, congestive heart failure, and schizophrenia. The authors choose autoencoders to counteract noise in EHR data and create a reduced representation of the patient records. The study used a heavily processed EHR repository of 700,000 patients. The authors annotated the dataset with the Open Biomedical Annotator [72] and extracted topics from clinical notes using regular expressions and Latent Dirichlet Allocation. EHR data needs heavy preprocessing to achieve good results [102] and the reduction of the feature space is key as EHR data has thousands of sparse features.
Trang et al. [128] proposed modeling a chronic patient’s long-term outlook using Long Short-Term Memory (LSTM) neural networks. A LSTM is a variation of Recurrent Neural Network in which nodes store internal lossy data to replicate memory. The authors used a modified LSTM to model long term health trends and improve on methods like Hidden Markov Model for chronic conditions [128]. The authors trained and tested the LSTM with 12 years of EHR data comprising two datasets, with each dataset covering approximately 12,000 patients and 53,000 hospital admissions. The authors noted that the model struggled with patients who had small records.

2.4.1 Predictive Models and Bayesian Statistics

In this thesis, a Bayesian model is used to demonstrate predictive medical diagnosis as a service. Bayesian Statistics are probabilistic methods that model systems using Bayes theorem [125]. Bayes Theorem (Eq. 2.1) is used to infer the probability of events based on prior observations. These methods can be classifiers or generative models.

\[
P(\text{Outcome}|\text{Data}) = \frac{P(\text{Data}|\text{Outcome})P(\text{Outcome})}{P(\text{Data})} \tag{2.1}
\]

where: \(P(\text{Outcome}|\text{Data})\) is the Posterior, \(P(\text{Data}|\text{Outcome})\) is the Likelihood, and \(P(\text{Outcome})\) is the Prior.

The simplest approach is Naive Bayes [136]. Naive Bayes is a classifier that models a set of features as random variables (RV) with the “naive” assumption that each is conditionally independent (Eq. 2.2). This simplifies learning and the inference process. Naive Bayes can perform well despite this simplification [136] and is used heavily in text mining [141].

\[
P(C, E_n) = P(C) \prod_i P(E_i|C) \tag{2.2}
\]

where: \([C] = \text{outcome}\) and \([E_n] = \text{evidence}\)
Another approach is Bayesian Network. Bayesian Network are Directed Acyclic Graph (DAG) with nodes representing RVs [136]. The space modeled is the joint distribution of the RVs but in a compact form. A connection between nodes indicates conditional dependency. The combined connections and probabilities create a model of the system representing interrelated events and their probabilities. By using Bayes Theorem and the graph structure, a query can traverse the network to infer the conditional probabilities of events given evidence. Bayesian Networks have advantages over Naive Bayes. The structure intuitively models the relationships between features and can represent complex models with conditional dependencies. Moreover, experts can use domain knowledge to define the graph structure, thereby bootstrapping the network. This is helpful as deriving the graphical structure of the Bayesian Network is otherwise NP-Hard [136].

Inferencing, computing the probabilities of an event given data, in complex Bayesian models is NP-Hard [136]. To overcome this issue, models can be trained using Expectation Maximization (EM) [136], Gibbs Sampling (a variation of Markov Chain Monte Carlo) [136], or Variational Methods [73]. The EM algorithm iteratively finds the probability distributions but can converge to local optima. Gibbs Sampling and Variational Methods use randomization or optimization techniques to approximate the distributions in order to avoid converging to a local optima.

2.4.2 Bayesian Applications to EHR Datamining

Lucas et al. [91] presents a review of Bayesian Networks for medical diagnosis. The representation of probabilistic knowledge as a graph creates a formalism that mirrors how medical domain experts see cause and effect relationships. It is an intuitive model, with easy to understand relationships and probabilities. Clinical experts can bootstrap the model allowing the application of prior diagnoses knowledge. Lucas demonstrates a theoretical application of Bayesian Networks to diagnose ventilator-associated pneumonia.
Sexias et al. [143] proposes using Bayesian Networks for Clinical Decision Support Systems targeting Dementia, Alzheimer’s, and Mild Cognitive Impairment. The authors reviewed similar CDSSs that used Bayesian Network, Neural Networks, and Multiple Criteria Decision Analysis. The study used 1,804 patient neuropsychological tests from four institutions for the dataset. The authors also used Synthetic Minority Over-sampling Technique (SMOTE) [30] to balance the dataset. Another processing step was the removal of attributes not related to diagnosis and records missing values. An expert predefined the network structure. The expert noted that several features that were removed for low counts were highly relevant to the classification. The authors generated five Bayesian Networks using the Waikato Environment for Knowledge Analysis (WEKA) software [57] along with the GeNiE Modeler and SMILE Engine [41]. The Bayesian Networks used a three-level template and the Dirichlet Distribution for the nodes. The expert-defined Bayesian network outperformed the auto-generated Bayesian Network but had similar performance to other methods. This study shows the benefit of using an expert to bootstrap the network.

Other approaches use Dynamic Bayesian Network (DBN) to account for time. Dynamic Bayesian Network are a generalization of Bayesian Networks and Hidden Markov Model that can model a system over time. Nachimuthu et al. [110] used DBNs to predict sepsis, a rapidly progressing condition in Intensive Care Units (ICU). The authors chose DBNs to account for the increase in evidence over time in an ICU. The study used 3,100 patients from an Intermountain Healthcare ICU dataset. The data was skewed towards control patients by 4 to 1 and was in various unstructured formats. The researchers extracted lab and vital signs for the first six hours. To reduce sparseness, the researchers aggregated the data to 1-hour intervals. The authors did not remove records with missing data and all variables were discretized. The input space to the model was a set of time increasing vectors of clinical data and observations for each patient. A physician labeled the data for sepsis diagnosis. The researchers created the DBN using Projeny, a front end to MATLAB’s Bayes Net Toolbox, and trained the network using the EM algorithm. The DBN structure was defined by
the researchers and used two time slices. Overall the network performed well, with all performance measures improving over time. Despite the promising performance, this study does have limitations. The authors noted that missing data prevented the diagnosis of specific types of sepsis such as septic shock.

Xu et al. [168] propose using a Hierarchical Bayesian Non-Parametric (BNP) to model patient’s responses to treatment. Patients have varied responses to treatments over time which motivated Xu to consider BNPs. The stated novel aspects were a continuous response over time, flexibility in modeling, and probability estimates at the patient level. The paper notes that longitudinal EHR data has the “time-varying confounding issue”, where treatments are present at a higher rate in sicker patients. This creates the impression that treatment caused the condition. Xu used the G-computation formula as a guide for creating the BNP. The dataset consisted of EHR time series data of 123 patients from the Boston Beth Israel Deaconess Medical Center. The patients averaged a 20.75-day stay and totaled 6,992 observations. The BNP modeled the patient’s kidney function in response to Ischemic Heart Disease treatments and Continuous Renal Replacement Therapies. The authors trained the model using Gibbs Sampling and Metropolis-Hastings, another MCMC method. Results were measured with Root Mean Squared Error (RSME) for Creatinine (a measure of renal health) and compared against baseline responses from the model using different distributions. The model performed well against fully and partially observed data. Overall, this paper presents interesting ideas related to modeling noise, using BNPs for time series data, and the issue of cause and effect in longitudinal data. Gaps in this paper were the failure to examine the scalability of the BNP method and effects of the small population size on the produced results.
2.4.3 EHR Datasets

Current EHR repositories have known issues with quality and sparseness. Weiskopf et al. [158] and Hersh et al. [60] reviewed the quality of EHR data. Weiskopf [158] examined 3.9 million patient records from the New York Presbyterian Hospital for completeness. The study tested records for documentation, breadth, number of visits, and predictive quality using logistical regression. The authors found that clinical notes increased after EHR adoption. Many records were sparse, only 45% included diagnosis and 13% included medication. The repository had limited longitudinal value, only 23% of the records had 5 follow-up visits after the initial visit. The author concludes that only 0.6% of the records are complete but recommends not generalizing the results to other hospital systems. Hersh [60] summarized the EHR issues from multiple studies. The author found common themes of inaccurate data, incomplete records, poor semantic encoding, data unrecoverable for research, unknown provenance, insufficient granularity, and incompatibility with research protocols. Both authors paint a dim outlook on using EHR data for clinical research. Moreover, Hersh [158] notes the limited usefulness of the International Classification of Diseases Revision 9 Clinical Modification (ICD-9-CM) [112] for studies.

The Health Insurance Portability and Accountability Act of 1996 (HIPAA) protects patient medical records from unauthorized access [156]. The majority of the reviewed literature on EHR datamining used a restricted dataset provided by a hospital. Yadav et al. [169] mentions that large scale studies before EHRs, used insurance claims data. Several open datasets target single conditions. Delen et al. [38] used the Surveillance, Epidemiology, and End Results Program (SEER) for predicting breast cancer survival rates. However, SEER is not a substitute for an EHR as it only covers cancer statistics in the United States. Xu et al. [168] used an open ICU EHR dataset, however, it does not include long term changes.

The Medical Expenditure Panel Survey (MEPS) [4] is a yearly survey of households by the Agency for Healthcare Research and Quality (ARHQ). ARHQ’s goal is
to survey current patient healthcare in the United States. In each survey, ARHQ asks households questions about conditions, providers, costs and medication. Each survey group is followed over a two-year period termed a panel with questions over 5 rounds (Fig. 2.2). There are currently 20 panels available, starting from 1996. Each panel includes approximately 15,000 households with a subset of the previous year’s respondents carried over. ARHQ warns that MEPS data is missing values, which affects the overall quality of the data. For example, in Panel 19, only 50% of the participants responded. ARHQ made attempts to address this issue by altering the survey design but the result is inconclusive. The questions and recorded information vary over the years.

ARHQ hosts the surveys online and provides the data files in SSP and fixed length format. Each survey is by year with two active panels per year [6](Fig. 2.2). ARHQ groups the information by household and event components. The household component covers survey questions while the event component tracks patient reported medical events. The household component contains tables covering topics such as conditions and medications. The link between the tables is the respondent’s DUPERSID, a concatenation of the respondent’s ID and the household ID.

Fig. 2.2.: MEPS Panel Design
3. ARCHITECTURE

The architecture (Fig. 3.1) of the proposed system is a centralized Peer-to-Peer model, consisting of: an index server, a peer client, and registered network services. The peer client is an application with a Graphical-User-Interface (GUI) for patients and providers to manage health records and interact with the network. The index server controls network access and provides resource lookup for the peers. Once connected to the network, a peer can post document metadata and request transactions. The target peer can then authorize the transaction. Peers periodically pull authorized transactions from the network. A peer carries out the transaction by directly contacting the target peer.

The user stores their health records on their own computer. Each new record has associated metadata (Fig. 3.2). The metadata includes the origin, creation time, and a SHA256 [9] hash of the data to establish its uniqueness. It is this metadata that is uploaded to the network for discovery, not the underlying record. Access is only granted by following the established transaction protocol and with the approval of the owner. This makes the user in control of their health records and the system a Personal Health Record.

The index server manages the network for the peers and performs the signaling, signup and lookup services for the network. This centralized model tracks the registered users, peer IP addresses, documents, transactions, and services.

Patients and Providers use the peer client to register in the network, login, manage local health records, lookup documents and services, upload records into the network, and issue requests for transactions. In addition, the client operates a P2P client-server subsystem to carry out approved transactions.

A service client is a special peer client for network services. Regular peers can request these services, which are logged and carried out as transactions. The service
Fig. 3.1.: System Architecture
client periodically checks the network for new service transaction requests and approves them. The service client does not store any records. It returns the result of the service application to the requesting peer.
3.1 Network Actions

There are several types of peer actions in the network.

Registration: The peer client presents network actions as a series of pages. A user can register in the network by filling out the registration information (Fig. 3.3). The peer client parses the form information and creates a registration message which is submitted to the network. The server validates this information, and if valid, a new account is created and a success response is returned to the client. Otherwise, an error is returned.

![Fig. 3.3.: Registration](image)
Login: The user logs into the network by entering their credentials. The client parses the information and creates a new network client. This network client creates a heartbeat process (Fig. 3.4) which continuously pings the server with route information on a predefined interval. Other peers use this route information to locate other clients.

Fig. 3.4.: Heartbeat Process
Search: The client can search for network resources (Fig 3.5). Resources include account information, posted documents, transactions, portals, and services. Each query can be refined to narrow the search. The client parses the query and sends the result to the index server. The index server validates the query, executes it, and returns the results of the search to the client.
Record Download: The client can download data from external sources. The index server holds a list of verified portal servers. The client queries the index server for this list and selects the desired portal. The user enters their portal login information and the client parses and creates a FHIR query. The client executes the query against the portal which returns the user’s documents (Fig. 3.6). If the query fails, the user is alerted. The downloaded documents are then compared against the local records. If the documents are the same, the record is not added to the database. Otherwise, a record is created and added to the peer database.

