Faculty of Computer Science
Bachelor of Science in Applied Computer Science
Free University of Bolzano

Advanced Database techniques in Dpdl

Dynamic Packet Definition Language
A portable data protocol for small Embedded Systems

Author:
Armin Costa
armincosta@dpdl.biz

Academic Year 2003/2004
2^{nd} Graduate Session 29.10.2004

Supervisor Prof. Alessandro Artale

Abstract

Probably, you may not have ever heard of Dpdl (Dynamic Packet Definition Language), so I'll briefly introduce it in a very compact form. Dpdl is a kind of scripting language I've started developing in summer 2003 in order to provide a fast and efficient data transport layer particularly optimized for small Embedded Systems like mobile phones and even 8-bit Microcontroller- Chips. In this thesis project I've integrated advanced database techniques in order to optimize further Dpdl. The system can be used in many application areas to store, control and visualize data. In order to demonstrate one of it's usage scenarios, I've implemented a client/server environment which uses Dpdl to provide data in form of installable services, which may be executed on a variety of devices (Mobile Phones, PDAs, etc.). I had more than once problems in dealing even relatively small data-sets on a mobile-phone application using standard technologies like XML (Extensible Markup Language). This leak of technology lead me to the conclusion to build a complete system optimized for providing that functionality on small devices.

Table of Contents

List of Figures

Chapter 1 – Introduction

This Chapter provides a short overview of the project's domain and tries to give a rough insight about the scope of Dpdl (Dynamic Packet Definition Language), the underlying architecture and standard technologies used in these fields.

1.1 Introduction to Dpdl

Nowadays the use of mobile devices plays an important role in every day's life and tend to be used massively to accomplish different tasks, most of them related to information retrieval or other kind of service transactions and data management facilities. As the technology in this field is relatively young, many functionalities and software components are yet missing, restricting so it's application area. Although the hardware for mobile devices is evolving very fast, these devices tend to present serious performance leaks for some sort of applications due to the limited amount of RAM-memory available, especially regarding memory usage, processing power and network bandwidth capabilities. One of these performance leaks is given for example by handling big data-sets, like for example the decoding of XML-data (Extensible Markup Language) if data is big. This lead to the fact that it's not easy to manage or transmit over a network relatively big data-sets on these devices. The standard technologies used to manage data on devices such as a Mobile Phones for example is XML, a meta-language used to transport and describe data. This data protocol has been proven not to be very efficient on small devices and can therefore be used merely to handle small data sets at time. This lead to the fact that those meta-languages (XML, WAP, cHtml) do not suite well for a “static service” environment, namely for information services that do not change frequently, have big data-sets and could therefore be ideally installed just once, and maybe updated periodically (for example: Phone-books, train schedules, etc…) through a protocol-like interface. In such a static service environment, data could be browsed by a user of the system even if no sort of network connection is given, imagine for example the possibility to browse a phone-book without the need of being connected. Standard protocols like XML for example, on memory scarce devices, have been implemented primarily to be used in a connection-oriented environment, like the internet for example, where possibly small peaces of information are exchanged asynchronously between client and server, and huge XML-sets of information are handled by powerful server-stations and DBMS (Database Management System). To overcome this restriction of XML and data management in general on memory scarce devices, I decided to implement a kind of scripting language, implementing a Database- and a Graphical GUI-engine , which is able to provide some specific functionalities for managing efficiently big data-sets in KB range architectures (140 KB – 3 MB). The purpose of this thesis-project is to implement advanced database techniques and a set of runtime options in the Dpdl core-engine in order to push it's performance. Dpdl runs also on a device such as the one shown in Figure 1.0.

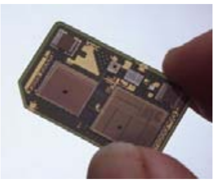


Figure 1.0
32 bit RISC processor running at 30 MHz with 256 Kbyte SRAM, 4 Mbyte Flash memory and 2 Kbyte EEPROM

1.2 Outline of the Thesis

2. Chapter document describes further the existing technologies used to handle data, especially on small scale architectures. It introduces Dpdl (Dynamic Packet Definition Language) and describes some missing aspects of standard technologies like XML, a standard meta-language used to transport data. As an important issue, a study about the chosen platform has been documented.

3. Chapter focuses on Dpdl and describes the concept and usage scenarios in which Dpdl can be employed. A comparison between XML and Dpdl is also provided in order to illustrate common and diverse features.

4. Chapter describes the functional requirements of the system at user level and contains a list of features and API functions included in the Dpdl language semantics.

5. Chapter gives a insight on how the system has been implemented and presents the relationship between various parts of the software along with the corresponding UML diagram.

6. Chapter gives a general overview of database technologies available and describes the data structure used to optimize the Dpdl system. Also a functional description of the swapping technique and compression algorithm implemented is documented.

7. Chapter is devoted to the evaluation of the system and compares execution results between XML and Dpdl on given data-sets. This part also includes a conclusion and judgment of the overall system.

Chapter 2 – State of the Art

This chapter introduces the idea of Dpdl, and gives an overview of existing technologies on Mobile devices, in particular XML. It further describes some aspects of Dpdl and states the missing functionalities and faults of XML on small devices. A further study is documented about the actual platform chosen, in fact a KJVM (Kilobyte Java Virtual Machine), and lists the used API's and software tools in this project.

2.1 The Background and the main ideas

Nowadays mobile devices are massively used in every day's life, just think of the popularity of Mobile Phones, PDA's and the presence of microcontroller-chips to control almost everything. In response to this trend, I thought that an additional API, particularly suited for small devices, could support the development and problem-solving in such an environment, and consequently extend the development possibilities in these areas. That's way I came up with the idea to create Dpdl, a small API and scripting language suited to format, describe and visualize data in a rather efficient way. Think of Dpdl as an XML-like meta-language.

Similar protocols used in the development of Software on small architectures usually implement XML, WML (Wireless Mark-up Language), cHtml (Compact Hyper Text Transfer Protocol), WAP, etc., in order to transport, control and present data. These protocols have the fault that they do not suite very well to handle huge data sets on limited hardware resources, like Mobile Phones or 8- or 16-bit Microcontroller-Chips. Dpdl in contrast is optimized for containing much more data statically in a compressed format. Dpdl can potentially be used also to contain XML, WML, cHtml, etc. and other data-formats in a compact, optimized and structured fashion, which are than decoded at runtime and being projected into a proper browser interface. How this has been achieved is the matter of this dissertation.

Execution inefficiencies of XML in Mobile devices are claimed very often in development forums through the internet, and I personally had such a problem once, which lead me to the conclusion to build the Dpdl system. This does not mean that XML is being considered useless, moreover it can be packed as a special built-in data type in Dpdl, extending so the development possibilities.

2.2 General considerations about XML

XML, as a meta-language, has very powerful construct by design, and is nowadays being used everywhere in computing, just think of Web-services with SOAP (Simple Object Access Protocol), messaging applications like mobile agents, it is used very often as a configuration script for applications and of course also as a pure data transport and presentation layer. The popularity of XML induces a high portability of data. However there are some questions opened, which makes XML not very suited for some sort of applications.

Is XML a Database?

An XML document is a database only in the strictest sense of the term. That is, it is a collection of data. In many ways, this makes it no different from any other file. After all, all files contain data of some sort. As a "database" format, XML has some advantages. For example, it is self-describing (the mark-up describes the structure and type names of the data, although not the semantics), it is portable (Unicode), and it can describe data in tree or graph structures. It also has some disadvantages. For example, it is verbose in terms of memory usage and access to the data is slow due to parsing and text conversion. This verbosity in terms of memory consumption applies specially to small devices.

A more useful question to ask is whether XML and its surrounding technologies constitute a "database" in the looser sense of the term, that is, a database management system (DBMS). The answer to this question is, "Sort of." On the plus side, XML provides many of the things found in databases: storage (XML documents), schemas (DTDs, XML Schemas, RELAX NG, and so on), query languages (XQuery, XPath, XQL, XML-QL, QUILT, etc.), programming interfaces (SAX, DOM, JDOM), and so on. On the minus side, XML lacks many of the things found in real databases: efficient storage, indexes, security, transactions and data integrity, multi-user access, triggers, queries across multiple documents, and so on. A side we consider the fact that these query languages mentioned above do not jet exist for the mobile-device versions of XML.

Thus, while it may be possible to use an XML document or documents as a database in environments with small amounts of data, few users, and modest performance requirements, this will fail in most production environments, which have many users, strict data integrity requirements, and the need for good performance on limited hardware resources.

A good example of the type of "database" for which an XML document is suitable is an .ini file -- that is, a file that contains application configuration information. It is much easier to invent a small XML language and write a SAX application for interpreting that language than it is to write a parser for comma-delimited files. In addition, XML allows you to have nested entries, something that is harder to do in comma-delimited files. However, this is hardly a database, since it is read and written linearly, and then only when the application is started and ended. Of course an .ini configuration file could also be formatted and decoded with Dpdl.

Why Use a Database?

The first question you need to ask yourself when you start thinking about XML and databases is why you want to use a database in the first place. Do you have legacy data you want to expose? Are you looking for a place to store your Web pages? Is the database used by an e-commerce application in which XML is used as a data transport? The answers to these questions will strongly influence your choice of database and middleware (if any), as well as how you use that database.

For example, suppose you have an e-commerce application that uses XML as a data transport. It is a good bet that your data has a highly regular structure and is used by non-XML applications. Furthermore, things like entities and the encodings used by XML documents probably aren't important to you -- after all, you are interested in the data, not how it is stored in an XML document. In this case, you'll probably need a relational database and a software to transfer the data between XML documents and the database. If your applications are object-oriented, you might even want a system that can store those objects in the database or serialize them as XML.

Summarizing we can say:
An XML document is a collection of data, so in the strictest sense it is a database.

It has some advantages as a database: (this also applies in some sort to Dpdl)

• self-describing, portable, can define data in trees or graphs
• flexible schemas (DTD's, XML Schema)
• query languages (XPath, XQuery, …)
• programming interfaces (SAX, DOM, …)

Also disadvantages:

• verbose, access to data is slow (needs parsing)
• lacks indexes, security, transactions, multi-user access,triggers, queries across multiple documents..
- NO query languages for XML on Mobile Devices (XPath, XQuery)

2.3 XML in Mobile Devices J2ME

More and more enterprise and Java technology projects are making use of XML as a medium to store data in a portable fashion. But due to the increased processing power demanded by XML parsers, J2ME (Java 2 Micro Edition) applications have been penalized a bit by this fact. Although there are lightweight XML parsers for J2ME like for example KXML (Kilobyte XML), but these tend to be too slow or inadequate for certain applications. You can use XML parsers in J2ME applications to interface with an existing XML service. For example, you could get a customized view of news on your phone from an aggregator site that summarizes headlines and story descriptions for a news site in XML format. But XML parsers tend to be bulky on mobile devices, with heavy runtime memory requirements on memory scarce devices.

XML Performance issues [4]

1. *Increase in size:* An XML parser is code-intensive and increases the overall size of an application. This is a particularly important consideration for resource-constrained MIDP devices. There are several optimization techniques you can use to fight code expansion. First, you should remove resource files that are not in use. You should also use obfuscators that will remove unused classes, unused methods, and variables from your JAD file.
2. *Heavy string parsing:* XML parsers use intensive string parsing to perform their jobs; this will add to the overhead in MIDP applications with low runtime memory. XML documents that J2ME applications parse need to be small and contain as much useful information as possible.
3. *Slow response time:* As the MIDP application parses a relatively large amount of XML data, the response time will increase. The XML files to be parsed should be small, and the parsing should be done in a thread of execution that is separate from the main application.

Refer to the Appendix D for code examples on how to use XML on J2ME.

2.4 Software and Programming Language used in the project

Development environment

There might be a good question before going on and develop a piece of software: “What platform is the target platform and what development environment should be used?”. This is an important approach before beginning with the actual implementation of particular software.

