Date of acceptance | Grade
October 2009 | Eximia

Instructor
Sasu Tarkoma

Dessy: desktop search and synchronization

Eemil Lagerspetz

Helsinki Monday 14th September, 2009

Pro gradu thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science

Tekijä — Författare — Author
Eemil Lagerspetz

Työn nimi — Arbetets titel — Title

Dessy: desktop search and synchronization

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level Pro gradu thesis	Aika — Datum — Month and year Monday 14th September, 2009	Sivumäärä — Sidoantal — Number of pages 54 pages + 0 appendix pages

Tiivistelmä — Referat — Abstract

Current smartphones have a storage capacity of several gigabytes. More and more information is stored on mobile devices. To meet the challenge of information organization, we turn to desktop search. Users often possess multiple devices, and synchronize (subsets of) information between them. This makes file synchronization more important. This thesis presents Dessy, a desktop search and synchronization framework for mobile devices. Dessy uses desktop search techniques, such as indexing, query and index term stemming, and search relevance ranking. Dessy finds files by their content, metadata, and context information. For example, PDF files may be found by their author, subject, title, or text. EXIF data of JPEG files may be used in finding them. User–defined *tags* can be added to files to organize and retrieve them later. Retrieved files are ranked according to their relevance to the search query. The Dessy prototype uses the BM25 ranking function, used widely in information retrieval. Dessy provides an interface for locating files for both users and applications. Dessy is closely integrated with the Syxaw file synchronizer, which provides efficient file and metadata synchronization, optimizing network usage. Dessy supports synchronization of search results, individual files, and directory trees. It allows finding and synchronizing files that reside on remote computers, or the Internet. Dessy is designed to solve the problem of efficient mobile desktop search and synchronization, also supporting remote and Internet search. Remote searches may be carried out offline using a downloaded index, or while connected to the remote machine on a weak network. To secure user data, transmissions between the Dessy client and server are encrypted using symmetric encryption. Symmetric encryption keys are exchanged with RSA key exchange. Dessy emphasizes extensibility. Also the cryptography can be extended. Users may tag their files with context tags and control custom file metadata. Adding new indexed file types, metadata fields, ranking methods, and index types is easy. Finding files is done with virtual directories, which are views into the user's files, browseable by regular file managers. On mobile devices, the Dessy GUI provides easy access to the search and synchronization system. This thesis includes results of Dessy synchronization and search experiments, including power usage measurements. Finally, Dessy has been designed with mobility and device constraints in mind. It requires only MIDP 2.0 Mobile Java with FileConnection support, and Java 1.5 on desktop machines.

ACM Computing Classification System (CCS):
H.3 [Information storage and retrieval],
H.3.1 [Content analysis and indexing],
H.3.3 [Information search and retrieval],
H.3.5 [Online Information Services]

Avainsanat — Nyckelord — Keywords
desktop search, synchronization, mobile computing, metadata

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Today's smartphones have a storage capacity of several gigabytes, just like the desktops of yesterday. As the amount of information grows, conventional search tools become less effective. To meet these challenges, we turn to desktop search. Recent desktop search applications have provided mostly searching for documents using their content. Few allow searching by user-assigned tags, file metadata, and context information. Providing user-friendly metadata is a subject of current research. As users often possess multiple devices and use them to access the same files, file synchronization becomes more important. Most desktop search systems that index files on one machine do not provide search capabilities on another device. None of today's desktop search systems include synchronization support. Current synchronization software for mobile devices copies information off the mobile device and stores it on a desktop computer. Concurrent change reconciliation is typically not supported. To my knowledge, search and synchronization of found results on mobile devices is not supported.

This thesis presents Dessy, a desktop search and synchronization system for limited devices. Dessy finds files by their content, metadata, and context information. It enables searching from the local file system, remote computers, and the Internet. Dessy supports remote searching and index synchronization to allow offline searching on another device and offload the indexing task to a more capable machine. Dessy supports synchronization of files and directory trees found by the search. It provides an interface for locating files for both users and applications. Dessy uses the Syxaw file synchronizer with XML-awareness to synchronize found files. Syxaw is designed for limited devices, and follows an efficient synchronization protocol that minimizes the number of network round-trips required for synchronization. Syxaw enables separate synchronization of file metadata and data, useful for index synchronization. It also provides Dessy with unique file identifiers suitable for efficient index metadata storage.

Recent years have seen the development of numerous online file synchronization services. Services such as Dropbox[1] and Ubuntu One[2] allow a user to synchronize files between her computers, and share folders and files with other users. Dropbox runs on Linux, Windows and Mac OS computers, while UbuntuOne is designed primarily for Ubuntu Linux. Recently developed services for mobile phones include Nokia's Ovi[3], designed for Nokia phones, and Apple's MobileMe[4] that runs on the iPhone. Both of them synchronize the user's calendar and files, and allow sharing files with others.

Dessy does not aim to be a sharing service, but it enables synchronization of a user's files. Dessy does not store a user's files online, but leaves them on the user's own computers. Contrary to the mentioned existing online synchronization ser-

[1] http://www.getdropbox.com

[2] https://ubuntuone.com

[3] http://www.ovi.com/services

[4] http://www.apple.com/mobileme

vices, Dessy also provides desktop search capabilities. Extensions to share files and searches can be developed.

The problem that Dessy addresses is that of efficient mobile desktop search and synchronization. Efficient desktop search means that files are found without iterating the file system, in reasonable time, and by at least their content in addition to their names. Efficient search can be accomplished by use of an index stored either on the local device or a remote system. To support search of local files, the device's files should be indexed. With Dessy, it is possible to download an index constructed on another computer, and find files using that index. This enables remote searching when connectivity is not available. Dessy can also find files residing on the remote computer using a weak connectivity network, such as GPRS or 3G, by sending queries to that computer, and letting it carry out the searches. Searches for files may also be carried out in the Internet through popular search engines, such as Google. Dessy is able to find files by their names, content, user-assigned tags, metadata, and context information, such as EXIF data of JPG files, author, subject, and keywords of PDF files, and so forth.

Efficient synchronization means that a minimal number of round trips are required to synchronize files with a remote computer. This is taken care of by the Syxaw file synchronizer, which Dessy uses for synchronization support. For files on the Internet, the Syxaw synchronization server cannot be used, and files must be downloaded normally. However, only files changed since the last synchronization are downloaded.

The following use–case illustrates Dessy. Mr. Smith is commuting, and reading a computer science article, *Dessy: Towards Flexible Mobile Desktop Search*, on his smartphone. The article refers to another article, titled *A three-way merge for XML documents*, by Mr. Lindholm. Mr. Smith decides that he should read *A three-way merge for XML documents* to obtain a proper understanding of *Dessy: Towards Flexible Mobile Desktop Search.* To read *A three-way merge for XML documents*, Mr. Smith types the search terms `three-way merge for XML documents` and `author:Lindholm` to Dessy, and clicks `Search`. Dessy reports that there are no local results on the smartphone, and none on Mr. Smith's computer either. However, Dessy shows promising results via Google Scholar: the result description contains the title, *A three-way merge for XML documents* and the name Lindholm. So, Mr. Smith chooses `Synchronize` on the Google Scholar result, and *A three-way merge for XML documents* is downloaded to the smartphone. Mr. Smith then opens it in his PDF reader and proceeds to reading it.

Finding files in Dessy is performed using virtual directories, similar to those in Semantic File Systems [GJSJWO91]. On desktop machines, these can be used from applications and by the user by directly browsing them with a file manager. On all devices, The Dessy GUI provides easy access to the features of the virtual directory system. On desktop machines, the virtual directories are also provided through an NFS [Now89] or FUSE[5] interface, which allows applications unaware of Dessy to browse them.

[5] File system in Userspace. `http://fuse.sourceforge.net/`

Dessy is extensible in many ways. Users can add *tags* similar to those used in other Desktop Search software to represent files' context associations. These are also used by Dessy–aware applications to add context information to files. For example, a photo taking application running on a mobile phone could tag taken photos with their location as given by the map software of the phone. Text files, such as meeting minutes, could be tagged by the user's current calendar event. Developers can easily add new file properties for finding files, such as author, artist, keywords, album, bitrate, and so on. For properties with complicated values, dynamic query aliases can be created. An alias is translated to a real query, and finds the results of that query. For example, the alias `mod-date:today` finds all files modified today (`mod-date:2009-09-14`). Another example could be the value `large` for an image property `size:`. The query `size:large` would find images of $1024x768$ pixels and larger. Other examples of query aliases include the special `sync:` and `sync-md:` directories, which allow synchronizing files and directories contained in their parent directory. Developers may also add custom entries in virtual directories. An example of this is showing the alias `today` to the user in the `mod-date:` virtual directory, even though no files have the value `today` as a modification date. This helps the user in learning the aliases, as she does not need to know about them beforehand.

The structure of this thesis is as follows:

Section 2 introduces the background of mobile desktop search and related work. Section 3 lists the design goals of Dessy. Section 4 describes the design of Dessy, and how it fulfills the design goals. Section 5 presents the current Dessy prototype. Section 6 evaluates the performance of the system and its energy efficiency. Section 7 discusses the results, and finally, Section 8 presents future ideas and concludes the thesis.

2 Background and related work

Due to recent improvements in desktop computer disk space availability, more files are stored on home and work computers. The inefficiency of finding files on one's own disk compared to the speed of Internet searches has prompted a rise in desktop search software development. The term *desktop search* is commonly used to refer to finding files on the user's own computer, as opposed to Internet document searches. A desktop search program usually incorporates indexing and querying for documents by their content, metadata, and/or context information. Metadata includes photo tags, EXIF info for JPEG files and file modification dates. Context information can be e.g. the location where a file was created, which project the file is part of, who it is shared with, and what files are often used with it. In this thesis, desktop search is used to refer to a piece of software capable of quickly finding files from the user's own device, based on, at least, the files' content. Specifically, finding files by their file names using exhaustive iteration of the directory tree at search time is not desktop search.

Recent years have seen a multitude of desktop search applications being developed.

Some of the most known examples of these are Apple's Spotlight[6], Novell's open-source project Beagle[7], the Tracker project[8], Copernic Desktop Search[9], Google Desktop[10], Microsoft's SiS [CDT06], and Windows Search[11].

Beside the usual user interfaces for queries, finding files has also been approached in a different manner, such as using virtual directories [GJSJWO91] and directory namespaces [HC03]. A *virtual directory* (also called a *virtual folder*) is a directory that does not exist in the file system hierarchy, but instead shows found documents according to search criteria specified for the directory. A benefit of the virtual directory abstraction is that saving searches just requires creating a shortcut to a folder in most desktop file managers. In Semantic File Systems [GJSJWO91], two types of virtual directories were used: *field virtual directories*, that denoted a field or property name, and *value virtual directories*, that indicated the value of the property that the parent field virtual directory identified. For example, a file having the value `bob` for the field `owner:` would be found in the virtual directory `owner:/bob`. A virtual directory-like approach is also present in later versions of Apple's Spotlight. There it is called *Smart Folders*. In Windows Vista, saved desktop searches are called *Search Folders*. In Semantic File Systems [GJSJWO91], the virtual directories were accessed through one server, and indexes were not transferrable to other devices. Dessy adopts the virtual directories for desktop searching, using the local device for index storage instead of a server. This allows offline browsing. In the rest of this thesis, field virtual directories will be called *property virtual directories* to better match their use in Dessy.

Many of today's desktop search applications look for files according to their content or fields specific to file types. In Connections [SG05], document creation and modification times were used to create links between documents to help the user find related files. This technique was used to improve a regular desktop search system. The use of context information and user-friendly metadata was explored by Cuttrell et al [CRDS06] and Hess et al [HC03] by using application and user–defined *tags*. A tag is an arbitrary keyword or a property — value pair assigned to an object, such as `tag:important` for an email or `location:france` on a set of photos. In Dessy, modification dates and other metadata properties can be used to search for a file. Adding custom metadata and context tags is supported.

In 2008, I worked on a natural language retrieval engine for grocery product descriptions [NLB$^+$08b, NLB$^+$08a]. The system indexes short product descriptions and names, written in Finnish, with natural language, abbreviations, colloquialisms, noisy and erroneous data, and Finnish case variation. The product descriptions are searched using natural language expressions found on a typical grocery list. The search engine is used to support a mobile grocery list assistant that recommends

[6] `http://www.apple.com/macosx/features/spotlight/`

[7] `http://beagle-project.org/Main_Page`

[8] `http://projects.gnome.org/tracker/`

[9] `http://www.copernic.com`

[10] `http://desktop.google.com`

[11] `http://www.microsoft.com/windows/products/winfamily/desktopsearch`

products to buy based on the user's current grocery list and shopping history. During the project, I studied and implemented techniques crucial for desktop search and indexing, including inverted file indexing, stemming and lemmatization, misspelling correction with edit distance, partial word match querying, and the BM25 ranking function. In the following sections, I will introduce these techniques, and the approach taken in Dessy.

2.1 Indexing

The ability of quickly finding files inherent in Desktop search software makes use of a technique called *indexing*. Indexing consists of recording properties and keywords of files in a special structure called the index. The files that match the stored keywords and properties can then be quickly found using the index. There are two popular indexing techniques: *Signature file* indexing and *inverted file* indexing.

In *inverted file* [HBYFL92] indexing, a list of file identifiers is stored for each keyword. When the keyword appears in a query, the list is added to the results or intersected with them, depending on whether the query was a boolean **or** or **and**. In *signature file* indexing, a hash function is used on the words in a file to determine the *signature* of the file. Queries are then hashed and matched against this signature to determine if files match. These methods of indexing are compared in [ZMR98]. According to the results of the comparison, inverted file indexing is superior. The index is smaller since compression can be used to reduce index size without sacrificing too much query performance, unlike when using signature file indexing. Inverted file query performance is generally better, since checking for false positives is not necessary. Inverted file indexing is used in Dessy.

A data structure called a *bloom filter* [Blo70] can be combined with any type of indexing. In addition to storing words in the index while indexing, the word is hashed with k hash functions, to obtain indexes to a fixed–size bit array, $w_1, w_2, ..., w_k$. Bits at these indexes are set to 1. During lookup, the query words are hashed similarly and compared with the stored bloom filter. If any bit on indexes $q_1, q_2, ..., q_k$ is not 1, the word does not exist in the index, and therefore will return no results. If all bits on the indexes q_i are set to 1, the word either exists in the index, or the bits were set when adding other words, and the query is a false positive. This enables an index to answer very quickly to the question whether a word exists in the index or not. False negatives are not possible with a bloom filter. Therefore, a bloom filter may be used for checking word existence before a real lookup. This should reduce search time in the case that the index has no results. Also, by optimising k and the size of the bit array, the percentage of false positives can be brought down to comfortable levels, such as under 1% or 0.1%. Note that with many complex to calculate hash functions, a lookup in the index may be faster without bloom filters. A large part of making bloom filters efficient is choosing fast and relatively uncorrelated hash functions.

A problem with using bloom filters for desktop search is that deletion of entries in a classic bloom filter is not possible. Unsetting the bits that a query word hashes to accomplishes deletion, but may also delete other words from the index that hash to at least one of the bits that were unset. To support deletion, counted bloom filters were created. In this extension, the bit array is replaced by an array of n–bit buckets, each bucket counting the number of elements hashed to that bucket. Addition would then increment the bucket value, and deletion decrement it. However, as the bucket size is a fixed n, buckets may reach the maximum value n, after which we cannot know about further additions to that bucket. At that point, deletion and addition should both leave the value at the maximum value, and the behaviour of the bloom filter becomes that of the classic bloom filter. Should too many buckets reach the maximum value, we can regenerate the bloom filter with a higher n by re–reading the index word dictionary. Dessy does not use bloom filters. Their applicability to Dessy for performance gain remains future work.

In many desktop search applications, such as Beagle and [GJSJWO91], indexing of different file types is done by *indexing helpers*. Indexing helpers are specialized property and text extractors designed to handle one or more file types. Using indexing helpers simplifies extending the desktop search application, since adding a new file type boils down to writing a new indexing helper. For easy file handling architecture and extensibility, Dessy uses indexing helpers.

In order to reduce the size of the index, it is possible to use word fragments of a certain length n, called *n-grams*, instead of words in an inverted index. In [MD83], this possibility was explored. As the number of alphabetical *2-grams* and *3-grams* is relatively small (676 and 17 576, respectively), using these instead of words limits the size of an index. However, this technique causes false positives, since a phrase can include the n-grams of another, even though it does not contain the words of the latter. For example, the query `duck`, consisting of the 2-grams `du`, `uc` and `ck`, would match a document containing the phrase `dumb luck`, since it has the word fragment `du` in `dumb`, and `uc` and `ck` in `luck`. Also, n-grams cannot be directly applied to character-word languages with thousands of symbols, such as Chinese. Dessy uses words in the inverted index to avoid false positives and stay applicable for a larger set of languages.

2.2 Language issues

Documents on a user's device may be in a variety of languages, depending on the user's preferred language. This section deals with the language-derived issues that have to be considered for indexing and querying for documents in different languages.

To be able to efficiently index documents in a given language, it is first necessary to identify the language the documents use. Indexing English, Finnish and Chinese is different: words in English are shorter and often easier to bring to their basic forms than in Finnish. In Chinese, detecting words is considered a difficult problem, but, alternatively, characters can be considered words, which makes indexing simple. Different languages require different, language-dependent preprocessing. Also, the language of user queries must be detected, in order to process them identically to the set of indexed documents. Language detection using n-grams was explored by T. Dunning in [Dun94]. Implementing language identification is outside the scope of this thesis. The Dessy prototype assumes that documents are in English.