Fig. 3.6.: Fetch FHIR Documents from Portal
Record Registration: A user can register a document in the network (Fig. 3.7). The client parses the user’s selection and grabs the requested document’s metadata. The client sends the metadata to the index server. The index server validates the request and checks that no network document has the same SHA256 as the posted document. If the request is valid, then the document is assigned to the user. Otherwise, the request is rejected.

![Record Registration Diagram](image-url)
Service Registration: A service provider can issue a request for the registration of a new service by sending a registration request to the network (Fig. 3.8). The network administrator checks the request and approves or denies the new service. If the service is approved, a new peer client is created for the service and the service becomes visible to the peers.

Fig. 3.8.: Registration and Approval of Service
3.2 Transactions

Transactions are exchanges of records between two peers. Each type of transaction is treated as a state machine, with the state of the transactions changing as conditions are met. The index server acts as the ground truth between the two peers and ensures that transactions are both valid and consistent. In addition, the index server provides an independent log of the transaction for audit purposes.

Fig. 3.9.: Network Transaction
3.2.1 Authorization and Operation

A peer, called the requester, can issue a transaction request on a network document to the index server. The client posts the document hash and the requested action to the server (Fig. 3.10). The server validates the request and creates a new transaction in the pending state (Fig. 3.11). The target peer can view the list of pending transactions. He or she can then approve, reject, or ignore these requests and posts the decisions to the index server. The server validates the peer and if the target approved the transaction, moves the transaction to the next state. If the target rejects the request, the state is changed to canceled and the transaction is non-executable by the clients.

Transaction Request {
   Action     string
   ServiceID  string
   UserID     string
   DocumentID string
}

Fig. 3.10.: Transaction Request Data

Fig. 3.11.: Transaction Request and Approval

Once a transaction is approved, the participating clients can pull the transaction from the server (Fig. 3.12). The transaction is added to each client’s list of transactions that need to be executed.
Transaction {
  ID        string
  Hash      string
  SenderID  string
  SenderIP  string
  ReceiverID string
  ReceiverIP string
  Port      string
  Action    string
  State     string
  Time      time
}

Fig. 3.12.: Transaction Data

Update {
  Tid      string
  Type     string
  Data     rawdata
}

Fig. 3.13.: Update Data
To execute transactions, the peers operate both as a client and a server. The requester initiates the connection with the target peer. The two peers carry out the protocol and state of the transaction. At the end of the action, the peers alert the index server of the result with an update message (Fig. 3.13). The index server, also operating as a state machine, checks the messages to see if a state progress condition is satisfied. If a transition condition is satisfied, the index server moves the transaction to the next protocol state. Otherwise, a failure occurs, and the clients will need to retry the transaction. The clients continue executing the transaction until the protocol is completed or an error occurs.

3.2.2 Protocol

The method of the transaction determines the type of state machine that is invoked. Each state machine is a template with the following states: pending, final, failed, and canceled. The pending state shows that the owner has not approved a transaction. The final state indicates the end of a successful transaction. The failed state indicates that a transaction failed. The canceled state shows that the owner of the underlying data canceled the transaction. All other states are variable and depend on the protocol. There are three protocols modeled in the network: Get, Push, and Service.
Get: The *Get* protocol transfers a document from the owner to the requesting peer (Fig. 3.14). It has three states: *pending*, *get*, and *final*. While waiting for approval, the transaction is in the pending state. The owner must approve or disapprove a transaction before the transaction can move to the next state. Once approved, the transaction is moved to the *get* state. The requester connects to the owner. The owner returns the approved document to the requester. The two clients alert the index server that the transfer is complete. The index server sets the transaction to *final* state, thus completing the transaction.

![Fig. 3.14.: Get Protocol](image-url)
Push: The *Push* protocol transfers a document from the owner to the accepting party (Fig. 3.15). It has three states: *pending*, *push*, and *final*. A push transaction starts in the *pending* state. The target approves the transaction and the index server moves it to the *push* state. The owner of the document opens a connection with the target and transfers the document. The target adds the record to their data store. The two peers then alert the index server that the transaction is complete. The index server moves the transaction to the *final* state, thus completing the transaction.

![Fig. 3.15.: Push Protocol](image-url)
Service: The *Service* protocol sends a document from an owner to a service and the service returns a new document in response (Fig. 3.16). It has three states: *waiting*, *service*, and *final*. The transaction starts in the *waiting* state. The service approves the transaction and the index service changes its state to *service*. The owner connects to the service and sends the agreed upon document. The service performs the requested action and returns the result to the owner. The owner adds the new document to their data store. The two peers alert the index server that the transaction is complete. The index server places the transaction in the *final* state.

Fig. 3.16.: Service Protocol
3.3 Hypertension Prediction Service

The Hypertension prediction service is a network service that runs a Naive Bayes classifier. The classifier predicts a patient’s risk of developing hypertension in the next year based on the patient’s historical health record. The classifier is trained with data from the Medical Expenditure Panel Survey (MEPS) [4]. Though not a patient health record, the MEPS dataset is used to demonstrate the proposed approach and service protocol. The classifier targets Hypertension (High Blood Pressure), ICD-9-CM Code 401 [112]. Hypertension was chosen as a demonstration of the proposed service as it is the highest occurring condition in MEPS.

The prediction service has two stages: development and deployment. During the development stage, the classifier is trained and tested using a MEPS dataset. The model is then deployed as a service in the proposed PHR.

Records are extracted from MEPS and transformed into a vector of patient demographics, prior conditions, and labels of posterior conditions. The selected features included 30 related ICD-9-CM conditions and 17 health related survey questions. This dataset was split into training and test datasets, with 70% of the records for training and the remaining 30% for testing. The output of the classifier is a binary prediction that the patient will develop Hypertension over the next year.

The Hypertension prediction classifier is then added to the network and operates as the Hypertension prediction service. A peer can connect to the service and pass the target health record which conforms to the classifier’s requested format. The service invokes the classifier on the supplied data and returns the result to the client as a new record.
3.3.1 Data Set

The dataset used to train and test the Hypertension Prediction Classifier is comprised of two MEPS tables: consolidated and conditions. The consolidated table contains the bulk of the respondent’s information, including demographics, insurance, income, etc. The condition table contains the self-reported conditions coded in ICD-9-CM for each respondent. To protect respondents, ARHQ only recorded the first three characters of ICD-9-CM [5], dropping the number of codes from 13,000 to less than 1,000. ARHQ further obfuscated the information of children and at-risk groups.

The highest occurring condition in Panel 19 is Hypertension (ICD-9-CM 401) with 3,078 instances. The high counts of Hypertension and accessibility to heart-related information motivated the choice of Hypertension as a target disease for this thesis. A review of heart condition literature helped prune the list of conditions to 30 ICD-9-CM codes covering heart and respiratory conditions (Table 3.1). The consolidated table has few direct measures of health such as Body Mass Index (BMI) [12]. However, several questions on advice from doctors, major health events such as stroke, and smoking information are also available [5]. The respondent’s answers to these questions are used as additional measure of the respondent’s cardiovascular health. This resulted in an additional 17 features. The selected features are included in Table 3.2.

Table 3.1.: Selected Condition Features

<table>
<thead>
<tr>
<th>ICD-9-CM Group</th>
<th>Meps Code</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>410-414: Heart Disease</td>
<td>410</td>
<td>Acute Myocardial Infraction</td>
</tr>
<tr>
<td></td>
<td>413</td>
<td>Angina Pectoris</td>
</tr>
<tr>
<td></td>
<td>414</td>
<td>Other forms of Ischemic Heart Disease</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>ICD-9-CM Group</th>
<th>Meps Code</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>415-417: Heart Failure</td>
<td>415</td>
<td>Acute Pulmonary Heart Disease</td>
</tr>
<tr>
<td>420-429: Circulatory System Diseases</td>
<td>424</td>
<td>Other Diseases of Endocardium</td>
</tr>
<tr>
<td></td>
<td>425</td>
<td>Cardiomyopathy</td>
</tr>
<tr>
<td></td>
<td>427</td>
<td>Cardiac Dysrhythmis</td>
</tr>
<tr>
<td></td>
<td>428</td>
<td>Heart Failure</td>
</tr>
<tr>
<td></td>
<td>429</td>
<td>Ill-defined Descriptions and Complications of Heart Disease</td>
</tr>
<tr>
<td>430-438: Cardio Brain Hemorrhage, Stroke</td>
<td>436</td>
<td>Acute, but ill-defined, Cerebrovascular Disease</td>
</tr>
<tr>
<td>440-449: Restricted Blood Flow</td>
<td>440</td>
<td>Atherosclerosis</td>
</tr>
<tr>
<td></td>
<td>441</td>
<td>Aortic Aneurysm and Dissection</td>
</tr>
<tr>
<td></td>
<td>442</td>
<td>Other Aneurysm</td>
</tr>
<tr>
<td></td>
<td>443</td>
<td>Other Peripheral Vascular Disease</td>
</tr>
<tr>
<td></td>
<td>444</td>
<td>Arterial Embolism and Thrombosis</td>
</tr>
<tr>
<td></td>
<td>447</td>
<td>Other Disorders of Arteries and Arterioles</td>
</tr>
<tr>
<td>451-459: Issues with Veins and Lymph System</td>
<td>454</td>
<td>Varicose Veins of Lower Extremities</td>
</tr>
<tr>
<td></td>
<td>455</td>
<td>Hemorrhoids</td>
</tr>
<tr>
<td></td>
<td>458</td>
<td>Hypotension</td>
</tr>
</tbody>
</table>

*continued on next page*
Table 3.1.: continued

<table>
<thead>
<tr>
<th>ICD-9-CM Group</th>
<th>Meps Code</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>459</td>
<td>Other Disorders of Circulatory System</td>
</tr>
<tr>
<td>490-496: Chronic Respiratory Issues</td>
<td>490</td>
<td>Bronchitis, not specified as Acute or Chronic</td>
</tr>
<tr>
<td></td>
<td>491</td>
<td>Chronic Bronchitis</td>
</tr>
<tr>
<td></td>
<td>492</td>
<td>Emphysema</td>
</tr>
<tr>
<td></td>
<td>493</td>
<td>Asthma</td>
</tr>
<tr>
<td></td>
<td>496</td>
<td>Chronic Airway Obstruction, not elsewhere classified</td>
</tr>
<tr>
<td>510-519: Secondary Respiratory Conditions</td>
<td>511</td>
<td>Pleurisy</td>
</tr>
<tr>
<td></td>
<td>514</td>
<td>Pulmonary Congestion and Hypostasis</td>
</tr>
<tr>
<td></td>
<td>518</td>
<td>Other Diseases of Lung</td>
</tr>
<tr>
<td></td>
<td>519</td>
<td>Other Diseases of Respiratory System</td>
</tr>
</tbody>
</table>

ARHQ conducted panels 17, 18, and 19 from 2012 to 2015 and the related data is stored in the H15 to H18 datasets. The dataset covers 26,563 patients. The split of each dataset creates a natural break point to test prior and post conditions. The first half of the panel (i.e., rounds 1 and 2, and the first part of round 3) is taken as the set of features to predict hypertension. The second half (i.e., second part of round 3 and rounds 4 and 5) is used to build the outcome label of whether the patient contracted Hypertension. Not all patients fully responded to the second half of the survey which reduced the usable vectors. The conditions and consolidated datasets for each panel.
Table 3.2.: Selected Consolidated Features

<table>
<thead>
<tr>
<th>MEPS Feature</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEX</td>
<td>Sex</td>
<td>Discrete</td>
</tr>
<tr>
<td>RACE1VX</td>
<td>Race (edited/imputed)</td>
<td>Categorical</td>
</tr>
<tr>
<td>AGELAST</td>
<td>Person’s age last time eligible</td>
<td>Categorical</td>
</tr>
<tr>
<td>HIBPDX</td>
<td>High Blood pressure diagnosis</td>
<td>Continuous</td>
</tr>
<tr>
<td>CHHDX</td>
<td>Coronary Heart Disease diagnosis</td>
<td>Discrete</td>
</tr>
<tr>
<td>ANGIDX</td>
<td>Angina diagnosis</td>
<td>Discrete</td>
</tr>
<tr>
<td>MIDX</td>
<td>Heart Attack (MI) Diagnosis</td>
<td>Discrete</td>
</tr>
<tr>
<td>OHRTDX</td>
<td>Other Heart Disease Diagnosis</td>
<td>Discrete</td>
</tr>
<tr>
<td>STRKDX</td>
<td>Stroke Diagnosis</td>
<td>Discrete</td>
</tr>
<tr>
<td>EMPHDX</td>
<td>Emphysema Diagnosis</td>
<td>Discrete</td>
</tr>
<tr>
<td>CHOLDX</td>
<td>High Cholesterol Diagnosis</td>
<td>Discrete</td>
</tr>
<tr>
<td>DIABDX</td>
<td>Diabetes Diagnosis</td>
<td>Discrete</td>
</tr>
<tr>
<td>ADSMOKE42</td>
<td>SAQ: Currently Smoke</td>
<td>Discrete</td>
</tr>
<tr>
<td>NOFAT53</td>
<td>Restrict High Fat/Cholesterol</td>
<td>Discrete</td>
</tr>
<tr>
<td>EXRCIS53</td>
<td>Advised to exercise more</td>
<td>Discrete</td>
</tr>
<tr>
<td>BMINDX53</td>
<td>Adult Body Mass Index</td>
<td>Continuous</td>
</tr>
<tr>
<td>POVCAT</td>
<td>Family Income as Percentage of Poverty Line</td>
<td>Categorical</td>
</tr>
</tbody>
</table>
are pulled from the ARHQ website and loaded into SAS [137]. SQL joins are then used to extract the targeted panels and features across the datasets.

