AN ARCHITECTURE FOR BUILDING COST-EFFECTIVE, FLEXIBLE AND INTEROPERABLE XML HEALTHCARD APPLICATIONS

A. Georgoulas* and D. Koutsouris*

* National Technical University of Athens/Biomedical Engineering Laboratory, Athens, Greece

ageorg@biomed.ntua.gr

Abstract: The work reported in this paper concerns the design and implementation of an object-oriented framework for developing cost-effective, flexible and interoperable healthcard solutions in Java, that make use of ISO 7816-compliant smart cards, on different target platforms such as Windows, network computers and handheld devices. The framework comprises both a methodology and a set of high-level Java Application Programming Interfaces (APIs) which facilitate healthcard applications development, covering all healthcare specific requirements and needs. Our implementation is being demonstrated and tested in terms of efficiency, compatibility, portability and performance using sample healthcard applications that have been developed for this reason.

Introduction

While the use of smart cards in healthcare promises to improve both the quality and the availability of healthcare services, providing a convenient and secure medium for storing and communicating medical information, the adoption of healthcard implementations within the National Healthcare Systems and IT strategies is relatively slow, blocked by a number of technical, administrative and medical barriers. The most important barrier seems to be that existing healthcard solutions do not comply with a number of requirements and qualitative parameters of e-health (and e-government in general) [1]: compatibility and interoperation with the existing IT infrastructure, extensibility, independence from specific suppliers & vendors and last but not least, cost-effectiveness.

The current paper addresses the above challenges by proposing a reference architecture for developing cost-effective, flexible and interoperable XML healthcard applications, using open source software.

The paper is organized as follows: In the next Section we present some background information regarding current smart card application development problems and limitations. The ensuing Section introduces the overall architecture followed by the specific implementation details. In the *Results* section the architecture's innovative features and performance testing outcomes are presented through selected pilot applications, while the last Section summarizes the results and draws the major conclusions.

Background Information

The closed architecture of today's smart card operation systems and lack of high-level Application Programming Interfaces (APIs), have turned card-application development to a very difficult and time-consuming task, requiring high-specialized programmers and dedicated software tools [2]. Current healthcard solutions are proprietary, expensive in development and maintenance, and in most cases face difficulties to interoperate and exchange information with existing medical applications or databases, due to lack of standards (both medical and technical) or partial implementation of existing ones. Applications usually work only with a specific type of card, reader and platform, resulting in increased dependence on specific vendors and suppliers.

Card Application Development: Smart card applications essentially comprise two parts: the application part resident on the card, henceforth called *card-resident application* and the application part running in the computer henceforth called the *card-using* or *host applicatio*n [2]. File system cards represents the most common smart card type in use today, being a reliable, secure and cost-effective solution (in contrast with the - yet expensive - Java Cards). Such a file system (defined in part 4 of the ISO7816 standard [3]) is built based on three components: the Master File (MF), which is the root of the file system, the Dedicated File (DF), which is comparable to a directory of a UNIX or PC file system, and the Elementary File (EF), containing the actual data.

The card-resident part comprises the data kept on the card in the context of an application. Thus, developing a card-side application involves building a complex file structure, where a single DF contains several DFs and EFs representing the required data fields. However, this approach has serious drawbacks: accessing these data requires a dedicated external application, which has knowledge of all card-side implementation details (file paths, file names and semantics) from the beginning. Furthermore, if a new standard evolves, regarding e.g. the Emergency Medical Record, the application file structure has to change completely in order to include new fields or remove others. This could be a daunting task, especially when talking for applications containing large number of individual fields. Therefore it is hard to communicate and keep up with the rest medical applications and databases of a healthcare system.

Architecture Overview

As shown in Figure. 1, the architecture comprises four separate layers, which establish the necessary interfaces between the system's distributed parts, from the healthcard to the end-user application [4]. In the following paragraphs we attempt a closer look to each layer, explaining the functionality and implementation details of the major structural modules.