As already mentioned, the Dpdl-System can ideally be used in Embedded Systems like Mobile Phones, PDA’s, various development Boards (ARM-Boards, etc.), 8-bit Microcontroller-Chips, virtually on every device as long as a C compiler is available for the target platform, because the JVM (Java Virtual Machine) is open source and can be recompiled for every device.

This wide range of platforms on which Dpdl is thought to run, answers clearly the question about what programming language is most suited. A study I carried out shows clearly the big advantage in using Java™-Technology as development platform (KJVM – Kilobyte Java Virtual Machine).

These are main reasons for this choice:

1. Java is supported in a wide range of popular Mobile Devices by Default (Nokia, Ericcsson, Siemens, etc.)
2. Java runs also on Compact .NET
3. KVM is open-source (ANSI-C source-code) and can be recompiled for every platform as long as a C-compiler is available for the target platform
4. Java is Object Oriented, which makes implementation easier
5. Java has a garbage-collector built-in in the KVM. This provides automatic memory management in memory scare devices and prevent runtime memory leaks.
6. Supports also Kni (kilobyte native Interface kni.h). This implies that java code can integrate native C++ code on the Symbian platform (Epoc32 for example).
7. A ROMizing technique can be used to improve Speed and Memory-usage, and also decrease by ca. 60% the application’s Memory-Footprint.
8. Java is very similar to C#, so a porting can be done in minimal time
9. There are lots of Microcontroller-Chips which executes Java natively. Thus, this makes possible to achieve very low execution times, almost as written in C.

Java needs a VM (Virtual Machine) in order to execute a program, as long as Java is not implemented in Hardware (special Microcontroller-Chips), like on small Microcontroller chips. The KVM is a Virtual Machine optimized for running on at least 150 kilobyte Heap and a processor speed of 8-32 Mhz.

Further KVM considerations:

1. written in ANSI C (25,000 lines of well commented Code)
2. VM core size 60 – 72 Kb (on Win32 72 Kb)
3. Source-Code available under SCSL (Sun Community Source License)
4. Modular design with compile-time options to optimize Speed vs. Size

5. Scalable and modular: CLDC (Connected Limited Device Configuration), CDC (Connected Device Configuration), MIDP → Very portable to new consumer devices (even 8-bit micro-controllers)
6. Bytecode-Interpreter can be optimized due to it's simplicity (use 2 interpreter-loops; move VM registers to hardware registers…etc.)
7. Garbage-Collector to ensure a proper memory layout without pointer leaks.

For the good reasons mentioned above, Dpdl has been implemented fully in standard Java™, more specifically in the "Java 2 Micro Edition" (J2ME) of the Java™ platform, fully compliant with MIDP1.0 and MIDP2.0 (CDC && CLDC).

See Appendix D for a complete list of all java enabled phones

Java™ Architecture [1]

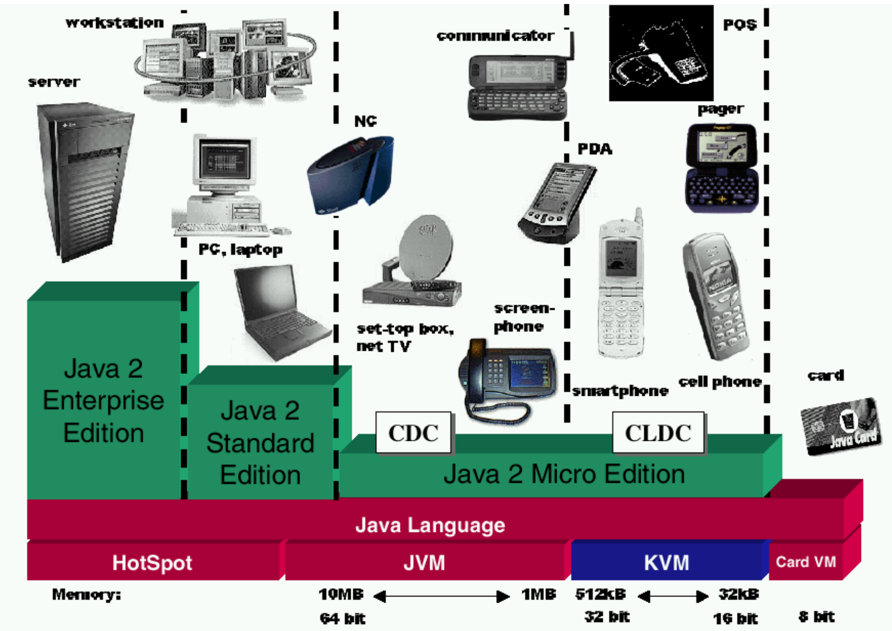


Figure 2.1 Java Architecture

API's and Tools used in the project

In order to develop and test the Dpdl system, different kind of tools have been used:

1. J2SE V 1.4.1 standard Java™ compiler and JRE
2. J2ME V 1.0 / V 2.0 the Java™ Mobile Edition development environment
3. MySQL DBMS a DBMS
4. NaviCat a Tool to manage MySQL DBMS
5. xmlenc V. 0.44 a tool to query databases and create XML files
6. KXML a lightweight java API for XML

Chapter 3 – Concept behind Dpdl

This chapter introduces Dpdl more accurately and explains how this little framework can be used in application development to manage data and which techniques have been employed to achieve high performance.

3.1 Description of the Dpdl system

Dpdl is like a meta-language similar to XML, but with a rather different approach. It can be potentially used in many application areas to store and describe data, or to contain other protocols in a compact fashion. The idea behind this scripting language is to integrate huge amounts of arbitrary, structured data (could also be data in other formats like XML, cHtml, WAP, etc..) in a well-defined, highly compressed data packet, which can be decoded by the underlying Dpdl-architecture. The system has been implemented in form of two components, a Dpdl-encoder and a Dpdl-decoder. The Dpdl-encoder is used to pack and format the data according to a defined Dpdl-script and has also the ability to query a Database with SQL-queries which are defined inside the Dpdl-script. The Dpdl-decoder is able to restore and query the data contained in the Dpdl-packet composed by the Dpdl-encoder and occasionally provide a graphical user interface (GUI) for a particular service implementation. Both, the Dpdl-encoder and the Dpdl-decoder usually use the same Dpdl-script, but this is implementation dependant.

Dpdl has been implemented considering a very special application domain, and therefore a very specific combination of optimization techniques have been employed to achieve the desired functionalities. Due to it's very compact data-format, Dpdl can be ideally involved in small microchip-architectures or in Wi-Fi networks, where highly compressed data and speed is an optimal development issue, especially for small devices like Mobile-Phones, Microcontroller-Chips or small Network Modules. The system implements compression algorithms, swapping techniques, indexing algorithms and further optimizations and security issues in order to be able to achieve the desired performance goal. The Dpdl scripting language has a parameterized data-compression engine build-in, which can be controlled by the software programmer through syntactical definitions inside a Dpdl-script. This means that data can be compressed and organized in defined data-chunks, which can be decoded selectively by the Dpdl-decoder. Specific database structures and swapping techniques have been employed to guarantee considerably low query-, execution- and memory-rates on small-scale architectures.

Dpdl, as a kind of scripting environment, allows a fast implementation of different kind of data management facilities and services, which can be executed nearly on every device, also 8-bit microcontrollers, because standard Java™ has been used for its implementation (cell phones, PDA's, microcontroller, etc.). Another reason for this high portability is the availability of software tools to translate simple Java into C, which improves execution by a factor of 20-40 and implies all-over compatibility.

The core-engine of Dpdl is on the decoder-side 21 KB, and even less on the encoder-side (16 KB). The extensions made in this version of Dpdl, which include automatic GUI-engine (Graphical User Interface), are in both cases 33 KB. Due to the very small size of the two Dpdl components (Dpdl-encoder and Dpdl-decoder), both can potentially be used in a mobile phone or a 16-bit Microcontroller-Chip. This model can be used in client/server applications or to exchange information asynchronously in controlling and automatition applications.

As already mentioned, the concept behind Dpdl, is that the same script defined for a given service is used by the Dpdl-encoder to retrieve and encode the information, and used by the Dpdl-decoder to restore the encoded data. In this way, different functionalities can be performed either on the server- , or on the client-side, by using the power of a DBMS (Database Management System) on one side, and an efficient inner “Dpdl-service” database with an automatic GUI-generation built-in, to complement the service framework on the other side. This can be a very powerful combination because it enables a dynamic definition and controlling of data. Additionally, Dpdl can potentially be used also to contain built-in XML (Extensible Markup Language), WML (Wireless Markup Language), etc. data-formats, as a specific service-decoding option is given at Dpdl language level. This feature expands further the usage possibilities, because data in XML format is very common, and Dpdl provides the functionality to organize this data. Dpdl also supports cHtml (Compact Hyper Text Transfer Protocol) and is thought to provide interoperability with popular image vector formats such as the Shape-data-format (Maps). In this way Dpdl can also be used to solve different data visualization issues.

Another idea behind Dpdl is to provide well structured information in form of nested Menu-selections, namely, Dpdl-scripts can contain other Dpdl-scripts with GUI-definitions or XML-coded data. In this way, a complete constraint-based Menu-engine is built-in by default.

3.2 Dpdl usage scenarios

Although Dpdl can be used in very different application environments to handle data, also as kind of protocol to configure or control Microcontrollers, however, the most typical case is a client/server application scenario. In such a situation, Dpdl acts on both sides, the client and the server. The two components, the Dpdl-encoder and Dpdl-decoder, are used one on the server-side and the other on the client-side respectively. In this case, the server uses the Dpdl-encoder to retrieve and encode the data according to a present query definition inside the Dpdl-script, and the client use the Dpdl-decoder to decode the Dpdl-packet downloaded from the server in some protocol specific way. The protocols used to perform the data retrieval by the client are left over to the software developer and the Dpdl-Engine provides a basic abstraction layer to simplify these operations (http, Gprs, Wi-Fi, Bluetooth, Irda, etc.). See Figure 3.1 for a visual overview of the environment.

There may be other cases, in which the roles mentioned above are inverted, or cases in which client and server integrate both components (Dpdl-encoder and Dpdl-decoder) in order to be able to exchange information asynchronously, like in controlling applications. Just to mention one more usage possibilities, I’ve used Dpdl also as transport to contain 3D-structures in a 3D-engine application on a PDA. However, here we’ll focus on a client/server environment, because it may be a common implementation. This client/server scenario applies also to the DEMO-Application I’ve implemented in order to demonstrate the application possibilities of Dpld. The DEMO-Application will be presented later on.

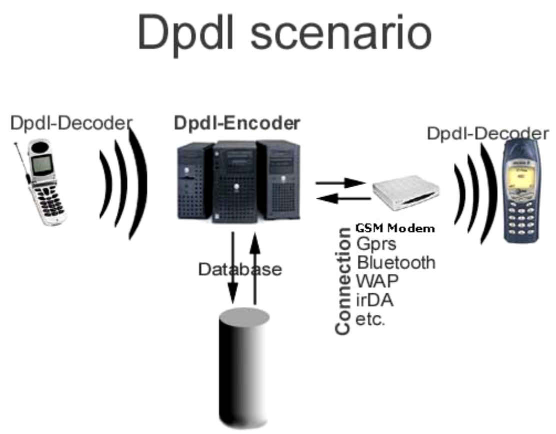


Figure 3.1 Application scenario using Dpdl

See Chapter 5 for details about the overall system architecture

3.3 Dpdl vs. XML

Although XML provides a powerful application interface, it misses some important concepts and ideas. Let's compare Dpdl and XML in order to show how the concept differs from standard protocols like XML.