### 3.3.2 Preprocessing

The compiled dataset is further processed (Fig. 3.17) with the Python Data Analysis Library (pandas) [97]. A custom pipeline converts the compiled prior to posterior pairs into a vector of demographic information, conditions prior the time split (i.e., historical information), and the conditions post time split (i.e., predictive outcome) for each patient. It consists of three stages: feature vector creation, label vector creation, and feature cleanup.

![Diagram](image)

**Fig. 3.17.** MEPS Data Preprocessing Pipeline

The first stage converts the feature vector and labels. It compresses the list of conditions for each respondent into a single vector (Alg. 3.1

The next stage compresses the labels into a single vector (Alg. 3.2). Labels are prepended with L to differentiate labels from features.

---

1 Error handling has been removed from all algorithm due to length.
The final preprocessing step converts the features to categorical values and replaces unknown responses with default values. BMI is converted into the standard ranges of underweight, normalweight, overweight, and obese [12]. The age is converted into ranges of 10 years. Unknown values for demographic questions are converted to either no or to an empty variable depending on the count.

The resulting dataset had 26,653 vectors. The ICD-9-CM features and labels are binary. The other features are categorical or binary. Missing prior conditions or labels are set to 0. This may cause under-reporting. However, this was unavoidable since MEPS does not have definite no answers for many cases. The feature vector layout is shown in Figure 3.18 and an example feature vector is included in appendix B.
Fig. 3.18.: Feature Vector

The dataset is randomly divided into training and test data. The split is 70%, or 18,657 records, for training and the remaining 30%, 7,996 records, for testing.

### 3.3.3 Algorithm

The proposed classifier uses Naive Bayes (Eq. 2.2). As mentioned in section 2.4.1, Naive Bayes is a supervised learning algorithm that requires labeled data. It is a restricted Bayesian Network with the assumption of conditional independence between features. This assumption makes inference of an outcome given evidence a product of the conditional probabilities of evidence. After inferring the probability of each event given the evidence, the most likely result, called the *maximum a posteriori* (MAP) [136], is selected as the prediction outcome.

### 3.3.4 Training and Testing

The model’s accuracy is evaluated with hypothesis testing. Hypothesis testing creates a 2x2 matrix of the results of the binary classification. This matrix is sometimes called a confusion matrix. It contains the true positive (TP), false positive (FP), false negative (FN), and true negative (TN) counts of the prediction versus the label [80, 136]. High FP or FN rates may be unacceptable in certain environments (e.g., cancer diagnosis). The confusion matrix is aggregated to derive the Precision (Eq. 3.1), Sensitivity (Eq. 3.2), and Specificity (Eq. 3.3) of the model. The Precision, also called the positive predictive value (PPV), measures the performance of classifying a true positive given a positive result. The Sensitivity measures the performance of correct identification of positive results. The Specificity measures the classification of events that return negative results.
\begin{align*}
\text{Precision} &= \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}} \quad (3.1) \\
\text{Sensitivity} &= \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \quad (3.2) \\
\text{Specificity} &= \frac{\text{TrueNegative}}{\text{TrueNegative} + \text{FalsePositive}} \quad (3.3)
\end{align*}
4. IMPLEMENTATION

The network server, peer client, and service client are implemented with Go (Golang) [53] and MongoDB [105]. Libraries used include mgo [114] and the FHIR intervention engine [103]. The Hypertension classifier subsystem is implemented with Python3 [132], SAS [137], and pandas [97].

4.1 Index Server

The index server supports account management, resource lookup, and transaction management (Fig. 4.1). It is programmed with Golang and uses MongoDB for persistent storage. Clients access the service through a Representational State Transfer (REST) [47] API.

4.1.1 Database

The network server connects to a MongoDB [105] server though the mgo [114] driver. MongoDB stores all documents in BSON [107], a binary JSON format. Tables in a MongoDB are called collections [106]. The database has the following collections: user, document, transaction, service, and FHIR (Fig. 4.2).

The user collection holds the account information of the network users. The user’s email address is the primary key.

The document collection holds the metadata of the documents that users have posted to the network. The document contents are not posted to the server. The hash field is the unique SHA256 hash value derived from the document’s contents. The UID field is a foreign key to the user collection to link the document to the owner.
Fig. 4.1.: Index Server Classes
### User

<table>
<thead>
<tr>
<th>ID</th>
<th>ObjectID PK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>String</td>
</tr>
<tr>
<td>Password</td>
<td>String</td>
</tr>
<tr>
<td>Email</td>
<td>String</td>
</tr>
<tr>
<td>VerCode</td>
<td>String</td>
</tr>
<tr>
<td>Verified</td>
<td>Boolean</td>
</tr>
<tr>
<td>HeartBeat</td>
<td>Time</td>
</tr>
<tr>
<td>IP</td>
<td>String</td>
</tr>
<tr>
<td>Port</td>
<td>String</td>
</tr>
<tr>
<td>Role</td>
<td>String</td>
</tr>
</tbody>
</table>

### Transaction

| ID | ObjectID PK |
| DocHash | Sha256 FK |
| SenderID | String FK |
| ReceiverID | String FK |
| Action | String |
| State | Boolean |
| Time | Time |

### Document

| ID | ObjectID |
| UID | String FK |
| Hash | Sha256 FK |
| Time | String |
| Origin | String |

### Service

| ID | ObjectID PK |
| UID | String FK |
| Info | String |
| DataType | String |
| Approved | Time |

### Portal

| ID | ObjectID PK |
| Type | String |
| Org | String |
| URL | String |
| Time | Time |

---

**Fig. 4.2.: Database Collection**
The transaction collection holds the network transactions. Each transaction record contains the state, requested data, and actors in a transaction. The actors, senderID and receiverID, are foreign keys to the user database. The hash field is a foreign key to the requested document. The ID field is a unique key assigned by the database to identify a transaction, allowing users to reissue a transaction without duplication.

The service collection holds the approved and unapproved services in the network. Network clients can only interact with approved services. A service record contains information on the data needed by the service and the foreign key UID of the linked peer who operates the service.

The portal collection holds connection information and details on FHIR portals provided by health providers.

4.1.2 API and Handlers

The Index server provides its services as a REST API. REST is a design pattern for network services in which resources are URL paths that are called using the standard HTTP routines (e.g. GET and POST). The network resources are organized as a hierarchy (Fig. 4.3), for example, the /network path includes resources for logged in clients.

Fig. 4.3.: REST Access
Contacting a REST endpoint invokes a series of HTTP handlers. The handlers evaluate a request and interact with the database. The full communication details and routines between clients and the index server are in section 4.4.

4.2 Peer Client

A high-level diagram of the peer client system components and their interactions is in Figure 4.4.

The Peer Client database is implemented in MongoDB. It has two collections: Record and PeerRecord (Fig. 4.5). Each collection is independent. The record collection holds the set of FHIR records pulled from portal servers. Each record consists of the FHIR record data following the FHIR V3 standard and the metadata which is generated when adding the record to the database. The FHIR V3 golang framework is taken from the Intervention Engine [103], which is licensed under the Apache license, Version 2.0 [14].
<table>
<thead>
<tr>
<th>Record</th>
<th>PeerRecord</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>Time</td>
<td>ObjectID</td>
</tr>
<tr>
<td>Origin</td>
<td>UID</td>
</tr>
<tr>
<td>Data</td>
<td>TID</td>
</tr>
<tr>
<td>Hash</td>
<td>Time</td>
</tr>
<tr>
<td>Sha256</td>
<td>Origin</td>
</tr>
<tr>
<td>PK</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>Hash</td>
</tr>
<tr>
<td></td>
<td>Sha256</td>
</tr>
<tr>
<td></td>
<td>PK</td>
</tr>
</tbody>
</table>

Fig. 4.5.: Peer Client Collections
New records are added to the database with the `Database.NewRecord` function (Alg. 4.1). This function generates the metadata for the record to be added, including: the origin, date, and a SHA256 hash of the data. The database method `StoreRecord` attempts to push the target record to the database.

```go
func NewRecord(time time.Time, url string, resource interface{}) (Record, error) {
    var record Record
    buffer := json.Marshal(resource)
    hash_byte := sha256.Sum256(buffer)
    session := d.rootSession.Copy()
    defer session.Close()
    c := session.DB("client").C("record")
    c.Insert(record)
}
```

Algorithm 4.1: New Record

The peer record collection holds the records acquired from other peers through transactions. This collection is similar to `record` but includes the `UID` to indicate the original owner and the `origin` field holds the ID of the transaction that transferred the record.

The Peer Client accesses the network through the Network Client (Fig. 4.6). The Network Client wraps the network calls and keeps the client logged into the network. The Network Client’s interactions with the network are discussed in Section 4.4.

The client GUI is implemented as a Web User Interface (WUI). The user must start the client and then use a browser to connect to the WUI running on localhost. The application is laid out as HTML pages (Fig. 4.7), which are served to the user.

The root page provides access to the initial pages of registration, login, local records, and help. The registration page allows a user to register using a form (Fig. 4.8). After submitting the form, the WUI parses the form and invokes the network client. If the registration request succeeds, the user is redirected to the login page. On the login page, the client can submit their credentials (Fig. 4.9). The submitted form is parsed and a new network client is launched to connect to the network. This connected network client is necessary for network page access. It will run in the background until the user logs out. If the network client is not running, the WUI will redirect the user back to the login page.
Fig. 4.6.: Network and WUI Classes
Fig. 4.7.: WUI Paths

Registration

Name: 
Password: 
Email: 
Submit

Fig. 4.8.: Registration

Login

Name: 
Password: 
Submit

Fig. 4.9.: Login
The local records page renders a table of the health records in the local database (Alg. 4.2). Each row shows the record metadata. A client can click on any individual record to show the content of the record (Fig. 4.10). In addition, if the user is logged into the network, the user can request that the document metadata be posted to the network.

```go
func (d *Database) AllRecords() ([]Record, error) {
    session := d.rootSession.Copy()
    c := session.DB("client").C("record")
    var records []Record
    c.Find(nil).All(&records)
    return records, nil
}
```

Algorithm 4.2: Local Records

**Local Record:**
a54d1503a72385cdb74e9ca93ccee4ef26128e70d35074f02b1dca20e5bc4cf9f

**Metadata**
a54d1503a72385cdb74e9ca93ccee4ef26128e70d35074f02b1dca20e5bc4cf9f 2019-06-21 17:48:27.345 +0000 UTC http://bapi.fhir.org/baseDtu3

**Data**

```
{ "id": "9914869", "birthdate": [ "precision": "date", "time": "2018-12-16T05:00:00Z", "gender": "male", "meta": [ "lastUpdated": [ "precision": "timestamp", "time": "2018-12-16T04:36:00Z", "versionId": "0" ] ], "name": [ [ "family": "Walker", "given": [ "Austin" ] ], "text": "Austin Walker", "use": "official" ] ], "resourceType": "Patient", "text": [ "div": "a003ccx xmlns=http://www.w3.org/1999/xhtml\"a003ccx xmlns=\"http://www.w3.org/1999/xhtml\"a003ccx xmlns=" }, "status": "generated" ] }
```

**Network Options**

[Post to Server]

Fig. 4.10.: Record

The network menu lists pages that require network access. It is the base menu to other network actions. Accessing the account page triggers the network client to retrieve the client’s account information from the index server. The returned information is rendered to the client (Fig. 4.11).