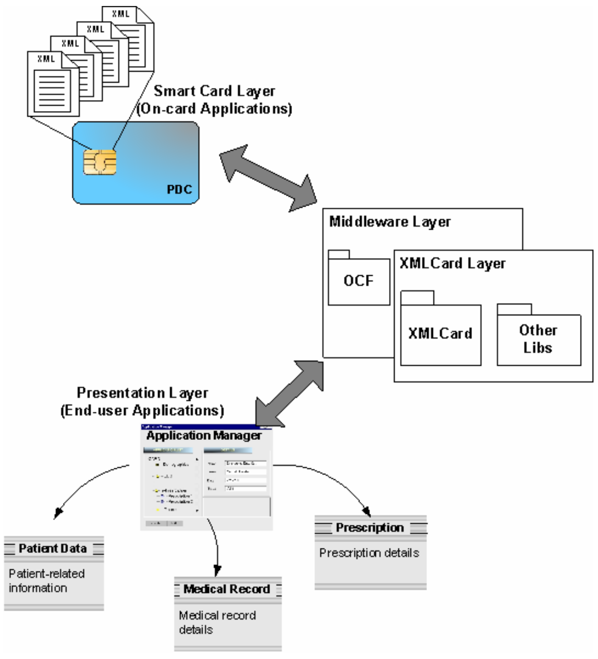
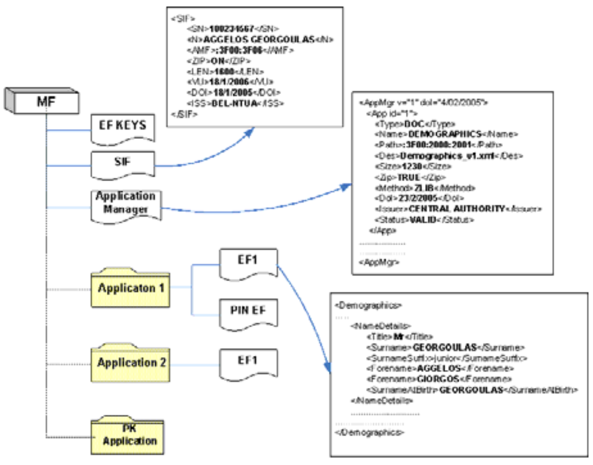


Figure 1: Overall Architecture

Smart Card Layer: It concerns the applications stored onto the smart card chip. The architecture introduces the XML document model for developing card-side applications. As shown in Figure 2, each card-side application (e.g. Demographics, e-Prescription, etc) is represented by an XML document, which holds both the application structure (meta-data) as well as the actual data. This document is stored on single card EF files, one for each separate application.

Thereby we avoid building complex card file structures, which is very difficult to modify or update in the future. Furthermore, the card application is self-descriptive since it holds all necessary information for data binding (application name, data items names, etc), and no longer depends on external resources.

Application management is implemented through the Application Manager component, an innovative control mechanism, which also facilitates multi-application capabilities on the card. The Application Manager, which comprises an XML document stored in a single EF card file, serves as the main entry point for interaction with the external application. This document contains all necessary information for managing the card-residing applications without prior knowledge of the card file structure.



```
<SIF>
    <SN>100234667</SN>
    <N>AGGELOS GEORGOULAS</N>
    <AMF>:3F00:3F06</AMF>
    <ZIP>ON</ZIP>
    <LEN>1600</LEN>
    <VU>18/1/2006</VU>
    <DOI>18/1/2005</DOI>
    <ISS>DEL-NTUA</ISS>
</SIF>
```

```
<AppMgr v="1" doi="4/02/2005">
  <App id="1">
    <Type>DOC</Type>
    <Name>DEMOGRAPHICS</Name>
    <Path>:3F00:2000:2001</Path>
    <Des>Demographics_v1.xml</Des>
    <Size>1230</Size>
    <Zip>TRUE</Zip>
    <Method>ZLIB</Method>
    <Doi>23/2/2005</Doi>
    <Issuer>CENTRAL AUTHORITY</Issuer>
    <Status>VALID</Status>
  </App>
...............
...............
<AppMgr>
```

```
<Demographics>
.....
    <NameDetails>
        <Title>Mr</Title>
        <Surname>GEORGOULAS</Surname>
        <SurnameSuffix>junior</SurnameSuffix>
        <Forename>AGGELOS</Forename>
        <Forename>GIORGOS</Forename>
        <SurnameAtBirth>GEORGOULAS</SurnameAtBirth>
    </NameDetails>
    ..........................
..........................
</Demographics>
```

Figure 2: Card-side application model

Security issues: the proposed model enforces to use the same security level for the whole document (e.g. one or two PINs, secure messaging, etc) for the whole document, since the whole application is stored within a single EF. However, the architecture offers the flexibility of using advanced security measures at the application level, such as XML Signature, XML Access Control, etc.