Stemming and lemmatization

Indexing English words in an efficient way involves either *stemming* or *lemmatizing* each keyword in a file, and each query word. Stemming is the process of reducing a word to a form shared by the different inflections or cases of the word. For example, `stemming`, `stemmer` and `stem` could be reduced to `stem`. This increases the recall of the search engine, since documents with different forms of the same word are found with the same query. Lemmatization means reducing word forms to their *lemma*, or "basic" grammatical form. While stemming does not guarantee the stem is a valid word, lemmatization does. Lemmatization also handles language specific special cases and can associate dissimilar word forms with the same lemma. For example, the words `better` and `best` will be reduced to `good` when using an English lemmatizer, but with a stemmer, they may both be reduced to `bes` or `bet` or have two different stems. Understandably, lemmatization requires a more detailed language model. For the English language, stemming is often good enough for desktop search purposes. The Porter stemmer [Por80] is popular for English systems. For languages with more complex inflection and case structures, such as Finnish, Lemmatization is preferable. The Malaga language analysis tool[12] can be used for word detection, case detection and stemming. With the Suomi-Malaga Finnish grammar[13], Malaga makes a powerful Finnish lemmatizer. Indexing character or syllable based languages such as Chinese is easier, since one can generally consider each character a word. The Dessy prototype uses the Porter stemmer to handle the English language. Other languages are not handled at the moment.

Misspellings

The target users of Dessy are regular people, and people often make mistakes. Accidental misspellings in query words can be taken into account by comparing query words with index words with a word distance metric. One such metric is edit distance, where the distance between two words is defined by the smallest number of character additions, deletions, or replacements that are required to produce the second word from the first. A good edit distance metric is the Levenshtein metric [Lev66]. If a word is found in too few documents, a possible cause might be that the word is misspelt. In this case, its edit distances to index words are calculated, and the closest matching word is added to the query. Edit distance may also be used to show the user a suggestion for another search with more matches, such as

[12] `http://home.arcor.de/bjoern-beutel/malaga/` [Retrieved on: 2008-08-09]

[13] `http://joyds1.joensuu.fi/suomi-malaga/suomi.html` [Retrieved on: 2008-08-09]

the "Did you mean ..." in Google. Misspellings are not supported in the Dessy prototype at the date of writing. However, adding misspellings support is easy through implementing a Dessy query plugin to translate queries by correcting misspellings.

2.3 Querying

Once the index has been constructed, querying for documents with keywords is a simple lookup operation. The index structure is addressed with query words, and the resulting sets of documents are combined by adding them together or intersecting them with each other, depending on if the query was a boolean OR or a boolean AND query. Before the query is performed, however, the same preprocessing, such as lemmatization and stemming, that was done in the indexing phase, needs to be performed on the query word. This maximizes the number of matches the query can obtain from the index.

Lookup and full-text search

Finding documents that contain words with in–word matches is called *full-text* search. Full-text search often gives more results than simple exact match lookup. In full–text search, the terms of the index have to be ordered, usually in a term dictionary, and must have a support index, so that searching it for partial matches can be made computationally inexpensive. Implementing full-text search in Dessy is left as future work. Implementing it for a MySQL[14]–based Dessy index is not difficult, since MySQL databases offer full-text and in–word matching. However, adding support for the mobile Java version, where MySQL is unavailable, necessitates finding or implementing a fast, ordered index storage structure for smartphones. One option would be to implement an index storage structure using SQLite[15], in C on Symbian, and using the Android APIs on Android devices, but this would reduce the number of platforms that Dessy can run on. Full–text search is among future work for Dessy.

Ranking

When searching through large amounts of data, the number of results shown to the user can be overwhelming. This is why the relevance of results needs to be measured, and they need to be presented in order of decreasing relevance. Both Internet search engines and desktop search systems employ a *ranking* function. Popular relevance metrics used in traditional information retrieval are the basic *Term Frequency - Inverse Document Frequency* or *TF-IDF* [SB87] measure and TF-IDF derived measures. The TF-IDF measure gauges the importance of a word by how often it appears in the document, and how often it appears in the collection. Occurrences

[14] MySQL open-source DBMS. `http://www.mysql.com`

[15] `http://www.sqlite.org/`

of a query word that is rare in the document collection rank a document higher than occurrences of a very common word. A TF-IDF derived measure called *Okapi BM25* [RWBG95] has been empirically shown to give good performance results. In order to support TF-IDF ranking, the index structure must contain the document frequency and the collection frequency of each term. In spirit of common text retrieval, Dessy supports ranking and uses BM25 ranking by default. The Okapi BM25 function can be written as follows:

$$BM25(Q,D) = \sum_{i=1}^{k} \log \frac{N - n_{q_i} + 0.5}{n_{q_i} + 0.5} \frac{(k_1+1)TF_{D,q_i}}{TF_{D,q_i} + k_1\left((1-b) + b\frac{dl}{avgdl}\right)} \tag{1}$$

Where $Q = q_1, q_2, ..., q_k$ and $D = t_1, t_2, t_3, ..., t_m$ are the query and document (term vectors) for which the rank is calculated, respectively. The variables k_1 and b are predefined constants. I use $k_1 = 2.0$ and $b = 0.75$. These values are commonly used with BM25 when it is not optimized for a specific collection. The variable TF_{D,q_i} is the term frequency of the query term q_i in D, i.e. how many times term q_i appears in the document. The variable n_{q_i} is the number of documents where q_i appears, that is, the size of the result set for the query term q_i, and N is the total number of indexed documents. dl is the length of document D in words, and finally, $avgdl$ is the average document length among indexed documents.

For the purposes of Dessy, BM25 must be extended. Dessy supports multiple properties, or fields, of information for a document. Ranking these without weighting them would cause loss of information structure. Document keywords and abstracts often contain a condensed description of the document, and as such should have higher relevance than body text. I therefore use the BM25 extension for multiple weighted fields [RZT04]. In the extension, a document consists of all the text in its fields, weighted by the field weight:

$$D' = \sum_{i=1}^{j} w_f \cdot f_{i,D}, \tag{2}$$

where w_f is the weight for field f and $f_{i,D}$ are the values of field i for D, vectors of terms. In effect, the extension modifies D by duplicating each term for each field f w_f times.

Document statistics are interpreted as per the modified document, e.g., $TF_{D',q_i} = w_f \cdot TF_{D,q_i}$ and document length is adjusted as if the terms of f appeared w_f times. In other words, $dl(D') = \sum_{t_i} TF_{D',t_i}$ where t_i are the terms in D.

I use the BM25 ranking method because TREC evaluations have shown the probabilistic relevance model that it is based on to perform well for various text collections [SWR00a, SWR00b]. The extensive empirical testing of the model and BM25 also increases the value of BM25 as a good choice for a ranking function.

2.4 Synchronization

In this thesis, *synchronization* refers to file synchronization, i.e. keeping distributed copies of data up-to-date and reconciling concurrent modifications. Many synchronization systems for mobile devices have been presented [ML05, LLS02, CVS04]. The majority of these deal with the synchronization of personal information, and are often coupled with personal information management (PIM) software, such as Evolution[16] and Microsoft Outlook[17]. The synchronization systems synchronize contact details, such as names and phone numbers, SMS messages and emails, and notes from phone note applications. They transform the data to a general format, such as comma-separated-values (CSV), and provide it to the user's PIM program on their computer. They generally do not handle synchronization of regular files. The SyncML synchronization protocol specification [Syn02], currently supported by many phone manufacturers, is vague enough to accommodate regular file synchronization. However, it has been used mostly for the purposes outlined above. A possible reason why phone synchronization software does not handle regular files is that phones generally do not expose a full file system to the user. Often the only way to access notes is through the phone note application, which refuses to open or create regular text files on the phone's file system. Many phones ship with a simplified viewer that shows the pictures, videos, notes and other content that has been created on the phone, but cannot display or create other content. In order to interact with the file system on the phone, users have to download file browsers from third parties. To copy personal information from the phone to a computer, or vice versa, proprietary synchronization applications usually need to be used, with a vendor-specific cable for connecting the device to a computer. Alternatively, Bluetooth may be used. Most mobile handsets allow browsing the file system with a computer connected over a Bluetooth link. However, the bandwidth limitations of current Bluetooth radios make them a poor alternative compared to an 802.11 [IEE99] wireless link used for networking between computers. While high-end handsets contain 802.11 wireless chips, and allow network connectivity through them, they do not allow data synchronization or remote file system browsing through the wireless link. To synchronize files through the wireless link, third party applications are required. In addition, with PIM synchronizers and by manually browsing the file system, the user needs to notice and deal with synchronization conflicts caused by concurrent modifications. Conflict reconciliation is usually not supported in these systems; the system simply copies data from the mobile device to the computer for safe keeping, replacing previous records.

File synchronization systems geared towards mobile devices, using minimal network resources, have been proposed. In [LLS02], a file system modification for detecting and transmitting user operations, or *operation shipping*, is presented. Instead of transmitting the changed files, or the file differences or *diffs*, the operations that caused the changes are transmitted. The system mandates that a separate entity, a

[16] `http://projects.gnome.org/evolution/`

[17] `http://office.microsoft.com/en-us/outlook/`

surrogate, is present in the network, has near-identical software to the mobile device client, and has strong connectivity to the intended synchronization endpoint. The surrogate then receives the script of operations that the user performs on her device, and replicates them. After replication, the files are compared for equality, and error correction may be used if differences remain. The surrogate then synchronizes the resulting file over the strong link to the server. Naturally, the surrogate requirement may be an obstacle for many users. Users who synchronize with multiple servers may need multiple surrogates if the servers are far apart. Also, in order to script actions in graphical programs, some kind of scripting extension is required. For programs that use the command line, an extension to the command interpreter will suffice. Furthermore, in some cases the transmission of user actions is inferior to transmitting a diff of the file, and in some cases the user actions have unreplicable side effects. For these cases, a diff transmission is superior. The authors introduce ways to identify the cases where diff is superior, and use operation shipping and diffs in tandem to achieve optimal performance.

In order to overcome application dependence of the system described above, the Mimic [CVS04] system monitors a user's files and actions. The user's mouse movement and keyboard activity is monitored on top of the window manager. The script of mouse and keyboard activity is then transmitted to the server, and executed there. This enables application-independent scripting, as long as the window managers are compatible and the user launches programs the same way on both systems. Keyboard settings must also be identical. Like the previous system, Mimic allows fallback to sending diffs when operation scripts cannot be used. The greatest obstacle to using Mimic and other operation shipping systems is the requirement that both the client and the server must have a near-identical execution environment. Since owners of mobile devices such as PDAs and mobile phones typically synchronize with desktop computers that have little in common with the mobile clients, operation shipping is rarely the best option.

For synchronization of files, a well-known system is Unison[18] [Bol05]. It is a cross-platform synchronization tool that allows users to specify file merging methods based on file names. The custom merging methods allow automatic reconciliation of changes when possible, and make the tool more flexible for a variety of file types. For example, one could use 3DM [Lin04] for merging of XML files, diff for merging text files, and binary diff programs for other data. When files are changed, Unison sends the diffs of changed files instead of complete files to reduce network usage. It maintains the last synchronized state on both endpoints, so that deletes, moves and updates can be detected and propagated to the other endpoint. Unison runs on Unix/Linux and Windows computers.

For synchronization of file data and metadata, Dessy uses the Syxaw XML-aware file synchronizer. Syxaw can synchronize file data and metadata separately. It maintains a link to a file's last synchronization endpoint, and handles synchronization conflict resolution. Syxaw uses 3DM for merging XML files. Dessy uses Syxaw for unique

[18] `http://www.cis.upenn.edu/~bcpierce/unison/`

file identifiers and synchronization of files and indexes. The Syxaw file synchronizer is introduced in Section 4.1.

2.5 Mobile Java

Dessy differs from most desktop search software in being mobile. Dessy can be used on desktop computers, MIDP 2.0[19] / CLDC 1.1[20] smartphones, and Java Foundation Profile PDAs. Devices that fulfill these requirements include all laptop and desktop computers that can run Java Standard Edition, most Nokia smartphones such as the 6120 Classic, E51, E61, and E71, Motorola smartphones such as the A780, E1, K1, L6, and the Motorazr series, Samsung, Siemens and SonyEricsson smartphones, the Blackberry phones, such as the 6710, 8900, and 9100, and many others.

While the MIDP 2.0 / CLDC 1.1 Mobile Java platform is very widespread among mobile handsets, in terms of desktop search, the platform is very limited. MIDP, the Mobile Information Device Profile, is a specification for Java APIs on mobile handsets. It relies on the Connected Limited Device Configuration (CLDC) specification for low-level functionality support, such as basic input/output and Java basic types. Neither the MIDP 2.0 nor the CLDC 1.1 allow accessing files in any way. The JSR 75 PDA optional packages for J2ME provide a very basic method for file system access[21], presenting a file in a fashion similar to a socket connection, with streams for reading and writing. The JSR 75 File API is available on most of the MIDP 2.0 handsets. The JSR 75 API is used in Dessy. For portability of Dessy, I have implemented a Standard Java File work-alike class that relies on the JSR 75 FileConnection for low-level functions. Also, the set of data structures and convenience classes available in MIDP 2.0 / CLDC 1.1 is almost non-existent. For this reason, I have included classes from public domain Java implementations, such as the Apache Harmony project[22]. I also created a workflow that allows the use of Java 1.5 features in MIDP, much like the J2ME Polish system [Vir05]. I was compelled to create my own workflow, since J2MEPolish proved insufficient for my purposes (See Section 5.3). With these additions, the changes in Dessy core classes between the MIDP 2.0 / CLDC 1.1 version and the desktop computer version are mostly cosmetic.

2.6 Desktop search and mobility

For a user on the move, synchronizing data and having the most recent files available regardless of location or the device used is important. Dessy supports synchronization of search results via the Syxaw file synchronizer with XML–awareness [TKL$^+$06]. Dessy allows searching of files residing on a remote computer using a phone or an-

[19] `http://jcp.org/aboutJava/communityprocess/final/jsr118/index.html`

[20] `http://jcp.org/aboutJava/communityprocess/final/jsr139`

[21] JSR 75, `http://jcp.org/aboutJava/communityprocess/final/jsr075`

[22] `http://harmony.apache.org/`

other computer. It also enables searching for desired content via popular search engines, such as Google.

To provide users with an easy to use search and synchronization system, one needs to study the users and how they use search and synchronization applications. Searching for local files is in principle similar to searching for documents on the Internet. The subject of searching the Web has been increasingly studied recently. For example, in [YMP08], the distribution of Yahoo search queries was analyzed. The study encompassed an SMS interface, a web-browser based one, and a Java search application. The users interacted with the search from their personal phones using one of these interfaces. The number of words per query was 2.35 on average. Aspects such as time taken inputting queries, session length, or number of results examined was not studied. The authors note that the results match earlier studies, such as [KB06]. In the latter, a study of mobile phone and PDA search behaviour was conducted on the Google Mobile search interface. Search logs from one month's time were examined, and statistics extracted. The study suggests that mobile users spend 56–63 seconds inputting a query on a mobile phone, and 27–35 seconds on a PDA. Queries consist of 2.3 and 2.7 words on average for phones and PDAs, respectively. The authors note that the query composition on phones is almost the same as on desktop machines. Users generally searched with 1.6 different queries per session (5–minute span), and rarely looked at results beyond the first page (10 results). 28.7% of consecutive queries in a session were refinements of the previous query, i.e. the first query was a substring of the second, the second a substring of the first, or the edit distance between them less than half the length of the second. The time required to input queries on a mobile phone together with the users' reluctance to scroll and click to the next page makes the first few results important. An application that displays a concise list of ranked results from multiple locations, together with content snippets would probably be appropriate for most users. For example Google displays results with snippets; it shows a few sentences from each result web page, and a thumbnail picture for image results. Dessy allows a simultaneous search of the phone filesystem, the user's remote computers, and the Internet.

2.7 Metadata, tags and context awareness

Most desktop search software only finds documents based on their content, name and filesystem metadata, such as creation and modification date. However, the mobile phone is, first and foremost, mobile, and as such, can be used in very different *contexts.* A common dictionary definition for context is *the facts or circumstances that surround an event.* For desktop search applications, context can identify when, where, and for what purpose a file was created or edited. This makes context an excellent search parameter. Instead of finding a file by its content or name, context–aware search applications can find the user's meeting minutes using his schedule, photos using the location they were taken in, and received files using the names of the senders or nearby devices. A lot of software has been developed for gathering context data on the mobile platform. Dessy could easily be coupled with, e.g., the BeTelGeuse [KLNA09] data gathering software. A photo taking application combined with Dessy and BeTelGeuse could tag new photos with the current context variable values, such as the current location, nearby Bluetooth devices, and calendar events. The photos could later be synchronized to the user's desktop and Flickr or other cloud services using Dessy. BeTelGeuse supports sensors connected to the phone via Bluetooth, such as GPS modules and accelerometers, several internal sensors, such as GSM location information, discovered Bluetooth device names, phone battery level, and activated profile name. BeTelGeuse is extensible, so writing extensions for Dessy and tagging image files is easy. When new image files are synchronized with a desktop machine, context data is propagated on the desktop. This allows search using context data on both the phone and the desktop. While using context data gathering software in conjunction with Dessy is among future work for Dessy, it is out of the scope of this thesis.

The next section discusses the goals of Dessy.

3 Design Goals

While the previous section implied most of the goals that Dessy aims to fulfill, this section lists them clearly. The goals reflect the aims of Dessy to solve the desktop search, remote desktop search, and file synchronization problem on mobile devices, while emphasizing extensibility.

1. Allow finding of files with at least their name, extension, content, and user-assigned tags. The basic requirement for a desktop search system is finding files by the words in their content. Dessy should also add support for user–created and application–dependent, Dessy–specific tags.

2. Index different file types with indexing helpers that are easy to develop. All desktop search engines index files to find them quickly later. Indexing using file–type specific handlers makes the indexing process easier to extend. Developers that wish to add support for a new file type need only write an indexing helper.

3. Handle at least the English language, by stemming words in files and queries. Dealing with the English language is a first step. In the future, Dessy should detect the language of documents and use the appropriate stemming or lemmatization tools.

4. Rank returned results by their relevance using a well–established relevance measure. The number of results returned by short boolean AND queries or long boolean OR queries can be overwhelming. The results must therefore be ranked, showing the most relevant results first.

5. Allow searching of files residing on a remote machine. The storage capacity of smartphones is increasing, but the gap between desktop computer and smart-

phone storage capacity is still large. Therefore, users often carry a subset of their files on smartphones. Searching for files on the desktop machine allows quick finding of specific files as they are required. Coupled with the synchronization goal below, this allows access to all indexed files on the desktop machine.

6. Allow easy adding of tags to files. Text files and word processor documents have easily indexable content. However, other file types, such as image and video files have less text–form content to index. This is why a desktop search engine should support custom tags that are applicable to all files.

