Project LiquidLan

Final Report

May 08, 2006

Scott Baldwin

Mentor: Dr. Chi-Ren Shyu

Senior Capstone

Table of Contents

Abstract

This report provides a detailed overview and analysis of my Senior Capstone project, LiquidLan. Windows Networks have been around for many, many years, but have always lacked a way to efficiently search them. At its core LiquidLan it is file sharing system for Windows Networks. It differs from the rest in that it provides a custom application client that hides all the underlying protocols from the user. Current implementations rely on webpage interfaces, severely limiting their functionality. LiquidLan has no such design limitations.

The main purpose of LiquidLan is to provide a fast search mechanism and feature-rich download management tools. The system has two major components: an application client and a search engine. In the sections that follow I will discuss the constraints, requirements, and alternative solutions that exist. A detailed look at the design and implementation will also be presented.

1. Problem Definition

1.1 Introduction

My capstone project, *LiquidLan*, solves an existing problem in a very innovative way. At its core, LiquidLan is a software layer that extends the functionality of Windows/Samba (SMB) networks. It consists of an application client for Windows and a search engine/server for Linux. The server scans Windows/Samba networks for public shares and indexes them into a database. The client is the front-end—it is what end-users see. The client can perform searches, manage downloads, and other related tasks.

LiquidLan is unlike any other implementation in that it uses an application client to make the underlying Windows/Samba network transparent to the user. Current implementations rely on webpage interfaces that display results as Universal Naming Convention (UNC) resource paths. The functionality therefore is limited by the capabilities of the Web browser. LiquidLan has no such design limitations.

One of the more innovative features of LiquidLan is a server address resolution mechanism called Dynamic LiquidLan Configuration Protocol (DLCP). The client will detect what network it is on and query dlcp.liquidlan.net to pull the server information for that network and automatically configure itself. This requires no user intervention! As a result, the user can literally download the installer, install the program, and start searching and downloading without any configuration. Many users will feel as though they are using a high-speed Gnutella network due to the client's look and feel.

Nothing quite like LiquidLan has ever been done before. LiquidLan is the only Windows Network search engine system I have seen that has a custom application client. I expect there to be high demand on college campuses, provided the administration is open to new ideas. Of course, controversy surrounds anything that is even remotely related to file sharing, so I expect that some users will have difficulty gaining acceptance from their administration to run LiquidLan on their campus network.

1.2 Background

The current state of the *underlying* technology is well-established. Windows/Samba networks have been around for many, many years, but have always lacked a way to efficiently search them. The main purpose of LiquidLan is to provide a fast search mechanism and feature-rich download management tools. LiquidLan can also facilitate the sharing of files through Windows API functions so that the user can easily share directories on their hard drive.

Windows/Samba networks are in fact peer-to-peer networks, but LiquidLan provides its services through a client-server communication model. An XML-based protocol is used to exchange messages between the client and server. The communication consists mainly of global configuration (from the server) when the client is initialized, search queries from the client and corresponding search results from the server.

For the server I'm extending the Seek42 project, an SMB network search engine originally written by a student at UMR. It is released under the GPL and is available to download at http://sourceforge.net/projects/seek42/. The problem with Seek42 is that it uses a webpage interface, which is very restrictive. The nice thing about Seek42 though is that its implementation is almost pure ANSI C, and so it should easily port to other environments.

For the client, I'm using an application framework based on the eMule (ED2K) client for Windows. It is also released under the GPL and is available to download at http://prdownloads.sourceforge.net/emule/. Since the ED2K protocol and functionality are so drastically different from that of LiquidLan, I basically threw out most of the core application logic and rewrote it. I have also added many new features to make it more ideal for LANs.

The LiquidLan layer runs on top of Windows/Samba networks—I'm not actually implementing a new protocol. The SMB/CIFS protocols are implemented in Windows and Samba along with all the networking functions (master-browser elections, authentication, etc). This underlying network is encapsulated by APIs and the Samba suite (http://www.samba.org/). I chose to use a client-server model instead of P2P for LiquidLan because it simplifies the implementation, and it also provides faster, more reliable searching. The instantaneous searching would be impossible under a P2P model. Also, the scalability that P2P provides isn’t necessary for local area networks.