Smart card capacity: Smart card's limited memory capacity seems to be the major problem regarding the use of XML documents for representing card-side applications. While XML format makes the use and interchange of data easier and more user-configurable, it substantially increases the size of the file over the size when the same data is represented in its raw format. This inherent inflation of the file sizes - or "bloat" - can represent an average size of 100-400%, being thus a critical problem when data has to be transmitted quickly or stored compactly as in our case [5]. The common smart cards in use today, offer about 8-16 Kbytes of available memory for storing data (may reach 32 to 64 Kbytes for more expensive cards). This is insufficient estimating that even a small XML medical document would have an average size of 4-10 Kbytes. The most common approach for overcoming this limitation is the application of lossless data compression techniques, which would decrease the size of the XML documents stored on the card, without wasting the benefits came up by the use of XML format. In this respect we performed a series of tests focusing on the performance of the available compression techniques. The outcome of these tests is presented in detail in the *Results* section.

Middleware Layer: The middleware layer serves as an intermediate between the on-card applications and the host system. The communication is carried out through the card reader device plugged (or integrated) to the host system. This requires the appropriate software components implementing the low-level communication protocols for data and commands interchange. The primary goal here is to hide complexity, while achieving maximum independence from card and reader manufacturers. Our choice for the middleware layer is the OpenCard Framework (OCF) [6]. The OpenCard Framework is an open standard, providing both a reference architecture and a set of java APIs that enable application developers and service providers to build and deploy smart card aware solutions in any OpenCard-compliant environment. The applications developed against OCF high-level APIs can work with different cards and readers by just plugging the corresponding *CardService* and *CardTerminal* implementations (usually offered by the manufacturers) into the architecture [2]. This approach ensures that differences or changes in the card operating system (e.g. use of other smart card), in the card reader or in the application management scheme used by the card issuer do not impact the user application code.

XMLCard Library: The XMLCard library is the core component of the architecture, providing a high-level interface to access and use the on-card applications. The library (*edu.biomed.ntua.xmlcard*) has been developed in pure Java and it exclusively depends to open source third party resources. Figure 3 describes the *xmlcard* package:

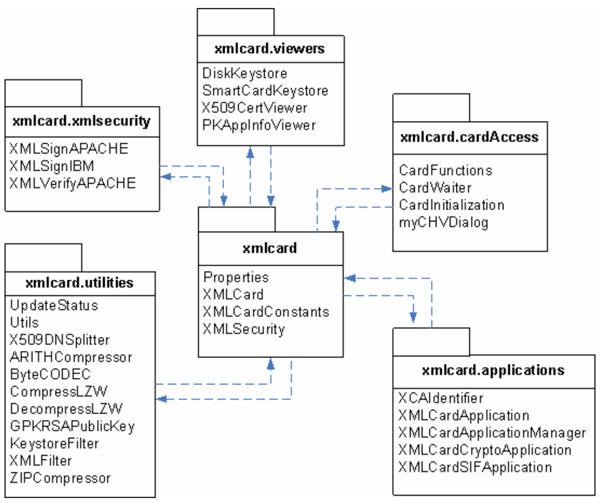


Figure 3: *xmlcard* package

The *XMLCard* class represents the actual smart card, providing to the programmers a high-level API to access and process the on-card applications. The *XMLCardApplication* class models the on-card XML application, hiding the underlying complexity, enabling the transparent exchange of XML documents between the host PC and the smart card. All smart card-related functionality (PIN protection, card file selection, etc) is handled in a lower level of the architecture, totally transparently to the end-application programmer. The API also includes a set of static methods (*XMLSecurity* class) in order to perform standard XML Security-related functionality (signing, verifying, etc) both on the PC and the smart card. Using this API, developers can link the end-user applications to the on-card applications with only a few lines of code, focusing more to the graphical interface of the application.