7. Provide a simple API for applications and users. To encourage development of applications such as file managers, photo browsers and synchronization front ends, Dessy should have an easy to use API. This API should also be used to add custom tags to files, enabling application–specific tagging of files, such as tagging newly taken photos with the current location and tagging created files, such as meeting minutes, with the user's calendar entries.

8. Allow synchronization of search result files, individual files, and the whole file system. The mobility of Dessy enables users on the move to search for files on their desktop. Synchronizing those files is a logical requirement. Some users may prefer finding files by the file system structure. Users can browse the file system with Dessy and synchronize individial files. For keeping several devices up to date with the remote file system, or a subset of it, the synchronization of entire directory trees is required. These are the synchronization requirements for Dessy.

9. Operate on multiple smartphone platforms. To bring search and synchronization services to as many platforms as possible, the desktop search system should be implemented in a way that allows it to run on the widest range of devices in the smartphone market.

10. Provide an easily extensible framework. A desktop search system is at the core of gathering information about the user's files and providing search facilities in a useful and modern manner. While file types evolve and new search features are designed, the desktop search system must be adapted. Dessy should allow adding file types, searchable file properties, and new search interfaces. Moreover, extending Dessy should be as easy as possible.

11. Secure the information stored within Dessy from third parties. For any application that operates on more than one computer, the security of data transmissions must be considered. Dessy should implement data encryption to protect transmitted data. All queries, search results, synchronization requests, file data and metadata should be protected from malicious users.

The next section describes the design of Dessy, in accordance to the goals above.

4 System Design

This section introduces the design principles of Dessy and details the design of the system. Section 4.1 introduces the Syxaw file synchronizer with XML-awareness [TKL$^+$06] used in Dessy for synchronization and the handling of files. Section 4.2 outlines the architecture of the desktop version of Dessy and the interaction of its main components. Section 4.3 discusses extensions done in 2008–2009 to the original Dessy design. Finally, Section 4.4 discusses security considerations.

4.1 Synchronization Model

For file synchronization, we use the Syxaw file synchronizer [TKLR06]. The model for sharing data in Syxaw is based on establishing *synchronization links* between local and remote *objects*, i.e. files and directory trees. The link is persistently stored along with other object metadata. When synchronizing an object *a* linked to an object *b*, we propagate changes made to these objects since the last point of synchronization. In the case that the objects are directory trees, the full contents, including contained files and directories, are synchronized. Links are unidirectional, meaning that only the device from which the link originates knows about the link. We use the term *client* for the link originator, and *server* for the link target.

From a synchronization point of view, we model data objects as follows. Each object is identified by a unique identifier (UID), which is an opaque string of bits. UIDs are used in the synchronization protocol, rather than file paths. Objects have both a *metadata* and a *data* part. These may be synchronized separately. In particular, for some collection of objects, we may retrieve the metadata before fetching the actual data. Finally, there are objects that enumerate other objects. The directory tree object, which provides a mapping between hierarchical file names and UIDs, is the most prominent example of such an object.

Syxaw synchronization links are illustrated in Figure 1 in the form of a photo archive setup. Let us call the fictive user of the setup Bob. Initially, Bob stored his pictures in a directory subtree (`photos/`) on his PC. He put unsorted images in the `incoming/` directory. When Bob got a camera phone, he created a synchronization link from its `Images` directory, in which the camera phone stores pictures, to the `incoming/` directory. This lets him synchronize new pictures to the PC over the network.

Later Bob got a laptop, and also decided he wanted to store his photos in a more durable location. For this purpose, he rented properly backed up storage space from an Internet Service Provider (ISP), created a synchronization link to the space, and regularly synchronizes the photos to this space, thus ensuring they are backed up. To be able to view and modify the photos on the laptop, he also creates a synchronization link from its `photos/` folder to the PC.

Our concurrency model is optimistic [SS05], which is well suited for the mobile environment [Lin03]. Optimistic concurrency does, however, introduce the need

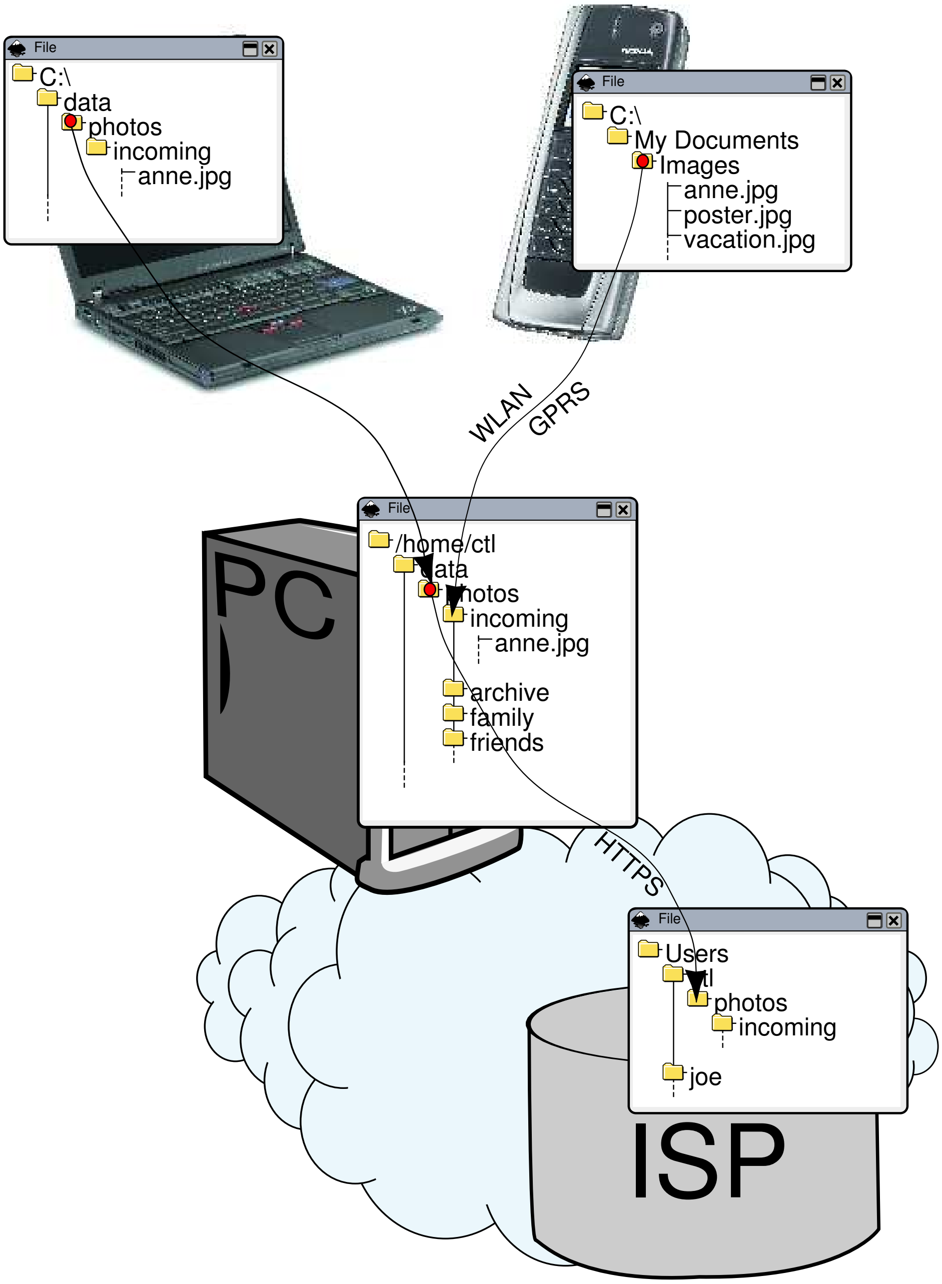

Figure 1: Syxaw synchronization links.

for data reconciliation. In addressing this need, we looked beyond the intrinsic requirements of the file synchronizer, towards a more general goal of providing a generic reconciliation framework for applications aware of the ongoing file sharing. The chosen design provides reconciliation services for XML data by utilizing the XML three-way merging algorithm developed by us in [Lin04].

We built the synchronization protocol using HTTP requests initiated by the mobile client. To minimize the impact of the high latency of current cellular data networks, we designed the protocol to make as few requests as possible. In particular, we batch object requests, so that a complete file system is typically synchronized in only two HTTP requests. To support varying pricing and hot-spots, data synchronization consists of two stages of network-intensive discrete runs, where the first stage synchronizes file metadata (i.e. the layout of the directory tree) and the second stage synchronizes file content. We contrast this to systems that impose a continuous, typically lighter, load on the network [PST$^+$97, K$^+$00, MES95].

The Syxaw file synchronizer supports synchronizing individual files, sets of files, and whole file systems. To synchronize search results, Dessy provides the file identifiers to Syxaw as a set of files, and they can be synchronized. This functionality fulfills design goal 8, since it allows synchronization of search result files, individual files, and the whole file system.

4.2 System Architecture

Dessy was originally designed to bring a synchronization-aware desktop search mechanism to mobile devices. The design was prototyped in 2007 and results were presented in *Dessy: towards flexible mobile search* [LLT07]. The Dessy desktop search system was designed in Java, and has been written to be portable to J2ME for use on smartphones. The structure of Dessy follows a modular design, as shown in Figure 2. File indexing, querying, metadata storage and synchronization are clearly separated. The Figure is colour–coded. Yellow denotes components of the query engine. These translate the user's queries to retrieval requests to the index and synchronization requests to Syxaw. Green denotes the indexing engine. It retrieves document identifiers from the index implementation based on keywords. The indexing engine also contains the interfaces for local file, remote computer, and Internet metadata sources. These allow searching for local files, files residing on remote systems, and files on the Internet, respectively. The red colour indicates the Syxaw file synchronizer. Syxaw handles the download and synchronization of files and the reconciliation of synchronization conflicts. Syxaw also maintains file identifiers and associates files with Dessy metadata.

In Dessy, every query is a path of *physical* and *virtual* directories. A physical directory exists on the local or remote file system managed by Dessy. If a physical directory path is given to Dessy, the query results will be the contents of the directory identified by the path. A *virtual directory* is a file property, or the value of such a property. Virtual directories do not exist in the host file sys-

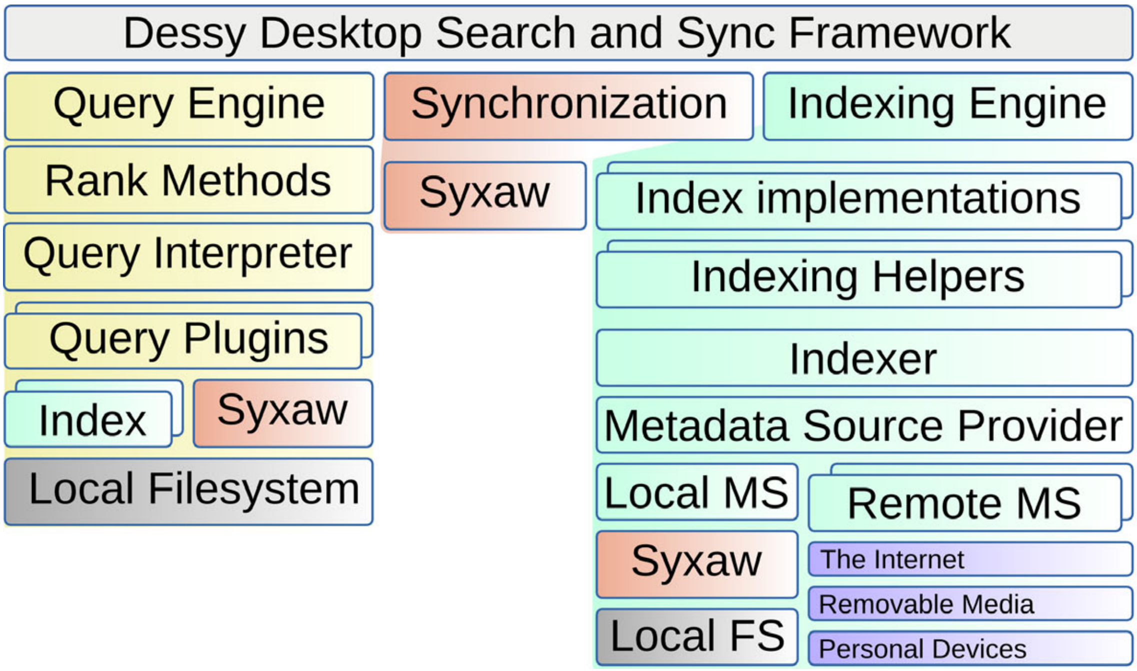

Figure 2: A structural breakdown of Dessy.

tem. A virtual directory path expresses a query, as a sequence of property — value pairs. There are two kinds of virtual directories: *property virtual directories* and *value virtual directories*. Property virtual directories identify a property, such as `author:`, `tag:`, `size:`, `mod-date:`, `name:` or `text:`. Value virtual directories follow a property virtual directory on the path, and specify the value of the property. For example, the virtual directory `author:/bob` states that the value of `author:` is `bob`. Virtual directory queries act upon the files contained in their parent physical directory. For example, the path `Pictures/tag:/laura` is a query for all the files in the physical directory `Pictures` that have `laura` as the value of the `tag:` property. Similarly, the path `Documents/text:/dessy/tag:/work/or:/tag:/important` queries for files in `Documents` with the word `dessy` in them, tagged `work` or `important`. The virtual directory `or:` is an *operator*. Operators change the way that virtual directory paths are interpreted. The `or:` operator aggregates the results from the partial path after it with the path before it. If successive `or:` operators are used, the depth of the *Boolean OR* increases. For example, `text:/dessy/tag:/work/or:/or:/tag:/ important` queries for files with both the word `dessy` and the tag `work`, aggregated with those tagged `important` (without requiring the word `dessy` in the file). This query path is therefore equivalent to `tag:/important/or:/text:/dessy/tag:/work`.

Querying for files is handled by the query engine. The query engine manages browsing virtual directories and requests file identifiers from the index. The same preprocessing that was done at indexing time for file content is performed on the query terms. As every path is treated as a query in Dessy, it is natural to handle browsing of both physical and virtual directories with the query engine. The query engine can be extended via query plugins that translate query aliases, or handle queries that have a special meaning. For example, the query alias *today* when specified as the value of *mod-date:*, refers to the current date, while a query that ends in *sync:* results in the found files being synchronized with their original source location. Regular directory reads are passed to Syxaw and finally to the underlying file system. Property virtual directory reads are treated as queries for possible values of that property. They are passed to the current index implementation, and found values are returned. Value virtual directory reads are treated as queries for files that have the property — value pair. Operators, such as `or:` and `and:` take effect in the query engine, and affect combining the component paths that they connect. For example, the virtual directory path `tag:/important/or:/text:/dessy/tag:/work` returns files which have the `tag:/important` property and also files which have both `text:/dessy` and `tag:/work` properties on them, while `tag:/important/and:/text:/dessy/tag:/work` is equal to `tag:/important/text:/dessy/tag:/ work`, and requires that returned files have all of the specified property values.

After the query results have been determined, they will be ranked with a rank method. Rank methods impose an ordering on the search results. The rank method may be as simple as alphabetical ordering based on the file name, or it may implement a search relevance metric, such as the TF-IDF measure or the Okapi BM25. Dessy uses the Okapi BM25 by default. The Okapi BM25 ranking was specified in Section 2.3. The inclusion of the Okapi BM25 ranking into Dessy completes design goal 4: Rank returned results by their relevance using a well–established relevance measure.

Synchronization and associating files with metadata in Dessy is done by the Syxaw file synchronizer (see Section 4.1). Syxaw associates file data and metadata with a UID that Dessy then uses to identify the file. Synchronization requests are passed to Syxaw in UID form. For remote hosts and the Internet, host names or IP addresses are used as prefixes of the UID to obtain *Globally Unique Identifiers* or GUIDs. These are necessary for multipoint synchronization. Dessy can use Syxaw to synchronize a group of files with their original hosts, regardless of which files come from which host.

The directory tree or filesystem assigned to Dessy is crawled by an indexer. The indexer schedules new and modified files for indexing. Developers can create a custom indexer, which for example ignores a sensitive file type, directories with a certain name, or only indexes the most heavily used files. The indexing of files is done by indexing helpers, which read their supported file types and generate summaries that are then stored into the index. To add support for a new file type, a developer only needs to write an indexing helper for that file type. An indexing helper may handle more than one file type. Dessy identifies files based on their MIME [BF93] type. In some cases, also the file extension is used. For example, files with text (ascii) data are handled by TextHelper, an indexing helper that just reads in the words in the file, paying no attention to the higher-level structure of the file. Dessy indexing helpers assume file content is in English and use the Porter stemmer to increase recall. The architecture of Dessy supports handling other languages as

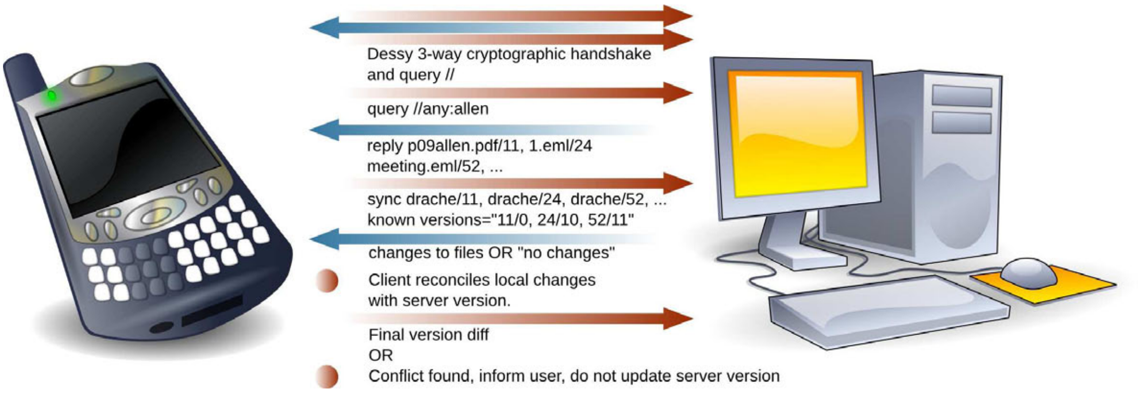
Dessy 3-way cryptographic handshake
and query //
query //any:allen
reply p09allen.pdf/11, 1.eml/24
meeting.eml/52, ...
sync drache/11, drache/24, drache/52, ...
known versions="11/0, 24/10, 52/11"
changes to files OR "no changes"
Client reconciles local changes
with server version.
Final version diff
OR
Conflict found, inform user, do not update server version

Figure 3: A typical Dessy remote search and synchronization sequence.

well. Text files with the `.eml` extension are handled by EmailHelper which looks for email fields such as `to:`, `from:`, and `subject:`, and saves them as separate properties. PdfHelper reads PDF files for metadata and document text, and adds these as properties such as `author:`, `keywords:`, `title:`, and `text:`. The use of indexing helpers satisfies design goal 2: index different file types with indexing helpers that are easy to develop. Furthermore, the design allows handling many languages; currently the system is designed for English. The current design fulfills design goal 3: Handle at least the English language, by stemming words in files and queries.