Some of the current systems include PySMBSearch, UntzUntz LAN Scan, Strangesearch, Netropolis, and Phynd. None of them come equipped with an application front end. To clarify what I mean by "application front-end" and "webpage front-end" consider this: An application front-end is what Napster had. A webpage interface is what you have when you search Google. The difference is that an application front-end can do lots of application-specific things that are not possible with a Web browser. All of the projects I mentioned above rely on a webpage interface.

There are few, if any, research publications that specifically talk about SMB search engines. The most relevant papers out there discuss Windows Networks and Samba in general or search engines in general. Please refer to section 1.3, Literature Review, for several articles that discuss various file sharing topics.

1.3 Literature Review

[1] Incentives in BitTorrent Induce Free Riding

In the last several years, BitTorrent has emerged as one of the most popular peer-to-peer file distribution protocols. One of the key reasons why BitTorrent is so popular is its effectiveness at encouraging cooperation among peers. In other words, there are fewer free-loaders (people who take but never give back) using BitTorrent networks than with other popular P2P file sharing systems.

While BitTorrent's "Give and Ye Shall Receive" attitude has paid off greatly, it is far from perfect. The author discusses many of the perceived weaknesses of BitTorrent's incentive mechanism. The author also offers an alternative solution based on some relevant theory, and experimental evidence to suggest that the alternative solution would outperform the current solution in practice.

The author first presents an overview of BitTorrent and its incentive mechanism. The protocols are complex, but the ideas are rather simple. Essentially, a BitTorrent network consists of users exchanging file chunks with one another. It starts out with a *.torrent* file that contains information to help the user find the master node (called a tracker) and verify the integrity of each chunk that is downloaded. Once the user is connected, the tracker will send it a list of all the peers. The peers will collaborate with each other to determine who has which chunks. The chunks allow greater parallelism, which in turn improves scalability. The scalability of BitTorrent is one of the primary reasons why it is so popular.

In a perfect world everyone would share, but we do not live in such a world. BitTorrent's protocol, however, has managed to effectively coerce users into sharing—it's called the incentive mechanism. Quite simply, this means that users are rewarded for sharing and punished for not sharing. BitTorrent's incentive mechanism is surprisingly simple. First, let N be the maximum number of peers a BitTorrent client is configured to upload to at any given time. The client will then upload to the N peers that are giving it the fastest download rates. All the other peers are denied service, and in turn they will deny you back. This strategy is effective however because those nodes aren't as generous anyway, hence the incentive to give. If you do not give, then eventually most of the peers will turn their back on you. BitTorrent has one card up its sleeve though, and it's called the "optimistic unchoking" (unchoking is BitTorrent lingo for uploading). The optimistic unchoking mechanism works as follows: the BitTorrent client will probe for faster links (that are not within the set of N peers) by uploading a chunk to it in order to reacquaint with that node. If the node turns out to be slow, then the BitTorrent client will once again ignore it and move on to test the next node not within N.

The author, however, argues that this incentive mechanism can be improved upon. The new mechanism is based on a strategy called *Tit-For-Tat*. The strategy is simple: *In the first exchange, the client will always cooperate (share). Thereafter, it does what the other peer did in the previous exchange*. [1] In other words, every node should share an even amount of upload bandwidth with every other node. If there is a selfish node (unwilling to share) then that node's actions will be reported by the tracker and all other nodes will ignore it, resulting in starvation for that node. The incentive then is obvious—share with others and they will share with you. It is simpler than BitTorrent's actual mechanism, and the author shows evidence that it can outperform the actual mechanism in practice.

The author identifies four important properties of an efficient incentive mechanism. First of all, the client must be nice; that is, never be the first to deny service. Second, the client must be retaliatory; that is, if another peer refuses to upload then the client must ignore that peer. Third, the client must be forgiving; that is, if that same peer decides later on to start sharing, then the client should forgive and reciprocate the kindness. Finally, the behavior must be clear and well understood. If all clients understand the rules, then the system should work. [1]

BitTorrent's incentive mechanism lacks some of these properties. For example, once a BitTorrent client is already uploading to N peers, it will have to ignore the rest, which isn't *nice*. [1] Consequently, the other peer's response will be mutual. Also, the "optimistic unchoking" often goes uncompensated, which results in leaking resources to free-loaders. This is especially undesirable for BitTorrent networks. Replacing the incentive mechanism with a strategy similar to Tit-For-Tat would likely prevent such problems.

