Journal of Health Informatics in Africa

8th Health Informatics in Africa Conference (HELINA 2013)
Peer-reviewed and selected under the responsibility of the Scientific Programme Committee

Aeetes: An App Generator for Sustainable and Secure Health Data Collection

Dominic Duggan [a,*], Daniel Golden [b], Wenbin Li [a], Mark Mirtchouk [b], Mark Mschedlivi [b], Justice A. Muh [c], Benjamin Muhoza [d], Leonard Ndongo [c], David Smith [a], Lian Yu [a]

[a] Stevens Institute of Technology, Hoboken, USA
[b] Bergen County Academies, Hackensack, USA
[c] Research for Development (R4D) International, Yaounde, Cameroon
[d] Women's Equity in Access to Care and Treatment (WEACT), Kigali, Rwanda

Background and Purpose: Aeetes represents an approach to patient data collection in LMIC that is intended to be scalable, sustainable and secure.
Methods: Existing approaches to data collection are typically either personal-computer-based or enterprise-based. Aeetes occupies a third space, that of peer-to-peer devices. It is intended to occupy a spot between the PC-based and enterprise-based approaches, addressing issues that make these approaches a poor match with some deployment scenarios.
Results: The Aeetes approach is characterized by a compiler-based approach to data collection app generation: A new data collection app is generated from an input specification of the data to be collected and the user interface for data collection. The benefit of this is greater software reliability, since the data model is made explicit in the code, and eventually greater security through analysis of the code for information flows. Aeetes also takes measures to protect patient confidentiality against loss of devices.
Conclusions: Aeetes represents an approach to moving away from PC-based approaches to data collection with poor security characteristics, to an approach that is more reliable and secure, while also being sustainable. It is hoped that some of the approach taken by Aeetes may eventually influence other systems such as OpenMRS.

Keywords: Health Impact Assessment, Public Health Informatics, Health Information Management, Software Design, Computer Security, Database Management Systems

1 Introduction

In general, healthcare information technology is regarded as essential to providing efficient and effective healthcare delivery. In countries ravaged by pandemics such as HIV/AIDS, healthcare data collection plays an important role in healthcare analysis and planning, revealing where resources need to be allocated and what treatments are efficacious. Any deployment of healthcare IT in LMIC must meet the goals of being scalable, sustainable and secure. *Scalability* refers to the ability of the technology to scale up from small field trials to large numbers of patients in production deployment. *Sustainability* refers to the ability of any such deployment to eventually become self-sufficient, once an initial deployment support framework is withdrawn. Finally, *security* refers to the protection of the confidentiality of patient data in such systems. Failure to protect confidentiality can undermine patient trust in the patient-doctor relationship, and in some cases expose patients to discrimination, blackmail and even death.

As part of developing data collection tools for epidemiological studies in several LMICs, we considered all three of these criteria, and evaluated several alternative approaches based on their fit with the conditions for IT deployment in the member countries. Our approach was designed to address what

*Corresponding author: Stevens Institute of Technology, Hoboken, NJ 07040, USA. Email: dduggan@stevens.edu, Tel: +1 (201) 216-8042
HELINA 2013 M. Korpela et al. (Eds.)
 DOI: 10.12856/JHIA-2013-v1-i1-60

were seen as deficiencies in some of the alternative approaches already in use. Those deficiencies were in respect of sustainability and security: For some of the alternative systems, it was not clear that they would be appropriate to the deployment situations that were envisaged, while for other systems security and confidentiality were serious concerns.

2 Materials and methods

In surveying the existing systems for patient data collection, we categorized these systems into a few broad categories.

The first category consists of *personal computer-based systems* that store patient data on desktop or laptop computers. Such systems are typically based on Microsoft Windows and the Access database management system. An example of such a system is the EPI-INFO system [1] provided by the Centers for Disease Control (CDC) , currently in use for example in clinics in Democratic Republic of Congo. SQLite is a standard database management system used for the Android and iOS platforms that in some ways is similar is Access, so a system similar to EPI-INFO could be developed for mobile devices.