Dessy can also crawl, index and search non–local filesystems. The *Metadata Source Provider* provides Dessy with current *Metadata Sources* to crawl. The `MetadataSource` interface serves as an abstraction for connected file systems. The local filesystem is just one of these; other examples include a user's Flickr account, Google calendar, and so on. With the rise of cloud computing, developers that wish to bring search to their corner of the cloud can write a `MetadataSource` that provides crawling facilities for their service to Dessy. Dessy can then seamlessly crawl, index, search, and synchronize this information.

Storage of file metadata into the index is handled by index implementations. File content, file names, and custom tags are all handled similarly by a Dessy index. Dessy includes a MySQL–based index and a proof–of–concept flat file database based one. The included Dessy indexes store all keywords as an identifier number to keyword dictionary. They allow quick finding of files with a keyword identifier. The resulting set of file identifiers includes word occurrence counts. A total word count for each file is also included. The number of indexed files, total number of values for each property, and average number of file occurrences for a property value are maintained for the whole file system, to allow quick recall of collection statistics. These are often used in ranking schemes. The design of Dessy does not restrict the index format; index implementations are free to store indexed properties as their authors see fit. The indexing helpers together with the index form the base of a desktop search system, and fulfill design goal 1: allow finding of files with at least their name, extension, content, and user–assigned tags.

Figure 3 illustrates the data flow and sequence of actions in a typical phone search and synchronization use case. First, the user starts the Dessy browser application on the phone. Next, she specifies an address or a host name for the client to connect to, let it be *drache*. The desktop drache is already running the Dessy Remote API and the Synchronization Server. The application connects to the server, makes the cryptographic key exchange, and requests the list of files for the server's root directory. The appliaction shows a view of the files on the Desktop, with a search box. Let us say the user is interested in documents that contain the name `allen` in them. The user types `allen` into the search box and presses search. The browser application sends the search request to the server, and receives the resulting file names and identifiers. If Dessy is configured to connect to Google, the query is also transmitted there. Furthermore, the query can be run locally, if the local file system has files that have been indexed. If the user wishes to examine the metadata of files, such as the thumbnails of images, the title, author, or keywords fields of PDF files, metadata can be requested as well. After deciding on relevant files based on text snippets and metadata, the user chooses the files to synchronize. To trigger the synchronization process, the user selects *synchronize* on the results from the Dessy remote file browser. If the files do not exists locally, the client then creates place–holder files on the phone, links them to the unique identifiers received from the server, and synchronizes them as a batch. The synchronization process works on top of HTTP, and communicates the requested set of files, and their local versions, to the server. If the local files are newer, the phone communicates those files to the server at the server's request later in the exchange. If the server files are newer, the server sends the client changes that have occurred since the last version the client had. If both versions have changes, the server communicates the changes since the last sychronized version to the client. If there are no previously synchronized versions, Dessy will download the whole file from the server. The client then tries to reconcile the changes by comparing them to the last common ancestor version and applying both change trees. If reconciliation succeeds, the client uploads the differences to the server. This is done also if there were local changes but no remote changes. If reconciliation is not possible, the file is left in a conflict state and the user is notified, so she can resolve the conflict manually. After synchronization is done, newly created files will reside in a local copy mirroring the remote file system structure.

4.3 Extensions

While the architecture of the system has not changed significantly from the prototype in 2007, some extensions have been developed. The 2007 prototype did not run on MIDP devices, and it could not search for files on remote computers. To do a remote search, the prototype required downloading the remote index, and doing the search locally. Furthermore, the prototype did not rank results in any order. For a document collection of any realistic size, ranking documents based on their relevance to the user's query is required. Without ranking, the user is required to sift through all the result files to find what she is looking for. The rest of this section will detail the extensions developed.

To enable running the software on modern smartphones, the Dessy desktop search system was ported to the MIDP 2.0 / CLDC 1.1 mobile Java platform. The work done in this regard is discussed in Section 5.3. The ability to run on mobile Java fulfills design goal 9: operate on multiple smartphone platforms.

The ranking system has been added. Like other parts of Dessy, the ranking system is extensible. A rank method implementation is chosen on startup. The rank method is identified by a Java system property. Dessy includes a simple BM25 rank method that implements the BM25 extension to multiple weighted fields [RZT04]. The method scopes the ranking to the queried property, if the user is querying for a single property. It can also rank documents taking into consideration the whole range of properties.

A subset of the Dessy API can now be accessed remotely. The remote Dessy API allows searching for files that reside on a remote computer using a phone, and then synchronizing found files. This is to support the common use case where a file is quickly needed on the phone, but synchronization of all files is either not practical or affordable. The available subset includes browsing virtual and physical directories, downloading files and their metadata, tagging, untagging, and deleting files. Synchronization may be available remotely, if a use case is devised where it is required. The phone may synchronize files found on the remote machine without this capability, since synchronization originates from the mobile device, not from the remote machine. The only utility of remote synchronization invocations would be to synchronize the desktop with another server. Unless the user wants to have a chain of machines designated to synchronize with each other, and then control the synchronization remotely, this is not required. The remote Dessy API, along with the `MetadataSource` interface, fulfill design goal 5: allow searching of files residing on a remote machine. Furthermore, the tagging and untagging operations of the Dessy API are available both remotely and locally. This enables development of applications that tag their files with any tags that their authors deem appropriate. This fulfills design goal 6: allow easy adding of tags to files.

Numerous extensible elements have been mentioned above. Changeable indexing helpers, index storage methods, query plugins, metadata sources, and rank methods make Dessy easily extensible. This fulfills design goal 10: provide an easily extensible framework.

4.4 Security considerations

The security of a traditional desktop search program, which operates within the confines of a single computer system, is usually the responsibility of the operating system. Since the desktop search software simply allows browsing the same files that are available through other access methods on the computer, the operating system that protects those files is the logical choice for protecting the desktop search software. To maintain privacy, a desktop search program should not find files that are considered hidden, and as such not to be found without knowledge of their existence and location.

While the file access rights of modern file systems and access restriction with password protected user accounts are sufficient for protecting a single computer system, when the desktop search program spans more than one computer, information security during transmission and the protection of the server side system from other parties becomes a concern. The desktop search system should protect the server with at least a passworded user account, preferably a passworded private and public key pair, or a more secure method of protection. The data transmitted between the server and client devices should be encrypted, so that nodes between the endpoints cannot eavesdrop on the user's queries or data. The data sent and received by Dessy can be split into blocks, as it is not stream–oriented, so block ciphers can be used. The encryption should be an adequate symmetric encryption method, with a public key – based key exchange, such as Blowfish and RSA key exchange. Use of encryption to protect data in Dessy fulfills design goal 11: secure the information stored within Dessy from third parties. Stronger encryption is a point of future work.

5 Implementation

This section discusses the implementation of Dessy. Section 5.1 discusses the original Dessy prototype, developed in 2007, and the CDC port that was tested on the Nokia 9500 Communicators, and how it differs from the desktop version. Section 5.2 shows an overview of the current implementation and discusses it in detail. Section 5.3 discusses the problems and solutions used in porting the system to MIDP 2.0 / CLDC 1.1. It also contains an evaluation of custom data structure performance on the MIDP 2.0 / CLDC 1.1 platform. Section 5.4 describes the limitations of the mobile version of Dessy compared to the desktop version. Finally, Section 5.5 describes the security implemented in Dessy.

5.1 Original Prototype

Dessy was developed in 2007 largely as a single developer project by Eemil Lagerspetz. The system relies heavily on the Syxaw file synchronizer, developed by Tancred Lindholm. Both of these use the Fuego middleware, developed in the Fuego Core project at HIIT. All of the software is written in Java for multi-platform support, easy prototyping and improved portability.

The original prototype of Dessy was developed as a synchronization and desktop search framework that would be portable to many platforms. Its purpose was to demonstrate the feasibility of desktop search on limited mobile devices, and enable synchronization of search results and small sets of files as opposed to the usual full device synchronization. Also, Dessy allows synchronizing files in both directions; to

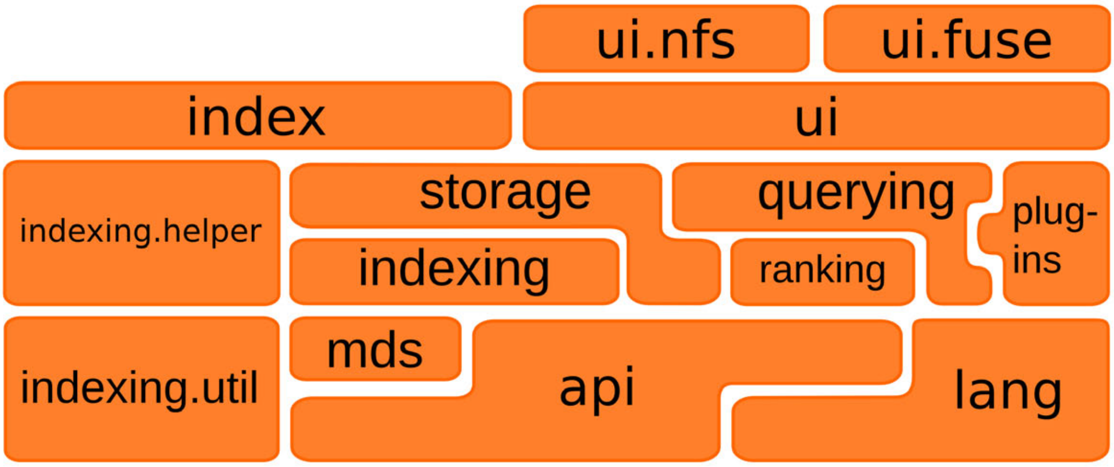

Figure 4: A stack overview of Dessy package structure

the mobile device and to a remote computer, incorporating remote changes to the mobile device as well as the other way around. The whole of the Dessy design was implemented and tested on GNU/Linux desktops. Fuego middleware and Syxaw were available for CDC, and so the system was also ported to CDC and tested on Nokia 9500 Communicators, with results shown in [LLT07]. In summary, the prototype allows a CDC phone to synchronize to a desktop, but it may not be used as a synchronization repository. The CDC version supports only the flat file database indexes, and index size is approximately 10% of text data in the corpus indexed. The prototype required indexes to be on the phone before searching could be carried out. The data of files was not necessary, so the phone could already synchronize subsets of desktop files by first synchronizing the index. Search performance on the phone was acceptable, with 3 seconds as the highest time for finding files, using 10 query terms, and the average search time well below one second. Indexing speed on the device was poor. Hence the additions to the design; in the new mobile version the desktop index can be used remotely for searching, with no need to download metadata or index on the mobile device.

5.2 Overview

This section discusses the implementation of Dessy in detail. The implementations of the phone and desktop version differ only slightly. I will discuss the desktop version here, as its feature set is a superset of the phone version. The implementation architecture is the same; see Section 5.4 for differences between the versions.

Figure 4 shows the package structure inside Dessy. The Dessy package prefix `fc.dessy` has been removed from the names in the picture, in the interest of saving space. The structure is shown as a stack, where elements require those below them in the stack. For example, the `api` package needs the `lang` package, and the `storage` package requires the `indexing` and `api` packages. One of the cornerstones of Dessy is the `api` package. It interfaces with the Syxaw file synchronizer, and the file system. It provides a unified interface to `querying`, `storage` and `index` classes. Classes in the `api` package include DessyApi, the main interface to Dessy functionality, the RemoteApi, a server that receives commands from a remote Dessy instance, IndexKeeper that manages the Dessy index and the indexing process, and the VirtualDir class, which identifies directories as value virtual, property virtual, and regular directories. The `api` package also includes the `SyncMonitor` class, which programs can implement to be notified of synchronization progress. The `api` package is the entry point for software developers that wish to use Dessy.

For developers wishing to extend Dessy to handle more file types, the `indexing.helper` package contains the required interfaces and base classes. Indexing helpers implement `IndexingHelper` interface and are created in the `HelperRegistry` class. The `BasicHelper` serves as an indexing helper skeleton, ready for filling in the file–type specific parts. To detect file types, the `indexing.util` package contains the `MimeTypes` class that detects file types based on their content. The class takes advantage of the *jmimemagic* open-source file type detection library. For the language processing required in reading text files, the `lang` package provides useful utilities. It contains a version of the Porter stemmer for English documents, the `TextLib` class that contains a list of stopwords and allows removal of nonprintable and unwanted characters and expansion of ligature marks in PDF files to their component characters. Finally, the `DateLib` class can be used for formatting dates and supporting the `today` and `yesterday` modification date query aliases. To index sources other than the local file system, a developer can implement the `MetadataSource` interface in the `mds` package. `MetadataSource` implementations are in effect iterators for remote file systems. A `MetadataSource` provides `StoredObjects` to the indexing process, in any order. It is expected to call `indexObject` on `IndexKeeper` in the `api` package for all indexable documents that it provides, when the `crawl` method of the `MetadataSource` is called. The `MetadataSourceProvider` class in the `mds` package keeps track of `MetadataSources` to index. The facilities provided by the `mds` package make the job of the `indexing` package easy. The `CrawlingThread` class calls `MetadataSourceProvider` and crawls through the files to index on all the `MetadataSources`. The `Indexer` implementation only needs to keep track of indexed files, and delegate the indexing to the active `Index` implementation, which records the properties and keywords of the file. The `indexing` package also defines a skeleton for quick `Indexer` development, the `Notifiable` interface implemented by classes that initiate indexing and wish to be notified of its progress, and the `Keywords` class, a container that holds the Dessy file keywords when files are synchronized with Syxaw over the network. The `index` package contains the Dessy `Index` implementations. They store the properties gathered from documents by the `IndexingHelpers`. To support ranking, they must also provide several statistics on the indexed document collection. These include e.g. the number of different values for a property in the collection, the number of occurrences of a property value in a file and in the collection, the file length in property values (words), collection size, and average document length. Dessy includes two Index implementations: MysqlIndex, which uses a MySQL database for storing and quick retrieval of file properties, and IntDbIndex which stores file properties in a file, using the Sdbm flat file database class. The Index implementations make use of the storage package, which allows lower–level interfacing with the Syxaw file storage system, provides convenience classes such as the Dictionary class that represents the word dictionary of an index, the WordCountSet class that implements an ordered counted set, good for efficient storage of word identifiers, and utilities for efficient serialization of Strings and numbers, usable on both mobile devices and desktop machines.

To query the Index implementation for documents matching a set of properties, the querying classes are used. The Query class serves as an interface for all queries, both those to list files found by a regular directory path, and those listing search results in a virtual directory path. It also handles synchronization requests for file data and metadata. Other classes in the querying package include QueryPlugins, a registry class for querying extensions, and the SyncThread class, which handles monitoring and execution of synchronization requests for virtual directories. To handle special virtual directories and translate query strings, a developer may implement the QueryPlugin interface in the querying.plugins package. The interface defines only two methods. The first is String handleDynamic(String), which handles a dynamic query, translating the query string to another. This may be used for creating shorthands to complicated property values, such as today for finding files modified on the current date, and lastWeek for finding files modified during the previous week. The other method defined by QueryPlugin is String[] getSpecials(SyxawFile). This method allows appending *special* virtual directories into another directory. It can be used to make the shorthands visible to the user, by showing today, yesterday, and lastWeek in the value listing for the property mod-date:, instead of dates in numeric format, such as 2009-05-11. Another package closely related to querying is the ranking package. It defines the RankMethod interface, which extends the standard Comparable interface. The active RankMethod is defined by a Java system property, fc.dessy.rankmethod, and is used as a Comparator in Query for ranking result uids. The RankMethod interface defines two additional methods: void setQueryWords(String[]) and void setQueryWords(LinkedList<List<Object>>). Both methods are used for informing the RankMethod implementation of its current context, that is, the set of property values used for ranking the documents. The second method allows the Query class to initialize the RankMethod with a list of boolean or: query paths, where each path is a list in itself, with the first element being a SyxawFile defining the physical path, followed by property — value pairs such as text:allen and from:william. Dessy includes the Bm25 RankMethod implementation. It implements the Bm25 ranking scheme, as described in Section 2.3. For the Dessy NFS and FUSE interfaces, the packages ui, ui.nfs and ui.fuse act as a bridge between the Dessy interfaces and the NFS and FUSE filesystem interfaces. Dessy includes an NFS server that shows the virtual directories of Dessy and allows browsing files and their metadata, and issuing synchronization commands for directories. The Dessy FUSE module allows the same operations. The browsing of virtual directories is implemented via the `readdir()` call. Both the NFS and FUSE interfaces allow readdirs in nonexistent directories, provided that they are virtual directories. They allow browsing of property values in property virtual directories, and results in value virtual directories, as well as browsing the regular directory tree. Finally, the metadata browsing works via the special `metadata:` virtual directory, which shows the metadata of documents as files.

The dependency patterns of Dessy packages could not be captured completely in the picture, however. For example, the picture does not show that `indexing.helper` requires the `querying.plugins` and `lang` packages, and most of the packages that require `api` are also required by it.