[2] The Impact of DHT Routing Geometry on Resilience and Proximity

One of the integral parts of any peer-to-peer file sharing system is its routing method. The performance, resilience, flexibility, and scalability of the overall system are all affected. A relatively new class of routing tables in P2P, called Distributed Hash Tables (DHTs), is the focus of the article.

DHTs partition ownership of a set of keys among all the nodes such that messages can be efficiently routed to the unique owner of a given key. [6] Each node is like a bucket in a hash table; each data object is associated with a key, mapping it to the IP-address of the node hosting it. [6] Infrastructures based on DHTs are extremely scalable and robust even in transient environments (continuous node arrivals and failures). In the event of node failures, recovery algorithms are used to repopulate the routing tables with live nodes so routing can continue. DHTs can be used to route around trouble before the recovery mechanisms kick in. [2]

As the title of the article suggests, the main focus is comparing the routing *geometries* of each algorithm. *Geometry* refers to the way in which neighbors and routes are chosen, and how its routing paths are geometrically interpreted. Some examples include the hypercube, ring, tree, and butterfly. [2] These various DHT geometries provide different levels of flexibility for neighbor and route selection.

When evaluating a given DHT algorithm, it is important not to use a "black-box approach" in which the entire algorithm is treated atomically. Instead one should break down the algorithm into its many design components and then evaluate them independently. This approach could lead to a hybrid design that incorporates the best components of all the algorithms. [2]

In judging which is best, the author believes that flexibility is a paramount consideration. Flexibility describes the amount of freedom available to choose neighbors and next-hop paths. Flexible neighbor selection can be based on other criteria in addition to the identifiers, such as proximity (i.e., latency). Given a set of neighbors and a destination, the routing algorithm determines the choice of the next-hop. The flexibility depends on how many options there are for the next-hop. If there aren't any, or only a few, then the routing algorithm is likely to fare poorly under high failure rates. [2]

The choice of routing geometry is critical to other routing design issues. One of the most important differences is the degree of flexibility. When comparing DHT algorithms one should use a component-based analysis to fully understand which parts of the design are smart and which parts need reworking.

[3] Tree-Based ALM using Proactive Route Maintenance

Traditionally, P2P systems have been based on unstructured networks, making it difficult to come up with efficient routing algorithms. An example method might involve forwarding messages with a Time-To-Live (TTL) field incremented at every hop. More flexible routing would require a more organized structure. [3] Fortunately for P2P, structured overlay networks have emerged in recent years.

An overlay network is a virtual network that is built on top of, or "overlays", many different physical networks. A single link between any two nodes may comprise multiple physical routers. Overlay P2P networks are self-organized and typically robust enough to handle high rates of node entry and failure. One of the most common overlay structures is a tree, called an overlay tree. The organization and routing of an overlay tree is directed by prefix-matching of each node's identifier. A group of nodes can resolve its parent through a group hash function. The parent node, in turn, chooses another node until the root node is reached and the tree is constructed. [3]

When a parent node departs, it is important to restore the backup route quickly since all the child nodes are disconnected. It usually takes several seconds to restore the overlay tree, but using a proactive approach can cut the interruption time in half. [3] The basic underlying idea is that each non-leaf node in the overlay tree pre-computes a backup route. [3] Upon departure of a parent, any node can then use the backup route to find and attach to another parent quickly. Thus affected nodes can receive data flow after lower interruption time than that of the reactive approach.

[4] An XML-based Conversational Protocol for Web Services

The eXtensible Markup Language (XML) is essentially a general-purpose markup language for defining special-purpose markup languages. It has become an increasingly popular messaging framework for Web services. Many new protocols are derived from XML, such as the well-known SOAP (Simple Object Access Protocol). SOAP, however, has inherent drawbacks in that it cannot converse (negotiate) with Web servers to perform specific operations. [4]

For example, several merchant sites may use different messages to provide the same service. Or, the merchant sites may use the same message formats for placing an order but use them in a difference sequence. Without prior knowledge of these protocols and any semantic differences between them, the client is unable to communicate with the merchants. With a dynamic communication protocol (such as the one proposed in the article), the client could download a protocol specification from each merchant and *discover* the protocols dynamically. The client could then implement the protocol and uses it to access the services. [4] This example illustrates how a conversational protocol can make Web services more interoperable.