Functionality	Dpdl	XML
Optimized to handle huge data-sets on small devices (mobile phones, PDA's, microcontroller, etc.)	X	
Primarily thought to be used in a connection-oriented environment (Gprs, internet, etc.)	X	X
Particularly Suited for a service environment where services remain static for a certain time period (ex. a phone-book)	X	
Huge data compression	X	
Given the big data-compression, it's ideal to be used in Wi-Fi networks, especially for mobile devices.	X	
Advanced database techniques (B-Tree, etc.) to improve effiency for a particular Dpdl service → in XML see XQuery below	X	
Automatic GUI generation defined by the script.	X	
Used as data-container	X	X
A diffused standard (configurations, Web-services, etc.)		X
Interface with geographical GIS maps (shape *.shp file format)	X	
Used to format a variety of documents		X

Dpdl can be used to contain XML, WBXML, cHTML, WAP, etc., in logical structures, extending so the application possibilities	X	
Same script is used on the server-side to encode the Service, and on the client-side to decode the service.	X	
Embeddable SQL-queries used on the server-side service generation.	X	
Data visualization on a client-device can be extended with XSL, CSS, cHTML, etc.	X	X
Services can be scaled for different devices → a nested service for example may be downloaded at usage time, or be splitted in multiple sections	X	
Embedded queries (XPath, XQuery) → in Dpd this option is not necessary, because Dpdl services are generated on server-side according to the current query definition in the actual Dpdl-script, and are optimized according to that.		X

3.4 The layout of a Dpdl packet

Here an example of how a Dpdl-packet is organized. The main parts are the header, the layout, and the data Chunks. The Header contains all information needed to restore the data, like security and integrity flags, various other flags and the Dpdl-script belonging to that Dpdl-Packet. The layout contains possible visualization layers like cHtml, WAP, CSS, etc. that are optionally used to interact graphically with services or just to display results. The data Chunks contain the coded Dpdl data, and if necessary a section reserved for B-Tree indexing information inside the chunk itself. Remember that the data itself is organized in CHUNKS, which reflects the definition of the data inside the Dpdl-script. A Chunk of data may also consist of 1 or n Dpdl-scripts, which provides a powerful recursion of data.

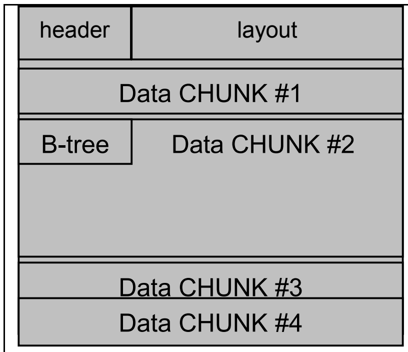


Figure 3.2 Layout of a Dpdl-Packet

Chapter 4 – Functional requirements of the System

In this chapter the functional requirements of the system will be presented in order to illustrate how the Dpdl system works at user level, and what are the main functionalities it should provide. The study and planning considers a software programmer in first order as primary user target.

4.1 The target user

As the Dpdl system consists of a set of software components which may probably be integrated in other software products, we consider in first place a software programmer as primary user. In order to be able to use the functionalities offered by Dpdl, the programmer will be provided with a language specification describing all features of the Dpdl scripting-language and an additional API documentation describing some specific functionalities of the Dpdl-engine itself. A trained programmer shouldn't take more than a couple of days to effectively be able to use Dpdl in all its features. Of course there may be also other possible users of the system, like for example System Administrators, Electrical Engineers, etc., but for now, as we first need to focus on the technical details of the language, we'll consider the software programmer in order to establish the core functional requirements of the system.

4.2 General requirements of the system

In order to ensure a greater usage and scalability of the system, there are a couple of baseline requirements that should be provided by the system.

Such baseline-requirements include:

1. small memory footprint of the components (Dpdl-encoder and Dpdl-decoder) and low memory usage
2. high portability
3. easily enable future extensions
4. plug-in architecture for algorithms and additional functionalities

1) small memory footprint and low memory usage

As mentioned, Dpdl is thought to be used primarily in small scale architectures with a memory heap of at least 120 Kb. In order to enable functionality in such environments, the design of the code has been kept as small as possible. To guarantee a further downsize of the application's code, an obfuscation technique has been employed. A further possibility given to keep the components even smaller, is to apply a technique called ROMizing, applicable to all Java applications running on a KJVM (Kilobyte Java Virtual Machine). In this case, the obfuscated Java code is traduced into native C by a translation software and than compiled directly into the KJVM). This technique guarantees also an increase in execution-speed by ca. 30-40%.

2) High portability

As the variety of small devices is very vast, just think of all different models of mobile phones for example, there was the need to use Java as development platform for the implementation of Dpdl. This ensures that Dpdl runs on a wide range of small architectures without the need of recompiling the code. For industry and controlling applications which may use Dpdl to transport or control data, there are also small micro-chips available which implement Java directly in hardware registers, increasing so execution speed drastically.

3) Extensible architecture

The Dpdl system has been designed considering the fact that it may be extended in various parts. This was a clear decision I made when developing the core Dpdl-engine. In this way, future functionalities can be easily integrated into the Dpdl system.

4) Plug-in architecture for algorithms

As we'll see, Dpdl implements well defined algorithms and data structures in order to efficiently handle data. The choice about what algorithms to use, for example for compression, suggested to design the system in a way where this algorithm could be easily replaced in future. This is an important point to consider, because in different situations, under certain circumstances, there may a raise the need to use another algorithm. We also do not know yet if the future reserves better, more efficient algorithms, therefore this was a crucial requirement.

4.3 Functional requirements

The functional requirements of the system at language level has been derived considering a few possible application areas in which both Dpdl components could be used. These considerations implied all basic functionalities needed to process and transport data. Additionally, also other functionalities useful to carry out security and visualization issues have been included. Dpdl is also thought to provide interfaces with popular GIS (Graphical Information System) data formats such as the Shape format.

Here are the most important functional requirements the system has to provide:

1. Define data entries containing Chunks of Data
2. Define and associate Data-sources and –queries
3. Define standard data types for Chunk-members
4. Define database index structures (B-tree)
5. Possibility to define GUI's (Graphical User Interface)
6. Define various options and tags

1) Define data entries containing Chunks of Data

The definition “Chunk of data” refers to a Data-set included inside the Dpdl packet which contain homogeneous data. A Dpdl packet may contain 1 to n data entries, which can contain 1 to n chunks of data, where a given chunk may also contain another Dpdl-packet (recursive feature). The user of the system can declare these data entries and chunks inside the Dpdl-script.

This definition for example defines a data entry named BolzanoPhone. This data entry contains one chunk with 3 fields as content (name, phoneNR, e-mail)

```
#defineD BolzanoPhone {
            CHUNK entry;
            entry.content { name , phoneNR , e-mail }
}
```

2) Define and associate Data-sources and -queries

As the same Dpdl-script is used twice, for encoding the data and for decoding the data, there was the need to provide a definition for specifying where the Dpdl-encoder should get the needed data being encoded. The programmer may declare Database-connections and sql-queries inside the Dpdl-script and than associate them to the appropriate data entry.

Here an example of how to declare database-connections and sql-queries:

```
#defineDB phone_bz | 129.124.89.2 | root root
#defineSQL query_ SELECT name, phoneNR, e-mail FROM PHONE_BZ
```

These definitions can than be associated to the corresponding data-entry in the following way by adding src and SQL conditions:

```
#defineD BolzanoPhone src phone_bz SQL query_ {
            CHUNK entry;
            entry.content { name , phoneNR , e-mail }
}
```

3) Define standard data types for Chunk-members

Dpdl allows all standard data types to be declared for chunk member data. Such types include **int, double, float, string, char, byte, Image, Vector, class, int[], double[], string[], byte[], Dpdl**, etc. The types can be extended easily in future, or even customized inside the Dpdl-script.

Here an example how to map the member data types for the previous defined data entry (the TAG(0x..) definition is just an optional field to enable faster searching):

```
#defineD BolzanoPhone src phone_bz SQL query_ {
            CHUNK entry;
            entry.content { name , phoneNR , e-mail }
            entry.name TAG(0xef) (string) = 20;
            entry.phoneNR TAG(0xefe) (string) = 15;
            entry.e-mail TAG(0xefee) (string) = 30;
}
```

4) Define database index structures (B-tree)

When data sets become big, there may be the need to define an efficient version of a given data-entry in order to be able to query the data in a fast way at decoding time. This is done by declaring a B-Tree in a given chunk. If the data needs to be queried, there is also the possibility to associate a constraint to a member variable of a chunk.

This example illustrates how to associate a constraint and how a defined data entry can be optimized by implementing a B-tree structure:

```
#defineD BolzanoPhone src phone_bz SQL query_ {
            CHUNK entry;
             struct BTree DENSE_INDEX *name
            entry.content { name , phoneNR , e-mail }
            entry.name TAG(0xef) (string) = 20;
             …
}
```

5) Possibility to define GUI’s (Graphical User Interface)

Another important functional requirement is to enhance the Dpdl scripting language with GUI features. In this way, a given Dpdl-packet can be executed like a standalone application, not requiring any other software interaction. Dpdl also supports other protocols like cHtml, WAP, etc., which can be integrated to show and format results in a user friendly fashion. An example to this would be a service like a phonebook on a mobile phone, where the complete user interface used to search the phonebook is generated by Dpdl. The results will be formatted according to a cHtml script defined or imported into the Dpdl-script.

This is how we could define a GUI-functionality for the data-entry presented above, where a cHtml script is defined and used to visualize the results queried by the user of ther service:

```
#defineProtocol-cHtml phonebook_visual {
                <html>
                <head>
                <meta http-equiv="Content-Type"
                    content="text/html; charset=iso-8859-1">
                  <title>Bolzano PhoneBook</title>
                  </head>
                  <body>
                  <h2><font size="2">Results:</font></h2>
                  <hr><font size="2"><b>Name: </b>&nbsp; $name</font>
                  <p><fontsize="2"><b>Tel:</b>&nbsp;$phoneNR</font></p>
                  <font size="2"><b>e-mail: </b>&nbsp; $e-mail</font>
                  <hr>
                  </body>
                  </html>
}
import virtual DATA none  {
      class BolzanoPhone volatile phonebook_visual {
          DATA::string const name;
          DATA::string using phoneNR;
          DATA::string using  e-mail;
          #defineGUI Default <Brunico_PhoneBook>  <please_enter_name_and_surname_here:>
      }
}
```

6) Define various options and tags

Another important functionality is to provide a way to control security options, compression options, Dpdl and KVM versions, Dpdl-script versions, Dpdl-engine models, and other features which may be integrated in future versions of Dpdl.

```
call(dpdlInterpreter)
::module dpdl_PHONEBOOK
::module_SPEC 23452
::model 836
::dpdlVersion 1.0

OPTIONS {
  TARGET = MOBILE_PHONE & PDA
  KVM = 1.0
  ZIP = true
  CHECKSUM = true
  SIGNATURE = true
  ENCRYPTION(RSA) = true
}
```

4.3.1 Use Case Diagram

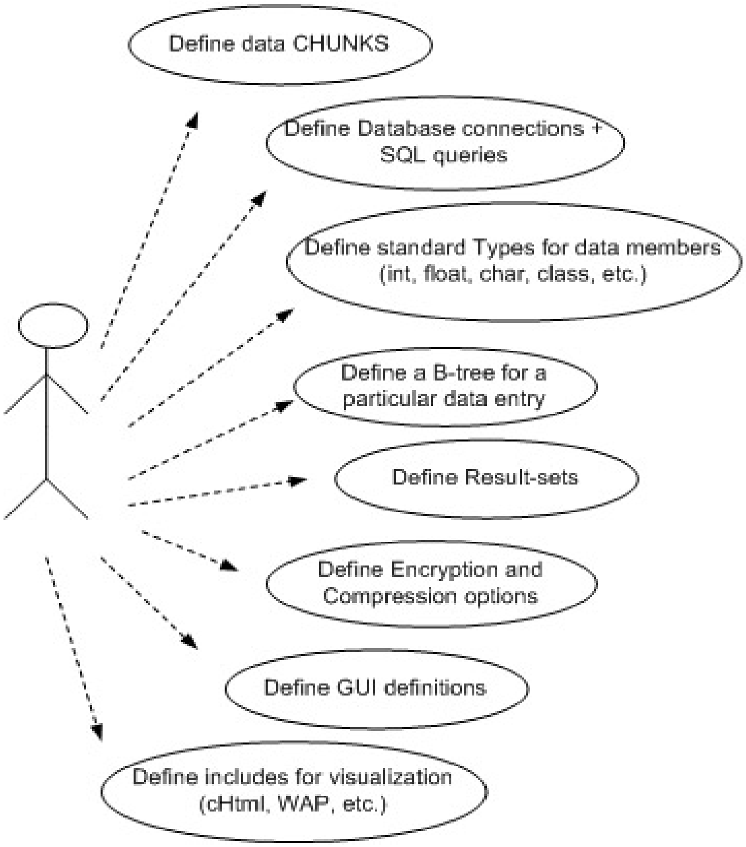

Define data CHUNKS
Define Database connections + SQL queries
Define standard Types for data members (int, float, char, class, etc.)
Define a B-tree for a particular data entry
Define Result-sets
Define Encryption and Compression options
Define GUI definitions
Define includes for visualization (cHtml, WAP, etc.)