The second category consists of *enterprise-based systems*, that adopt a client-server distributed system approach to patient data management. We refer to these systems as "enterprise" because they are typically built on an enterprise software stack. For example, the IQCare system [2] is built on SQL Server and the Microsoft .NET software platform. The popular OpenMRS open source system [3] is built on the Spring and Hibernate Java frameworks for building Web-based enterprise applications. Clients of an OpenMRS system typically execute as AJAX applications in a client Web browser, although it is also possible for client applications to execute on mobile devices (e.g. Sana Mobile and Open Data Kit [4]-[6]).

There is a third category of systems, that is in some ways still in a nascent state: That of *devices organized into a peer-to-peer local network*. They are distinguished from personal computer systems by the use of a network to coordinate the data on the devices, and they are distinguished from client-server systems by the absence of a central server. Our approach fits in this third category, and we argue that this is an important architecture to be considered for future IT development for healthcare delivery in LMIC.

In our view, personal computer systems for patient data management have the following issues. In one country where we collect data, informed consent is required of patients that are involved in the study. This informed consent is collected electronically, and data collection should prevent the entry of data for patients without such consent. But informed consent and data entry are performed by different personnel, on different devices. Second, most such systems are, as noted, based on Windows and Access. The benefit of this approach is that these systems are broadly familiar to clinic workers, and it is for example feasible for relatively technically unsophisticated staff to use form-based tools to formulate ad-hoc database queries. However there is a fundamental security flaw in such systems: the very ability of users to manage these systems introduces the possibility of social engineering attacks, where users install software apps that include malware such as Trojan horses. The unfettered ability of users to self-manage these systems is a legacy of business decisions made early in the history of personal computers, and the use of techniques such as virtualization and UAC to rectify this situation has met with limited success.

Our main concern with enterprise approaches is sustainability (ironically, a strong point with personal computer systems). An enterprise system composed of a Web server, application container, a persistence framework and a database server is a fairly sophisticated system to manage. For example, Tierney et al [7] report on the deployment of OpenMRS systems in three African countries. While the experience with OpenMRS was positive, deployments struggled once funding for clinic IT staff was withdrawn. With a national commitment to OpenMRS, as in Rwanda, it is possible to train a pool of professionals who can support OpenMRS deployments, and the relatively small size of the country enables a support strategy based on sending centralized IT staff to fix local problems. In a larger country, this strategy will not work for small rural clinics that may be many hours of travel away from where IT staff is located.

Our concerns with secure device management led us to consider the Android platform. Mobility was not an issue in this decision, and in fact mobile phones are a poor choice for data entry and were not a consideration. The attraction of Android is that it provides facilities for restricting user device management and enabling remote device management. There is already third-party commercial off-the-shelf (COTS) software, exploiting hooks provided by the Android kernel that can be used to restrict the applications that users can install on Android phones. Restricting the applications that can be installed by the user on the device is a key factor in protecting the device against malware. For this reason, it was

 DOI: 10.12856/JHIA-2013-v1-i1-60

decided early on that Web browsers (including embedded Web browsers using Webviews) should be discouraged on a device, because of the potential attack vector provided by mobile Javascript code. In addition, the ability to remotely manage such devices has been convincingly demonstrated, for example by Google removing malware-containing apps from consumer phones. This ability to manage devices "by remote control" was a key reason for our decision to deploy on Android devices.

There are several different hardware platforms to choose from in our deployment. Our current deployment is based on the ASUS Transformer Prime tablet, accommodating a docking station that includes a keyboard and trackpad. Android laptops are also available, such as the Go Note laptop from the UK. In future, we will be able to use desktop devices such as the HP Slate, that provides a 21" display on an Android device that connects to external peripherals such as a keyboard and mouse. Non-display devices such as the CompuLabs Utilite personal computer provide an ARM-based PC that can run either Ubuntu or Android, and includes WIFI, Bluetooth, Gigabit Ethernet and USB connections. Such devices may be useful for example as local backup devices. We can expect other forms of embedded devices, such as "smart glasses" as evidenced by Google Glass, to join this increasingly rich ecology of devices that our approach is based on.