The library is highly configurable through a flexible and extendible properties mechanism. Using this feature, the users can configure a variety of system operation parameters, like the preferred XML signature method, the cryptographic keys to use to use, etc. Furthermore, the developers of end-user applications can use this mechanism to add their custom parameters.

Presentation Layer: It is the upper layer of the architecture and includes the various end-user medical applications that make use of the healthcard. As it was described earlier, using the *XMLCard* API the user applications interact with the healthcard transparently, actually exchanging XML documents. The next step is to present the application data to the end-users through an interactive graphical user interface (GUI). Even there is no constrain about the type of the end-user application, the use of java and XML format enables utilization of more *flexible* solutions for data presentation than just developing an ordinary desktop application (like a standard java swing application). For example, using XSL transformations we can present the application within a common web browser in the context of a web application. Separating the document's content and the document's styling information allows displaying the same document on different media (like PC screen, mobile devices, etc), and it also enables users to view the document according to their preferences and abilities, just by modifying the *stylesheet*. Furthermore, any time a medical application is modified (e.g. to conform to the latest related standard) there is no need to change the end-user application code, except updating the corresponding stylesheet. Another promising approach is the usage of techniques that separate the application logic (which can be written in Java) from the user interface, which can be described in a markup language as XML [7].

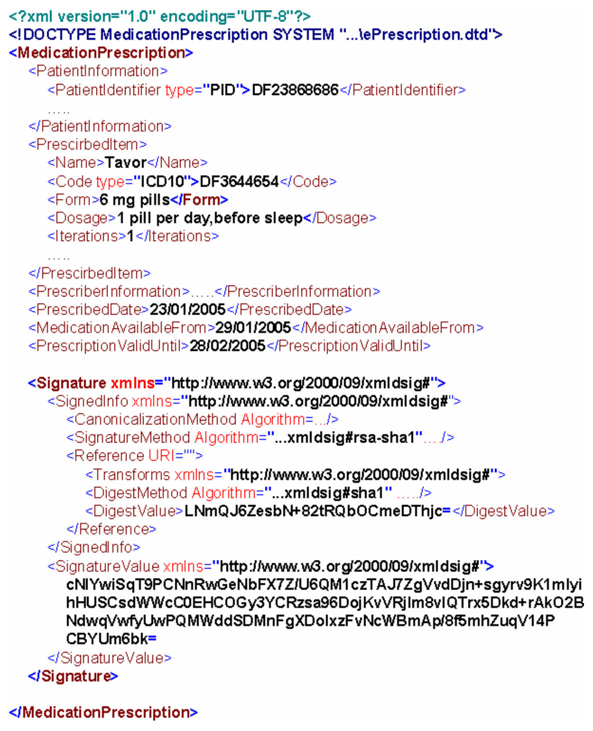
Application Deployment: As it is mentioned earlier, the architecture does not set any constrain regarding the type of the end-user application. Similarly, application deployment can be performed in the way that fits better to the specific application. If we talk about a web application, this is available through a common web browser, so there is no need to go through a specific installation procedure. However, if we are talking of a rich desktop application which needs installation to the target PC or other device, it is clear that we need a more flexible approach, in order to fully exploit the architecture features (portability, cross-platform interoperability, easy of use, etc). In this respect, we have tested and propose the use of Java Web Start. Java Web Start [8] provides a platform-independent, secure, and robust deployment technology. It enables developers to deploy full-featured applications to end-users by making the applications available on a standard web server. Java Web Start caches resources locally on the disk, but also provides a secure execution environment and a virtually transparent updating facility for applications. For security reasons, Java Web Start applications run by default in a restricted environment, known as a *java sandbox*. However, if the application's JAR files are digitally signed by a trusted authority we can easily provide for functionality that goes beyond what is allowed in the *sandbox*.

Results

The architecture's features were tested and presented through a pilot healthcard system supporting some basic functionality. For the Patient Data Card (PDC) we have chosen two characteristic medical applications, the *Electronic Medical Summary* (eMS) and the *Electronic Prescription*, while the Healthcare Professional Card (HPC) was mainly used for performing security functions (e.g. digital signature).