Figure 4.1 Use case diagrams

Refer to Appendix A to see a complete Dpdl-script for a phonebook service

Chapter 5 – Design of the System

5.1 General overview of the Dpdl-system

The Dpdl system has been implemented in form of two components, namely the Dpdl-decoder and the Dpdl-encoder. As already mentioned, a very common application example using the Dpdl scripting language is a client/server application. However, how the two components interact in this case, is just one of the possible application scenarios in which Dpdl can be used. That is because Dpdl can be included in any other software products in form of a library. In this case, the server (Dpdl-encoder) generates Dpdl-data for a client application such as a Mobile Phone for example (Dpdl-decoder). The data contained may be for example phone-book entries, train and bus schedules, etc. Both components receive as input instructions a Dpdl-script, which is then parsed and processed in order to encode (server application) or restore (client application) some kind of data.

Whenever the server-side application receives a request from a client application, the Dpdl-script belonging to that particular service is parsed by the Dpdl-encoder and according to the definitions contained in the Dpdl-script, it retrieves the data form the specified sources and encodes a Dpdl-packet. The Dpdl-packet contains both, the data and the Dpdl-script. The composed Dpdl-packet is then transferred in some protocol independent way (Gprs, Bluetooth, Wi-Fi, etc.) to the client application (Mobile Phone, PDA, etc.), which stores the packet in permanent storage. Whenever the client-user accesses the service, the Dpdl-decoder parses the Dpdl-script contained in the Dpdl-packet and allocates the decoder and GUI-engine accordingly. At this point the Dpdl-service is ready to be browsed by the user.

The DEMO application build as part of the project is thought to run in such a client/server environment. The Data-Flow shown in Figure 5, shows how the user interacts with the system in order to install various services implemented with Dpdl. The user sends a usage request to the system. The system answers with a list of services accessible (Train- and Bus-schedules, Phone-book, etc.). The user selects the service and subsequently the Server query and generates, or loads from the internal cache a highly compressed Dpdl-Packet containing 1000 of data entries. The DEMO client-application can anytime run implement a searchable phone-book agenda on a Mobile-Phone containing 10.000 entries in less than 290 KB.

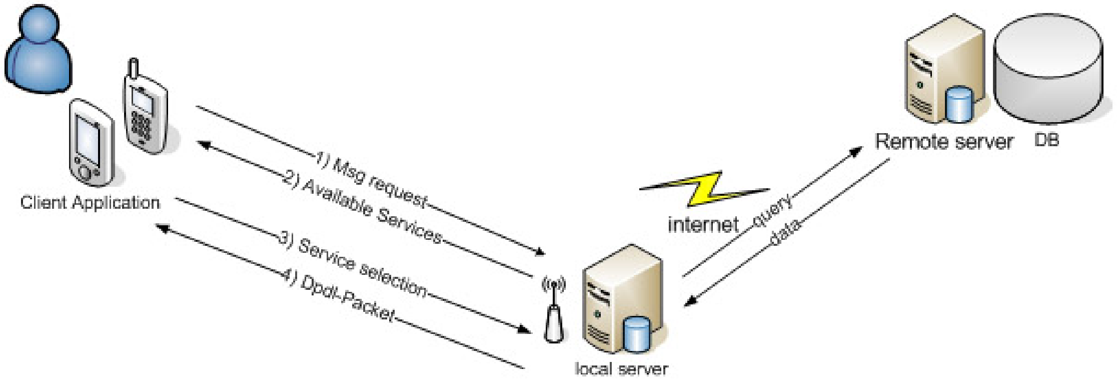


Figure 5 client/server scenario in the DEMO application

5.2 Dpdl-Decoder (client-side application)

The Dpdl-Decoder is part of the client-application. The Dpdl-Decoder receives as input a Dpdl-Packet stored in the permanent storage of the device. When the service is being accessed, it parses the contained Dpdl-script, and allocates the Decoder and DB-Engine. Now the system is ready to retrieve and visualize the requested data. The runtime management ensures that memory and execution requirements are held as low as possible and fit in the device specifiy memory-range. If a Dpdl-script is accessed twice, the decoder won't be reallocated, in this way execution time is reduced further.

The Dpdl-Decoder can be used in two way:

1. As standalone application not requireing other additional software interaction. In this case Dpdl provides a default way of browsing the service.
2. As part of more complex applications. In this case the Dpdl-Decoder is controlled by the overall application throught the Dpdl-API

Description of the System:

As represented in Figure 5.1 The Dpdl-Decoder gets as input a Dpdl-Packet. The Dpdl-Script inside the data packet is parsed by the Dpdl language parser. According to the definitions inside the script, the Dpdl system allocates all necessary system-environment and resources in order to be executed in a fast and memory contained way. The system is provided with an internal runtime- and cache-management facility in order to impove processing speed. If the same Dpdl-Packet is accessed twice, it may nor be reallocated in subsequent accesses.

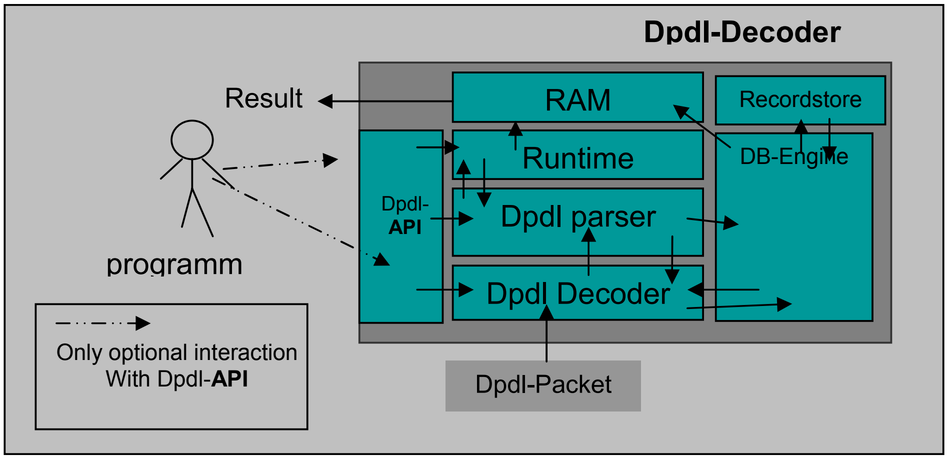


Figure 5.1 System architecture of the Dpdl-Decoder

5.3 Dpdl-Encoder (server-side application)

The Dpdl-encoder, represented in Figure 5.2, in this scenario, is installed on the server-side component. As part of the Dpdl-engine, the programmer can optionally control the operations through an API (Application Programmer Interface) inside other software. The Dpdl-encoder receives as input a Dpdl-script and allocates the encoder and DB-engine according to the definitions inside the Dpdl-script. At Dpdl-packet creation time, Dpdl retrieves the data from some DBMS (Database Management System) and encodes the service-data requested in form of a Dpdl-packet. The Dpdl-Packet is then ready to be downloaded by a client-application (Mobile Phone, PDA's, etc.).

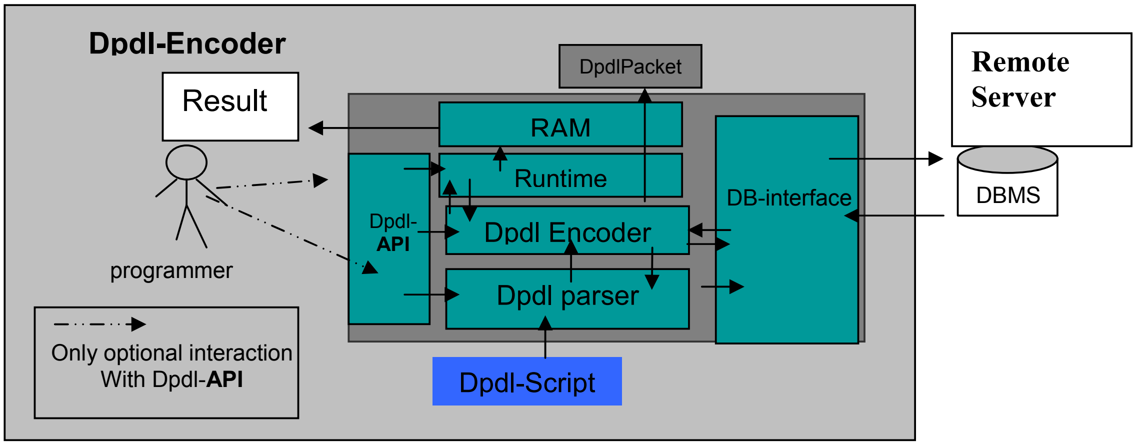


Figure 5.2 System architecture of the Dpdl-Encoder

5.4 UML Diagram

I started developing Dpdl in the summer 2003, and than in January 2003, for the thesis project, I made a refactoring of the core-component in order in integrate database algorithms for faster searching. Doe to the big size of code, It would be practically impossible trying to visualize the whole UML diagram on this paper, so first I'll provide a more general representation of the system.

The system consists of of major parts:

1. Dpdl-Engine (with an **API** Interface)
2. Dpdl-Parser
3. Dpdl-Decoder (decodes the data) or Dpdl-Encoder (encodes the data)
4. A module implementing B-Tree algorithms
5. Dpdl Dpdl-Visualizer and Dpdl GUI-Engine. This component is not included in the core Dpdl version, because certain systems may not need data visualization features, for example small Network Modules

The Dpdl-Engine component is the main controller of the whole system because it integrates the **API** function calls which enable to control more accurately Dpdl from other applications. Remember that Dpdl can be executed automatically or with parameterized functions. Please refer to Figure 5.5 for the very basic functionalities.