Our concerns with sustainability have led us to adopt a peer-to-peer architecture for devices within a clinic, rather than the client-server architecture favoured by enterprise approaches. Our motivation is that IT deployment in a clinic without professional IT support should be as simple as installing consumer electronic devices in a residential household. Effectively users should be able to switch on a device and immediately see it working[1]. "Underneath the hood," the new device may for example use a discovery protocol, for example using WIFI broadcast, to find peers to communicate with. But the device is capable of continuing to function in standalone mode if no peers are discovered. In our architecture, data is replicated across devices in a clinic, and devices communicate peer-to-peer to share data, transparently to the users of these devices. Security is clearly a critical issue in this architecture, particularly since devices may be relatively portable.

3 Results

In this section, we give a detailed overview of the Aeetes approach to generating and supporting data collection tools.

3.1 User Interface

Our initial prototype of a data collection tool was based on Open Data Kit. ODK records questionnaires as XForms documents, constructed off the device using a form builder such as ODK Build. An Android app, ODK Collect, allows these forms to be displayed and filled in on an Android device, typically a smartphone. A Web service backend in the app then uploads these filled-in forms to a Web application. We adapted a version of ODK Collect, off the main development branch, that has a particularly rich user interface for tablets, developed for the New York City Department of Health. Although clients' first impressions of the app were favourable, more experience with the user interface was less satisfactory. The UI reflects the origins of ODK as a data collection system for cell phones. The user experience with this interface was inferior to systems such as Access, that allow flexible navigation around the form.

Our second version therefore developed a native (hand-written) Android interface, with the interface designed to mimic as much as possible the user experience with paper-based forms. The user experience

[1] Similar consideration lay behind the Jini system [8], developed by Sun Microsystems for network appliances.

 DOI: 10.12856/JHIA-2013-v1-i1-60

with this interface was more satisfactory. The issues with this version were with the development, and with the backend handling of data. Because of the amount of data being collected, the development was extremely repetitive and tedious, which in turn increased the possibility of programmer errors, as well as making maintenance and routine form changes more difficult than they should be. This mitigated against a sustainable approach to data collection. Another issue, shared with the original ODK app, is that data collection is essentially "untyped." The app itself has no knowledge of the data model for the data being collected. One of our eventual objectives with this research is to be able to analyse code that handles sensitive patient data, to ensure that potentially third-party code does nothing to leak sensitive information or violate patient confidentiality. A first step towards this goal is to make explicit the structure of the data being collected, and then make sure that each data item is handled (by software) in an appropriate manner.

The Aeetes approach built on these early prototypes, to take a new approach to developing data collection tools. As with the second prototype, the user interface is a native Android interface, with the potential for questionnaire designers to provide an arbitrarily rich user experience, using all the power of the Android GUI. However, rather than developing this interface by hand, it is instead generated by the Aeetes compiler, which is at the heart of the approach. The Aeetes markup language is used to describe both the user interface and the data model for an application. The markup language is based on the language for describing Android user interfaces, including views and view groups. It augments this with elements for specifying forms, such as sections and questions. At the heart of the markup language is a `<select>` element for specifying structured data collections, including drop-down lists, radio-button lists, check-box lists and tables.

From this input specification of the user interface, both an Android user interface, and an HTML preview, are generated. The Android UI includes a static description of the user interface for forms (in the Android markup language), a string resource file (one for each of the languages of the study), and Java classes for the user interface for each form. The HTML preview uses HTML5 elements to provide a preview of the forms that can be viewed through a Web browser. This is intended to allow data analysts, whose input is used to design the forms, to view and critique the forms without requiring access to an Android device. Currently the markup specification for a form is done using an XML editor, through a dialogue between data analysts and someone conversant with the markup language for forms. An interfactive form builder, based on the HTML preview, is under development, to allow data analysts to design forms directly. However to realize the full power of the Android UI, we expect that manual customization of the form markup specification will always be necessary.