5.3 Porting to J2ME

While the original mobile version of Dessy targeted the CDC Foundation Profile Java mobile platform, since then Nokia and other smartphone manufacturers have dropped the device class that supports Foundation Profile Java. The Foundation Profile Java version would now work only on old Nokia Communicator phones and PDAs that support Foundation Profile. Therefore, to make Dessy function on smartphones and other MIDP 2.0 / CLDC 1.1 platforms, porting work had to be done. A publicly available framework for porting Java 1.4 and 1.5 code to MIDP 2.0 / CLDC 1.1, J2MEPolish, already exists [Vir05], but it proved insufficient for my purposes. The framework does not provide the Java Collections hierarchy, and lacks the ordered sets and maps that are crucial for fast indexing. Also, I failed trying to run the J2MEPolish Java 1.5 data structures example. The preverifier was unable to verify the code that dealt with Java 1.5 enums converted to MIDP 2.0 / CLDC 1.1. Therefore, I ported the system to MIDP 2.0 / CLDC 1.1 myself.

The original mobile (CDC) version used Java generics and other Java 1.5 features, with the help of Retroweaver, and the extensive set of classes that Foundation Profile devices provide. Therefore, I was faced with a choice: remove all Java 1.5 features and either design my own data structures or obtain implementations, or search for alternate solutions.

After some trial and error, I was able to combine using Retroweaver and the preverifier from Sun's Wireless Toolkit (WTK) 2.5.2 into a successful build process that allowed using some Java 1.5 features. Due to the nature of implementation of the Retroweaver runtime classes, using the `enum` structure was out of the question, but other features, such as expanded for loops, automatic object-wrapping of primitive types, and java generic types worked well.

Then there was the issue of data structures. I did not want to depend on Sun's code, since its licensing differs from the GPL. I therefore took a subset of the Apache Harmony classes, and started porting them to Retroweaver-powered MIDP 2.0 / CLDC 1.1. An issue worth mentioning here is that the Java Collections API uses the interfaces `Comparable`, `Iterable` and `Iterator` quite a lot, but MIDP 2.0 / CLDC 1.1 is missing these completely. I will discuss these next.

For `Comparable`, this means that a generic ordered data structure that accepts `Comparable` as its data type would not work with primitive types on MIDP. This is unacceptable, since we very often want to use our data structures with primitive types. My solution to this is to make the sorted data structures accept Objects, and check their comparability with another method that looks for `Comparables` and primitive types separately. This solves the problem of using primitive types with sorted data structures, as long as the programmer remembers to implement `Comparable` in custom sortable data types.

The `Iterable` issue is a bit more difficult to resolve. The J2SE expanded for loops are expanded for `Iterables` and array types only. Here, `Iterable` means `java.lang.Iterable`, which is not available on MIDP 2.0 / CLDC 1.1. However, Retroweaver changes all expanded for loops into regular loops, referring to its own `Iterable_` class. To have a custom data structure be expanded-for-loop-iterable, it would then have to implement `java.lang.Iterable`. Unfortunately, implementing `java.lang.Iterable` means adding a method that returns `java.util.Iterator`, which we do not have on MIDP and which Retroweaver does not provide. Adding our own `java.util.Iterator` and `java.lang.Iterable` is not possible, since adding to the core packages is not allowed. I chose to add a placeholder class for `java.util.Iterator` and `java.lang.Iterable`, and let Retroweaver convert the expanded for loops using its own `Iterable_`. This means classes that support expanded for loops must have the `iterator()` method that returns the `java.lang.Iterable`. So, for returning a normal, properly functioning (MIDP) `Iterator`, where it is desired, a different method name must be used. I chose `iteratorM()` (for `Iterator` MIDP). Expanded for-loops can be used for array types without modifications.

The problem with `Iterator` is similar to the problem with `Iterable`. Since MIDP 2.0 / CLDC 1.1 does not have Iterator at all, the data structures from Apache Harmony and other custom data structures would have to implement a different `Iterator` interface. I created `fc.util.midp.ds.Iterator` for this purpose, and had them implement it. The use of `Iterator` in Dessy and the whole of the Fuego Core middleware is common, so it was easier to have them than do without.

I made a test application, CollectionsTestMidlet, to see how the Harmony data structures worked on MIDP 2.0 / CLDC 1.1. The test application also showcases expanded for loops and autoboxing. The tested data structures were `SortedSet` and `TreeSet` from Apache Harmony. The test application runs on a MIDP 2.0 / CLDC 1.1 phone.

In addition to testing the functionality on a phone, I measured the performance of a `TreeSet` of `Longs`. I present the performance evaluation for this here, as good performance of this sorted data structure is required for indexing as well as local searching on the mobile phone. This evaluation must precede evaluation of Dessy as a whole, so that the data structures used with Dessy can be proven to have sufficient performance to make the system viable on the MIDP 2.0 / CLDC 1.1 platform.

Treeset performance evaluation

In an inverted index, words and files are generally handled as numbers, preferably longs to allow a large value space. The results of a query are found by first looking up the word identifier for the query word in a word dictionary, and then looking up the set of file identifiers for that word. To efficiently store sets of numbers, they must first be sorted so that *counted sets* can be used. In a counted set, numbers are stored as increments from the previous number; for example, the set $1, 2, 20, 22, 26$ can be stored as $1, 1, 18, 2, 4$. In this compact representation, the closer numbers are, the better they can be compressed. The storage density can be enhanced by storing the required number of bytes only; a number less than 255 larger than the previous number in the set can be stored using a single byte. This motivates the use of sorted sets such as `TreeSet`: they can be used to achieve high storage efficiency with index structures, adding a small overhead for keeping the set sorted when inserting new numbers. The sorted sets are used when indexing and when querying. When indexing, the sorted set is incrementally enlarged as more files match a query during the course of indexing. The set is finally stored as a counted set by storing the first number in the set normally, and then storing the rest of the numbers as increments from the preceding number. When querying, the counted set is read from the index and the increments are added to preceding numbers to create the original sorted set of file identifiers, the set of results to be returned. The performance measurements were gathered from two phones, the Nokia E51 and the Nokia 6120 Classic, and the Sun WTK phone emulator running on a GNU/Linux desktop machine. For comparison, performance of the same `TreeSet` was measured on Java 1.5 running on two computers: a GNU/Linux mini laptop, and the GNU/Linux desktop machine. In the experiment setup, each of the configurations ran the test shown in Algorithm 1 (pseudocode):

Figure 5 shows the results. The measurements are in milliseconds on a logarithmic scale, and represent the time taken for 100 000 operations. The measured operations are listed in the Figure. The test was run 300 times on the five platforms. Table 1 shows the results in a table with standard deviation (σ). The small standard deviation for the `add`, `remove` and `contains` operations for phones (less than 4.2% for `add`, less than 8.1% for `remove` and less than 4.8% for `contains`) suggests that these measurements give a good picture of the expected performance of TreeSets on MIDP 2.0 / CLDC 1.1 using current handsets. The large standard deviation for the better performing operations, `tailSet` and iteration, is probably due to background processes of the phone. The performance gap between desktop machines and phones is very large; the amount of time used by the phones was more than two orders of magnitude larger than that of the desktop machines in the `add`, `remove`, and `contains` operations. However, for `tailSet` and iteration of elements, the difference is only one order of magnitude. An interesting observation is that the WTK emulator seems to be slower than the phones for `tailSet` and iteration. This speaks

```
s ← new SortedSet<Long>
// test addition:
for i ← 1 to 100 000 do
    s.add(i)
end
record the time taken
// test contains:
for i ← 1 to 100 000 do
    s.contains(i)
end
record the time taken
r ← random(100 000)
// test tailSet:
for i ← r to 100 000 do
    s.tailSet(i)
end
for i ← 1 to r do
    s.tailSet(i)
end
record the time taken
// test iteration:
iterate through s
record the time taken
// test removal:
for i ← r to 100 000 do
    s.remove(i)
end
for i ← 1 to r do
    s.remove(i)
end
record the time taken
```

Algorithm 1: Experiment pseudocode.

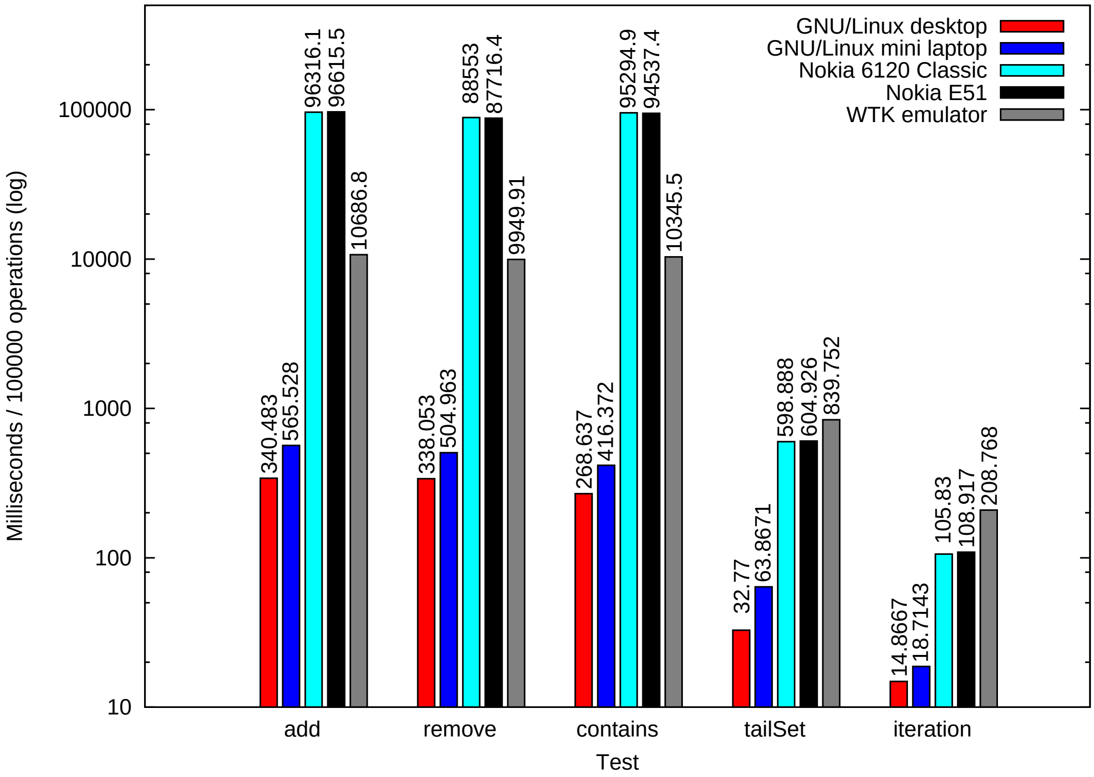

Figure 5: The performance of TreeSet operations on MIDP 2.0 / CLDC 1.1 compared with Java 1.5

Configuration	**add**	σ	**remove**	σ
GNU/Linux desktop	340.48	20.62	338.05	20.65
GNU/Linux mini laptop	565.53	127.60	504.96	127.28
Nokia 6120 Classic	96316.10	2276.87	88553.00	2697.46
Nokia E51	96615.50	4006.49	87716.40	7062.87
WTK emulator	10686.80	672.82	9949.91	639.79
Configuration	**contains**	σ	**tailSet**	σ
GNU/Linux desktop	268.64	12.69	32.77	6.16
GNU/Linux mini laptop	416.37	81.62	63.87	15.35
Nokia 6120 Classic	95294.90	2999.00	598.89	94.28
Nokia E51	94537.40	4465.54	604.93	86.34
WTK emulator	10345.50	673.84	839.75	91.81
Configuration	**iteration**	σ		
GNU/Linux desktop	14.87	2.83		
GNU/Linux mini laptop	18.71	6.05		
Nokia 6120 Classic	105.83	47.26		
Nokia E51	108.92	69.59		
WTK emulator	208.77	22.71		

Table 1: TreeSet performance and standard deviation

in favor of using iterated data structures on phones. Based on these measurements, I conclude that the `add`, `remove`, and `contains` operations are much slower on phones than on desktop computers, and take around 1 ms each to complete. When indexing, a word identifier is added to a TreeSet of word identifiers for the indexed file. This means that roughly O(documentLength) ms of time is spent indexing a file. Indexing a file with 1000 unique words takes at least 1 second. This means that indexing large sets of files in the current indexing framework is not desirable on phones, and downloading an index from a desktop running Dessy, or searching for files remotely is a more attractive alternative.

The 1 ms estimate for an addition also leads me to believe that adding user-defined tags and metadata is fast enough for comfortable use on the phone. Even tagging large sets of files should be possible in reasonable time.

In summary, the work on enabling Java 1.5 features and the Java Collections framework data structures on MIDP 2.0 / CLDC 1.1 devices was largely successful. The missing `enum` and some of the complexity that has to do with the `Iterable` and `Iterator` interfaces could be remedied by refactoring retroweaver. For the purposes of the Fuego Core project, I am not certain this would be worth the effort.

5.4 Mobile Version Limitations

The initial Dessy prototype was written in Java 1.5 and ported to CDC. It was tested on GNU/Linux desktops and laptops, and Nokia 9500 Communicators. The results can be found in [LLT07]. While the system was fast enough in querying on the device, indexing proved slow for bigger datasets. Since then, Dessy has been ported to MIDP 2.0 / CLDC 1.1, and tested on two Nokia mobile phones, the Nokia 6120c and the Nokia E51, and the Sun WTK phone emulator. The prototype shows a directory view of a remote desktop. It can successfully synchronize chosen files and folders with a server running the Dessy Remote API and the Synchronization Server. The prototype needs no prior knowledge of the server file system structure. In addition to physical directories, the prototype is able to browse virtual directories, that is, search results of user queries. These are synchronized the same way as regular files.

As the MIDP 2.0 / CLDC 1.1 platform does not provide certain data structures, and access to certain system services, the mobile version of Dessy has a few limitations. For instance, the Synchronization Server cannot be run on the mobile version, because it uses Servlets which are unavailable on MIDP 2.0 / CLDC 1.1. This means that the phone running Dessy may not be linked to as a synchronization repository. This limitation may be lifted by reimplementing the server without the use of Servlets. However, The phone is fully able to synchronize with a desktop running a Synchronization Server. Therefore, this limitation does not remove crucial functionality.

In the current version, some Dessy indexing helpers and Metadata source providers are unavailable. For example, the PDF indexing helper cannot be used on the phone,since the PDFBox PDF file reading library uses AWT classes extensively. This means that indexing of PDF files on the phone is not possible. However, the user may index PDF files on a desktop and then synchronize the files, receiving the same information indirectly on the phone. Also, the Flickr metadata source provider is not available, since it uses the Java 1.5 Collections and java.util.Iterator directly. Refactoring would be necessary to support synchronization with Flickr on the phone. It is, however, possible to synchronize Flickr to a desktop, and then synchronize those photos on the phone.

5.5 Implemented Security

As dicussed in Section 4.4, a system spanning multiple devices must protects its traffic from eavesdropping and the devices from intrusion. In the current Dessy prototype, the Dessy server is not protected by a password; an attacker with knowledge of the Dessy protocol and network access to the machine where the Dessy server is running can connect to it and search or browse the user's files. This could be remedied with a simple passworded user account. This is left as future work. The current security measures have been implemented as a simple server class that provides access to the Dessy API. Adding a password check to this framework should be simple. It must be stated that the lack of password protection for the Dessy server leaves the security of Dessy incomplete, and fails to fulfill design goal 11: secure the information stored within Dessy from third parties. The implementation of Dessy falls short of the design in this respect.

While server access is not restricted, the Dessy prototype server and client generate public and private keys, and use RSA key exchange to agree on a symmetric key for the encryption of traffic between them. The RSA key used is randomly generated, of 512 bytes in strength. The symmetric key is then used for the duration of the session. New symmetric keys are generated for each new session. The prototype uses the BouncyCastle cryptography suite for encryption on both the client and server. For data transmission, Dessy uses the Blowfish engine. The key used is a randomly generated sequence of 32 bytes. The cryptography strength and the used engines can be changed in the `PkEncryption` and `SymmetricEncryption` Dessy classes.

6 Evaluation

This section presents the evaluation of Dessy. Section 6.1 presents the evaluation methodology, and Section 6.2 details the experiment setups and results of the experiments.

The evaluation of a desktop search system is a complex task, as such a system contains many measurable quantities, and it is not clear which of them should be prioritized. Examples of measurable quantities include retrieval performance, indexing speed, index size, index compression rate, index synchronization speed, file synchronization speed, query speed on the phone, query speed on the desktop, and user-perceived delays in the user interface.

In addition, there are other concerns that a software program must address, such as the efficiency of the user interface, i.e. how many actions the user must perform in order to complete a task. The learning curve of the system should be evaluated, and it should be easy to adopt for new users. Also, the network costs involved in using the system must be taken into account when operating on the phone network.

The qualities most visible to the user relate to the speed and ease of use of common operations of the system, and the system's ability to find desired documents. Since synchronization and file searching are very user-oriented tasks, I choose to evaluate the system from the user's perspective.

6.1 Methodology

In order to validate Dessy as a capable search engine, its retrieval performance, i.e. ability to retrieve relevant results given a query, must be examined. I measure retrieval performance based on the precision and recall measures. The *precision* of an information retrieval engine, given a query, is simply the ratio of relevant documents from all retrieved documents, i.e.

$$P(Q) = \frac{\mathit{Relevant\ documents} \cap \mathit{Retrieved\ documents}}{\mathit{Retrieved\ documents}} \tag{3}$$

A precision of 1 means that the retrieval engine returns all relevant documents for the query and no others. If the engine always returns a lot of documents, a precision of 1 can also mean that all the relevant documents are returned before other documents. A precision of 0 means that the engine returns only irrelevant documents for the query. The precision measure cannot be used without relevance judgments. A number of experts must judge all the documents in the collection as relevant or irrelevant for a set of queries, and then this information may be used to judge the precision of information retrieval engines, when the queries are executed using those engines.

The *recall* measure is the ratio of relevant documents retrieved from the set of all relevant documents, i.e.

$$R(Q) = \frac{\mathit{Relevant\ documents} \cap \mathit{Retrieved\ documents}}{\mathit{Relevant\ documents}} \tag{4}$$

A recall of 1 therefore means that the query results contain all the relevant documents. It is trivial to achieve a recall of 1 by returning all the documents in the collection; A high recall value, by itself, does not tell much about an information retrieval system. However, it is possible to evaluate precision and recall together. By setting a level of recall, and considering the precision of the retrieval engine at that recall level, we can see how much "chaff" a retrieval engine returns along with the relevant results, and how the chaff increases (and accuracy decreases) when a higher level of recall is required.