**Account Info**

{ "id": "5cb7cc0ed144410ba97b4e5", "name": "John Doe", "role": "peer", "status": "unknown", "login-time": "2019-07-21T22:59:08.000Z" }

Fig. 4.11.: Account Page
The network document page triggers the network client to retrieve the list of documents in the network. The documents are rendered as a table of each record’s metadata (Fig. 4.12(a)). Each document is a link, which when accessed, invokes the network to retrieve the list of network services. The transaction options are rendered to the client and the user can choose a transaction option from the list (Fig. 4.12(b)). After selection, a request form for the transaction (Fig. 4.12(c)) is presented. Once the form is completed, the client sends the request to the network. If the request is successful, the client is redirected to the transaction page. Otherwise, the client receives a network error.
List of Network Records

<table>
<thead>
<tr>
<th>Id</th>
<th>Uid</th>
<th>Hash</th>
<th>Time</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>5d34e7175c13bee8b6535437</td>
<td>5cb7c0e6114f410b97b4e5 a54d1503a72385db74e9ca93cece4ef26128e70d350740023b6ca20e5bdccf6f</td>
<td>2019-06-21 17:48:27.345 +0000 UTC</td>
<td><a href="http://hapi.fhir.org">http://hapi.fhir.org</a></td>
<td>Transaction Request</td>
</tr>
<tr>
<td>5d34ecf75c13bee8b6535437</td>
<td>5cb7c0e6114f410b97b4e5 c519a9e6aebdf4615df8bfb2dfbe4e57a62a92f57f7a9f91eb898f4c12ca968</td>
<td>2019-06-21 17:48:27.345 +0000 UTC</td>
<td><a href="http://hapi.fhir.org">http://hapi.fhir.org</a></td>
<td>Transaction Request</td>
</tr>
<tr>
<td>5d34e8075c13bee8b6535437</td>
<td>5cb7c0e6114f410b97b4e5 3d8eeb89db694053ae23545b6897ed7e1e156fd33710ac908853625b489</td>
<td>2019-06-21 17:48:27.345 +0000 UTC</td>
<td><a href="http://hapi.fhir.org">http://hapi.fhir.org</a></td>
<td>Transaction Request</td>
</tr>
</tbody>
</table>

(a) Document List

Transaction Request

Record Metadata

{5d34e7175c13bee8b6535437 5cb7c0e6114f410b97b4e5 a54d1503a72385db74e9ca93cece4ef26128e70d350740023b6ca20e5bdccf6f 2019-06-21 17:48:27.345 +0000 UTC http://hapi.fhir.org/baseDstu3}

Basic Transactions

- Choose an option
  - Transfer
  - Push

Services List

<table>
<thead>
<tr>
<th>Service List</th>
<th>Id</th>
<th>Name</th>
<th>Uid</th>
<th>Heartbeat</th>
<th>Info</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>5cbf35247cd87aeebdf7eaf68 meps</td>
<td>5cbf35247cd87aeebdf7eaf65 2019-07-21 22:59:39.313 +0000 UTC info about the meps predictor heartbeat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) Transaction List

Service Request Form

Record Metadata

{5d34e7175c13bee8b6535437 5cb7c0e6114f410b97b4e5 a54d1503a72385db74e9ca93cece4ef26128e70d350740023b6ca20e5bdccf6f 2019-06-21 17:48:27.345 +0000 UTC http://hapi.fhir.org/baseDstu3}

Service Metadata

{5cbf35247cd87aeebdf7eaf68 5cbf35247cd87aeebdf7eaf65 info about the meps predictor heartbeat} { meps 2019-07-21 22:59:49.312 +0000 UTC}

Request Form

Enter your message here...

(c) Transaction Request

Fig. 4.12.: Network Documents and Transaction Request
The transaction page triggers the network client to retrieve the transactions in which the client is a participant. The results are rendered as a table for the user (Fig. 4.13).

**Fig. 4.13.: Transaction Page**

The user can view the list of records that have been transferred to the client on the peer record page. The peer record page renders the records in the `PeerRecord` collection as a table (Fig. 4.14).

**Fig. 4.14.: Peer Records Table**

Other network pages that the client can access include the service page and the survey page. The service page lists the active services in the network. The list is generated by a network client request to the index server. The survey page allows the client to fill out the equivalent MEPS survey data for the MEPS Hypertension service.
4.2.1 FHIR Client

An access of the portal page triggers a network request to fetch the list of portals maintained by the index server. The returned list is rendered to the client as a table. Each portal can be further accessed to make a request. A portal request allows the client to fill out a form to request records in accordance with the FHIR protocol. The form is submitted, parsed, and a FHIR query (Fig. 4.15) is constructed. The FHIR query is then sent to the target server (Alg. 4.3). The returned FHIR records are added to the database, except in cases where the record is not new.

![FHIR Query](http://hostname/Patient/{user}/$everything)

Fig. 4.15.: FHIR Query

```go
func GetPatientRecord(logger *log.Logger, conn fhir.Connection) ([]model.Record, error) {
    httpClient := &http.Client{Timeout: time.Second * 10}
    fhirClient := fhir.NewClient(logger, httpClient)
    entries := fhirClientFetchRequest(conn)
    time := time.Now()
    records := make([]model.Record, len(entries))
    for i, entry := range entries {
        records[i] = model.NewRecord(time, conn.Url, entry.Resource)
    }
    return records, err
}
```

Algorithm 4.3: Fetch FHIR Records

4.2.2 P2P Subsystem

The P2P subsystem manages the transactions with other peers (Fig. 4.16). It operates independently from the user’s input, with the user only controlling the launch and closing of the system. The P2P subsystem acts as a client and server, pulling and carrying out transactions from the network and serving transaction requests from other peers.
Fig. 4.16.: Peer-to-Peer Classes
The system polls the index server for new transactions with the `fetcher.updateJobs` routine (Alg. 4.4). These transactions are compared against the already running set of transactions, termed `jobs` (Fig. 4.17). If the transaction is new, then a new job is created by the factory `JobBuilder` class and stored in a hash table of jobs. Each job is given its own thread to prevent blocking the main client.

```
... ticker := time.NewTicker((api.HeartBeatInterval / 2) * time.Second)
for {
    select {
        case <-ctx.Done():
            ps.logger.Println("ending transaction update")
            return
        case <-ticker.C:
            err := ps.updateJobs(uid)
    }
}
...
func (ps *Fetcher) updateJobs(uid string) error {
    transactions := ps.netclient.RunningTransactions()
    err = ps.jboard.AddJobs(transactions)
    return nil
}
```

Algorithm 4.4: Fetch Jobs

A job’s operation is defined by the user’s role in the transaction. A user that is the requester will try to execute client-side transactions, while a receiver will wait for a client to connect before attempting a transition. A transition is a cycle of the current state of the protocol and any messages. Both operations implement a loop (Alg. 4.5) to poll the server for transaction updates, attempt transitions, and for sending updates back to the network or caller (Alg. 4.6 and 4.7). After polling the network for the transaction, the transaction’s state is examined. If the transaction state is in the final state, the job loop is exited. Otherwise a state transition is performed. The job sends the returned transition messages to the index server and if a server job, the client.
Fig. 4.17.: Job Classes
ticker := time.NewTicker(1 * time.Second)
for {
    select {
        case <-ctx.Done():
            return
        case <-ticker.C:
            run := j.ClientCycle() 'or' j.ServerCycle()
            if !run {
                return
            }
    }
}

Algorithm 4.5: Job Cycle

```go
func (j *Job) ClientCycle() (bool, error) {
    update := j.client.CheckTransaction(j.Tid)
    if update.State == "final" {
        return false, nil
    }
    temp := j.dispatch.ClientExecute(update)
    j.client.UpdateTransaction(temp)
}
```

Algorithm 4.6: Client Job

```go
func (j *Job) ServerCycle(message *api.Message) (*api.Message, error) {
    update := j.client.CheckTransaction(j.Tid)
    if update.State == "final" {
        return nil, errors.New("final")
    }
    sender, index := j.dispatch.ServerExecute(update, message)
    j.client.UpdateTransaction(index)
    return sender, err
}
```

Algorithm 4.7: Server Job
4.2.3 TCP Library

Peers use a small library to pass messages based on TCP (Fig. 4.18). The TCP library uses a standard Request-Response pattern utilizing JSON (Alg. 4.8, 4.9) which can pass arbitrary data. The client side is shown in algorithm 4.10 and the server side and handler are shown in algorithms 4.11 and 4.12. Each client registers a p2p listener which dispatches (Alg. 4.13) to the correct job when a peer connects. If the job is not present the connection is terminated.

Fig. 4.18.: TCP library
```go
// Algorithm 4.8: Request

type Request struct {
    Route string
    Type string
    Data json.RawMessage
}

// Algorithm 4.9: Response

type Response struct {
    Status string
    Type string
    Data json.RawMessage
}

... address := createAddress(c.ip, c.port)
tconn, err := net.DialTimeout("tcp", address, IdleTimeout)
c.tconn = tconn
...

func (c *Client) Do(request *Request) (*Response, error) {
    encoder := json.NewEncoder(c.tconn)
    decoder := json.NewDecoder(c.tconn)
    err := encoder.Encode(request)
    checkErr(err) // if err close connection, check if OpErr
    var response Response
    err = decoder.Decode(&response)
    checkErr(err) // if err close connection, check if OpErr
    return &response, nil
}

// Algorithm 4.10: Client Side TCP

func (c *conn) serve(cid int) {
    defer func() {
        c.tconn.Close()
    }()
    decoder := json.NewDecoder(c.tconn)
    encoder := json.NewEncoder(c.tconn)
    for {
        var request Request
        err := decoder.Decode(&request)
        checkErr(err) // close connection if err, check err.type
        response := c.mux.Dispatch(&request)
        deadline := time.Now().Add(IdleTimeout)
        c.tconn.SetDeadline(deadline)
        err = encoder.Encode(response)
        checkErr(err) // close connection if err, check err.type
    }
}

// Algorithm 4.11: Serve Connection
```
func (s *Server) ListenAndServe() error {
    listener := net.Listen("tcp", ":"+s.port)
    s.listener = listener
    go func() {
        var cLock sync.Mutex
        connMap := make(map[int]*conn)
        cid := 0
        OuterLoop:
        for {
            tconn, err := s.listener.Accept()
            checkErr(err) // break to OuterLoop if closed else continue
            tconn.SetDeadline(time.Now().Add(IdleTimeout))
            c := newConn(tconn, s.logger, s.mux)
            cLock.Lock()
            cid++
            id := cid
            connMap[id] = c
            cLock.Unlock()
            s.wg.Add(1)
            go func(conn_id int) {
                defer s.wg.Done()
                c.serve(id)
                cLock.Lock()
                delete(connMap, conn_id)
                defer cLock.Unlock()
            }(id)
            for _, v := range connMap {
                v.tconn.Close()
            }
        }
        s.wg.Wait()
        s.result <- nil
    }()
    return nil
}

Algorithm 4.12: Listen for Connection

func TransactionHandler(logger *log.Logger, board *Board) p2p.Handler {
    return func(r *p2p.Request) (*p2p.Response, error) {
        var message api.Message
        json.Unmarshal(r.Data, &message)
        tid := message.Tid
        response := board.HandleJob(&message)
        jsonData := json.Marshal(response)
        return &p2p.Response{"ok", "message", jsonData}, nil
    }
}

Algorithm 4.13: Server Dispatch
4.3 Service Client

The Service Client allows the execution of approved services for the network (Fig. 4.19). It implements similar functionalities to the Peer Client with several key differences. A service client only implements the Transaction Service Protocol. The Service Client has no local database. It only invokes the approved program. Additionally, on regular intervals it pulls all unapproved service transaction and approves them (Alg. 4.14). This approach allows the service client to remain independent of the index service while providing the network the necessary services on demand.

Fig. 4.19.: Service Client Subsystems

* Subsystem was modified from client version.
```go
func (a *Approve) approveTransactions() error {
    transactions, err := a.netclient.OpenTransactions()
    handleErr(err)
    for _, t := range transactions {
        err := a.netclient.OkayTransaction(t.ID)
        handleErr(err)
    }
    return nil
}
```

Algorithm 4.14: Approval All Transactions

A service registers with the network using the Network Client’s `ServiceRegister` function. This sends a service registration request to the index server. The index server creates a new service and a peer database record using the received information but leaves the service unapproved. The network admin can examine the service requests and approve them.

4.4 Network

The Peer and Service clients connect and interact with the Network through the Index Server’s by invoking the appropriate REST API (Fig. 4.3) of the Index Server. These API are discussed next.