The reason for form-specific classes in the output of the compiler is that the Aeetes compiler supports "typed" data collection, based on generating a model of the patient data being collected, and generating user interface code that is specialized to that model. Each form has a corresponding activity class, indexed by the patient data model, while each section of the form has a corresponding fragment class for managing the screen for that form section, indexed in turn by the corresponding submodel for that section of the form. Each input control is in turn indexed by the model for the data item being collected by that control. The intention is to eventually leverage this arrangement to augment the data model with security information, ensuring for example that only someone with proper authorization can view or edit a particular field in a form.

A data collection study typically consists of several forms, such as an enrolment form, a follow-up form and a lost-to-follow-up (LTFU) form. There is typically a high degree of overlap between these forms. For example, a form may specify laboratory test results, drug regimens, adherence to treatments, etc. The language for these forms must be carefully chosen to be consistent across multi-lingual studies (One of the countries we work with has at least two different languages, across different regions), and the skip logic must also be consistent between the forms. A markup document therefore describes all of the forms in a study, with sections and questions annotated to specify in which forms they appear, in the study. The main reason for describing forms this way, instead of separating the forms into separate specifications, is that it enables an overall data model for the study to be generated from the input specification, and each form related to the parts of this model for which it is designed to gather data.

 DOI: 10.12856/JHIA-2013-v1-i1-60

3.2 Data Model

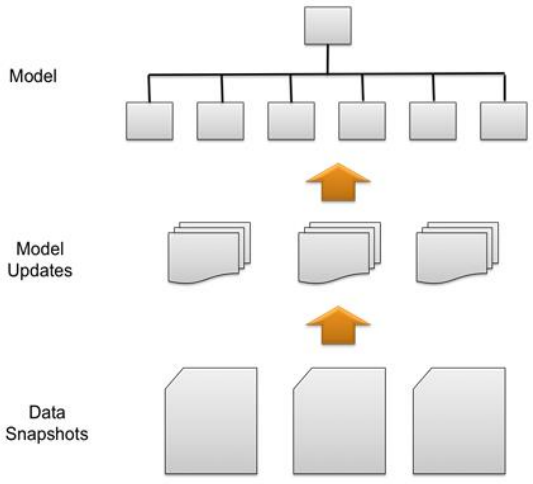
A major distinction between the Aeetes approach, and that of tools such as ODK and frameworks such as OpenMRS, is that we support typed data collection. The Aeetes compiler builds an explicit model of the patient data being collected, and statically ensures that the application gathering data is doing so in a type-safe manner, i.e., data is being handled in a manner consistent with the format of the data storage on the computer. As already noted, the motivation for taking this approach is to eventually support static code analysis to ensure that patient data is being handled by software in a secure fashion. For example, sensitive information should not be output to storage unencrypted, and should not be displayed on the screen for users without sufficient authorization to view the data.

By extracting the data model from the description of the data collection, we are able to relate each form generated from that description to the part of the patient data that it is responsible for collecting. The default choice is to extract the data model directly from the description of the questionnaire, since the two are usually closely related. For example, each section corresponds to a class for a persistent data object in the model. However where necessary, it may be useful to specify the model separate from the questions in that section, and a mapping used to describe where input data in the questionnaire is stored in the data model.

The default mapping from question type to data type is straightforward. Drop-down lists and radio-button lists are compiled to enumeration types, with the common special case of Boolean data recognized by the compiler. Checkbox lists, and tables with nullable rows, are compiled to lists. Where the rows in a checkbox list or table may have structured content, we may generate a class to box that content. In the case where different rows in a list have different types associated with them, we generate an abstract base class for the overall type, and concrete subclasses for each row in the list. Parts of a model, generated from a form specification, are also affected by the role of the form in the study. If a question is only asked in one form, and that form only administered once (e.g. in an enrolment form), then the data collected by that question comprises a field in the model. If the question is asked in multiple forms, or in a form that is administered several times (e.g. in a follow-up form), then the data model specifies a list of values, each one with an associated encounter date.