To provide an acceptable user experience, the Dessy user interface must give instant feedback on user actions, and common tasks must complete quickly. I assume that most users are comfortable with a wait time of one second between starting a search and receiving results. Therefore, I set this as a goal for Dessy; searches must complete in under one second. The other common actions in Dessy are synchronizing files, adding file tags, deleting files, and synchronizing file metadata. Adding new tags is a fast operation, since it simply adds a single property — value pair to the index for the given file. If the index used with Dessy is reasonably quick in indexing, adding one property — value pair has performance equal to adding a file with one word inside it. This should always complete in under 100 ms on all platforms that Dessy runs on, and have an expected completion time of much less. Deleting files constitutes removing a file from the file system. This need not touch the index at all; the files' indexed information may be removed later, on the background. This makes deleting a fast operation as well. Therefore, the two operations that must be examined closer are synchronization of file data and metadata. In the following, I will first discuss these two operations, and then the search speed.

To measure file data synchronization performance, I record the time taken when synchronizing a file, along with the power and memory usage of Dessy. In the experiments, the system synchronizes files of varying lengths. Files are synchronized one at a time; the performance for synchronizing a group of files will be higher when compared to synchronizing each file separately, since the Syxaw synchronizer supports file group synchronization. During the synchronization experiment, synchronizing index contents (Dessy metadata) for files was disabled. This is to decouple the performance of file data synchronization from metadata synchronization. Also, initial experiments showed that synchronization performance greatly improves on mobile phones when index contents are not synchronized.

For synchronization of file metadata, I did not run extensive experiments. It was observed that the operation is too costly on mobile phones, probably because of the prohibitive cost of file seeks within the flat file database where index words are stored. Adding file metadata to an inverted index constitutes adding one property — value to file identifier entry for each property — value pair in the metadata, and storing the list of words for the file as well. This can result in as many seeks as there are property — value pairs in the metadata, when using a hash–based storage method. Finding a better storage method on mobile phones is future work for Dessy. In the current system, it is actually faster to index files into a flat file database index on the server, and then transmit the index as a file to the client, rather than transmitting property — value pairs for a subset of files.

To achieve the goal of searching in under a second, we measured the time it takes to search for files remotely using a mobile phone as the search client. In the experiment, we generate an average of 2.3 keywords for each search, as observed for mobile Internet searches in [KB06]. I assume that the number of keywords follows the normal distribution. The query distribution mimics real user searching with Dessy. The experiment script executes queries constantly, faster than one each second, and should give reasonable upper limits of power usage in intensive use. I record search speed by timing each search exchange, from the time the request is sent to the server, to the point when the results are ready to display to the user. I also measure memory usage within Java, and power usage during the experiment.

While speed is important, on all mobile devices, concerns of battery consumption have to be addressed. The most power–hungry elements of a mobile phone are the processor and communications hardware. The power usage of the phone when running a piece of software, as compared to the phone idle state, can give intuitive suggestions of the "power–hungriness" of that software. To measure the impact of Dessy on the client device, I measure the power usage of Dessy on the Nokia E51. Power usage measurements are taken for all the experiments introduced before. For comparison, I measured the power usage when the device is idle, with GSM on, and Bluetooth and WLAN off. Finally, I measure power usage with a popular mobile application, Google Maps.

6.2 Results

The evaluation results for Dessy are divided into three main categories. First, I evaluate Dessy as a search engine and obtain precision and recall values for Dessy. These can be used to compare Dessy with other search engines that have been used with the same dataset. After this, I measure the time taken in synchronizing files, and the energy usage of synchronzation. Finally, I measure the time taken when searching for files.

Dessy as a search engine

To validate Dessy as a capable search engine, I recorded precision and recall with the freely available CACM dataset[23] of the SMART system[24]. While the dataset is not very well suited for evaluating desktop search applications because of its short documents and long, natural language queries, it gives an indication of Dessy's performance. The dataset has also been used with commercial desktop search applications, which enables direct comparison between the systems. To obtain the recall and precision results, I split the original `cacm.all` file into separate files, according to their identifier numbers, given by the `.I` lines in the `cacm.all` file. I then created a Dessy indexing helper called CacmHelper, which would index the properties `.T`, `.A`, `.K`, and `.W` as Dessy properties `title:`, `author:`, `keywords:` and `text:`, respectively. The data set was then indexed with this helper. Queries specified in `query.text` were then carried out. Dessy stopwords and those in the CACM dataset file `common_words` were removed from the data when indexing and from

[23] `ftp://ftp.cs.cornell.edu/pub/smart/cacm/`

[24] The SMART system is available from `ftp://ftp.cs.cornell.edu/pub/smart/`.

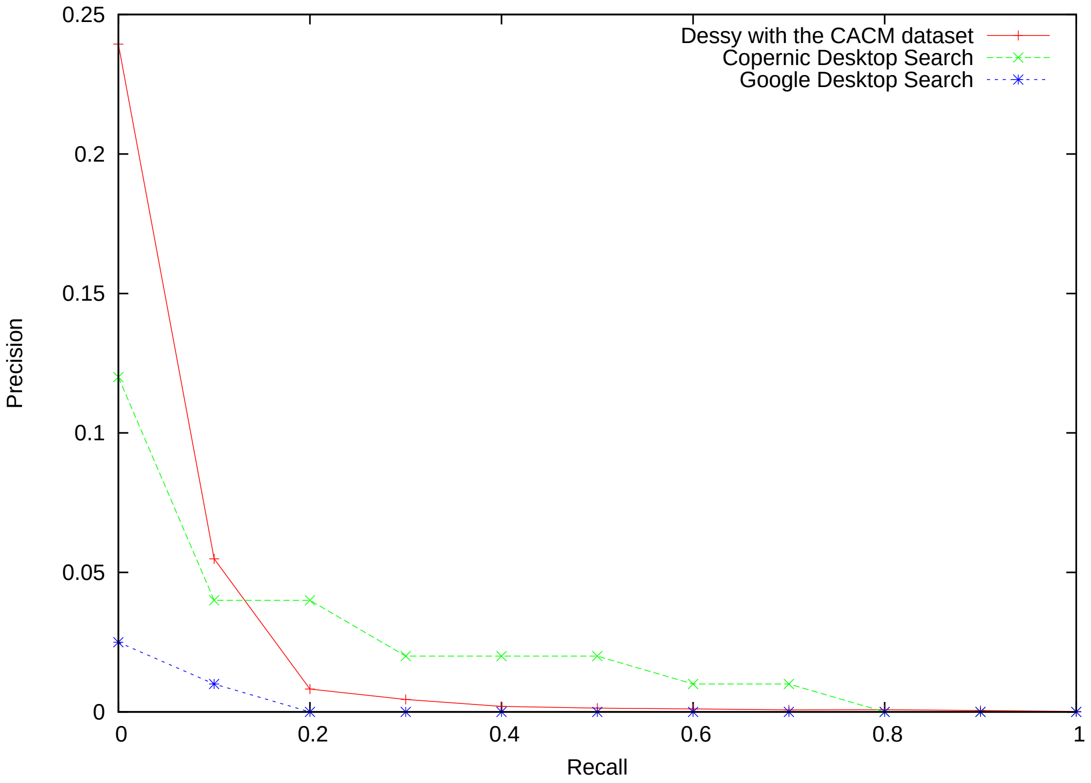

Figure 6: Precision and recall of Dessy and two commercial systems with the CACM dataset.

queries when querying. Figure 6 shows the results, along with those of Copernic Desktop Search and Google Desktop Search.

In Figure 6, the x–axis displays recall, while the y–axis displays precision. As recall increases, it is often the case that precision decreases. According to the results, Dessy falls short of the Copernic Desktop Search commercial desktop search software but bests the results of Google Desktop Search for this dataset. The results of Google Desktop Search and Copernic Desktop Search have been obtained from an extensive desktop search software comparison found online[25]. In the comparison, Windows Desktop Search is consistently lower in precision than Copernic Desktop Search, although the difference is small. Windows Search Companion, Yahoo Desktop Search and Google Desktop Search all fall short of Dessy. Although the initial results are positive, since the dataset is not well suited for desktop search, these findings should not be regarded as conclusive. Further retrieval performance comparison with existing systems should still be conducted.

[25] `http://kalio.info/Desktop_Search_Comparison/`

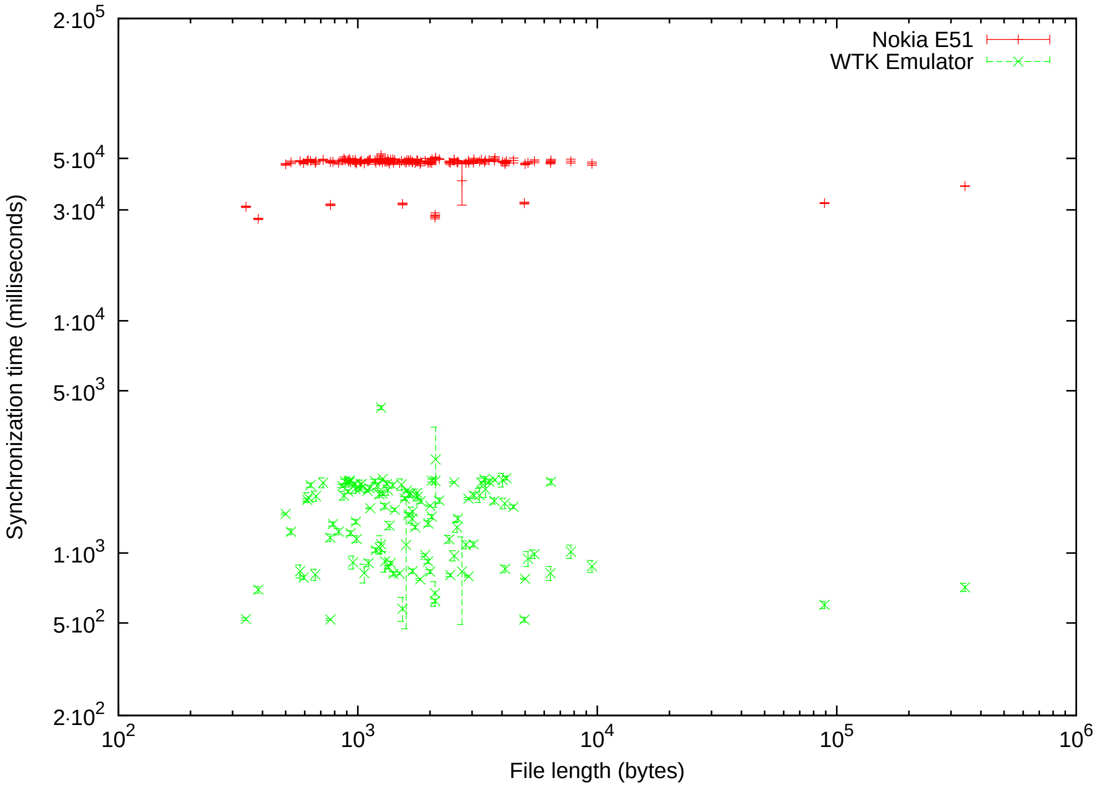

Figure 7: Synchronization time measurements for a small subset of the Enron email dataset.

Synchronization speed

One of the most significant delays associated with search and synchronization is the time the user spends waiting for a file to synchronize. While good user interfaces allow the user to perform other tasks while the system is synchronizing files, the user is conscious of the waiting state, and will prefer quick synchronization. When synchronizing files according to a user-defined schedule, being quick is replaced as a primary goal by being cheap. Synchronizing when a network hot spot is reached may be preferable to synchronizing in a low network connectivity area. It must be noted that the Syxaw file synchronizer supports synchronizing a large batch of files as well as single files. I evaluate the synchronization of single files only, because this is closer to the expected user behaviour. A user will typically search for files from her desktop, identify files of interest, and synchronize them. Typically, a user does not synchronize a big batch of files while on the move, but rather only files of interest. The following evaluations deal with synchronization speed of individual files.

The first results show synchronization times for two datasets. Both datasets were synchronized on the Sun WTK emulator and the Nokia E51, both connected to the synchronization server via an IEEE 802.11 wireless link. Synchronization of the Dessy index was disabled for these experiments. This significantly increases synchronization performance on the Nokia E51, e.g. from 60 seconds to 20 seconds synchronization time. Time differences are larger for files with more index data, such as large text files. As users generally synchronize small sets of files, I expect that synchronizing the index for local searching by default on the phone is not necessary. It is possible to synchronize the index later, or index the files using the Dessy running on the phone. These configurations will be used throughout the experiments, and will be called the Nokia E51 and the emulator.

The first dataset is a small subset of the Enron email dataset, similar to the one used in Dessy measurements in 2007. This subset has been extended with some Dessy source code files, and one email file with an OpenDocument Presentation attached. The files of the dataset are arranged into a directory structure, so that emails of a given email user are under a subdirectory called `inbox` in a directory named after the user's username. These directories are stored in a directory called `maildir` on the root level of the dataset. The added files are in other directories at the root level. Figure 7 shows the synchronization time in milliseconds for each file length, on a logarithmic scale, with standard deviation as error bars. I use a logarithmic scale to accommodate the results from both the WTK emulator and the Nokia E51. The file length is shown on the x–axis, on a logarithmic scale. This is to accommodate the gap between short email files and the email with the presentation. The results suggest that synchronization of small files takes around 50 seconds on the Nokia E51, with occasional variance. It is curious how some files always synchronize in around 30 seconds, even though they are larger than those that synchronize in 50 seconds. This may be caused by Syxaw's scanning of directory trees. The slower to synchronize files may be in deeper subdirectory trees, and thus require more directory nodes to be compared with the server. Most of the files can be synchronized on the emulator in under five seconds. On the emulator, variance is less pronounced, but occurs on the same file lengths.

When the first results did not show synchronization times for files of many different lengths, I created a new data set from random UK English words. I picked the words from the *UK English Wordlist With Frequency Classification*[26] collection, designed for spellcheckers. I created a script that generates files with a specified number of random words. I then created files with 20 to 900 000 random words. To save experimentation time, I then pruned this collection so that it had files of varying lengths, widening the difference in file length as the files got longer. The final collection had 57 files ranging from 188 bytes to 5.2 megabytes. The files were stored directly at the root of the dataset. The dataset contained no subdirectories.

I measured synchronization times on the Nokia E51 and the emulator for this dataset. Figure 8 shows the synchronization times for the Nokia E51 and the emulator. As before, the synchronization times are shown as points with standard deviation around them as error bars. The x–axis is linear, since file length increases linearly in this dataset. The time to synchronize seems to grow linearly on both

[26] `http://http://www.bckelk.ukfsn.org/words/wlist.zip` [downloaded 2009-05-27]

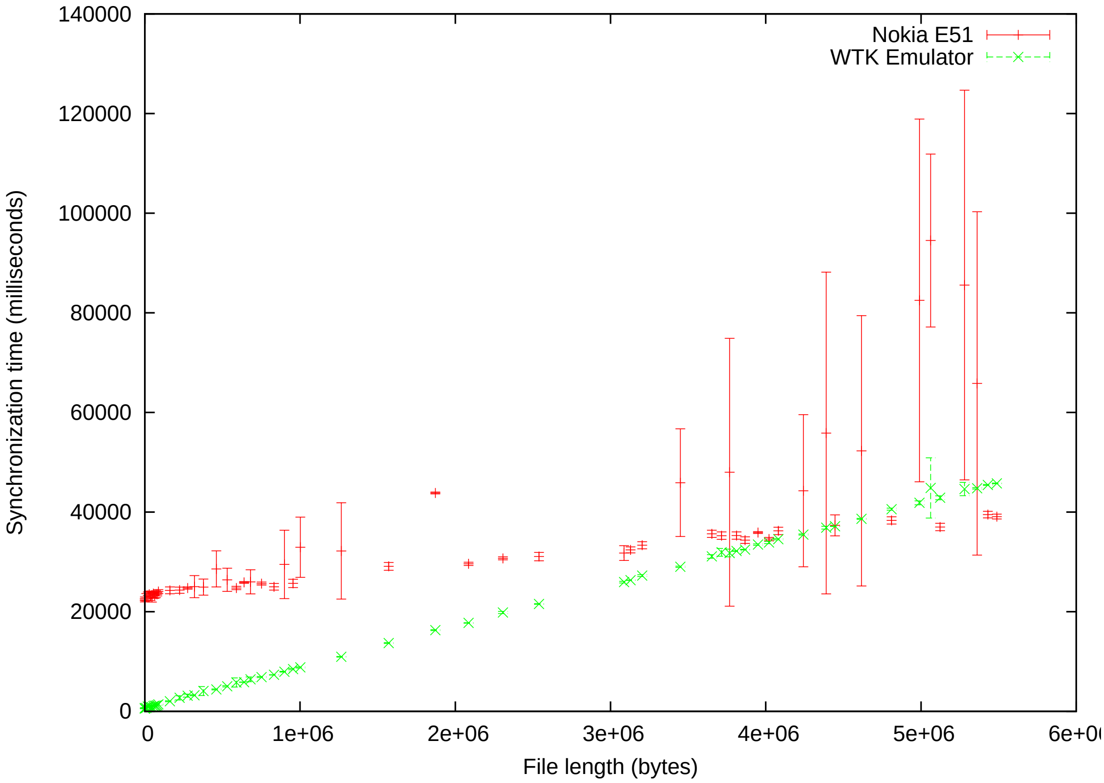

Figure 8: Synchronization time measurements for files from 0.2 KB to 5 MB, containing random English words.

the emulator and the E51. Minimum synchronization time starts off from above 20 seconds on the E51, and around a second on the emulator. Both on the emulator and the E51, average synchronization time grows linearly to around 40 seconds. The linear growth is especially pronounced on the emulator. On the E51, there is large variance for a couple of files 5 MB in size. The variance may be caused by Java garbage collection or phone power saving. The phenomenon does not seem to be frequent; some large files synchronize quickly with almost no variance, while a couple do not. Regardless, it would appear that synchronization of larger files is both feasible and reasonably fast on the Nokia E51. However, the user still needs to wait for the synchronization to complete. This suggests that more work should be done on background synchronization and scheduled synchronization. On the other hand, the E51 is a relatively basic smartphone in today's market. On newer and faster devices, I expect synchronization speed to increase.