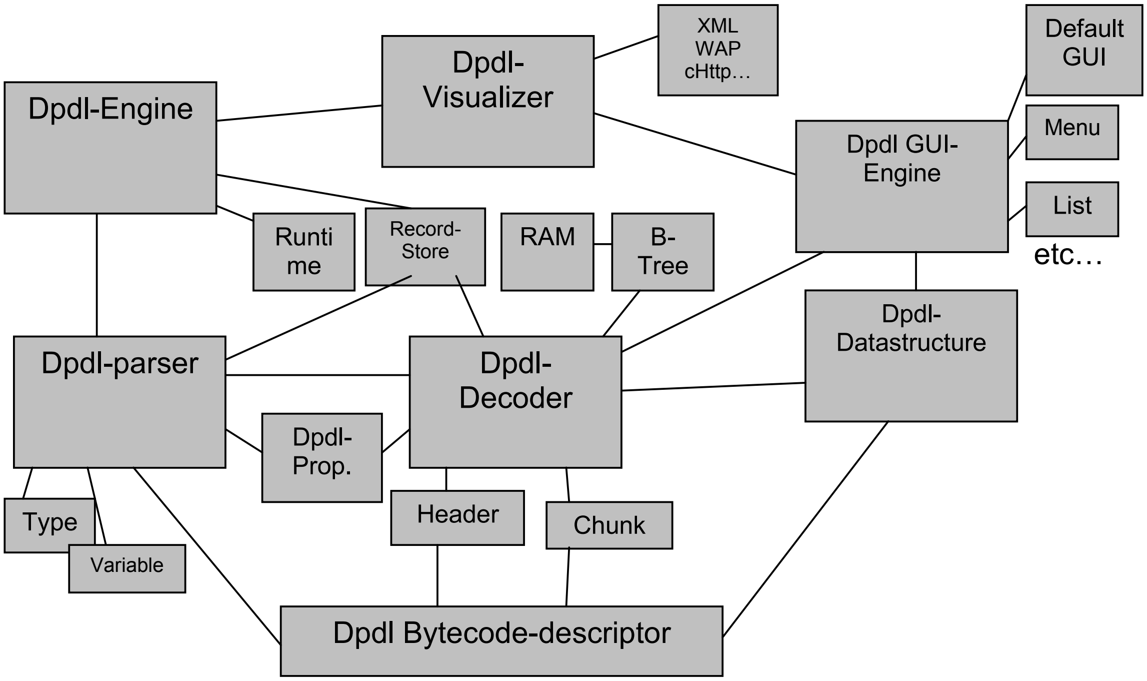


Figure 5.3 UML-Diagram of the Dpdl-Decoder

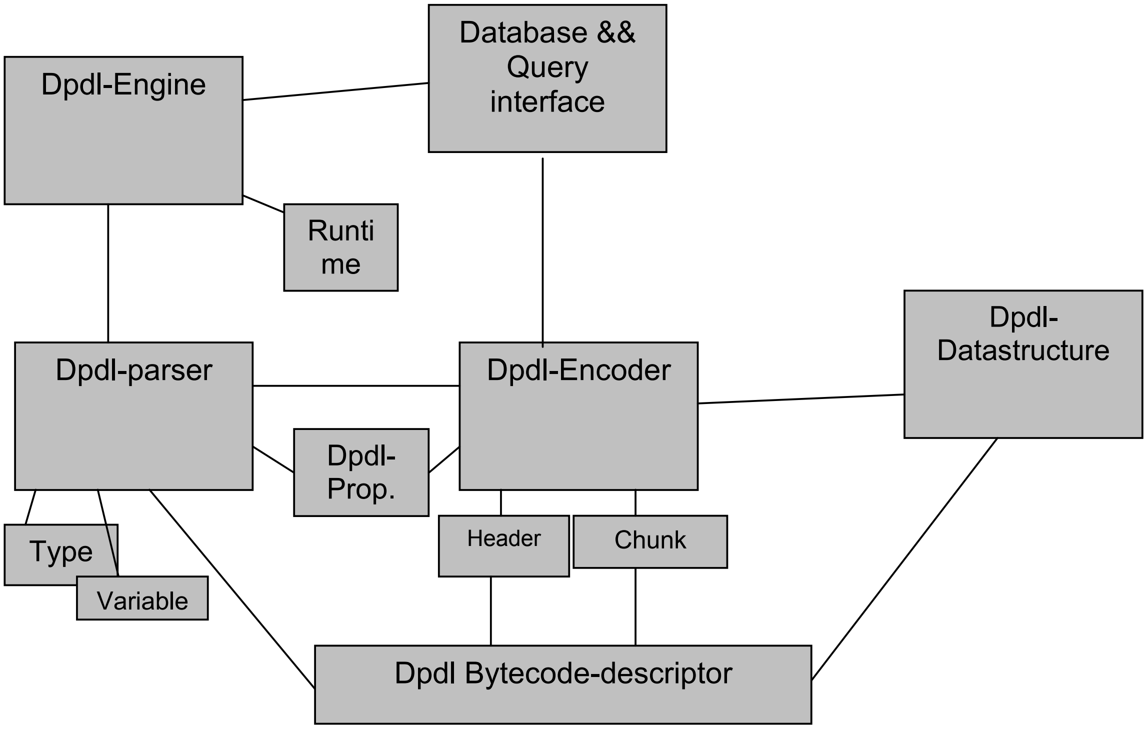
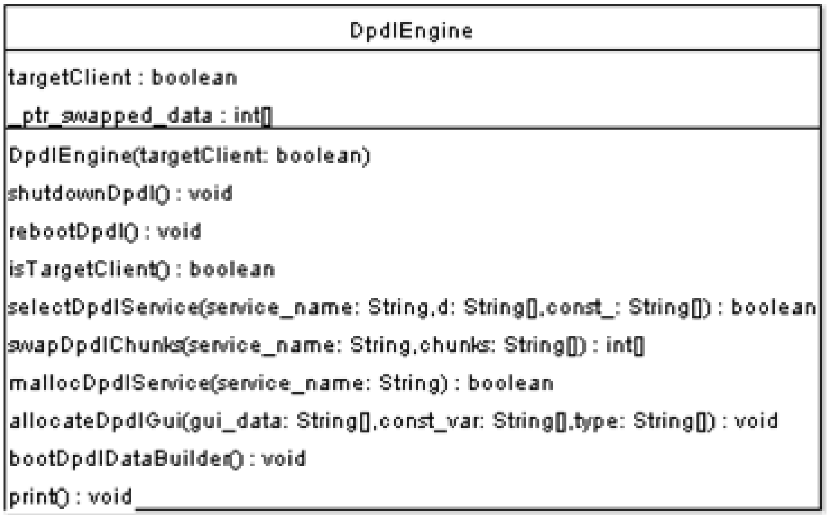


Figure 5.4 UML-Diagram of the Dpdl-Encoder

This system has become a very complex piece of software with over 25.000 lines of code. It was quite impossible trying to visualize on this paper the complete class-diagram containing all components. Therefore the system has been generalized substantially, including just the most important class, namely the API-Interface of Dpdl.

Class-Diagram for the Main class

This is the class represents the basic functionalities of the Dpdl-API interface.



```
DpdlEngine
targetClient : boolean
_ptr_swapped_data : int[]
DpdlEngine(targetClient: boolean)
shutdownDpdl() : void
rebootDpdl() : void
isTargetClient() : boolean
selectDpdlService(service_name: String,d: String[],const_: String[]) : boolean
swapDpdlChunks(service_name: String,chunks: String[]) : int[]
mallocDpdlService(service_name: String) : boolean
allocateDpdlGui(gui_data: String[],const_var: String[],type: String[]) : void
bootDpdlDataBuilder() : void
print() : void
```

Figure 5.5 Class-Diagram of Dpdl-Engine API-Interface

Chapter 6 – Implementation of the System

This chapter focuses on the implementation of the system and describes the concepts for the algorithms and techniques used to optimize Dpdl. In particular it describes the data structures used and how swapping techniques have been employed to guarantee functionality even in very scarce memory heaps and stacks. In order to guarantee a more effective data-compression, a LZ77 compression algorithm has been translated from C into standard java, and included in the Dpdl-System.

6.1 General overview of database-technology

In order to improve the underlying data management system in Dpdl, a combination of Database technology have been used to accomplish the implemented optimization. This choice, as we'll see, is related to the concept and domain of Dpdl.

There are different ways to browse and lookup data, depending on how big the data-sets are, and depending on what hardware facilities are being used:

1. Multi-Level Indexes
2. B-trees

1) Multi-Level Indexes

If a given data index fits into main memory, you need only one disk access to retrieve a tuple with a specific value. If the index does not fit into main memory and you intend to use binary search, you first have to read the middle block, then read the middle of the next half etc. A multilevel index is the optimal choice in such a situation, especially for memory-scarce devices such as Mobile-Phones, where a memory limit is reached very soon. In principle, you have a sparse index for the index file. The most common type of multi-level index is a B-tree. The in-memory versions of hash tables must be modified for use with a secondary storage:

2) B-trees

A B-tree is a data structure that:
• automatically maintains as many levels of index as is appropriate (normally three levels)
• manages the space on the blocks so that every block is between half used and completely full. No overflow blocks are needed
• automatically balances the tree A B-tree is like a balanced binary search tree
• you look for key values by traversing the tree, to the left if the key is less than the node value But also unlike a binary tree:
• packs *n* key values and *n*+1 pointers in each node
• typically, a node is a disk block with space for many (hundreds) key/pointer pairs

Difference between B-tree and B+-tree

Here we'll briefly describe the differences between B-trees and B+-trees in order to understand why a given solution has been chosen for this particular project considering the whole concept behind Dpdl.

B+-tree [5]

In B+ trees, the leaf nodes are linked by a chain, which connects them in the order imposed by the key, as illustrated in Figure 6.1. This chain allows the efficient execution even of queries with a selection predicate that is satisfied by an interval of values. In this case, it is sufficient to access the first value of the interval (using a normal search), then scan sequentially the leaf nodes of the tree up to a key value greater than the second value of the interval. In the key-sequenced case, the response will consist of all the tuples found by this type of search, while in the indirect case it will be necessary to access all the tuples using the pointers thus selected. In particular, this data structure also makes possible and ordered scan, based on the key values, of the entire file, which is quite efficient.

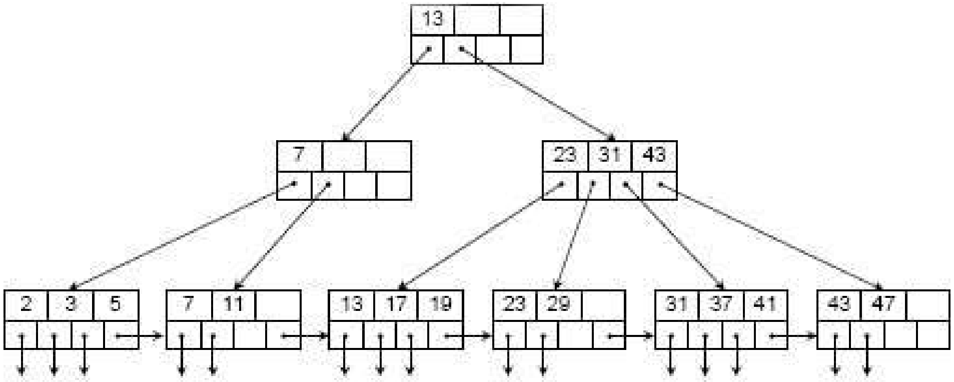


Figure 6.1 B+-Tree structure

B-tree [5]

In B trees, there is no provision for the sequential connection of leaf nodes. In this case, intermediate nodes use two pointers for each key value. One of the two pointers is used to point directly to the block that contains the tuple corresponding to key Ki, interrupting the search. The other pointer is used to continue the search in the sub-tree that includes the key values grater than Ki and less than Ki+1, as shown in figure 6.2. The first pointer Po highlights the sub-tree corresponding to key values less than K1, while the last pointer PF highlights the sub-tree corresponding to key values greater than KF. This technique saves space in the pages of the index and at the same time allows the termination of the search when a given key value is found on intermediate nodes, without having to go through each level.

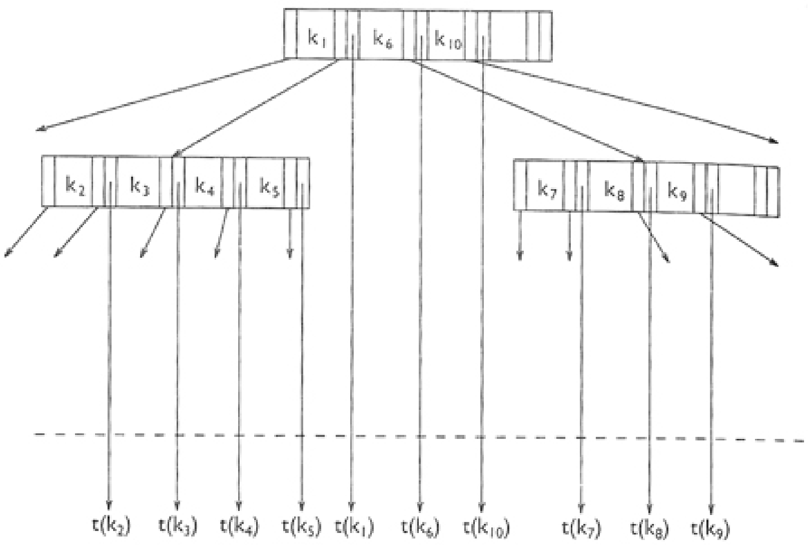


Figure 6.2 [5] B-Tree structure

6.2 The concept of Dpdl and the implemented B-tree algorithm

Although there might be a further improvement in Dpdl in near future, but for now, I intended Dpdl to be used in an environment, where powerful server-stations generate data for small, memory scarce devices. That is, when data is updated, the whole data part is downloaded again and encoded by the server-component of Dpdl. That's also why Dpdl is optimally used in an environment, where huge data-sets remain unchanged for a given period of time, and can therefore be used also if no sort of Connection is available on the target device, for example Bus- and Train-Schedules, Phone Books, City-Maps, etc.