3.3 Data Handling

There are several levels at which the data being collected may be viewed:



Ultimately the objective is to build a patient data model, based on observations collected from various sources. In a data collection tool, these sources are forms filled in by clinicians as a result of patient encounters. However there is an intermediate stage between the data collection and its recording as part of the data model. Consider for example drug treatment regimens that a patient is currently undertaking. Data collection based on paper forms will record for each visit the dosage and frequency of

 DOI: 10.12856/JHIA-2013-v1-i1-60

administration for each of the drug treatments that a patient is receiving. Data models such as OpenMRS and HIV Cohorts Data Exchange Protocol (HICDEP) model treatment in terms of starting date, ending date, dosage and frequency. Data collection must therefore infer, based on differences between treatments from one encounter to the next, changes that have occurred in the treatment regimens. This is reflected by the intermediate stage above, between data collection and recording the data in the model.

An alternative approach is, rather than inferring changes in treatment from the data collected, to directly record the updates themselves. To do so requires a technically small, but conceptually large, change to the app. Rather than simply entering forms and uploading this data to a date warehouse, as with ODK, we may retain at least some part of the data in a database on the device. Data entry then consists not simply of filling in parts of a form that is initially empty, but also in some cases consists of updating parts of the database (in this case, to change drug treatments). This can avoid inference errors from data collection, and also serve as a useful data quality check at the clinic level. There are also other considerations. Providing even a simple EMR may enhance acceptance of the data collection tools in a clinic, since users will experience a tangible benefit from the use of the tools in their workflows.

There are other reasons for working at the level of data updates rather than data snapshots. In the peer-to-peer setting for this app, devices share data using "gossip protocols." Periodically a device selects a peer and exchanges its data with that peer. There are two broad approaches to performing this sharing. One approach is to exchange the parts of a database that have been modified on one device but not the other. Another approach is for each device to keep an operation log, recording the update operations that have been performed on the data. A device then exchanges its data with another device by sending the operations not yet seen on the peer device. Once received at the peer device, the update operations are replayed there to bring the peer device up to the same state as the source device. The advantage of this update-based approach is that there are protocols that substantially reduce the amount of metadata that must be exchanged for the peer devices to identify the information that must be exchanged, while there are issues with extending these protocols to synchronization based on exchanging parts of the database [9].

To support the update-based approach, parts of the patient data model may be cached on the device. A new form may then be populated with the cached contents of the previous instance of this form for the patient, to allow direct updating of the date. These updates, along with updates for filling in other parts of the forms, are exchanged peer-to-peer with other devices, and eventually uploaded to the data warehouse for data analysis. Conceptually this uploading could be performed on a continuous basis, providing the data warehouse with an "almost-real-time" view of the clinic data. Practically, there are good reasons to batch the updates for uploading.

3.4 Security

Security of the data being collected is obviously paramount. This is particularly true when sensitive patient data is being stored on relatively small devices that may be lost or stolen. One of the challenges with deploying our approach has been to convince our collaborators not to adopt desktop computers for their clinic IT systems. Their understandable motivation has been that it is more difficult for a desktop computer to be stolen than a laptop or tablet. Nevertheless, the days of the desktop personal computer are numbered, and we have prevailed upon them to accept the future. The challenge remains then to make sure that patient data is properly protected.

It might be considered that, as long as no identifying patient information is kept on the device, data loss is no worse than publishing anonymized patient data. For example, for the purposes of epidemiological studies, patients can be identified by a study identifier rather than their medical record number. There are several objections to this. First, there are variables in the data, such as date of birth, that are considered quasi-identifiers, but which epidemiologists wish to collect for analysis. There are now well-known de-anonymizing attacks, such as the Netflix attack, that can be used to expose the identity of parties in a dataset from which identifying information has supposedly been removed. These attacks are based on correlating information in the dataset with another dataset that still retains identifying information. Second, there are scenarios where it may in fact be necessary to store the patient record number on the device. For example, much of the data useful for studies is stored in register books of various kinds, as mandated by government regulations, with patients identified in registers by their medical record number. Requiring a data entry person to look up a patient's study identifier for every line

 DOI: 10.12856/JHIA-2013-v1-i1-60

in a register book is infeasible. This mapping must be performed by the app itself, even though the medical record number should never be part of the data that is uploaded to the data warehouse.