Energy usage

During the synchronization experiments, I recorded power usage of the phone, along with total memory usage. I used the *Nokia Energy Profiler*[27] for measuring these values. Since the "Standby mode" of the phone uses power on its own, and the Energy Profiler may also consume power, I ran the tool with no other programs running on the phone to establish a baseline. The results of this initial experiment are shown in Figure 9. The Energy Profiler seems to use more memory as experiment time increases. It may be that it buffers measurements and occasionally saves them to a temporary file. This affects the memory usage measurements of the Energy Profiler. Power usage of the Energy Profiler is low; around 0.05 W with the backlight off, and 0.1 W with the backlight. So, I expect that the backlight uses around 0.05 W of power. I averaged the power usage and memory usage of the Energy Profiler, and use these values for comparison in further power measurement Figures. They are shown in all Figures as *Average Energy Profiler memory usage* and *Average Energy Profiler power usage.*

Figures 10 and 11 show memory usage in megabytes along with power consumption, in watts, while running the synchronization experiments. The x–axis displays time. In Figure 10 the time is in hours, minutes and seconds, while in Figure 11 the time is in minutes and seconds. The y–axis shows the amount of memory used in megabytes, on the left, and the amount of power consumed, in watts, on the right. The memory usage represents the whole memory usage of the phone. In addition to memory being used by Dessy, the Java virtual machine, the Energy Profiler, and background processes running on the phone consume memory. The results suggest that using Dessy on the E51 adds 5 – 10 megabytes to the device's memory consumption. While 5 megabytes is expected, 10 seems high. Later in this section I will compare memory measurements from inside Dessy to those recorded by the Energy Profiler.

[27] `http://www.forum.nokia.com/Tools_Docs_and_Code/Tools/Plug-ins/Enablers/Nokia_Energy_Profiler/`

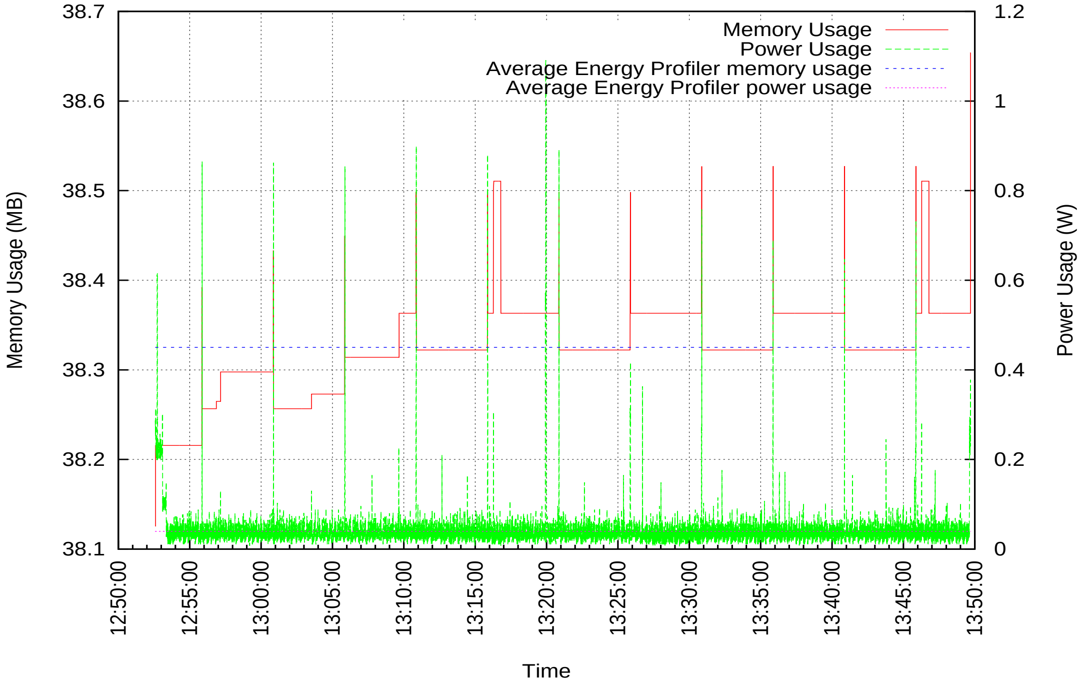

Figure 9: Energy and memory usage of the Energy Profiler on the E51.

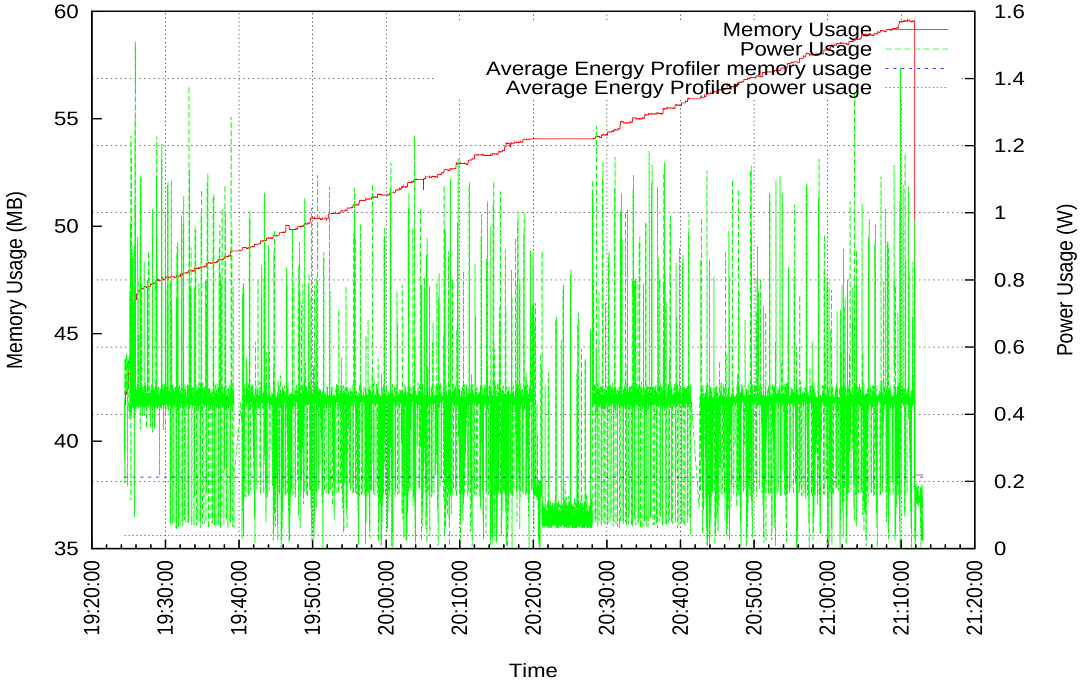

Figure 10: Power and memory usage of Dessy on the E51 with the first dataset.

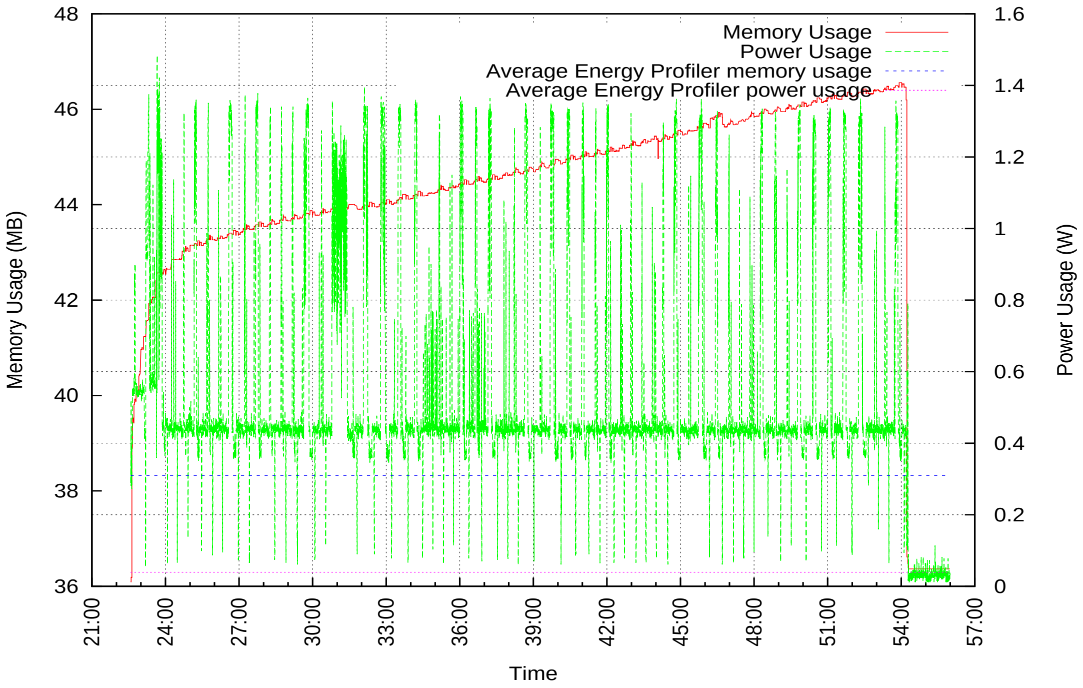

Figure 11: Power and memory usage of Dessy on the E51 with the second dataset.

Power consumption oscillates between 0.4 W and 0.8 W for the most of the test. When Dessy is started, power consumption spikes briefly above 1.2 W. The power consumption of the phone in the idle state is around 0.2 W. While Dessy uses a lot more power than an idle phone, this is to be expected. In the scenarios, Dessy is constantly synchronizing files. It is actively using the processor, the communication hardware, and the disk. When Dessy is on standby, ready to execute user queries, or displaying results for the user, its power usage is low. I therefore note that the power usage seems to be within acceptable limits. See for comparison the power usage of Google Maps on the E51, in Figure 12, when the user is actively scrolling and zooming the map, so that the program needs to download new maps almost constantly. The figure shows time in minutes and seconds. The power usage of Google Maps seems to stay above 0.6 W while downloading, and memory usage, as recorded by Energy Profiler, seems similar to that of Dessy.

To check that the memory usage measurements are correct, I modified the experiment to run the Java Garbage Collector (`System.gc()`) after each sychronization, and recorded the total memory and free memory as given by `Runtime.totalMemory()` and `Runtime.freeMemory()`. I then subtracted the free memory from total memory, to obtain the amount of memory used by Dessy. I graphed the total and used memory values over a course of five experimentation runs with Dessy, on the enron dataset subset. The results are in Figure 13. The x–axis is simply a running number. The y–axis shows the amount of memory used, in megabytes. It seems that Dessy uses at most 3.5 megabytes for each run, while

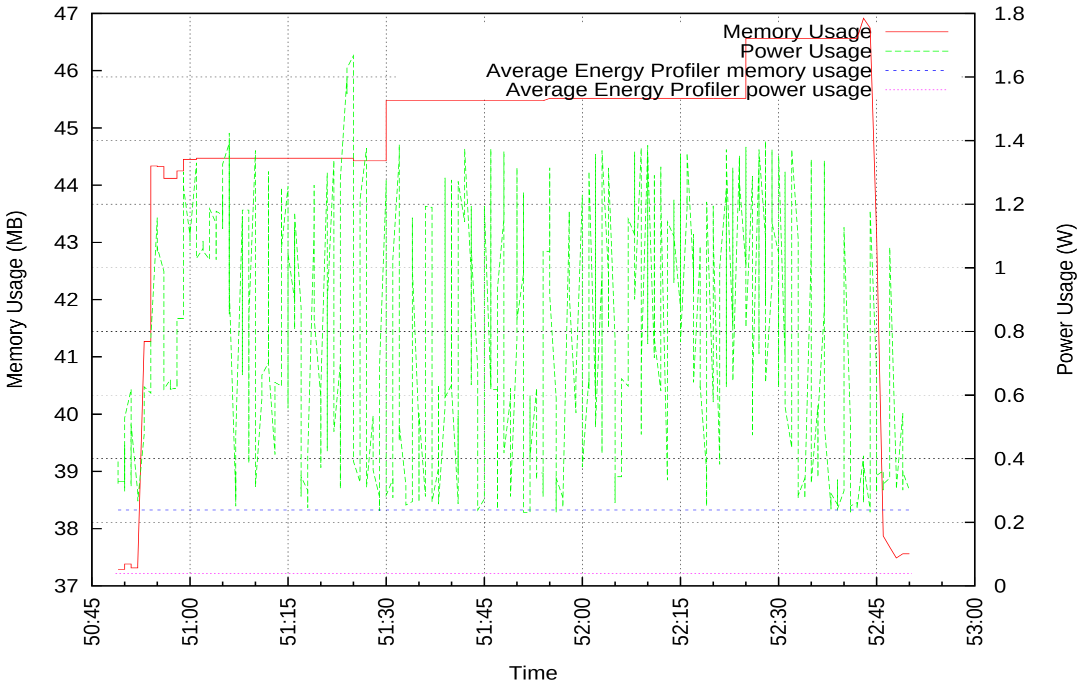

Figure 12: Power and memory usage of Google Maps on the E51.

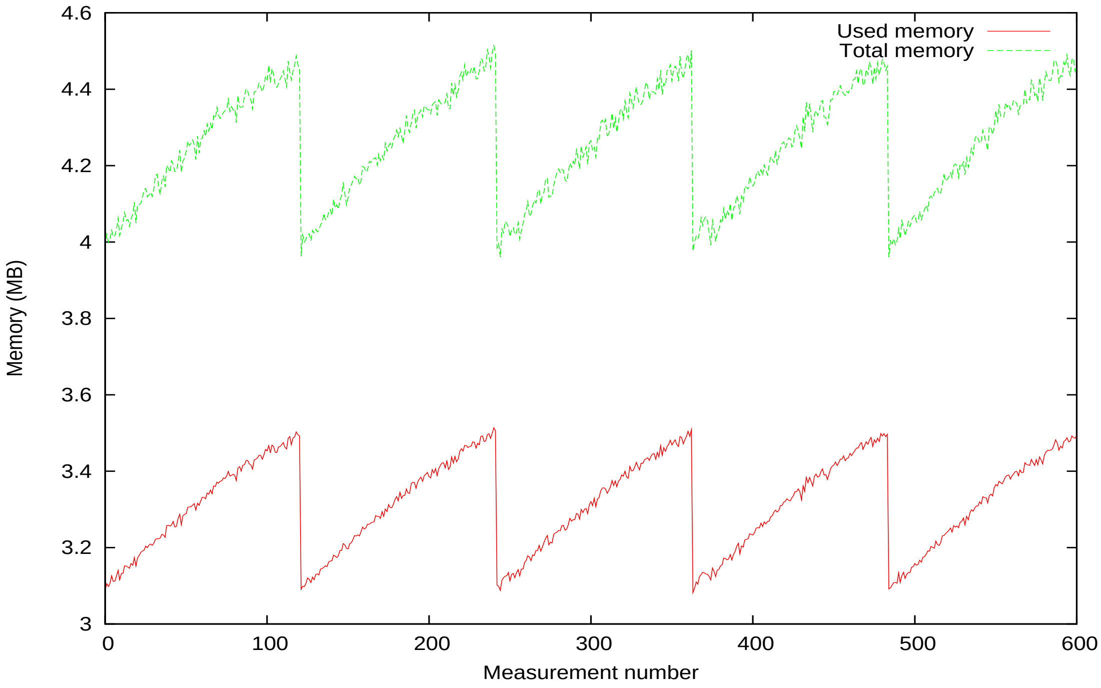

Figure 13: Memory usage of Dessy measured inside Java.

the Java Virtual Machine reserves around 4.5 megabytes for use. This is in line with the minimal memory use numbers from the Energy Profiler. It is likely that the memory use of the Energy Profiler itself skews the measurements. I therefore recommend that researchers measuring Java memory usage on Nokia phones use the Java internal tools, and do not rely on Energy Profiler for memory measurements.

Query performance

Finally, I measured the time it takes to search for files remotely, and receive results on the Nokia E51. In this experiment, the phone connects to a laptop running the Dessy remote API, and requests files with random queries. The laptop used the UK English words dataset introduced earlier in this section. It had a full Dessy index, using the Dessy MySQL index implementation. The queries issued by the phone consisted of property — value pairs, with the property always set to `text:`, as the files had no other index metadata than text words. The query words were picked from the UK English words dataset. The number of query words was normally distributed, with the mean being 2.3 query words, as observed in mobile search studies [YMP08, KB06]. The standard deviation was 1. This reflects usual search behaviour; searches with 1, 2, or 3 words are common. The experiment timed the whole operation from sending the query to the point when results are usable on the Nokia E51. Each experiment run consisted of one thousand queries, run subsequently. I ran the experiment five times, and calculated the mean and standard deviation. The results can be seen in Figure 14. In the Figure, the x–axis shows the number of results obtained from the server. The y–axis displays the time elapsed between sending the query and receiving the results. The experiment results suggest that searching is quick enough on the phone, with the current Dessy MySQL index. The searches completed on average in under half a second, and all completed in under 900 milliseconds. This well achieves the goal of searching in under a second, as specified in the previous section. Furthermore, the search time of one second is extremely short compared to the average time that users spend inputting the queries. In the Google mobile search study [KB06], users took 56 – 63 seconds on average to type a query to the mobile version of Google. While this experiment did not perform local searching or searching via Google in parallel, that is also possible. The user's wait time may be further shortened when searching multiple sources, as the return time can vary between sources. As a final note, the results were recorded when connected to a laptop via IEEE 802.11 wireless. On GPRS or 3G, network latency will make the search wait times longer. However, the search only requires one network round–trip, so latencies should only increase by a constant factor.