[5] Design Choices for Content Distribution in P2P Networks

Many choices must be made when it comes to designing a P2P architecture. Two popular solutions are the tree and mesh-based organizations. The distributed nature of P2P, along with its lack of centralization and millions of users make realistic test simulations difficult. This in turn makes it difficult to determine which architecture (tree or mesh) is truly more efficient.

In the mesh approach, nodes self-organize into groups (typically between 20 and 200 nodes) called neighbors. A cooperation strategy must exist in the protocol so that neighbors cooperate with one another to leverage the available bandwidth and to rapidly distribute the content. [5] In many cases, there must be an incentive mechanism to cooperate.

In the tree approach, nodes are organized in a tree-like structure. The tree starts out with a root node and branches down until the leaf-nodes are reached. Three common tree based architectures are Linear, Tree^k, and PTree^k. [5] The main difference between these architectures is whether they can run in parallel and whether all nodes (including leaf-nodes) contribute resources.

The author demonstrates experimentally that meshes outperform trees on average. The cooperation strategy mentioned above plays an important role in the performance of mesh overlays. The main factors are the peer selection strategy, the chunk selection strategy, and the network degree. [5] The peer selection strategy determines how peers select other peers to provide with bandwidth. *Least Missing* and *Most Missing* are commonly used peer selection strategies. [5] The chunk selection strategy determines the order in which chucks are exchanged. The network degree specifies the maximum number of active exchanges (uploads/downloads) allowed per node at any given time.

In conclusion, P2P design choices must be made carefully. After the architecture is chosen (e.g., mesh), the designer must determine other policies and strategies, such as peer selection, chunk selection, and network degree and ensure that these policies are cohesive with one another. The designer must also decide the degree of parallelism and rules of cooperation for the peers.

1.4. Goals & Objectives

I have many goals and objectives in place for this project. I will gain the satisfaction of creating a useful file sharing application that is open source and free of charge to others. I will also learn a lot about multi-threaded programming, graphical user interface programming, networking, XML-based protocol design, and object oriented design. My ultimate goal for LiquidLan is for it to be the best LAN file sharing solution in existence. I would also like to see several universities deploy it on their networks.

The development requires skills in several technologies that I had to learn outside the curriculum (Visual C++/MFC, socket programming, multi-threaded programming, the Windows API, debugging with gdb and the Visual Studio debugger).

1.5 Overall Approach

The design process model I have used is very similar to the waterfall process model. During the requirements analysis process I looked at current systems, and decided on what I wanted to model my own system after. Eventually that analysis is what led me to use Seek42 and eMule as the server base and client framework, respectively. I started off with a clean framework thanks to these quality open source projects. The implementation and testing stages have been the longest stages. I have made substantial modifications and extensions to the client and server, during which I have used the feedback I get from friends to improve upon the design and functionality.

I think there are many advantages to my development approach. For one, it is simple and efficient. My emphasis on a clean application framework has resulted in a very reusable and extendable code. The class interfaces are clean and have strong cohesion. I also thoroughly debug everything. Basically, I do everything possible to create high-quality software.

Below is the LiquidLan system diagram. It illustrates how LiquidLan runs on top of SMB network protocols (implemented in the WinAPI and Samba). The LiquidLan software layer provides users with greater functionality, including the ability to search the entire network and manage downloads.