Our threat model assumes that any attacker who has access to a device can eventually bypass any access restrictions on the device, be they in the app or in the underlying Android/Linux operating system, and access any data stored on the device. Our strategy for protecting the data on a device is to keep the data encrypted in storage, and require an attacker to have three things in order to be able to compromise the data. Although Android supports encryption of the file system, we do not make use of this for several reasons. File system encryption is based on password-based encryption, and this does not provide a sufficient level of security. Furthermore, there is a problem if the user forgets their (Android) password. We instead implement our own encryption of patient data, and provide escrow for information that is required to access the data.

The data for a device is encrypted using an AES secret key. This provides a much higher level of security than a user password. Clearly storing this password on the device itself is insufficient, given our threat model. We instead store the secret encryption key off the device, as a QR code on a card. A user who logs in to access patient data is required to present the QR code. The app takes a picture using the device's camera, decodes the encryption key stored in the QR code, and can then use this key to decrypt data stored on the device. Obviously if both a device and its QR code are stolen, or if the QR code is photographed and the device stolen, this scheme will be subverted. Our recommended practice is that the QR code stays under the supervision of the administrator at all times, stored in a locked cabinet that only the administrator has access to. A user logging in will need to physically go to the administrator to obtain the QR code. A device that has been inactive for a period may require a user to authenticate with their password, while the device remembers the QR code. After a longer period, such as a lunch break, the device should forget the QR code and require another visit to the administrator to obtain the code. The ultimate goal is to ensure that an attacker who steals a device does not have access to the QR code anywhere on the device, and therefore cannot compromise the confidentiality of the data.

We still plan for the worst-case scenario, where a QR code is compromised in some way. We pair each device with its own private encryption key. We do this using a master password, unique for each device, and used as an authentication key for the encryption key. The key stored as a QR code is the encryption key, encrypted in turn using the master password. Both the encryption key and the master password are created in advance, with the QR code, by central IT administration. As part of installing the app on a device, the administrator enters the master password and presents the QR code to the camera. The app initialization decodes the password-encrypted encryption key in the QR code, decrypts and authenticates the encryption key using the master password, and then encrypts the master password using the administrator password and stores it in a record for the administrator in the user database on the device. Every time the administrator creates a new user account, the master password is encrypted using that user's password and stored with their record in the user database. Therefore every user, once they have authenticated with their user password, has access to the master password in their use of the app. To log in, a user must present both the QR code and their user password. Once the user is authenticated, their password is used to decrypt the master password, and the master password is in turn used to decrypt and authenticate the encryption key provided in the QR code.

An attacker wishing to access the data on the device must therefore have the physical device itself, a copy of the QR code that is specific to that device itself, and the password of a user on that device. Clearly all of these can be obtained via an insider attack, unless we have procedures such as described above for preventing theft of the QR code. The user password provides a line of defence against the scenario where for example a thief breaks in and steals both a device and a QR code. The last line of defence is a "kill switch" for the device, administered in both a push-based way (using Google Cloud Messaging) and in a pull-based way (if the app attempts its normal background processing by contacting the data warehouse in the cloud).

Peer-to-peer data sharing is complicated by the use of per-device encryption keys, since we cannot assume a single shared encryption key for sharing all data across devices. Instead we require every device to have its own RSA public-private key pair, generated as part of installing the app on the device. Devices share their public encryption keys as part of discovery, and a source device sharing updates with a sink device must first generate an AES session key for the duration of the synchronization, encrypt the session key using the sink device's public key, encrypt updates to be sent to the sink device using the session key,

 DOI: 10.12856/JHIA-2013-v1-i1-60

and then send the encrypted session key and updates to the sink device. A very similar protocol is used for pushing updates to the data warehouse.

4 Discussion

Aeetes is not intended necessarily to replace existing PC-based and enterprise-based approaches to data collection. Rather it fills a gap that we have perceived in such tools, between simple Access-based approaches and more sophisticated approaches such as OpenMRS. We believe (or least hope) that the days of Access-based approaches are numbered. Our reason for hoping this is true is due to the security issues with Windows-based, Access-based approaches to data collection and storage. Today the private records of hundreds of thousands of HIV/AIDS patients reside unencrypted on Windows devices potentially infected by malware. The Aeetes approach suggests a possible migration path away from Access to more sophisticated and more secure approaches.