For the tests we used the popular Gemplus GPK 8K/16K file system cards, as well as the IBM MFC card. Since GPK cards offer additional cryptographic capabilities on the card, it was used for both PDC and HPC, while MFC card was used only for PDC. Regarding the card reader devices, we use simple contact readers, like Gemplus GemPC410 and Schlumberger Reflex 60, connected to serial or USB port. For both cards and readers, the OCF plug-ins are freely available by the manufacturers over the internet [9] [10].

e-Prescription application: For the purposes of our study, we developed a custom prescription document (Figure 4), based on the most common standardized data sets, like for example the ones proposed by NETLINK project [11] and CEN/TC 251 [12].

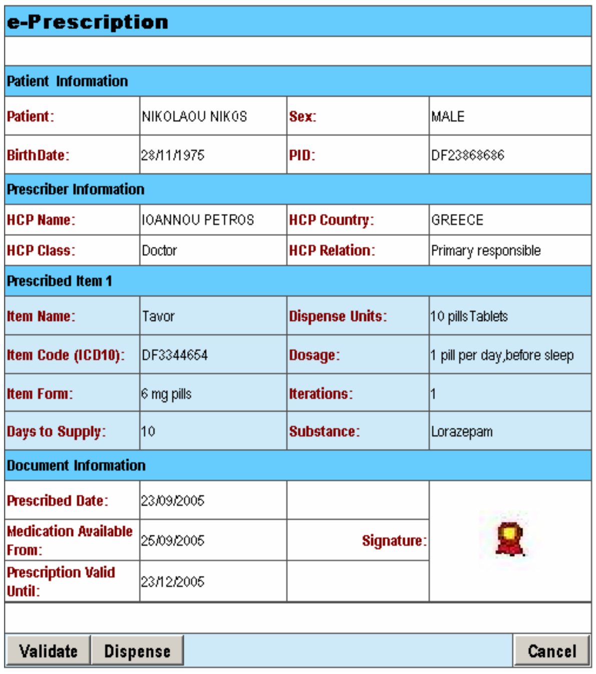


```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE MedicationPrescription SYSTEM "...\ePrescription.dtd">
<MedicationPrescription>
   <PatientInformation>
      <PatientIdentifier type="PID">DF23868686</PatientIdentifier>
      .....
   </PatientInformation>
   <PrescirbedItem>
      <Name>Tavor</Name>
      <Code type="ICD10">DF3644654</Code>
      <Form>6 mg pills</Form>
      <Dosage>1 pill per day,before sleep</Dosage>
      <Iterations>1</Iterations>
      .....
   </PrescirbedItem>
   <PrescriberInformation>.....</PrescriberInformation>
   <PrescribedDate>23/01/2005</PrescribedDate>
   <MedicationAvailableFrom>29/01/2005</MedicationAvailableFrom>
   <PrescriptionValidUntil>28/02/2005</PrescriptionValidUntil>

   <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
      <SignedInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
         <CanonicalizationMethod Algorithm=.../>
         <SignatureMethod Algorithm="...xmldsig#rsa-sha1"..../>
         <Reference URI="">
            <Transforms xmlns="http://www.w3.org/2000/09/xmldsig#">
            <DigestMethod Algorithm="...xmldsig#sha1" ...../>
            <DigestValue>LNmQJ6ZesbN+82tRQbOCmeDThjc=</DigestValue>
         </Reference>
      </SignedInfo>
      <SignatureValue xmlns="http://www.w3.org/2000/09/xmldsig#">
         cNIYwiSqT9PCNnRwGeNbFX7Z/U6QM1czTAJ7ZgVvdDjn+sgyrv9K1mlyi
         hHUSCsdWWcC0EHCOGy3YCRzsa96DojKvVRjlm8vIQTrx5Dkd+rAkO2B
         NdwqVwfyUwPQMWddSDMnFgXDolxzFvNcWBmAp/8f5mhZuqV14P
         CBYUm6bk=
      </SignatureValue>
   </Signature>

</MedicationPrescription>
```

Figure 4: Digitally signed e-Prescription XML document

The architecture also provides the flexibility to use different document standards in the future without any modifications on the applications' code. The integrity and the origin of the prescription are protected using W3C-compliant XML digital signatures [13] generated by the HPC (replacing handwritten signatures used today).