4.4.1 Registration

A peer client can register for an account by calling the Network Client function `Register` (Alg. 4.15). The Network Client creates the JSON registration packet (Alg. 4.16) and sends it to the Index Server endpoint `/signup`. The index server parses and validates the request (Alg. 4.17). If the client is new and the request is valid, a new User is created and pushed to the database (Alg. 4.18). Any subsequent network actions require that the user post their login information as part of the HTTP basic authentication header.
func Register(logger *log.Logger, netURL string, name string, pass string, email string) error {
    var httpClient = &http.Client{Timeout: timeout}
    signup := api.Signup{name, pass, email}
    jsonReq, err := json.Marshal(signup)
    url := netURL + api.SignUpRoute
    request.Header.Set("Content-Type", "application/json")
    response, err := httpClient.Do(request)
    if response.Status != "200 OK" {
        return errors.New(response.Status)
    }
    return nil
}

Algorithm 4.15: Client Register

type Signup struct {
    Name string `json:"name"`
    Password string `json:"password"`
    Email string `json:"email"`
}

Algorithm 4.16: Signup JSON

func (s *Service) Signup(w http.ResponseWriter, r *http.Request) {
    body, err := ioutil.ReadAll(r.Body)
    var signup api.Signup
    err = json.Unmarshal(body, &signup)
    err = s.database.NewUser(signup.Name, signup.Password, signup.Email)
    w.WriteHeader(http.StatusOK)
}

Algorithm 4.17: New User Handler

func (d *DataBase) NewUser(name string, password string, email string) error {
    session := d.rootSession.Copy()
    user := User{Name, Password, email, role}
    c := session.DB("index").C("user")
    err := c.Insert(user)
    return err
}

Algorithm 4.18: New User Database
### 4.4.2 Login

The **Login** function logs the client into the network. It spawns a background process that continuously pings the API heartbeat endpoint within the network’s timeout period of 10 seconds (Alg. 4.19).

```go
func (client *Client) launchHeartBeat() {
    var ctx context.Context
    ctx, client.cancel = context.WithCancel(context.Background())
    client.wg.Add(1)
    go func(client *Client, ctx context.Context, status chan<- conn) {
        defer client.wg.Done()
        for {
            select {
                case <-ctx.Done():
                    return
                case <-ticker.C:
                    checkedin, err := client.Heartbeat()
                    var result conn
                    checkErr(err) // set conn to result
                    client.mutex.Lock()
                    client.stat = result
                    client.mutex.Unlock()
            }
        }
    }(client, ctx, client.status)
}
```

```go
func (client *Client) Heartbeat() (bool, error) {
    var httpClient = &http.Client{Timeout: timeout}
        credentials.Password}
    jsonReq, err := json.Marshal(heartbeat)
    url := client.netURL + api.HeartBeatRoute
    ))
    request.Header.Set("Content-Type", "application/json")
    response, err := httpClient.Do(request)
    if response>Status != "200 OK" {
        return false, nil
    }
    return true, nil
}
```

Algorithm 4.19: Launch Heartbeat

When the index server receives a valid request (Alg. 4.20), it checks the IP of the client and the requested port. It then updates the client’s user record with the new connection information (Alg. 4.21). A client is not considered part of the network if it fails to ping the network in the allotted time.
type Heartbeat struct {
    Name string `json:"name"
    Password string `json:"password"
    Port string `json:"port"
}

Algorithm 4.20: Heartbeat JSON

func (s *Service) HeartBeat(w http.ResponseWriter, r *http.Request) {
    body, err := ioutil.ReadAll(r.Body)
    var heartbeat api.Heartbeat
    err = json.Unmarshal(body, &heartbeat)
    check, err := s.database.CheckCredentials(heartbeat.Name, heartbeat.
        Password)
    ip, port, err := net.SplitHostPort(r.RemoteAddr)
    err = s.database.HeartBeat(heartbeat.Name, heartbeat.Password, ip)
    w.WriteHeader(http.StatusOK)
}

Algorithm 4.21: Server Heartbeat

When the user elects to logout of the network, the close routine is invoked which
shuts down the heartbeat process (Alg. 4.22).

func (client *Client) Logout() error {
    client.cancel()
    client.wg.Wait()
    client.stat = conn{errors.New("no heartbeat thread running"), false}
    return nil
}

Algorithm 4.22: Close Client
4.4.3 Resource Lookup

The peer can lookup network resources from the index server using various client network functions and their corresponding REST endpoints. These functions are similar in implementation and a generic form of each is provided in algorithms 4.23, 4.24, and 4.25. Resource lookup can return details on: Users, Documents, Services, Portals, and Transactions.

```
func (client *Client) ListData() ([]api.Data, error) {
    var httpClient = &http.Client{Timeout: timeout}
    url := client.netURL + api.PortalsRoute
    request, err := http.NewRequest("GET", url, nil)
    request.SetBasicAuth(client.credentials.Name, client.credentials.Password)
    response, err := httpClient.Do(request)
    if response.Status != "200 OK" {
        return nil, errors.New("failed request")
    }
    body, err := ioutil.ReadAll(response.Body)
    var data api.Data
    err = json.Unmarshal(body, &data)
    return pb.Entries, nil
}
```

Algorithm 4.23: Client Resource Request

```
func (s *Service) GetDataList(w http.ResponseWriter, r *http.Request) {
    datas, err := s.database.GetDataList()
    apiData := make([]api.Data, len(datas))
    for i, d := range datas {
        apiData[i] = d.BuildAPIData()
    }
    bundle := api.DataBundle{count, apiData}
    jsonResp, err := json.Marshal(bundle)
    w.Header().Set("Content-Type", "application/json")
    w.Write(jsonResp)
}
```

Algorithm 4.24: Index Server Handler

```
func (ds *DataBase) GetDataList() ([]Data, error) {
    session := ds.rootSession.Copy()
    c := session.DB("index").C("data")
    var data []Data
    err := c.Find(nil).All(&data)
    return datas, nil
}
```

Algorithm 4.25: Database Access
### 4.4.4 Posting Documents

The user can post documents to the network using the network client’s *Post-Document* function (Fig. 4.26). The client constructs an HTTP `POST` request using the provided document metadata (Alg. 4.27) and sends it to the network `/network/document` endpoint (Alg. 4.26).

```go
func (client *Client) PostDoc(record *model.Record) (bool, error) {
    var httpClient = &http.Client{Timeout: timeout}
    jsonReq, err := json.Marshal(apiDoc)
    url := client.netURL + api.DocsRoute
    request.SetBasicAuth(client.credentials.Name, client.credentials.Password)
    request.Header.Set("Content-Type", "application/json")
    response, err := httpClient.Do(request)
    if response.Status != "200 OK" {
        return false, nil
    }
    return true, nil
}
```

Algorithm 4.26: Client Post Document

```go
type Doc struct {
    ID string `json:"id, omitempty"
    UID string `json:"userid, omitempty"
    Hash [sha256.Size]byte `json:"hash, omitempty"
    Time time.Time `json:"time, omitempty"
    Origin string `json:"origin, omitempty"
}
```

Algorithm 4.27: Record JSON

The server parses and validates the request. If valid, the server pushes the document along with the owner ID to the *document* collection (Alg. 4.28). The *document* collection is unique on the hash of the document. The result is returned to the client.
func (s *Service) PostDoc(w http.ResponseWriter, r *http.Request) {
    body, err := ioutil.ReadAll(r.Body)
    var postdoc api.Doc
    err = json.Unmarshal(body, &postdoc)
    username, _, ok := r.BasicAuth()
    user, err := s.database.ReturnByName(username)
                      Origin: postdoc.Origin}
    err = s.database.PushToDb(&doc)
    w.Header().Set("Content-Type", "text/plain; charset=utf-8")
    w.WriteHeader(http.StatusOK)
}

Algorithm 4.28: Server Add Document

4.4.5 Requesting and Approving Transactions

The /network/transaction path provides peers access to transactions. A client requests a transaction by sending a POST request with the JSON TransactionRequest (Alg. 4.29) to the index server (Alg. 4.30). The server verifies the request, and if valid, creates a new transaction and sets it to the pending state in the database. The new transaction ID is returned to the client.

type TransactionRequest struct {
    Action string `json:"action"`
    ServiceID string `json:"serviceId,omitempty"`
    UserID string `json:"userId,omitempty"`
    DocumentID string `json:"docid"`
}

Algorithm 4.29: Transaction Request JSON
func (client *Client) RequestTransaction(treq api.TransactionRequest) (string, error) {
    var httpClient = &http.Client{Timeout: timeout}
    jsonReq, err := json.Marshal(treq)
    uri := client.netURL + api.TransactionsRoute
    request.SetBasicAuth(client.credentials.Name, client.credentials.Password)
    request.Header.Set("Content-Type", "application/json")
    response, err := httpClient.Do(request)
    if response.Status != "200 OK" {
        return "", nil
    }
    body, err := ioutil.ReadAll(response.Body)
    var t api.Transaction
    err = json.Unmarshal(body, &t)
    return t.ID, nil
}

Algorithm 4.30: Request Transaction

A client can request the list of all waiting transactions by posting a GET request to/network/transaction endpoint with the query string parameter of state=waiting.

The server checks that the request is valid and queries the database for the transactions in which the client is the receiver and the state is waiting (Alg. 4.31). The resulting list of transactions (Alg. 4.32) is returned to the client.

func (ds *DataBase) OpenUserTransaction(username string) ([]Transaction, error) {
    user, err := ds.ReturnByName(username)
    uid := user.ID
    session := ds.rootSession.Copy()
    c := session.DB("index").C("transaction")
    var transactions []Transaction
    query := bson.M{"$and": []bson.M{"$or": []bson.M{"sender": uid, "receiver": uid}, "state": "waiting"}}
    err = c.Find(query).All(&transactions)
    return transactions, nil
}

Algorithm 4.31: Open Transactions
type Transaction struct {
    ID string `json:"id"`
    DocHash string `json:"dochash"`
    SenderID string `json:"sender"`
    SenderIP string `json:"senderip"`
    ReceiverID string `json:"receiver"`
    ReceiverIP string `json:"receiverip"`
    Port string `json:"receiverport"`
    Action string `json:"action"`
    State string `json:"pending"`
    Time time.Time `json:"time"`
}

Algorithm 4.32: Transaction JSON

A client can approve a transaction by sending a \texttt{POST} request to the
\texttt{/network/transaction/{TID}} endpoint. The \{TID\} path variable refers to the
transaction ID. The server validates the user and updates the transaction state from
\textit{pending} to the next protocol state (Alg. 4.33). If the transaction is not in the \textit{waiting}
state, an error is returned to the client.

```go
func (ds *DataBase) OkayTransaction(username string, tid string) error {
    user, err := ds.ReturnByName(username)
    uid := user.ID.Hex()
    session := ds.rootSession.Copy()
    c := session.DB("index").C("transaction")
    sel := bson.M{"$and": []bson.M{bson.M{"_id": bson.ObjectIdHex(tid)},
                                  bson.M{"receiver": bson.ObjectIdHex(uid)},}}
    upd := bson.M{"$set": bson.M{"state": "transfer"}}
    err = c.Update(sel, upd)
    return nil
}
```

Algorithm 4.33: Approve Transaction
4.5 Transaction

The transaction begins processing after the receiver approves. The transactions classes are shown in Figure 4.20.

```
<<Interface>>

transaction.FSM
+ ClientExecute(api.Transaction): api.Message, err
+ ServerExecute(api.Transaction): api.Message, err
+ IndexExecute(index.Transaction, string, api.Message):
```

```
api.Transaction
+ ID: string
+ Hash: string
+ SenderID: string
+ SenderIP: string
+ ReceiverID: string
+ ReceiverIP: string
+ Port: string
+ Action: string
+ State: string
+ Time: time.Time
```

```
model.Database
- logger: log.Logger
- rootSession: mgo.Session
~ NewDatabase(): Database
- setupDatabase(): void
+ StoreRecords([Record]): []Result
+ StoreRecord(Record): err
+ FindRecord(string): Record, err
+ AllRecords(): []Record, err
+ StorePeerRecord(PeerRecord): err
+ FindPeerRecord(string): PeerRecord, err
+ AllPeerRecords(): []PeerRecord, err
```

```
p2p.Client
- logger: log.Logger
- ip: string
- port: string
- tconn: net.Conn
~ NewClient(): Client
+ Connect(): err
+ Do(Request): Response, err
~ createAddress(net.IP, string)
+ Close(): err
```

```
api.Message
+ Tid: string
+ Type: string
+ Data: json.RawMessage
```

```
transaction.ClientProtocol
- logger: log.Logger
- db: model.Database
~ NewClientProtocol(): ClientProtocol
+ ClientExecute(api.Transaction): api.Message
```

```
transaction.ServerProtocol
- logger: log.Logger
- db: model.Database
~ NewServerProtocol(): ServerProtocol
+ ServerExecute(api.Transaction): api.Message, api...
```

```
transaction.Mux
- logger: log.Logger
- mutex: sync.RWMutex
- route: map[string]FSM
~ NewMux(): Mux
+ Register(string, FSM): err
+ ClientExecute(api.Transaction): api.Message, err
+ ServerExecute(api.Transaction): api.Message, api...
+ IndexExecute(index.Transaction, string, api.Message):
```