There is much that we would still like to accomplish, to make Aeetes a viable long-term alternative to Access systems. Our point of comparison for Access-based approaches is the EPI-INFO system supported by CDC. The two advantages cited for EPI-INFO are: (a) the ability to automatically generate a database schema from a user interface, and (b) the ability to program extensions, particularly for data analyses. The Aeetes compiler provides the former ability, and our next task is to develop a framework for supporting the latter. One issue here is that the Android platform has a difficult programming model, using asynchrony and callbacks ubiquitously to keep long-running computations off the main UI thread, and with an application life cycle model that is more complicated than that for more conventional operating systems that simply swap processes in and out transparently to application programmers. We are developing a domain-specific programming language for Aeetes extensions with the intention that programmers of extensions will be able to avoid the use of callbacks entirely. The kinds of extensions that we are focused on are on-device analyses of patient data, based on the experience with EPI-INFO.

We have not said anything about databases, because for now we have found it unnecessary to use databases. The app contains several content providers (for users, forms, form instances, etc), implemented using SQLite, but mostly these are simple tables. The implementation of role-based access control is the only place where we found it necessary to model relationships in the database schema. The relationships in patient data are one-to-one or one-to-many, represented in our schemas as lists. So a document-oriented NoSQL database such as Couchbase would suffice to store patient data. For now, we are storing patient data in files because of the need to store the data encrypted. It also keeps the runtime of a non-standard database management system outside our trusted computing base. We expect to revisit this at some point in the future, since some form of map-reduce-based parallel processing may be necessary to support efficient data analyses.

OpenMRS has demonstrated the value of having an extensible component framework for building EMRs in an open source community. Although designed initially as a data collection system, Aeertes could become the foundation for an EMR, less ambitious and simpler than OpenMRS, but perhaps more appropriate to some situations where IT support is lacking. It is therefore intended to occupy a different market niche from OpenMRS. We have some hope that some of Aeertes approach, in particular the use of a compiler to generate concept schemas, may eventually make its way into the OpenMRS community, since it can support more reliable and eventually more secure data handling.

Other systems have also explored the space of peer-to-peer architectures for data collection. AndroidOpenMRS combines an interface implemented using ODK with an implementation of the OpenMRS concept dictionary in SQLite, and includes peer-to-peer data replication. The focus in Aeetes has been on an improved user interface, by compiling from input specifications to native Android user interfaces, explicitly modelling the data being collected for reliability and security, protecting the confidentiality of patient data against attacks such as device theft.

Acknowledgements

Thanks to Kathryn Anastos, Paul Biondich, Hamish Fraser, Edward Friedman, Bobby Jefferson, Andrew Kanter and Tom Routen for helpful conversations. This work was supported by NIH grant number

 DOI: 10.12856/JHIA-2013-v1-i1-60

1U01AI096299-01 (Central Africa IEDEA), in collaboration with Kathryn Anastos, Donald Hoover, Denish Nash and Qiuhu Shi.

References

[1] *"EPI INFO."* Download available from http://wwwn.cdc.gov/epiinfo.
[2] *"IQCare".* Download available form http://fgiqcare.codeplex.com.
[3] *"OpenMRS."* Download available from http://www.openmrs.org.
[4] *"Sana Mobile."* Download available from http://sana.mit.edu.
[5] *"Open Source Data Collection in the Developing World."* Y. Anokwa, C. Hartung, W. Brunette, A. Lerer, G. Borriello. IEEE Computer, 2009.
[6] *"Evaluation of an Android-based mHeath System for Population Surveillance in Developing Countries."* Rajput et al, AMIA, 2011.
[7] *"Experience Implementing Electronic Health Records in Three East African Countries,"* Tierney et al, AMIA Annual Symposium Proceedings, 2010.
[8] *"Jini Technology: An Overview."* S. I. Kumaran, Pearson Education, 2001.
[9] "Replicated Data Management for Mobile Computing." D. Terry, Morgan & Claypool, 2008.

 DOI: 10.12856/JHIA-2013-v1-i1-60