Figure 5 shows how the document is presented in the context of a desktop Java application. The e-Prescription application is based on the existing paper-based business case. The application scenario includes two basic functions, *prescribing* and *dispensing*. The link between these two functions is the PDC, which carries the actual information (prescription document), forming a *Virtual Portable Network* between the prescribing and the dispensing agents.

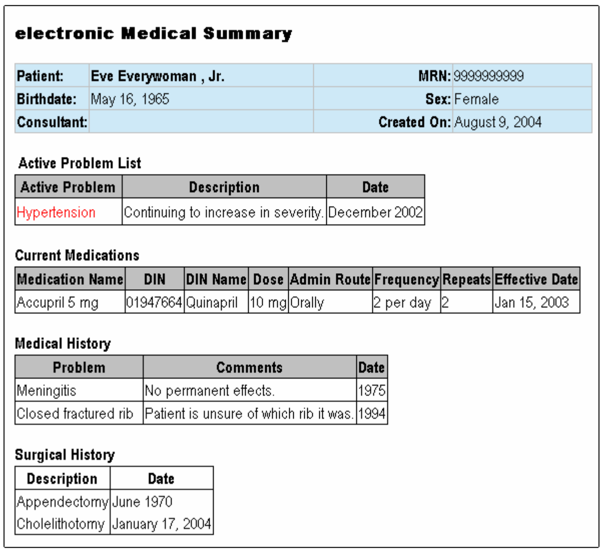


e-Prescription

Patient Information			
Patient:	NIKOLAOU NIKOS	**Sex:**	MALE
BirthDate:	28/11/1975	**PID:**	DF23868686
Prescriber Information			
HCP Name:	IOANNOU PETROS	**HCP Country:**	GREECE
HCP Class:	Doctor	**HCP Relation:**	Primary responsible
Prescribed Item 1			
Item Name:	Tavor	**Dispense Units:**	10 pills Tablets
Item Code (ICD10):	DF3344654	**Dosage:**	1 pill per day,before sleep
Item Form:	6 mg pills	**Iterations:**	1
Days to Supply:	10	**Substance:**	Lorazepam
Document Information			
Prescribed Date:	23/09/2005		
Medication Available From:	25/09/2005	**Signature:**	
Prescription Valid Until:	23/12/2005		

Figure 5: e-Prescription application

eMS application: An electronic medical summary is a subset of patient data suitable for communication among primary health care practitioners and other health care providers for the purpose of sharing the care of a patient. An e-MS for a patient includes all the information (i.e. Current Medications, Medical History, Surgical History, Allergies, Immunization, etc) essential for ensuring the patient receives exemplary health care from every health care provider involved in his or her care, including primary care physicians and third-party care providers such as specialists, diagnostic technicians, and tertiary (emergency) care providers. Figure 6 shows how the e-MS application (as defined by the HL7/CDA standard [14]) is presented within a common web browser in the context of a web application.



electronic Medical Summary

Patient:	Eve Everywoman , Jr.	MRN:	9999999999
Birthdate:	May 16, 1965	Sex:	Female
Consultant:		Created On:	August 9, 2004

Active Problem List

Active Problem	Description	Date
Hypertension	Continuing to increase in severity.	December 2002

Current Medications

Medication Name	DIN	DIN Name	Dose	Admin Route	Frequency	Repeats	Effective Date
Accupril 5 mg	01947664	Quinapril	10 mg	Orally	2 per day	2	Jan 15, 2003

Medical History

Problem	Comments	Date
Meningitis	No permanent effects.	1975
Closed fractured rib	Patient is unsure of which rib it was.	1994

Surgical History

Description	Date
Appendectomy	June 1970
Cholelithotomy	January 17, 2004

Figure 6: e-MS application

XML Compression Testing: Text compression algorithms have been the subject of extensive research over many years and are at an advanced stage of development. Any algorithm of this type can be easily applied to the text representation of XML documents. Our test involve compressing a variety of XML documents (more than 1000 files) using the most common available compression algorithms. The documents chosen for the corpus reflect a wide a variety of sources and file sizes (0 to 100.000 bytes), in order to avoid skewing our results to a particular kind of data. Each document is compressed through a custom Java application developed for this purpose, using the following (free) compression libraries implemented in java: *zlib* [15], *bzip2* [16], *gzip* [17] and *Arithmetic Coding* [18].