In order to see if synchronization and searching affect the memory usage of Dessy, I also recorded the memory usage when searching. The memory usage can be seen in Figure 15. The Figure represents memory usage of five consecutive runs of the search experiment explained above. The values are comparable to those recorded while synchronizing. It appears that searching raises the memory usage by a megabyte or two; in the synchronization case, Dessy used 3.5 megabytes, and here the bar rises

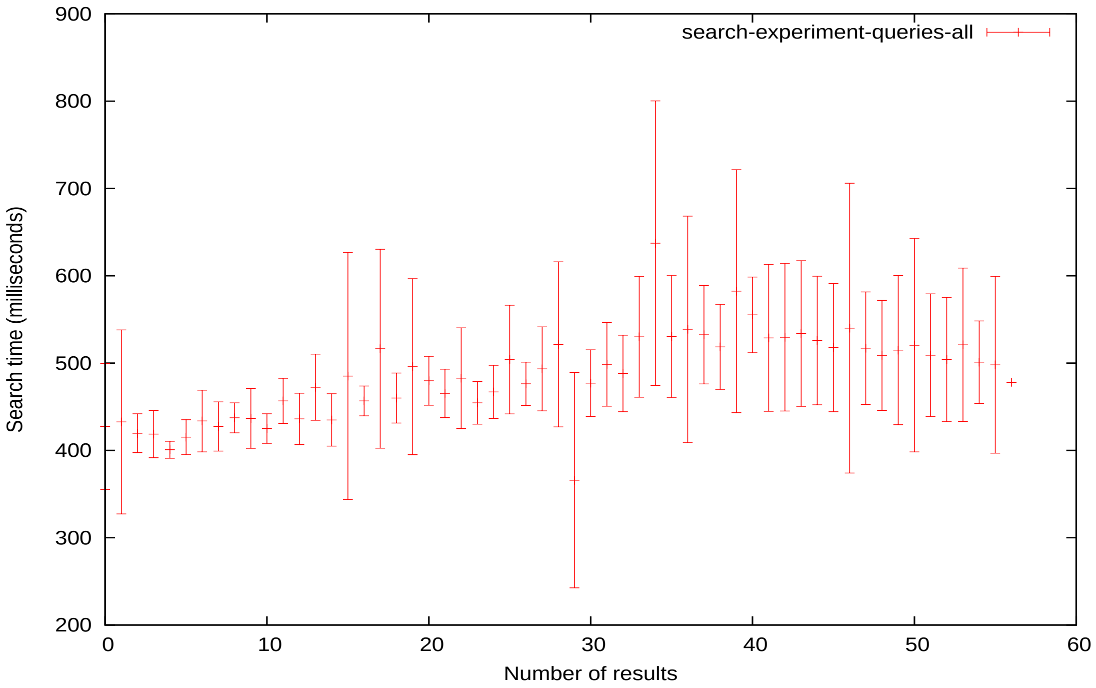

Figure 14: Remote query performance on the Nokia E51.

to 5.8 – 6 megabytes.

7 Discussion

This section discusses the impact of the results of the evaluation and lessons learned. The results in the previous section suggest that Dessy synchronization speed is adequate for common user tasks. While synchronization is not instantaneous, wait times are usually below one minute, and users may perform other tasks while synchronization takes place. For example, typing up searches and searching for files is possible at the same time with synchronization of previous search results. The search speed of Dessy is acceptable on mobile phones. On the Nokia E51, it stays below 900 ms in all cases, and under half a second on average. This enables users to quickly refine their searches, and gives users the impression of responsiveness.

The memory and power usage of Dessy are within reasonable limits, and are comparable to those of other software designed for the mobile platform, such as Google Maps. Dessy uses an average of 3.5 to 6 megabytes of memory when running on the mobile phone, actively carrying out searches and synchronization. This is a fraction of current handset main memory, which is from 30 to 100 megabytes. Newer models have a larger memory capacity. The power usage of Dessy is comparable to Google Maps, and oscillates between 0.4 W and 0.8 W. The idle power usage of the phone is below 0.1 W.

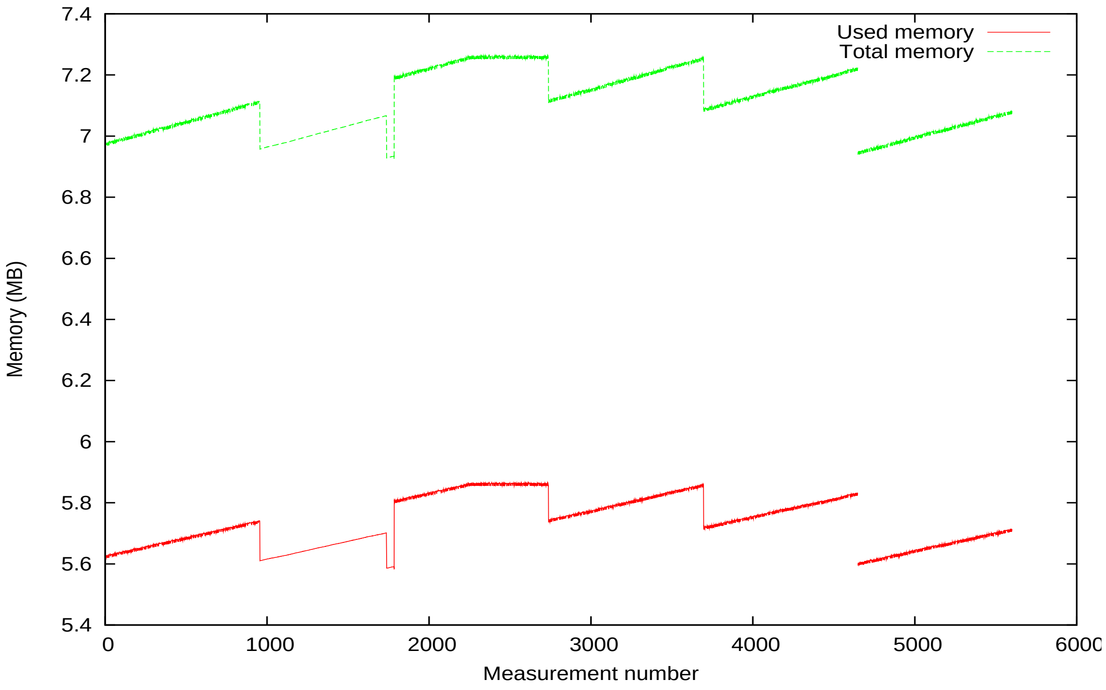

Figure 15: Remote query experiment memory usage on the Nokia E51.

However, there is room for improvement in Dessy. For example, synchronizing parts of the index with a server is slow on mobile phones. To improve this, a faster index storage system should be used on the phone. Also, the retrieval performance results were inconclusive since the CACM dataset is not well–suited to desktop search. Its long natural language queries are very different from the queries that users type into Internet search interfaces or in their desktop search software. Evaluating Dessy with a large collection with relevance judgments that is well–suited to desktop search remains future work.

For future mobile desktop search systems, performance is an important consideration. While prototyping in Java is easy, the J2ME platform makes desktop search development for mobile phones harder than it need be. The lack of a fast file access API for J2ME makes it a poor choice for desktop search. Python for Nokia phones could be a viable alternative. SQLite, file access, and many other features of the Symbian system are available in Python on Nokia phones. On the other hand, the performance of C/C++ and a wider range of available capabilities are advantages to consider. Also, the J2ME features are needlessly restricted in the name of platform security. J2ME applications without a Certificate Authority – signed certificate cannot practically read or write files. I would therefore discourage use of J2ME if other, multi–platform solutions are available.

To save power on mobile devices, the energy usage of a desktop search system can be reduced by running only the query handler at all times, and starting the search interface on user request. Also, using J2ME demands some memory for running the

Java virtual machine. This can be avoided by using another language, such as C, C++ or Python.

While performance is important, extensibility must be emphasized. A fast desktop search system, even if just a core, could be used by developers of desktop search on mobile platforms. Extensibility for new file types, changeable user interfaces, a flexible API for applications, and changeable index storage systems can make a well–written desktop search system useful for many user groups for years to come.

8 Conclusions and Future Work

This thesis presented Dessy, a desktop search and synchronization system for mobile devices. Dessy is extensible in many ways, through new indexing helpers, alternative index storage methods, and query plugins. Dessy contains a limited storage flat file database index, suitable for mobile devices, and a MySQL index for faster servers. Dessy separates metadata extraction, storage and use, and allows synchronizing parts of the index between different index implementations. Dessy can synchronize data and metadata separately. A device does not require index information of a file in order to synchronize the file's contents. In addition to searching for locally indexed files, the Dessy search API is accessible through TCP connections. For example, a phone with a Dessy client can search for files residing on a desktop machine with the Dessy server. Searching for files on the Internet via Google is also supported.

Our work on the Dessy prototype supports the conclusion that desktop search is feasible on mobile devices. The experimental results suggest that remote searches with Dessy on a typical IEEE 802.11 network complete in around 500 ms on average, and that synchronization of a file up to 5 megabytes in length takes approximately 40 seconds on average. These results lead us to conclude that desktop search and synchronization are both feasible and reasonably fast on current mobile handsets. I expect that software such as Dessy becomes commonplace as new mobile devices with more processing power, faster storage and faster connections are developed.

Earlier in this thesis, I have mentioned several points of future work. The current storage API for MIDP 2.0 / CLDC 1.1 is inadequate for Dessy. To remedy the situation, I could rewrite Dessy in Python or C, but both would leave me with less supported platforms than I can reach with Java. Another solution is to write a storage module in C or Python, and have the Dessy process communicate with it when data needs to be stored or retrieved. I believe this is a solution that can make Dessy synchronization and index storage faster on mobile devices. To improve the coverage of Dessy queries, a full–text search index could be implemented. Currently, Dessy indexes use only stemming, and can only find exact stem matches from the index. With full–text search, Dessy could find files that contain the query as a part of a longer word. This is desireable to accommodate compound words, grammatical word variation, and approximate queries. Other future goals for Dessy include language detection, handling more languages, and query misspelling correction. Another possible avenue of future work is exploring the coupling of context data gathering software such as BeTelGeuse with Dessy. This type of software can offer Dessy further property — value pairs to find files with. For example, a text document could be found with the title of a project, or the name of a person on a user's calendar entry. Photos could be automatically geotagged with GPS or GSM location information, and found by the street, district, city and country name where they were taken. Photos could also be tagged with persons nearby, through nearby detected Bluetooth devices. Scheduled synchronization, and synchronization of chosen files when a network hotspot is detected was not realized in the Dessy prototype. This is a feature worth exploring.

In summary, Dessy is a good example of desktop search coupled with synchronization that works on mobile phones, but our work is far from done. The prototype implements basic features, such as indexing files, searching for remote and local files, synchronizing them, tagging them with custom metadata and synchronizing index data between Dessy instances. However, we have outlined several advanced features that could be implemented to make Dessy more useful to users.

References

BF93 Borenstein, N. and Freed, N., *RFC 1521: MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies.* Internet Engineering Task Force, September 1993. URL `http://www.ietf.org/rfc/rfc1521.txt`.

Blo70 Bloom, B. H., Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13,7(1970), pages 422–426.

Bol05 Bolso, E. I., File synchronization with unison. *Linux Journal*, 2005,132(2005), pages 6–6.

CDT06 Cutrell, E., Dumais, S. T. and Teevan, J., Searching to eliminate personal information management. *Communications of the ACM*, 49,1(2006), pages 58–64.

CRDS06 Cutrell, E., Robbins, D., Dumais, S. and Sarin, R., Fast,flexible filtering with phlat. In Grinter, R. E. et al. [GRA+06], pages 261–270.

CVS04 Chang, T.-Y., Velayutham, A. and Sivakumar, R., Mimic: raw activity shipping for file synchronization in mobile file systems. *Mobisys 2004 Workshop on Context Awareness*, June 2004, pages 165–176.

Dun94 Dunning, T., Statistical identification of language. Technical Report, 1994.

GJSJWO91 Gifford, D. K., Jouvelot, P., Sheldon, M. A. and James W. O'Toole, J., Semantic file systems. *SOSP '91: Proceedings of the thirteenth ACM* *symposium on Operating systems principles*. ACM Press, 1991, pages 16–25.

GRA$^+$06 Grinter, R. E., Rodden, T., Aoki, P. M., Cutrell, E., Jeffries, R. and Olson, G. M., editors. *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM Press, April 2006.

HBYFL92 Harman, D., Baeza-Yates, R., Fox, E. and Lee, W., Inverted files. In *Information Retrieval: Data Structures and Algorithms*, Frakes, W. B. and Baeza-Yates, R., editors, Prentice Hall, Upper Saddle River, New Jersey, USA, 1992, pages 28–43.

HC03 Hess, C. K. and Campbell, R. H., An application of a context-aware file system. *CHI '03 extended abstracts on Human factors in computing systems*, Cockton, G. and Korhonen, P., editors, April 2003, pages 339–352.

IEE99 Institute of Electrical and Electronic Engineers, Piscataway, New Jersey, USA, *IEEE Std 802.11 — Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, March 1999.

K$^+$00 Kubiatowicz, J. et al., Oceanstore: An architecture for global-scale persistent storage. *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

KB06 Kamvar, M. and Baluja, S., A large scale study of wireless search behavior: Google mobile search. In Grinter, R. E. et al. [GRA$^+$06], pages 701–709.

KLNA09 Kukkonen, J., Lagerspetz, E., Nurmi, P. and Andersson, M., BeTelGeuse: A Platform for Gathering and Processing Situational Data. *IEEE Pervasive Computing*, 8,2(2009), pages 49–56.

Lev66 Levenshtein, V. I., Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10, pages 707–710.

Lin03 Lindholm, T., XML three-way merge as a reconciliation engine for mobile data. *Third ACM International Workshop on Data Engineering for Wireless and Mobile Access*, September 2003, pages 93–97, URL `http://www.hiit.fi/files/fi/fc/papers/mobide03-pc.pdf`.

Lin04 Lindholm, T., A three-way merge for XML documents. *ACM Symposium on Document Engineering*, Munson, E. V. and Vion-Dury, J.-Y., editors. ACM Press, October 2004, pages 1–10, URL `http://www.hiit.fi/files/fi/fc/papers/doceng04-pc.pdf`.

LLS02 Lee, Y.-W., Leung, K.-S. and Satyanarayanan, M., Operation shipping for mobile file systems. *IEEE Transactions on Computers*, 51,12(2002), pages 1410–1422.

LLT07 Lagerspetz, E., Lindholm, T. and Tarkoma, S., Dessy: Towards flexible mobile desktop search. *Proceedings of the Fourth ACM SIGACT-SIGOPS International Workshop on Foundations of Mobile Computing*, 2007.

MD83 Mah, C. P. and D'Amore, R. J., Complete statistical indexing of text by overlapping word fragments. *ACM SIGIR Forum*, 17,3(1983), pages 6–16.

MES95 Mummert, L., Ebling, M. and Satyanarayanan, M., Exploiting weak connectivity for mobile file access. *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, December 1995, pages 143–155.

ML05 Mei, H. and Lukkien, J., A remote personal device management framework based on syncml dm specifications. *MDM '05: Proceedings of the 6th international conference on Mobile data management*, New York, NY, USA, 2005, ACM, pages 185–191.

NLB$^+$08a Nurmi, P., Lagerspetz, E., Buntine, W., Floréen, P. and Kukkonen, J., Product retrieval for grocery stores. *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 2008.

NLB$^+$08b Nurmi, P., Lagerspetz, E., Buntine, W., Floréen, P., Kukkonen, J. and Peltonen, P., Natural language retrieval of grocery products. *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM'08)*. ACM, 2008.

Now89 Nowicki, B., *RFC 1094: NFS Network File System Protocol Specification*. Internet Engineering Task Force, March 1989. URL `http://www.ietf.org/rfc/rfc1094.txt`.

Por80 Porter, M., An algorithm for suffix stripping. *Program*, 14,3(1980), pages 130–137.

PST$^+$97 Petersen, K., Spreitzer, M., Terry, D., Theimer, M. and Demers, A. J., Flexible update propagation for weakly consistent replication. *Proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, September 1997, pages 288–301.

RWBG95 Robertson, S. E., Walker, S., Beaulieu, M. M. and Gatford, M., Okapi at trec-4. *TREC-4*, 1995, pages 73–96.

RZT04 Robertson, S., Zaragoza, H. and Taylor, M., Simple BM25 extension to multiple weighted fields. *12th ACM Conference on Information and Knowledge Management*, November 2004, pages 42–49.

SB87 Salton, G. and Buckley, C., Term weighting approaches in automatic text retrieval. Technical Report, Ithaca, NY, USA, 1987.

SG05 Soules, C. A. N. and Ganger, G. R., Connections: using context to enhance file search. *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, Herbert, A. and Birman, K. P., editors, October 2005, pages 119–132.

SS05 Saito, Y. and Shapiro, M., Optimistic replication. *ACM Computing Surveys*, 37,1(2005), pages 42–81.

SWR00a Sparck Jones, K., Walker, S. and Robertson, S. E., A probabilistic model of information retrieval: development and comparative experiments Part 1. *Information Processing and Management*, 36, pages 779–808.

SWR00b Sparck Jones, K., Walker, S. and Robertson, S. E., A probabilistic model of information retrieval: development and comparative experiments Part 2. *Information Processing and Management*, 36, pages 809–840.

Syn02 SyncML Initiative, *SyncML Sync Protocol, version 1.1*, February 2002. URL `http://www.syncml.org/docs/syncml_sync_protocol_v11_20020215.pdf`.

TKL$^+$06 Tarkoma, S., Kangasharju, J., Lindholm, T., Ramya, S. K. and Raatikainen, K., Specification of Fuego middleware service set 2006. Technical Report, Helsinki Institute for Information Technology, Helsinki, Finland, June 2006.

TKLR06 Tarkoma, S., Kangasharju, J., Lindholm, T. and Raatikainen, K., Fuego: Experiences with mobile data communication and synchronization. *17th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, September 2006, URL `http://dx.doi.org/10.1109/PIMRC.2006.254072`.

Vir05 Virkus, R., *Pro J2ME Polish.* Springer-Verlag, New York, LLC, July 2005.

YMP08 Yi, J., Maghoul, F. and Pedersen, J., Deciphering mobile search patterns: a study of yahoo! mobile search queries. *The Seventeenth World Wide Web Conference*, April 2008, pages 257–266.

ZMR98 Zobel, J., Moffat, A. and Ramamohanarao, K., Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23,4(1998), pages 453–490.