Due to the mentioned gap of memory availability between small Embedded Systems and powerful server- and PC-stations, Dpdl is thought primarily to browse information, while the data-update and insertion is performed by a DBMS's on a server-stations. Therefore, insertion and deletion has been left over for now. But this additional feature can be easily integrated in Dpdl, probably in near future.

The fact mentioned above affect in a well determined way how the B-tree algorithm has been implemented in this case.

The chosen B-tree algorithm

Again, as Dpdl is targeted primarily to small Embedded Systems, the implementation of the B-tree has been approached considering these storage diversities. Another point that lead to this solution, is that a given server generate a defined Dpdl-Packet using a DBMS. According to the query definition defined in the Dpdl-script, the data is than retrieved and encoded. The Dpdl implementation relies and actually implements therefore the condition, that the data is encoded in a already sorted order in the Dpdl-Packet. Because data is encoded in sorted order by the Dpdl-Encoder on the server-side component, there is little reason to implement a B+-tree, because the same effect is achieved by using a normal B-tree. A normal B-tree has in this case some advantages, namely, it saves space for node-data, reduces the number of nodes needed to store a given data-set and makes queries faster because in many cases just one additional intermediate node has to be loaded in memory. However the system also includes a cache of Nodes, so queries get faster if executed frequently. In this case, even with a normal B-tree, range queries can be carried out in a normal way by finding the first element of the range and then read the data until the upper limit of the range is reached. A further pro is that in many cases the Dpdl-Decoder on the Mobile Device has to load just one B-Tree node in order to find the result, and this Node may be in a cache, additionally improving performance for many queries.

Here a code snipped for the Node data-structure.

```
class Node {
	int index;
	int size;

	 int n = 32;
	String[] keys = new String[n + n + 1];
	int[] values = new int[n + n + 1];
	int[] children = new int[n + n + 2];


	Node() {


	}
};
```

Small devices, especially EEPROM'S and JME devices, are characterized to have a completely different or highly platform dependant storage mechanism compared to standard storages found on PCs and server stations. On EEPROM'S for example the operating system is even not given. We also consider the fact that all J2ME applications run in a "sandbox" environment, and I/O operations have several boundaries. This point affects the implementation of the B-tree on small devices, because storage operations in J2ME devices are allowed to access data using a RMS (Record Store Management System), which is limited in it's functionality. In order to guarantee memory savings and a further level of portability, an address relocation technique has been used in the B-tree implementation. These address translation is used to address chunks of data stored in a ROM (Read only Memory). Another reason to introduce this address relocation technique, is to keep the Node data of the B-tree as compact as possible. The available storage operations on J2ME device result in a very slow implementation with relatively big data-sets, and are therefore in this case just used to store and organize the Dpdl-packets.

6.3 Swapping and Compression Techiques

Dpdl has a language semantics that allows to declare multiple chunks of data with the corresponding database source from which to retrieve the data at the server-side component of Dpdl (Dpdl-Encoder). These chunks are compressed each individually. When a Dpdl-script is invoked, is can be executed in two modes either automatically or controlled accurately by the programmer through the API.

Example of a swapping declaration

This command for example operates on an installed Dpdl-Packet called BOLZANO_PHONE and processes the contained Chunk definition called BolzanoPhone

```
swapDpdlChunks("BOLZANO_PHONE", new String[]{"BolzanoPhone"});
```

When such a function is called, the Dpdl-Engine decompresses the Chunks specified in the function and swaps the resulting data to a temporary storage area in order to be processed further. This temporary storage is deleted automatically by the Dpdl runtime-system when another Dpdl-script is being processed, or may be deleted or preserved by the developer through the Dpdl API. This technique allows to contain multiple chunks of data in a highly compressed manner and to allocate and cancel them selectively. In other words is allows to install a vast set of Dpdl-Packets on a mobile device.

A sample Dpdl-Script containing multiple chunks of data

```
call(dpdlInterpreter)
::module dpdl_Example
::module_SPEC 23452
::model 836
::dpdlVersion 1.0

OPTIONS {
   TARGET = MOBILE_PHONE & PDA
   KVM = 1.0
   ZIP = true
   CHECKSUM = true
   SIGNATURE = true
   ENCRYPTION(RSA) = true
}

#defineDB DBa | 129.124.89.2 | root root
#defineDB DBb | 129.124.89.2 | root root


#defineSQL query_1 SELECT * FROM DBa
#defineSQL query_2 SELECT * FROM DBb


#defineProtocol-cHtml  layout   phone_book.html

import extern SystemData;

import virtual DATA none  {
    class data1 volatile layout   {
            ….here definitions
    }
}
#defineD data1 src DBa SQL query_ 1 {
      …..here definitions
}
import virtual DATA none  {
    class data2 volatile layout   {
            ….here definitions
    }
}
#defineD data2 src DBb SQL query_ 2 {
      …..here definitions
}
```

In the Phone-Book example these chunks could be organized (A-E, F-J, etc.…)

6.4 The compression algorithm used (LZ77)

The default compression algorithm used in Dpdl is LZ77 [6] (Lempel-Ziv substitutional compression scheme), however the algorithms can easily be plugged in the system and controlled by the Dpdl scripting language. I've chosen this algorithm for a specific reason, that is, because it's compression it's quite compact, and because data decompression, which is usually carried out on a client (Mobile Phone for example) with limited memory, is very fast.

Terms used in the algorithm

- *Input stream*: the sequence of characters to be compressed;
- *Character*: the basic data element in the input stream;
- *Coding position*: the position of the character in the input stream that is currently being coded (the beginning of the *lookahead buffer*);
- *Lookahead buffer*: the character sequence from the coding position to the end of the input stream;
- The *Window* of size W contains W characters from the coding position backwards, i.e. the last W processed characters;
- A *Pointer* points to the match in the window and also specifies its length.

The principle of encoding

The algorithm searches the window for the longest match with the beginning of the lookahead buffer and outputs a pointer to that match. Since it is possible that not even a one-character match can be found, the output cannot contain just pointers. LZ77 solves this problem this way: after each pointer it outputs the first character in the lookahead buffer. If there is no match, it outputs a null-pointer and the character at the coding position.

The encoding algorithm

1. Set the coding position to the beginning of the input stream;
2. find the longest match in the window for the lookahead buffer;
3. output the pair (P,C) with the following meaning:
    - P is the pointer to the match in the window;
    - C is the first character in the lookahead buffer that didn't match;
4. if the lookahead buffer is not empty, move the coding position (and the window) L+1 characters forward and return to step 2.

Example:
The encoding process is presented in Table 1.

- The column **Step** indicates the number of the *encoding step*. It completes each time the encoding algorithm makes an output. With LZ77 this happens in each pass through the step 3.
- The column **Pos** indicates the coding position. The first character in the input stream has the coding position 1.
- The column **Match** shows the longest match found in the window.
- The column **Char** shows the first character in the lookahead buffer after the match.
- The column **Output** presents the output in the format (B,L) C:
    - (B,L) is the pointer (P) to the **Match**. This gives the following instruction to the decoder: "Go back B characters in the window and copy L characters to the output";
    - C is the explicit **Character**.

Input stream for encoding:

Pos	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Char	A	A	B	C	B	B	A	B	C

Table 1: The encoding process

Step	**Pos**	**Match**	**Char**	**Output**
1.	1	--	A	(0,0) A
2.	2	A	B	(1,1) B
3.	4	--	C	(0,0) C
4.	5	B	B	(2,1) B
5.	7	A B	C	(5,2) C

Decoding

The window is maintained the same way as while encoding. In each step the algorithm reads a pair (P,C) from the input. It outputs the sequence from the window specified by P and the character C.

Practical characteristics

The compression ratio this method achieves is very good for many types of data, but the encoding can be quite time-consuming, since there is a lot of comparisons to perform between the lookahead buffer and the window. On the other hand, the decoding is very simple and fast. Memory requirements are low both for the encoding and the decoding. The only structure held in memory is the window, which is usually sized between 4 and 64 kilobytes.

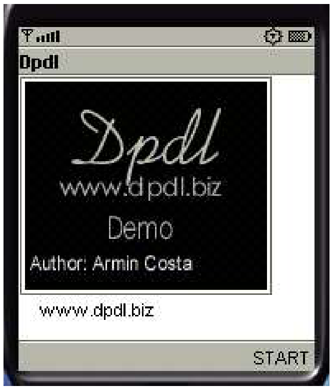
6.5 Implementation of the Dpdl-DEMO application

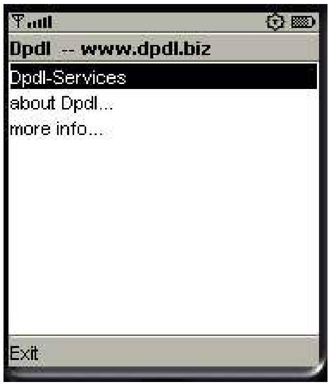
In order to test and demonstrate some of the functionalities provided by Dpdl, as part of the thesis project, I've implemented some sort of DEMO application to show one of the 1000 possible application-areas of Dpdl.

The DEMO application consists of a software-package, installable on almost every java-enabled phone and PDA, where different kind of information can be installed by the user from different sources (Gprs, internet, local access points, etc.) in form of Dpdl-Packets. Many services contained in the Dpdl-Packets can be browsed offline without necessarily being connected to a network, because the services are static (ex. Phone-Book) and data remains untouched for a certain time period . The main reason for this project, is that a User may ideally be willing to access a wide set of possible services, even if no sort of network-connection is available. A phone-book agenda of a little city or a train-schedule could be example services .

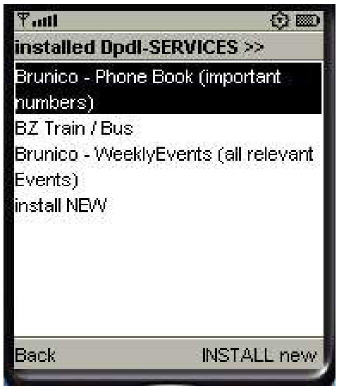
The Dpdl-DEMO application has been implemented in form of a Java™ MIDlet, which extends the Dpdl-Engine in order to perform data-allocation and retrieval. The DEMO application runs on both, J2ME V1.0 and J2ME V2.0. The input to this client-application is a Dpdl-Packet containing all service data. The Dpdl-Packets itself are generated by a server-application with the Dpdl-Encoder.

This pictures below show the main User interface of the DEMO application running on a Mobile Phone. Notice the section called "Dpdl-Services", where the user of the system has installed a variety of services in form of Dpdl-Packets. Additional services may be installed or requested by the user.