Fig. 4.20.: Transaction Process
A client requests the list of running transactions by posting a GET request to the /network/transaction path with the query string parameter of state=running. The network server validates the request and then creates two MongoDB pipelines to pull the list of running transactions in which the user is participating (Alg. 4.34). The pipelines are necessary to join the connection information from the User collection with the results from transaction collection. The two pipelines (Alg. 4.35) are separated by the clients role in the transaction, one for requester and another for the receiver. The two results are returned to the user as a single list of transactions. The clients add the new transactions to their respective job pools.

```go
func (ds *DataBase) RunningUserTransaction(username string) ([]Transaction, error) {
    user, err := ds.ReturnByName(username)
    uid := user.ID
    session := ds.rootSession.Copy()
    c := session.DB("index").C("transaction")
    senderTransactions := []Transaction
    var receiverTransactions []Transaction
    open := bson.M{"$match": bson.M{"state": bson.M{"$nin": []string{"final", "waiting"}}}"
    senderID := bson.M{"$match": bson.M{"sender": uid}}
    ljReceiverID := bson.M{"$lookup": bson.M{"from": "user", "localField": "receiver", "foreignField": ".id", "as": "out"}}
    combine := bson.M{"$project": bson.M{"_id": 1, "sender": 1, "receiver": 1, "receiverip": bson.M{"$arrayElemAt": []interface{}{"out.ip", 0}}, "dochash": 1, "state": 1, "time": 1}}
    receiverID := bson.M{"$match": bson.M{"receiver": uid}}
    senderPipe := []bson.M{senderID, open, ljReceiverID, combine}
    receiverPipe := []bson.M{receiverID, open}
    err = c.Pipe(senderPipe).All(&receiverTransactions)
    err = c.Pipe(receiverPipe).All(&senderTransactions)
    transactions := append(receiverTransactions, senderTransactions...)
    return transactions, nil
}
```

Algorithm 4.34: Running Transactions

```go
db.transaction.aggregate([{$facet: {"sender": [{"match": {receiver: ObjectId($UserId) }}, {$lookup: { from: "user", localField: "receiver", foreignField: ".id", as: "out"}}],"$project":{"_id": 1, "sender": 1, "receiver": 1, "recip": {"$arrayElemAt": ["$out.ip", 0]}}}, {"$match": {sender: ObjectId($UserId)}}}], "reciever": [{"match": {sender: ObjectId($UserId)}}])
```

Algorithm 4.35: Example Pipeline Query
Clients in the *sender* role initiate a connection with the *receiver* and attempt a transition from the current state of protocol’s state machine. Transitions in the network are modeled as a set of interfaces (Alg. 4.36). Each role uses the last known transaction data (Alg. 4.32) and, in the case of the server or index roles, any messages (Alg. 4.37). The job sends the returned transition execution messages to the client or the index server.

```go
1 type ClientFSM interface {
2     ClientExecute(*api.Transaction) (*api.Message, error)
3 }
4 type ServerFSM interface {
6 }
7 type IndexFSM interface {
8     IndexExecute(*index.Transaction, string, *api.Message) error
9 }
```

Algorithm 4.36: Transition Interfaces

```go
1 type Message struct {
2     Tid string `json:"tid"`
3     Type string `json:"type"`
4     Data json.RawMessage `json:"data"`
5 }
```

Algorithm 4.37: Message JSON

To correctly dispatch to the protocol and state, a *mux* acts as a dispatch layer between the interface and the routines. A *mux* is a design pattern to register a path to an action. In this case, the mux allows for state machine transitions to be registered to a transaction protocol and state. A caller passes a path to a requested action and the mux returns the registered action. The implemented *mux* fulfills all three interfaces, allowing the *mux* to hide all concrete details from the caller. Each concrete routine is registered by a path idiom of */{action}/{state}*. When the *mux* is invoked, it examines the transaction’s state and action and dispatches it to the correct routine. The results are returned to the caller. If no routine is registered, an error is returned.

After executing a transition, the client tries to update the index server of the result by calling the Network Client *UpdateTransaction* function (Alg. 4.38) with the
Message JSON packet. The function posts the packet to the transaction endpoint. The network server checks if the transaction is valid, pulls the current transaction from the database, and attempts an index side transition with the transaction and message information. To ensure correctness, this transition attempt is similar to an atomic compare-and-swap operation [3]. If the update succeeds, then the transaction is moved to its next state.

```go
func (s *Service) UpdateTransaction(w http.ResponseWriter, r *http.Request) {
    body, err := ioutil.ReadAll(r.Body)
    var update api.Message
    err = json.Unmarshal(body, &update)
    username, _, ok := r.BasicAuth()
    tid := update.Tid
    transaction, err := s.database.ReturnTransactionByID(tid)
    err = s.indexFSM.IndexExecute(transaction, username, &update)
    w.Header().Set("Content-Type", "text/plain; charset=utf-8")
    w.WriteHeader(http.StatusOK)
}

func (ds *DataBase) UpdateTransaction(tid string, last string, update string) error {
    session := ds.rootSession.Copy()
    c := session.DB("index").C("transaction")
    sel := bson.M{
        "_id": bson.ObjectIdHex(tid),
        "state": last
    }
    upd := bson.M{
        "$set": bson.M{
            "state": update
        }
    }
    err := c.Update(sel, upd)
    return err
}
```

Algorithm 4.38: Update Index Transaction

### 4.5.1 Get

A Get transaction transfers a record from the owner to the requester (Alg. 4.39). The sender initiates the connection with the TCP library and sends the transaction information to the receiver. The receiver checks the request, pulls the record from its local record database, and returns the result to the sender. The sender adds the returned document to its peerrecord collection. After the exchange the two clients send messages to the index server to finalize. The index server checks the messages and moves the transaction to the final state.
func (ct *ClientTransfer) ClientExecute(transaction *api.Transaction) (*api.Message, error) {
    if transaction.State != "transfer" {
        return nil, errors.New("wrong state")
    }
    message := api.Message{Tid: transaction.ID}
    m, err := json.Marshal(message)
    request := p2p.Request("/", "message", m)
    ip := net.ParseIP(transaction.ReceiverIP)
    port := transaction.Port
    p2pClient := p2p.NewClient(ct.logger, ip, port)
    err = p2pClient.Connect()
    defer p2pClient.Close()
    response, err := p2pClient.Do(&request)
    record, err := getRecord(response)
    err = ct.database.StorePeerRecord(peerRecord)
    data, _ := json.Marshal(api.Basic{"done"})
    indexUpdate := api.Message{transaction.ID, "basic", data}
    return &indexUpdate, nil
}

    hash := current.DocHash
    record, err := st.database.FindRecord(hash)
    update, err := json.Marshal(record)
    updateIndex, err := json.Marshal(api.Basic{"done"})
    sender := api.Message{current.ID, "record", update}
    broadcast := api.Message{current.ID, "basic", updateIndex}
    return &sender, &broadcast, nil
}

func (i *IndexTransfer) IndexExecute(transaction *index.Transaction, user string, message *api.Message) error {
    tid := transaction.ID.Hex()
    err := i.db.UpdateTransaction(tid, transaction.State, "final")
    return nil
}

Algorithm 4.39: Get Algorithm

4.5.2 Push

A Push transaction transfers a record from the requesting peer to the target peer (Alg. 4.40). The requesting peer connects to the target by using the p2p library and passes the agreed upon document to the target. The target peer checks the document and saves it to their PeerRecord collection. The two peers alert the index server of
the transfer. The index server checks the message and transitions the transaction to the final state.

```go
func (cp *ClientPush) ClientExecute(transaction *api.Transaction) (*api.Message, error) {
    if transaction.State != "transfer" {
        return nil, errors.New("wrong state")
    }
    record, err := cp.database.FindRecord(transaction.Hash)
    push, err := json.Marshal(record)
    message := api.Message{transaction.ID, "record", push}
    m, err := json.Marshal(message)
    request := p2p.Request{"/", "message", m}
    ip := net.ParseIP(transaction.ReceiverIP)
    port := transaction.Port
    p2pClient := p2p.NewClient(cp.logger, ip, port)
    err = p2pClient.Connect()
    defer p2pClient.Close()
    response, err := p2pClient.Do(&request)
    data, _ := json.Marshal(api.Basic{"done"})
    indexUpdate := api.Message{transaction.ID, "basic", data}
    return &indexUpdate, nil
}

    record, err := getRecord(message)
    err = sp.database.StorePeerRecord(peerRecord)
    updateIndex, err := json.Marshal(api.Basic{"done"})
    sender := api.Message{current.ID, "basic", updateIndex}
    broadcast := api.Message{current.ID, "basic", updateIndex}
    return &sender, &broadcast, nil
}

func (i *IndexPush) IndexExecute(transaction *index.Transaction, user string, message *api.Message) error {
    tid := transaction.ID.Hex()
    err := i.db.UpdateTransaction(tid, transaction.State, "final")
    return nil
}
```

Algorithm 4.40: Push Algorithm

### 4.5.3 Service

A Service transaction involves one regular peer and a service (Alg. 4.41). The peer connects to the service using the P2P library. After connecting, the client passes the agreed upon record to the service. The service invokes the requested application
and returns the result to the client. The client saves the result to its database as a new record. If the operation is successful, the client and service both alert the index server. The index server checks the messages and moves the transaction to the final state.

```go
func (c *ClientProcess) ClientExecute(transaction *api.Transaction) (*api.Message, error) {
    if transaction.State != "transfer" {
        return nil, errors.New("not process state")
    }
    record, err := c.db.FindRecord(transaction.DocHash)
    jsonRecord, err := json.Marshal(record)
    message := api.Message{transaction.ID, "record", jsonRecord}
    m, err := json.Marshal(message)
    request := p2p.Request{"/", "message", m}
    ip := net.ParseIP(transaction.ReceiverIP)
    port := transaction.Port
    p2pClient := p2p.NewClient(c.logger, ip, port)
    err = p2pClient.Connect()
    defer p2pClient.Close()
    response, err := p2pClient.Do(&request)
    var respM api.Message
    err = json.Unmarshal(response.Data, &respM)
    surveyResult := model.SurveyResult{string(respM.Data)}
    t := time.Now()
    newRecord, err := model.NewRecord(t, transaction.ID, surveyResult)
    err = c.db.StoreRecord(&newRecord)
    data, _ := json.Marshal(api.Basic{"done"})
    indexUpdate := api.Message{transaction.ID, "basic", data}
    return &indexUpdate, nil
}

    peerRecord, err := getRecordMess(message)
    j, err := json.Marshal(peerRecord.Data)
    out, err := s.classifier.Run(j)
    updateIndex, err := json.Marshal(api.Basic{"done"})
    sender := api.Message{transaction.ID, "record", []byte(out)}
    broadcast := api.Message{transaction.ID, "basic", updateIndex}
    s.logger.Println(sender, broadcast)
}

func (i *IndexProcess) IndexExecute(t *index.Transaction, user string, m *api.Message) error {
    tid := t.ID.Hex()
    err := i.db.UpdateTransaction(tid, t.State, "final")
    return nil
}
```

Algorithm 4.41: Service Algorithm
4.6 Hypertension Prediction Service

The MEPS Hypertension prediction service has three stages of implementation: training, testing, and deployment. In the training stage, the processed training data is feed into the model constructing the Naive Bayes classifier (Fig. 4.21). The training data feature list consisted of the selected 30 ICD-9-CM prior conditions and the 17 consolidated features. The model is saved as a binary object [133] that can be invoked every time the service receives a request. The model is then deployed as part of the service client. The service application reloads the model, reads the input JSON record, classifies the record for Hypertension, and writes out the result.