As resulted from our tests, brute-force compression seems to be the best choice, since XML is text with a lot of repetition, so it compresses surprisingly well. Some of the results are shown in the following composite diagram (Figure 7).

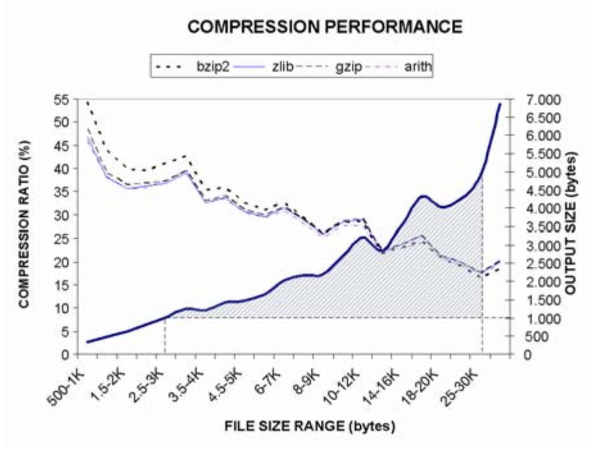


Figure 7: Compression performance diagram

The left axis shows the compression ratio (i.e. the size of compressed file as percentage of the original file), while the right axis represents the final size of the compressed file (using *zlib*), which is essential for our case. In general, all methods perform similarly well, having an average compression ratio of better than 4:1. *Zlib* has slightly better performance (average compression rate 4,28:1), especially for files size 500-10.000 bytes, while only *gzip* was slightly better for larger files (>10.000 bytes).

Our experiment also includes testing the space requirements of the same medical application using both the card files–based and the XML document method. As mentioned earlier, using the card-file method each data item is stored in the corresponding EF file on the card. Each card file (EF or DF) is represented within the Card Operating System through its "descriptor". For example, in GPK8K cards each EF has a 22 bytes long descriptor, while each DF has a 20 bytes long descriptor. Thus, if an application has 50 distinct data fields the additional space required (overhead) on the card will be 50*22 = 1100 bytes.

If the same information is represented using XML format, the outcome will be an XML document much bigger than the actual data size (due to XML "bloat"). However, our tests show that applying text compression techniques the document size is dramatically reduced, especially as the original document complexity and size grows. The results presented in the following table concern three sample medical applications, the e-MS, the Emergency Medical Set (EDS) as defined by the CEN/TC 251 specification [12] and a custom implementation of en electronic prescription document (electronically signed).

Table 1: Space requirements testing results

Application:	e-MS	EDS	e-Prescription (signed)
Size (XML):	12574	3240	4292
Size (data)	2069	808	527
No of fields	185	49	52
EF overhead	4070	1078	1144
zlib size	**1221**	**998**	**1530**
Card File	**6139**	**1886**	**1671**

Taking for example the eMS application, if we use the card file approach it will require building a complex structure of 185 EF files on the smart card, using a total space of (at least) 6139 bytes for storing 2069 bytes of information. Using the XML document approach (in conjunction with *zlib* compression) will require a single EF with total capacity of only 1221 bytes. In every case the compressed XML document requires less space, even when it is digitally signed. As shown in the *Compression Performance* diagram (right axis), for the range of 5000 to 16000 bytes (which is the majority of the medical XML documents used) the XML document will be compressed to less than 2.000-4.000 bytes, while for the range of 2000 to 5000 bytes (which represents the majority of administrative applications) the XML document will be compressed to less than 1.000-1.500 bytes. This means that a common smart card (with average memory of 12000 bytes) may hold an adequate number of distinct XML medical applications, overcoming the “bloat” that XML causes.