The user can install Dpdl-packet as needed, depending on how much storage space it has at disposal on his device. Newer Mobile phones have pluggable memory cards (32 MB – 500 MB or even more), so the number and size of a Dpdl-Packet can be relatively high. This screenshot shows the list of all Dpdl-Packets installed on the device (phonebook, train- and bus-schedules, weekly events).


installed Dpdl-SERVICES >>

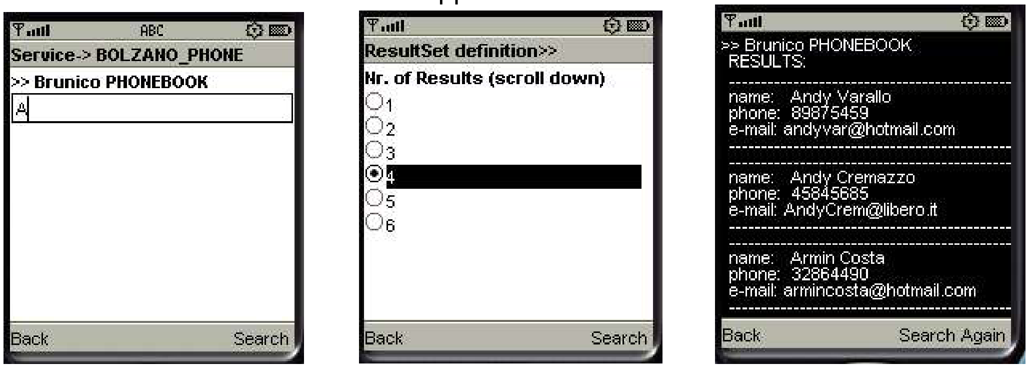
Brunico - Phone Book (important numbers)
BZ Train / Bus
Brunico - WeeklyEvents (all relevant Events)
install NEW

Back INSTALL new

Once the user has a set of Dpdl-Packets installed on his device, he may be willing to browse those services. As in this case the service data is static (data does not change for a given period of time), the user can browse the services without present network connection.

The example below shows a phonebook agenda implemented with a standalone Dpdl-script:

Figure 6.3 Screen-Shots of the DEMO Application


(1)

ABC
Service-> BOLZANO_PHONE
>> Brunico PHONEBOOK
A
Back Search

(2)

ResultSet definition>>
Nr. of Results (scroll down)
○1
○2
○3
●4
○5
○6
Back Search

(3)

>> Brunico PHONEBOOK
RESULTS:
----
name: Andy Varallo
phone: 89875459
e-mail: andyvar@hotmail.com

----
name: Andy Cremazzo
phone: 45845685
e-mail: AndyCrem@libero.it

----
name: Armin Costa
phone: 32864490
e-mail: armincosta@hotmail.com

Back Search Again

1) When the user executes the Dpdl-Packet, the Dpdl-Decoder constructs automatically the proper user interface (GUI) needed by the user to interact with the services contained in the Dpdl-packet. In this case, the loaded Dpdl-script contains just one data definition belonging to the phonebook-agenda, and therefore the system executes automatically this entry by creating an input mask. At this point, the user can insert a name to be searched in the agenda.

2) The user may also choose how may results he want to display

3) Finally the Result is shown to the user

Chapter 7 – Evaluation of the System

This Chapter provides briefly some tests in order to monitor the execution-rates of both, XML and Dpdl. In order to have a comparative overview of the execution rates, first I'll present general tests for a standard protocol like XML on different platforms, namely on a Desktop PC station and on different Mobile Devices. Than, In order to test and demonstrate the execution of Dpdl, a comparison between identical Dpdl-data and XML-data well be presented.

7.1 A general testing of XML on different platforms

In order to have a more complete picture of the execution capabilities of XML, a test suite consisting of two kind of data-sets will be executed on two different platforms. First we test the performance with medium sized data sets (ca. 100 - 200 KB). Than we'll perform another test using big data-sets.

Tests

1. XML on a PC workstation (on small- and big-datasets)
2. XML tested in J2ME on different Mobile devices

1) XML on a PC workstation

These tests have been performed with different XML parsers on a Red Hat 7.2 Linux system with a 1.4-GHz Athlon processor and 256 MB of RAM. The tests used Sun's JRE (Java Runtime Environment) 1.3.1 running under Linux kernel version 2.4.18.

a) XML tested on small data sets (on the x-axis the execution time in millisecons, y-axis different parsers):

- Soap_list.xml (131 KB): A generated SOAP document
- much_ado.xml (197 KB)
- periodic_table.xml (114 KB)
- xml.xml (192 KB)

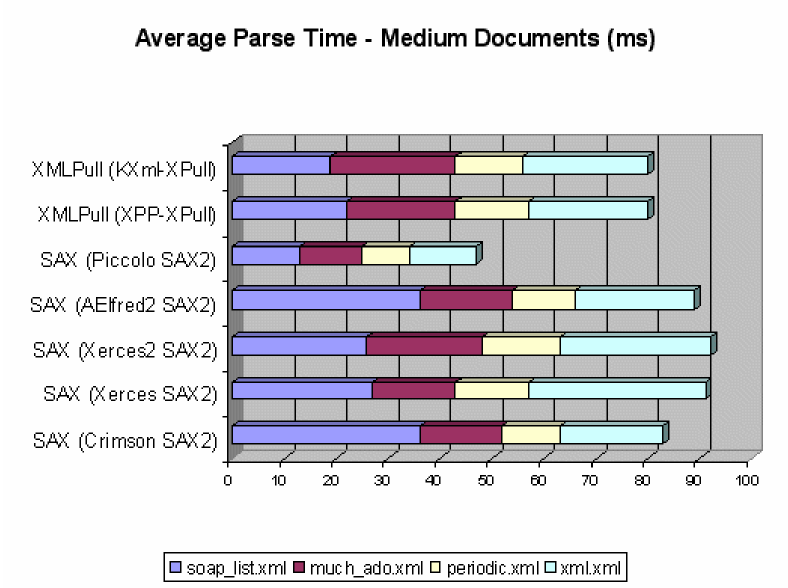


Figure 7.1 [3]

b) XML tested on big data sets (10,000 Data Entries) (2.9 MB)

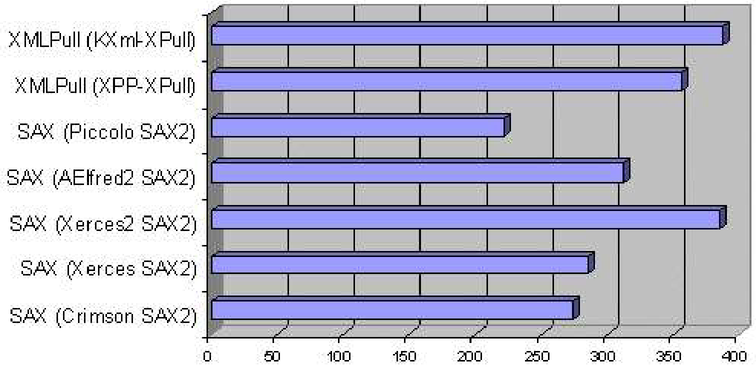


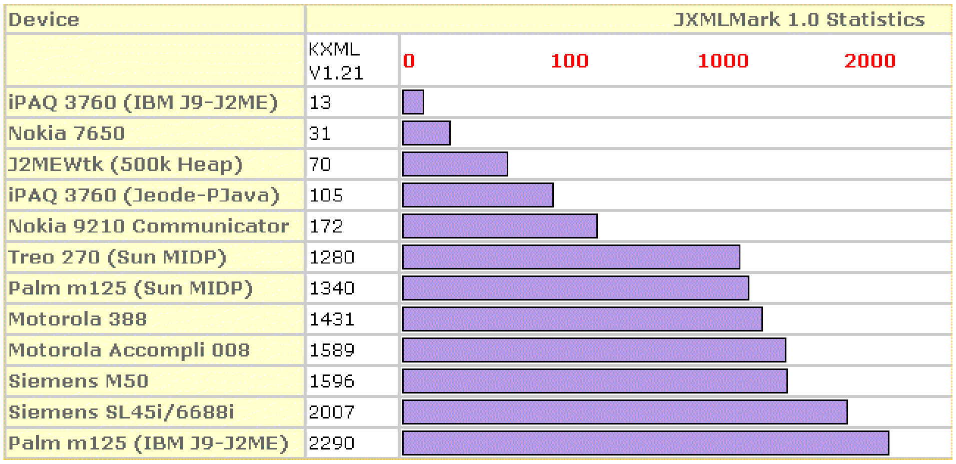
Figure 7.2 [3] XML performances on big-sized documents on a PC

2) XML tested in J2ME on different Mobile devices

Parsing XML on Mobile Phones is a common issue on Mobile Phones, but can be used merely to handle small data sets other then huge ones. Among lightweight XML parsers, kXML (kilobyte XML) is one of the most popular and standard APIs for XML parsing and can be used in the MIDP environment. Thus, we use kXML as the base API to benchmark the speed of parsing XML.

Due to limited memory, J2ME mobile devices can only parse small XML files. We created a simple and small XML file, shown below, for our benchmark. DTD (document type definition) is not needed since kXML does not validate DTD (which costs processor time and memory). Refer to picture Figure 7.3 to see the execution rates of XML decoding the file below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<message_list>
<message>
<header>
<status>xxxStatus</status>
<command>xxxCommand</command>
<messageId>xxxMessageId</messageId>
<processingRule>xxxProcessingRule</processingRule>
</header>
<body>
<a>1</a>
<b bvalue="123" />
<c cvalue="1234">cvalue</c>
</body>
</message>
</message_list>
```



Device	JXMLMark 1.0 Statistics	
	KXML V1.21	0 100 1000 2000
iPAQ 3760 (IBM J9-J2ME)	13	
Nokia 7650	31	
J2MEWtk (500k Heap)	70	
iPAQ 3760 (Jeode-PJava)	105	
Nokia 9210 Communicator	172	
Treo 270 (Sun MIDP)	1280	
Palm m125 (Sun MIDP)	1340	
Motorola 388	1431	
Motorola Accompli 008	1589	
Siemens M50	1596	
Siemens SL45i/6688i	2007	
Palm m125 (IBM J9-J2ME)	2290	

Figure 7.3 [2] Performance of XML on mobile devices (1 data entry)

7.2 Testing Dpdl and XML with identical data on Mobile Devices

A sample Test using Dpdl:

I tested Dpdl on the J2ME (Java 2 Micro Edition) V2.0 Emulator which may be at least ca. 20-30 x slower than a Nokia 7650 for example.

The dataset I use consists of 10.000 phone entries in a Phone-Book implementation. Each entry is 78 bytes. The encoded Dpdl-Packet is 300 Kb in size. Once the Engine has been allocated, the average execution time to find an entry in the phone-book is 220 Milliseconds.

The equivalent XML file (10.000 entries) is 1,31 MB in size and execution time and memory requirement is a multiple.

Dpdl:

Data-set:	10.000 Entries
Dpdl-Packet:	300 Kb
Execution time:	220 Milliseconds to find a given key in the Data-set

Equivalent xml data:

Data-set:	10.000 Entries
XML-size:	1,31 MB
Execution time:	* (at least 20-70 seconds for a sequential scan)

*hard to estimate in this case, because XML query languages for small devices haven't already been implemented (XPath or XQuery) and Filtering the data in a linear way, is very slow and memory consumptive.

7.2 Conclusion

Self Assessment

This paper should be enough to understand the basic idea and usage of Dpdl. However, in order to study more accurately the system and application environments, we would need an additional paper showing the architecture in detail and provide further application examples. Up to now I can state that Dpdl is a well working piece of software, stable enough to be used in other software applications. But as we all are aware of, before a system gets really complete, an additional requirement analysis and prototype testing has to be done in order to complete the system. It was not an easy task to build the system with performance and overall compatibility in mind. One big problem I had during the development on Dpdl, was the missing of basic API functions in J2ME devices, which took away several weeks of precious time because I had to hack around in the JVM in order to implement the missing functionalities from scratch. The choices I've done in the implementation, were made in order to guarantee overall compatibility. This task was not easy, because I had to take into account several trade-offs even when using simple API functions for accessing the permanent storage of a device, because especially those I/O functions are very often bound to a particular microchip architecture. Due to the availability of software tools to translate simple Java into C, I can claim that Dpdl has been made compatible with every device worldwide as soon as a C compiler is available for the target platform. This compatibility was one of the fundamental functional requirements of the system, because the variety of small devices is immense.

Future work on Dpdl

I tried to keep the code design of the system as modular as possible in order to allow easily future development. I'll for sure keep the project alive and I've already some very clever ideas to optimize further Dpdl and some basic ideas about additional functionalities. I plan to publish a web-page for Dpdl in order to gather input form programmers willing to test the system. I you ask me, I'll use Dpdl whenever I need performance on small devices.

Bibliography

[1] SUN Microsystem, *www.javasoft.com*
[2] J2ME devices: Real-world performance, *www.javaworld.com*
[3] XML documents on the *run, www.javaworld.com*
[4] IBM, *www.ibm.com*
[5] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi and Riccardo Torlone , *Database Systems concepts, languages and architectures*, McGraw-Hill 1999
[6] LZ77 *http://burks.brighton.ac.uk/burks/foldoc/44/69.htm*

Appendix A

The Dpdl-implementation of the PhoneBook example