![Fig. 4.21.: Hypertension Prediction Classes](image-url)
type SurveyResult struct {
    Value string `json:"result", bson:"result"`
}

Algorithm 4.42: Returned Data

4.6.1 Naive Bayes Model

The Naive Bayes is modeled as the nbayes class (Fig. 4.21). The class constructor
self-calls the learn function with the passed data to construct the model. The class
wraps a hash table of features each with a Conditional Probability Table (CPT).
Each feature is of the RV class. The Naive Bayes creates a database of conditional
probabilities by constructing a RV object for each feature and then adding it to the
hash table (Alg. 4.43).

```python
def learn(self, train_data):
    total = train_data.shape[0]
    label_true = train_data[self.label].value_counts().get(1)
    label_false = total - label_true
    p_label_true = label_true / total
    p_label_false = label_false / total
    label_table = {"0.0": p_label_false, "1.0": p_label_true}
    self.label_table = label_table
    feature_table = {}
    for feature, value in self.feature_list.items():
        gb = train_data.groupby([self.label, feature])
        count = gb.size().reset_index(name="count")
        count.set_index([self.label, feature], inplace=True)
        r = rv(count, label_table, [self.label, feature])
        feature_table[feature] = r
    self.feature_table = feature_table
```

Algorithm 4.43: Model Construction

The RV class creates the CPT by counting the features frequency intersection
with the label and then conditioning it on the probability of that label (Alg. 4.44).
The RV class prob method then retrieves a requested conditional probability from the
CPT.
def __init__(self, count_table, label_table, name):
    self.counts = count_table
    self.name = name
    self.label_table = label_table
    self.create_prob()

def create_prob(self):
    prob_table = self.counts / self.counts.sum()
    r = prob_table.loc[0.0, :].apply(lambda x: x / self.label_table["0.0"])
    prob_table.loc[0.0, :].update(r)
    r = prob_table.loc[1.0, :].apply(lambda x: x / self.label_table["1.0"])
    prob_table.loc[1.0, :].update(r)
    self.prob_table = prob_table

Algorithm 4.44: Calculating Conditional Probability Table

The hypertension event is a Boolean value, either true or false. To classify instances, the Naive Bayes constructs the probability of both events using the provided evidence (Alg. 4.45). The probability of each event is calculated by taking the product of all the features with the evidence. The function returns the label with the higher probability.

def classify(self, record):
    sum_cond_prob_x_0 = 1
    for var, value in self.feature_list.items():
        val = record[var]
        r = self.feature_table[var]
        cond_prob_var = r.get_prob(0.0, val)
        sum_cond_prob_x_0 = sum_cond_prob_x_0 * cond_prob_var
        p_cond_0_x = sum_cond_prob_x_0 * self.label_table["0.0"]

    sum_cond_prob_x_1 = 1
    for var, value in self.feature_list.items():
        val = record[var]
        r = self.feature_table[var]
        cond_prob_var = r.get_prob(1.0, val)
        sum_cond_prob_x_1 = sum_cond_prob_x_1 * cond_prob_var
        p_cond_1_x = sum_cond_prob_x_1 * self.label_table["1.0"]

    if p_cond_0_x > p_cond_1_x:
        cl = 0.0
    elif p_cond_0_x < p_cond_1_x:
        cl = 1.0
    else:
        cl = 0.0
    return cl

Algorithm 4.45: Classifying Instances
4.6.2 Performance

An initial test of the classifier resulted in a 95% accuracy. This performance was caused by cases of prior hypertension, which are easy to classify. After removing prior instances of hypertension from the dataset, the classifier performance is inline with the reviewed literature. The results are shown in Table 4.1. The table indicates that the model struggles when predicting positive results. A potential source of improvement is the addition of network records to the training data over time.

Table 4.1.: Classifier Performance with Panels 17, 18, and 19

<table>
<thead>
<tr>
<th>Performance Metric</th>
<th>Percentage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>75.2</td>
</tr>
<tr>
<td>Precision</td>
<td>58.9</td>
</tr>
<tr>
<td>Sensitivity</td>
<td>66.3</td>
</tr>
<tr>
<td>Specificity</td>
<td>79.1</td>
</tr>
</tbody>
</table>
5. CONCLUSION

A Personal Health Record would benefit patients, providers, and third parties. Patients can control and track the access of records, aggregate multiple providers, and organize the records according to their wishes. Providers can use the PHR as a hub to other providers and to interact with patients. Third parties can directly work with patients through the PHR rather than go through a provider or HIE.

A Peer-to-Peer Personal Health Record Network is patient-centric, highly scalable and can be made resilient to faults. Moreover, peers directly exchange records, limiting a record’s exposure. Finally, the network allows for new services to be added at run time.

The traceability of a record accesses is important in making the patients aware of how their records are used. A log of transactions provides a trace of each access to record where each access had to be approved by the patient.

Predicting conditions from historical health data can help improve patient care. Patients and providers can proactively prevent conditions and plan for future care. EHR data provides a broad view of the public’s health which can be used to build comprehensive predictive models for important disease conditions. These models can be deployed as a service in the PHR in order to help identify potential conditions.
The system defined and implemented in this thesis is a prototype Peer-to-Peer Personal Health Record network. The network is maintained by an index server which is accessed through a REST API. Peer clients connect to the server to post the metadata of records and to request transactions. Peers directly exchange health records, carrying out the requested transactions independently and only using the index server to confirm and log progress. These transactions can be one of three types: get, post, and service. get and post transfer documents between clients. Service transactions provide services to other peers. To demonstrate this concept, a prototype service implementing a Hypertension predictor trained on MEPS data, was deployed to the network.

An important direction for future work is the implementation of a client for mobile devices. A majority of patients and health providers use smartphones. This application presents some challenges. Though golang can run on both Android and iOS, the support is still experimental as of 2019 [54]. The GUI would need to be reworked as the mobile support is mostly bindings around iOS and Android native languages. The client database storage would also need to be migrated to Couchbase Lite [35] or SQLite [62].

The current document representation only makes the metadata of the document available to the network. This limits a peer’s ability to query for documents. The owner must inform the peer of the document contents and is a direction for future improvement.

The fine grain access control modeled in this thesis is difficult to propose to providers, as evidenced by the harsh reaction of providers to patient control of health records [140, 152]. Additionally, patients approving each transaction is cumbersome. Designing levels of access, with fine-grained control over sensitive records and coarse-grained over others, will be considered for future work.
A core issue with a centralized P2P model is the limited scalability compared to a pure P2P model. However, transactions in the pure model would be more difficult to implement, and the overall maintenance of the framework would increase. Exploring alternate architecture models can help improve the scalability of the network with limited increase in overhead.

In the current implementation the transactions are not fully secure. A document’s ownership is unverified, a party can repost modified documents, and the transmission of records is unencrypted. These flaws need to be addressed and the system needs to conform to HIPAA regulations. Transactions are modeled using a simple state machine design. This model is limited. Distributed programming paradigms like Tuple Space and Bitcoin’s stack language would allow for more expressive and extensible transactions to be injected at run time. However, this is difficult to implement in untrusted scenarios and would require further research.
REFERENCES
REFERENCES


APPENDIX
APPENDIX A: Hypertension Survey Info

```plaintext
Type Heartsurvey Struct {
    Age string 'json:"AGELAST"_bson:"AGELAST"'
    Sex string 'json:"SEX"_bson:"SEX"'
    Race string 'json:"RACEV1X"_bson:"RACEV1X"'
    Highbloodpressure string 'json:"HIBPDX"_bson:"HIBPDX"'
    Agina string 'json:"ANGIDX"_bson:"ANGIDX"'
    Coroneryheartdisease string 'json:"CHDDX"_bson:"CHDDX"'
    Heartattack string 'json:"MIDX"_bson:"MIDX"'
    Otherheart string 'json:"OHRTDX"_bson:"OHRTDX"'
    Stroke string 'json:"STRKDX"_bson:"STRKDX"'
    Emphysema string 'json:"EMPHDX"_bson="EMPHDX"
    Highcholesterol string 'json:"CHOLDX"_bson="CHOLDX"
    Diabetes string 'json:"DIABDX"_bson="DIABDX"
    Smoke string 'json:"ADSMOK42"_bson="ADSMOK42"
    Restrictfood string 'json:"NOFAT53"_bson="NOFAT53"
    Moreexercise string 'json:"EXRCIS53"_bson="EXRCIS53"
    Bmi string 'json:"BMINDX53"_bson="BMINDX53"
    Poverty string 'json:"POVCAT"_bson="POVCAT"
    C410 string 'json:"410"_bson="410"
    C413 string 'json:"413"_bson="413"
    C414 string 'json:"414"_bson="414"
    C424 string 'json:"424"_bson="424"
    C425 string 'json:"425"_bson="425"
    C426 string 'json:"426"_bson="426"
    C427 string 'json:"427"_bson="427"
    C428 string 'json:"428"_bson="428"
    C429 string 'json:"429"_bson="429"
    C436 string 'json:"436"_bson="436"
    C437 string 'json:"437"_bson="437"
    C440 string 'json:"440"_bson="440"
    C441 string 'json:"441"_bson="441"
    C442 string 'json:"442"_bson="442"
    C443 string 'json:"443"_bson="443"
    C444 string 'json:"444"_bson="444"
    C447 string 'json:"447"_bson="447"
    C453 string 'json:"453"_bson="453"
    C454 string 'json:"454"_bson="454"
    C455 string 'json:"455"_bson="455"
    C458 string 'json:"458"_bson="458"
    C459 string 'json:"459"_bson="459"
    C490 string 'json:"490"_bson="490"
    C491 string 'json:"491"_bson="491"
    C492 string 'json:"492"_bson="492"
    C493 string 'json:"493"_bson="493"
    C496 string 'json:"496"_bson="496"
    C514 string 'json:"514"_bson="514"
    C518 string 'json:"518"_bson="518"
    C519 string 'json:"519"_bson="519"
}
```
Algorithm A.1: Hypertension Survey Info
APPENDIX B: Hypertension Survey Info

Table B.1.: Feature Vector

<table>
<thead>
<tr>
<th>header</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>401</td>
<td>0</td>
</tr>
<tr>
<td>410</td>
<td>0</td>
</tr>
<tr>
<td>413</td>
<td>0</td>
</tr>
<tr>
<td>414</td>
<td>0</td>
</tr>
<tr>
<td>424</td>
<td>0</td>
</tr>
<tr>
<td>425</td>
<td>1</td>
</tr>
<tr>
<td>426</td>
<td>0</td>
</tr>
<tr>
<td>427</td>
<td>0</td>
</tr>
<tr>
<td>428</td>
<td>0</td>
</tr>
<tr>
<td>429</td>
<td>0</td>
</tr>
<tr>
<td>436</td>
<td>0</td>
</tr>
<tr>
<td>437</td>
<td>0</td>
</tr>
<tr>
<td>440</td>
<td>0</td>
</tr>
<tr>
<td>441</td>
<td>0</td>
</tr>
<tr>
<td>442</td>
<td>0</td>
</tr>
<tr>
<td>443</td>
<td>0</td>
</tr>
<tr>
<td>444</td>
<td>0</td>
</tr>
<tr>
<td>447</td>
<td>0</td>
</tr>
<tr>
<td>453</td>
<td>0</td>
</tr>
<tr>
<td>454</td>
<td>0</td>
</tr>
<tr>
<td>455</td>
<td>0</td>
</tr>
</tbody>
</table>

*continued on next page*
Table B.1.: continued

<table>
<thead>
<tr>
<th>header</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>458</td>
<td>0</td>
</tr>
<tr>
<td>459</td>
<td>0</td>
</tr>
<tr>
<td>490</td>
<td>0</td>
</tr>
<tr>
<td>491</td>
<td>0</td>
</tr>
<tr>
<td>492</td>
<td>0</td>
</tr>
<tr>
<td>493</td>
<td>0</td>
</tr>
<tr>
<td>496</td>
<td>0</td>
</tr>
<tr>
<td>514</td>
<td>0</td>
</tr>
<tr>
<td>518</td>
<td>0</td>
</tr>
<tr>
<td>519</td>
<td>0</td>
</tr>
<tr>
<td>ADSMOK42</td>
<td>1</td>
</tr>
<tr>
<td>AGELAST</td>
<td>2</td>
</tr>
<tr>
<td>ANGIDX</td>
<td>2</td>
</tr>
<tr>
<td>BMINDX53</td>
<td>2</td>
</tr>
<tr>
<td>CHDDX</td>
<td>2</td>
</tr>
<tr>
<td>CHOLDX</td>
<td>1</td>
</tr>
<tr>
<td>DIABDX</td>
<td>2</td>
</tr>
<tr>
<td>DUPERSID</td>
<td>26974101</td>
</tr>
<tr>
<td>EMPHDX</td>
<td>2</td>
</tr>
<tr>
<td>EXRCIS53</td>
<td>1</td>
</tr>
<tr>
<td>HIBPDX</td>
<td>2</td>
</tr>
<tr>
<td>l401</td>
<td>0</td>
</tr>
<tr>
<td>MIDX</td>
<td>2</td>
</tr>
<tr>
<td>NOFAT53</td>
<td>1</td>
</tr>
<tr>
<td>OHRTDX</td>
<td>1</td>
</tr>
</tbody>
</table>

continued on next page
Table B.1.: *continued*

<table>
<thead>
<tr>
<th>header</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>POVCAT</td>
<td>1</td>
</tr>
<tr>
<td>RACEV1X</td>
<td>1</td>
</tr>
<tr>
<td>SEX</td>
<td>2</td>
</tr>
<tr>
<td>STRKDX</td>
<td>2</td>
</tr>
</tbody>
</table>
APPENDIX C: Example Client Usage of System

Welcome to the Peer Health Client

- registration
- login
- record
- help
- log

Fig. C.1.: Welcome Page
Fig. C.2.: Patient Registration

Registration

Name: patient
Password: Password1
Email: patient@email.com
Submit

Fig. C.3.: Patient Login

Login

Name: patient
Password: password
Submit
Welcome to the Peer Health Network

- account
- records
- transactions
- jobs
- portal
- peerrecord
- record
- survey
- logout