Conclusively we could say that development process was very fast both for the card-side and the host-side parts. Most of the effort was put into the design of the user interface (layout, user options, etc) and the packaging of the application. Development cost was very low, and no additional cost for software licenses was required in any stage of the process, since all used libraries and tools were either open source or freely available. The applications worked fine with all tested smart cards and readers without any modification in the source code. On-card applications (XML documents) were updated with no difficulties, while the GUI was automatically adapted to the new documents through the corresponding stylesheets. Security functionality was also tested (especially PIN protection and digital signatures) in terms of transparency, usability, performance and reliability, showing satisfactory behaviour.

Conclusions & Future Work

This paper presents a reference architecture for developing cost-effective, flexible and interoperable healthcard solutions in Java, that make use of ISO 7816-compliant smart cards, on different target platforms such as Windows, network computers and handheld devices.

From the technical point of view, the proposed architecture brings an innovative approach to card application development, providing the tools for transparent interaction between the on-card and the end-user applications. The architecture’s novelty relies less in the technologies used, than in the way it combines them in order to provide a comprehensive, end-to-end methodology for building open source healthcard systems. This will enable the evolvement of new, value-added healthcard services and applications, thus accelerating the acceptance and deployment of smart card solutions within the National Healthcare Systems and IT strategies.

Work so far includes the development of the core *XMLCard* library, which is based on freely available open source java libraries. *XMLCard* is being tested in terms of efficiency, compatibility, portability and performance. Furthermore, sample on-card and end-user medical applications are being developed using various presentation techniques.

Future work includes overall performance testing of the whole supported functionality (compression, cryptographic operations, XML transformations), as well as testing the capability of smooth integration and interoperation with existing systems and applications.

References

[1] DIETZEL G.T.W., (2003), ‘The electronic Health Card as a Tool for Seamless Care’, Advanced Health Telematics and Telemedicine - The Magdeburg Expert Summit Textbook, Edited by: B. Blobel and P. Pharow, Vol. **96** in the "Studies in Health Technology and Informatics”, (Ed): IOS Press, pp. 213-217

[2] HANSMANN U., NICKLOUS M.S., SHACK T., SELIGER F. (2000), ‘Smart card application development using Java’, (Ed): Springer.

[3] International Organisation for Standardization, ISO/IEC 7816-1,2,3,4,5,6 Standards, Internet Site Address: www.iso.org

[4] GEORGOULAS A., GIAKOUMAKI A., KOUTSOURIS D. (2003), ‘A Multi-layered Architecture for the Development of Smart Card-based Healthcare Applications’, Proc. 25th Annual International Conf. of the IEEE Engineering in Medicine and Biology Society (EMBS), Cancun, Mexico, 2003.

[5] MEDGGINSON D. (2005), ‘XML Performance and Size‘, (Ed): Addison Wesley Professional, Retrieved April 15, 2005, Internet site address:

[6] www.awprofessional.com/articles/article.asp?p =367637. The OpenCard Framework (OCF), Internet site address: http://www.opencard.org

[7] XUL language (XML User Interface Language), Internet site address: http://xul.sourceforge.net/

[8] Java WebStart Technology, Internet site address: http://java.sun.com/j2se/1.5.0/docs/guide/javaws/index.html

[9] The OpenCard @ Gemplus Portal, Internet site address: http://www.gemplus.com/techno/opencard/

[10] Reflex60 Open Card Framework Driver, Forge Information Technology, Internet Site Address: www.forge.com.au/Research/products/freeware.html

[11] Netlink G8 Healthcare Data Card project, Internet Site Address: http://www1.va.gov/card/

[12] European Standardization of Health Informatics, CEN/TC 251, Internet Site Address: www.centc251.org/

[13] The Internet Engineering Task Force (IETF), RFC 3075, ‘XML-Signature Syntax and Processing’, Internet Site Address: www.ietf.org/rfc/rfc3075.txt.

[14] HL7 Standards, “The Clinical Document Architecture (CDA)”, Internet Site Address: http://www.hl7.org/

[15] Zlib compression library, Internet site address: http://www.zlib.net/

[16] Bzip2 compression library, Internet site address: http://www.bzip.org/
[17] Gzip compression library, Internet site address: http://www.gzip.org/
[18] Compression via Arithmetic Coding, Internet site address: www.colloquial.com/ArithmeticCoding/