```
/*##########################################################
              Dpdl
  Author:   Armin Costa
  contact: armincosta@dpdl.biz
  ---------------------------------------------------
  EXAMPLE:       City Phone-Book
  ###########################################################
**/
call(dpdlInterpreter)
::module dpdl_PHONEBOOK
::module_SPEC 23452
::model 836
::dpdlVersion 1.0
// Definition part
OPTIONS {
    TARGET = MOBILE_PHONE & PDA
    KVM = 1.0
    Dpd-LZZ = true
    CHECKSUM = true
    SIGNATURE = true
    ENCRYPTION(RSA) = true
}

#defineDpdlEncoding UTF-8     // encoding && decoding definitions
#defineDpdlDecoding UTF-8

#defineDB phone_bz | 129.124.89.2 | root root          // database connectivity

#defineSQL query_ SELECT * FROM PHONE_BZ       // query definition

#defineProtocol-cHtml phonebook_visual phone_book.html       // visualization layer

import extern SystemData;       // System class
// Visualization-part
import virtual DATA none  {
        class BolzanoPhone volatile phoneB_css {// volatile→data can be formatted with other protocols
              DATA::string const name; // constraint on variable name
              DATA::string using phoneNR;
              DATA::string using  e-mail;
              #defineGUI Default <Brunico_PhoneBook>  <please_enter_name_and_surname_here:>
        }   //#defineGUI → automatic GUI generation
}
// Data-Container part
#defineD BolzanoPhone src phone_bz SQL query_ {
                CHUNK entry [6]; //  means that [6] is the number of max. possible Resultset
                struct BTree DENSE_INDEX  *name //  build a primary dense Index on name
                entry.content { name , phoneNR , e-mail }
                entry.name TAG(0xef) const (string) = 20;
                entry.phoneNR TAG(0xefe) (string) = 15;
                entry.e-mail TAG(0xefee) (string) = 30;
}
```

Appendix B

XML Example in J2ME

```
<?xml version="1.0"?>
<!DOCTYPE rss PUBLIC
"-//Netscape Communications//DTD RSS 0.91//EN"
"http://my.netscape.com/publish/formats/rss-0.91.dtd"
>
<rss version="0.91">
<channel>
<title>Meerkat: An Open Wire Service</title>
<link>http://meerkat.oreillynet.com</link>
<description>
Meerkat is a Web-based syndicated content reader based on RSS ("Rich Site
Summary").
RSS is a fantastic, simple-yet-powerful syndication system rapidly gaining
momentum.
</description>
</channel>
</rss>
```

J2ME java code to decode the XML file [4]

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.io.*;
import java.io.*;
//kxml imports
import org.kxml.*;
import org.kxml.parser.*;
public class ParseXML extends MIDlet implements CommandListener {
private Command exitCommand; // The exit command
private Command displayXML; // On execution, it displays title and description
                             // on phone screen
private Display display;     // The display for this MIDlet
// UI Items for display of title and description on phone screen
private static TextBox t;
private static String textBoxString = "";
// XML String
private String xmlStr = "";
public ParseXML() {
   display = Display.getDisplay( this );
   exitCommand = new Command( "Exit", Command.EXIT, 2 );
   displayXML  = new Command( "XML", Command.SCREEN, 1 );
  // The XML String in form of RSS
  StringBuffer xmlString = new StringBuffer();
  xmlString.append("<?xml version=\"1.0\"?>
                        <!DOCTYPE rss PUBLIC \"-//Netscape Communications//DTD RSS
0.91//EN\"");
  xmlString.append("\"http://my.netscape.com/publish/formats/rss-0.91.dtd\">");
  xmlString.append("<rss version=\"0.91\">");
  xmlString.append("<channel><title>Meerkat: An Open Wire Service</title>");
  xmlString.append("<link>http://meerkat.oreillynet.com</link>");
  xmlString.append("<description>Meerkat is a Web-based syndicated content
                                         reader based on RSS (\"Rich Site
Summary\").
                                         RSS is a fantastic, simple-yet-powerful
syndication
                                         system rapidly gaining momentum.");
  xmlString.append("</description><language>en-us</language>");
  xmlString.append("</channel>");
  xmlString.append("</rss>");
  xmlStr = xmlString.toString();
 }
public void startApp() {
    // The textbox displays title and description from a RSS String
    t = new TextBox( "MIDlet XML", "kXML", 256, 0 );
    t.addCommand( exitCommand );
    t.addCommand( displayXML );
    t.setCommandListener( this );
    display.setCurrent( t );
 }
/**
Pause is a no-op since there are no background activities or
record stores that need to be closed.
*/
public void pauseApp() { }
/**
Destroy must cleanup everything not handled by the garbage collector.
In this case there is nothing to cleanup.
*/
public void destroyApp(boolean unconditional) { }
/*
Respond to commands, including exit. On the exit command, cleanup
and notify that the MIDlet has been destroyed.
*/
public void commandAction(Command c, Displayable s) {
  if ( c == exitCommand ) {
    destroyApp( false );
    notifyDestroyed();
  }
  else if ( c == displayXML ) {
    try {
      viewXML();
    }
    catch( Exception e ) {
        e.printStackTrace();
    }
   }
}
// This function sets up kxml parser and calls traverse() to parse the whole XML
String
public void viewXML() throws IOException {
   try {
    byte[] xmlByteArray = xmlStr.getBytes();
    ByteArrayInputStream xmlStream = new
    ByteArrayInputStream( xmlByteArray );
    InputStreamReader xmlReader = new
    InputStreamReader( xmlStream );
     XmlParser parser = new XmlParser( xmlReader );
     try
     {
       traverse( parser, "" );
     }
     catch (Exception exc)
     {
       exc.printStackTrace();
     }
    return;
}
catch ( IOException e ) {
        return ;
   } finally {
  return ;
   }
 }
/**
Traverses the XML file
*/
public static void traverse( XmlParser parser, String indent ) throws Exception
{
  boolean leave = false;
  String title = new String();
  String desc = new String();
  do {
  ParseEvent event = parser.read ();
  ParseEvent pe;
  switch ( event.getType() ) {
   // For example, <title>
   case Xml.START_TAG:
   // see API doc of StartTag for more access methods
   // Pick up Title for display
    if ("title".equals(event.getName()))
      {
      pe = parser.read();
      title = pe.getText();
    }
   // Pick up description for display
    if ("description".equals(event.getName()))
    {
        pe = parser.read();
        desc = pe.getText();
    }
    textBoxString = title + " " + desc;
    traverse( parser, "" ) ; // recursion call for each <tag></tag>
    break;
    // For example </title?
   case Xml.END_TAG:
    leave = true;
    break;
    // For example </rss>
   case Xml.END_DOCUMENT:
    leave = true;
    break;
    // For example, the text between tags
   case Xml.TEXT:
    break;
   case Xml.WHITESPACE:
    break;
    default:
    }
    } while( !leave );
    t.setString( textBoxString );
  }
}
```

Appendix C

A full featured, standalone Dpdl-script (Note: use a C++ syntax Highlighter to view the script)

```
/*#####################################################
          Dpdl
   Dynamic Packet Definition Language
        www.dpdl.biz
      copyright© 2003-2006


 this is a prototype-script written in Dpdl
 (Dynamic Packet Definition Language ) showing
 how services could be implemented on the Dpdl platform

 Author:   Armin Costa
 contact: armincosta@dpdl.biz
 ----------------------------------------------------
 EXAMPLE:  City Phone-Book / Bus-Schedules / Event-Calendar

#####################################################
**/
call(dpdlInterpreter)
::module dpdl_CITY_SERVICE_BZ
::module_SPEC 23452
::model 836
::dpdlVersion 1.0

OPTIONS {
  TARGET = MOBILE_PHONE & PDA
  KVM = 1.0
  ZIP = true
  CHECKSUM = true
  SIGNATURE = true
  ENCRYPTION(RSA) = true
}
```

```
#defineDpdlEncoding UTF-8
#defineDpdlDecoding UTF-8

#defineDB DB_bz | 129.124.89.2 | root root

#defineSQL query_PhoneBook SELECT name, phoneNR, e-mail FROM PHONE_BZ end
#defineSQL query_Trains SELECT * FROM BZ_Train end
#defineSQL query_Buses SELECT * FROM BZ_Buses  end
#defineSQL query_Events SELECT * FROM br_events end

#defineProtocol-cHtml phonebook_visual {
                <html>

                <head>
                <meta http-equiv="Content-Type"
                    content="text/html; charset=iso-8859-1">
                 <meta name="GENERATOR" content="Microsoft FrontPage 5.0">
                 <title>Bolzano PhoneBook</title>
                 </head>

                 <body>

                 <h2><font size="2">Results:</font></h2>
                 <hr>
                 <font size="2"><b>Name: </b>&nbsp; $name</font>
                 <p><font    size="2"><b>Tel:</b>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;
$phoneNR</font></p>
                 <font size="2"><b>e-mail: </b>&nbsp; $e-mail</font>
                 <hr>
                 </body>
                 </html>
}
#defineCSS-StyleSheet train_css train_display.css
#defineCSS-StyleSheet events_css train_display.css


#defineDpdlService important_phone Http 129.283.48.3:7903/BZ_important_phone_nr
#defineDpdlService bussiness_phone Http 129.283.48.3:7903/BZ_bussiness_phone_nr
#defineDpdlService private_phone Http 129.283.48.3:7903/BZ_private_phone_nr

import extern SystemData;

import virtual DATA none  {
      class BolzanoPhone volatile phonebook_visual {
           DATA::string const name;
           DATA::string using phoneNR;
           DATA::string using  e-mail;
           #defineGUI                  Default                  <Brunico_PhoneBook>
<please_enter_name_and_surname_here:>
      }
}
#defineD BolzanoPhone src DB_bz SQL query_PhoneBook {
            CHUNK entry [6];
            struct BTree DENSE_INDEX hashing *name
            entry.content { name , phoneNR , e-mail }
            entry.name TAG(0xef) const (string) = 20;
            entry.phoneNR TAG(0xefe) (string) = 15;
```

```
                entry.e-mail TAG(0xefee) (string) = 30;
}
//////////////////////////
import virtual DATA none {
        class Train volatile train_css {
              MAP::string const target;
              MAP::string using trains;
              #defineGUI List ONCE target #defineSUBGui Train.subselection_1
        }


        class Bus {
              MAP::string const target;
              MAP::string using busses;
              #defineGUI Menu ONCE target
        }
}
#defineD Train src DB_bz SQL query_Trains {
                CHUNK entry [1];
                entry.content { target , trains }
                entry.target TAG(0xef) const (string) = 30;
                entry.trains TAG(0xefe) (string) = 200;
                entry.reservation_form TAG(0x99f)(class) for(CANVAS) __stdcall(visualize());
}
#defineD Bus src DB_bz SQL query_Buses {
                CHUNK entry [1];
                entry.content { target , busses }
                entry.target TAG(0xef) const (string) = 30;
                entry.busses TAG(0xefe) (string) = 120;
}
///////////////////////////
import virtual DATA none {
        class Events volatile events_css {
              MAP::string const day;
              MAP::string using events;
              MAP::string using optional;
              #defineGUI Menu ONCE day
        }
}
#defineD Events src DB_bz SQL query_Events {
                CHUNK event [1];
                event.content { day , events , optional }
                event.day TAG(0xefe) const (string) = 20;
                event.events TAG(0xe98f) (string) = 800;
                event.optional TAG(0xef) (string) = 50;
}
```

Appendix D

For a complete list of Java™ enabled devices refer to:

www.javasoft.com

http://java.fritjof.net/