Fig. C.4.: Network Page

Network Portals

<table>
<thead>
<tr>
<th>Id</th>
<th>Type</th>
<th>Organization</th>
<th>Url</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5c6093b044460a588a9d6d594f83</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. C.5.: Portal Selection
Fig. C.6.: Downloading Records from a Registered Portal
### Local Records

<table>
<thead>
<tr>
<th>Hash</th>
<th>Time</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:49 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:45:50 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:50 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:45:51 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:51 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:45:52 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:52 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:45:53 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:53 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:45:54 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:54 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:45:55 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:55 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:45:56 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:56 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:45:57 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:57 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:45:58 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:58 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:45:59 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:45:59 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:46:00 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:46:00 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:46:01 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:46:01 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:46:02 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:46:02 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:46:03 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:46:03 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:46:04 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:46:04 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:46:05 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:46:05 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:46:06 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:46:06 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:46:07 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:46:07 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:46:08 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
<tr>
<td>825401d9b1</td>
<td>2015-07-22 07:46:08 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
</tr>
<tr>
<td>2015-07-22 07:46:09 +0000 UTC</td>
<td>figshare.com/96a75f5668</td>
<td></td>
</tr>
</tbody>
</table>

Fig. C.7.: Viewing Records
Local Record:
a54d1503a72385c9749ca93c8e4ef26128e70d35074f03b1dca20e5bc4cf9f

Metadata
a5441503a72385c9749ca93c8e4ef26128e70d35074f03b1dca20e5bc4cf9f 2019-07-22 07:49:59.403 +0000 UTC http://soap.fhir.org/baseData3

Data

Network Options
Post to Server: SUCCESS

Fig. C.8.: FHIR Data for a Single Record and Posting to Network
C.1 Provider Requesting a Patient Health Record

List of Network Records

<table>
<thead>
<tr>
<th>Id</th>
<th>UId</th>
<th>Hash</th>
<th>Time</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>5d3569a7f5c13beeb53a7c2 5d35692175c13beeb53a5ab a54e1503a72385c0b74e9eaf9c3ce4ef63126e70d35074df2bc2d4a2c2b0ebc0ef9f</td>
<td>2019-07-22 07:44:52.666 +0000 UTC</td>
<td><a href="http://hapi.fhir.org/baseDstu3">http://hapi.fhir.org/baseDstu3</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. C.9.: Provider: Network Records

Transaction Request

Record Metadata

(5d3569a4f75c13beeb53a7c2 5d35692175c13beeb53a5ab a54e1503a72385c0b74e9eaf9c3ce4ef63126e70d35074df2bc2d4a2c2b0ebc0ef9f 2019-07-22 07:44:59.403 +0000 UTC http://hapi.fhir.org/baseDstu3)

Basic Transactions

- Choose an option
  - Transfer
  - Push

Submit

Services List

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>UId</th>
<th>Heartbeat</th>
<th>Info</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>5cbf35247cd67aeabf7eaa52</td>
<td>meps</td>
<td>5cbf35247cd67aeabf7eaa52</td>
<td>2019-07-22 07:46:55.231 +0000 UTC info about the meps predictor heartinfo</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. C.10.: Provider: Selecting Patient Record for a Transaction
Transfer Request

Record Metadata

Fig. C.11.: Provider: Sending Request for Get Transaction

List of Network Transactions

Fig. C.12.: Patient/Provider: Transaction Waiting for Approval
### List of Network Transactions

<table>
<thead>
<tr>
<th>Id</th>
<th>DocHash</th>
<th>SenderId</th>
<th>ReceiverId</th>
<th>Action</th>
<th>State</th>
<th>Time</th>
<th>Authorize</th>
</tr>
</thead>
<tbody>
<tr>
<td>5dd56a1b18440be1012aeb35c3a5dd153172c12f3d5c6b744dca92c1dedf262188e70d25a7d3f36b10daa2e8b0c0c9f5d45696375c13b3ebb53a0b5b8d5d5602175c13b3ebbb3c1a9f</td>
<td>5dd56a1b18440be1012aeb35c3a5dd153172c12f3d5c6b744dca92c1dedf262188e70d25a7d3f36b10daa2e8b0c0c9f5d45696375c13b3ebb53a0b5b8d5d5602175c13b3ebbb3c1a9f</td>
<td>07:47:16.285 +0000 UTC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. C.13.:** Patient: Approving Transaction

<table>
<thead>
<tr>
<th>Id</th>
<th>DocHash</th>
<th>SenderId</th>
<th>ReceiverId</th>
<th>Action</th>
<th>State</th>
<th>Time</th>
<th>Authorize</th>
</tr>
</thead>
<tbody>
<tr>
<td>5dd56a1b18440be1012aeb35c3a5dd153172c12f3d5c6b744dca92c1dedf262188e70d25a7d3f36b10daa2e8b0c0c9f5d45696375c13b3ebb53a0b5b8d5d5602175c13b3ebbb3c1a9f</td>
<td>5dd56a1b18440be1012aeb35c3a5dd153172c12f3d5c6b744dca92c1dedf262188e70d25a7d3f36b10daa2e8b0c0c9f5d45696375c13b3ebb53a0b5b8d5d5602175c13b3ebbb3c1a9f</td>
<td>07:47:16.285 +0000 UTC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. C.14.:** Provider/Patient: Transaction Complete
Fig. C.15.: Provider: Viewing Peer Records

Fig. C.16.: Provider: Transferred Peer Record
### C.2 Patient Requesting the Hypertension Service

![Image of a patient filling out a survey for the Hypertension Service](image)

**Heart Info Survey**

<table>
<thead>
<tr>
<th>Item</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
</tr>
<tr>
<td>Age</td>
<td>0-9</td>
</tr>
<tr>
<td></td>
<td>10-19</td>
</tr>
<tr>
<td></td>
<td>20-29</td>
</tr>
<tr>
<td></td>
<td>30-39</td>
</tr>
<tr>
<td></td>
<td>40-49</td>
</tr>
<tr>
<td></td>
<td>50-59</td>
</tr>
<tr>
<td></td>
<td>60-69</td>
</tr>
<tr>
<td></td>
<td>70-79</td>
</tr>
<tr>
<td></td>
<td>80-89</td>
</tr>
<tr>
<td></td>
<td>90+</td>
</tr>
<tr>
<td>Sex</td>
<td>Male</td>
</tr>
<tr>
<td></td>
<td>Female</td>
</tr>
<tr>
<td>BMI</td>
<td>Under 25</td>
</tr>
<tr>
<td></td>
<td>25-29.9</td>
</tr>
<tr>
<td></td>
<td>30-34.9</td>
</tr>
<tr>
<td></td>
<td>35-39.9</td>
</tr>
<tr>
<td></td>
<td>40+</td>
</tr>
<tr>
<td>Heart Rate</td>
<td>Normal</td>
</tr>
<tr>
<td></td>
<td>Tachycardia</td>
</tr>
<tr>
<td>Blood Pressure</td>
<td>Normal</td>
</tr>
<tr>
<td></td>
<td>Hypertension</td>
</tr>
<tr>
<td>Diabetes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Asthma</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Allergies</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>History of Heart Attack</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>History of Stroke</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Family History of Hypertension</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>History of Diabetes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>History of Asthma</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>History of Allergies</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Other Conditions</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Other Conditions</td>
<td>Other</td>
</tr>
<tr>
<td></td>
<td>Other</td>
</tr>
<tr>
<td>How much do you exercise?</td>
<td>Regularly</td>
</tr>
<tr>
<td></td>
<td>Occasionally</td>
</tr>
<tr>
<td></td>
<td>Rarely</td>
</tr>
<tr>
<td></td>
<td>Never</td>
</tr>
<tr>
<td>What is your smoking status?</td>
<td>Non-smoker</td>
</tr>
<tr>
<td></td>
<td>Smoker</td>
</tr>
<tr>
<td></td>
<td>Former smoker</td>
</tr>
<tr>
<td>More physical activity related to reduce smoking and heavy drinking</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Do you need help with nutrition?</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>No</td>
</tr>
<tr>
<td>What is your weight?</td>
<td>Underweight</td>
</tr>
<tr>
<td></td>
<td>Normal</td>
</tr>
<tr>
<td></td>
<td>Overweight</td>
</tr>
<tr>
<td></td>
<td>Obese</td>
</tr>
<tr>
<td>Family history as a percentage of the genetic link</td>
<td>10%</td>
</tr>
<tr>
<td></td>
<td>20%</td>
</tr>
<tr>
<td></td>
<td>30%</td>
</tr>
<tr>
<td></td>
<td>40%</td>
</tr>
<tr>
<td></td>
<td>50%</td>
</tr>
<tr>
<td></td>
<td>Over 50%</td>
</tr>
</tbody>
</table>

**Fig. C.17.: Patient Filling out Survey for Service**
Local Record:
8a7df0d23ce520c27b5fcb8bff55dcaf1aad1562db8f161ab31e0a846a6422cc

Fig. C.18.: Survey as a Record

Metadata
8a7df0d23ce520c27b5fcb8bff55dcaf1aad1562db8f161ab31e0a846a6422cc 2019-07-22 07:50:39.968 +0000 UTC local

Data
["410": 0.0, "412": 0.0, "414": 0.0, "434": 1.0, "435": 0.0, "438": 0.0, "439": 0.0, "429": 1.0, "486": 0.0, "437": 0.0, "440": 0.0, "441": 1.0, "442": 1.0, "443": 0.0, "444": 1.0, "445": 0.0, "453": 0.0, "454": 0.0, "455": 0.0, "458": 0.0, "459": 1.0, "460": 0.0, "491": 0.0, "862": 0.0, "493": 1.0, "496": 0.0, "514": 1.0, "515": 0.0, "516": 0.0, "AXGIDNX": 2.0, "AGELAST": 3.0, "ANGIDNX": 2.0, "BMIND53": 1.0, "BIDNIX": 1.0, "CHOLIDX": 0.0, "EMPHIDX": 2.0, "EMPHIDX": 2.0, "EMPID3": 1.0, "HIBPDX": 2.0, "MIXD": 2.0, "NQPAT53": 1.0, "ORHDX": 1.0, "POVCAT": 3.0, "RACEVIX": 6.0, "SEX": 1.0, "STIRID3": 2.0 ]

Network Options

Fig. C.19.: Survey Posted to Network
Transaction Request

Record Metadata

Basic Transactions

Choose an option
- Transfer
- Push

Submit

Services List

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Uid</th>
<th>Heartbeat</th>
<th>Info</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>5cbf35247cd6a6e67ae8b7fe65</td>
<td>meps</td>
<td>5cbf35247cd6a6e67ae8b7fe65</td>
<td>2019-07-22 07:50:39.866 +0000 UTC</td>
<td>info about the meps predictor heartinfo</td>
<td></td>
</tr>
</tbody>
</table>

Submit

Fig. C.20.: Selecting Service Transaction

Service Request Form

Record Metadata

Service Metadata

Request Form

Enter your message here...

Submit

Fig. C.21.: Requesting Service Transaction
### List of Network Transactions

<table>
<thead>
<tr>
<th>Id</th>
<th>DecHash</th>
<th>SenderId</th>
<th>ReceiverId</th>
<th>Action State</th>
<th>Time</th>
<th>Authority</th>
</tr>
</thead>
<tbody>
<tr>
<td>5d356a04e0eb1c1502e63e0350e6b7d4e0ba9300ded26126f7650517803081561e20e83d6cfdf</td>
<td>5d356a04e0eb1c1502e63e0350e6b7d4e0ba9300ded26126f7650517803081561e20e83d6cfdf</td>
<td>5d356a04e0eb1c1502e63e0350e6b7d4e0ba9300ded26126f7650517803081561e20e83d6cfdf</td>
<td>5d356a04e0eb1c1502e63e0350e6b7d4e0ba9300ded26126f7650517803081561e20e83d6cfdf</td>
<td>2019-07-22 07:47:46.805 +0000 UTC</td>
<td>2019-07-22 07:47:46.805 +0000 UTC</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. C.22.: Service Transaction Finished**

### Local Record:

8dc8f3631ae6036ec5b777070e72545c01c83267ec8c7a93b8f04736a44a4885

**Metadata**

8dc8f3631ae6036ec5b777070e72545c01c83267ec8c7a93b8f04736a44a4885 2019-07-22 07:52:46.18 +0000 UTC 5d356a04e0eb1c1502e63e0350e6b7d4e0ba9300ded26126f7650517803081561e20e83d6cfdf

**Data**

`{ "result": "true" }`

**Network Options**

*Push to Server*

**Fig. C.23.: Hypertension Service Result**