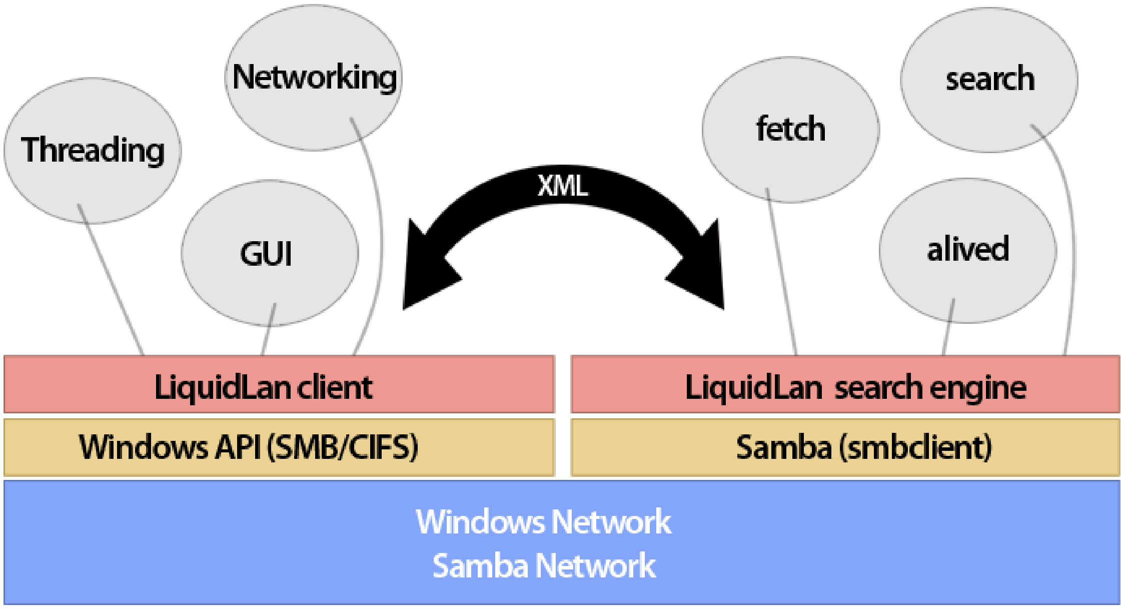


Figure 1.0

2. Requirements Analysis

2.1 Introduction

The purpose of this paper is to present an analysis of the requirements involved in the design and implementation of my Senior Capstone project, LiquidLan. Before I start, however, I should state that the most important requirement of all is that it meets my goals and everything I had envisioned from the onset. My goal is quite ambitious—that LiquidLan be the ultimate service of its kind. So clearly I intend for LiquidLan to be more than just a working prototype. As such, the design and implementation of the system will comprise many requirements that must be carefully looked at and addressed.

For readers that are unfamiliar with what LiquidLan will accomplish, I will first present an overall description of the system. I will then discuss all the internal, external, and regulatory constraints. I will list the system components, development tools, and software interfaces and libraries that will be used. Finally, I will delve into the performance and resource requirements.

2.2 Overall Description

LiquidLan is a software layer that extends the functionality of Windows/Samba (SMB) networks by providing fast search capabilities. It consists of an application client (the client) for Windows and a search engine (the server) for Linux and other Unix platforms. The server scans Windows/Samba networks for public shares and indexes them within a flat-file database. The client is the front-end—it is what *most* end-users will see. The client can send search requests, process search responses, manage downloads, and perform all the standard functions expected from a high-end file sharing client.

LiquidLan differs from the competition in that it provides an application client that effectively hides the underlying Windows/Samba network from the user. Current implementations rely on webpage interfaces that render search results as Universal Naming Convention (UNC) resource paths (as hyperlinks). The functionality in these systems is limited by the capabilities of the Web browser. LiquidLan has no such design limitations.

LiquidLan will also be capable of configuring itself automatically for the network it is running on. The client will use a lookup service on liquidlan.net to pull the server information for that network and automatically configure itself. Of course, the server operator for that network will need to manage the record on liquidlan.net. As a result, a user on the network can literally download the installer, install the program, and start searching and downloading without any configuration. Many users will feel as though they are using a high-speed Gnutella network due to the client's look and feel.

2.3 System Requirements and Constraints

2.3.1 Operating environment (external constraints)

The operating environment differs for the client and server. The client is compatible with NT-based versions of Windows (NT/2000/XP/Vista/etc.). Unfortunately the client cannot run on Windows 9x (95/98/ME) due to the Windows 9x API lacking a necessary function in kernel32.dll. LiquidLan requires CopyFileEx() for facilitating SMB file transfers. The Windows 9x API does have a routine called CopyFile(), but it does not provide a call-back interface for reporting back transfer progress. In NT, the routine was renamed to CopyFileEx() because of this extended functionality.

The client's development environment is Microsoft Visual Studio 2005. The code base is entirely C++; it uses the Microsoft Foundation Classes (MFC) and is **not** managed by the *.net* framework. Visual Studio has an excellent integrated debugger that I find extremely useful.

The server is designed for Linux and BSD, but should easily port to any Unix-like environment (e.g., Solaris and Mac OSX). The server must be compiled so a compiler and linker must be available. The environment must also have Samba installed, or at least the smbclient tool, so that the server can scan hosts on SMB networks. The results from these scans are indexed (saved to a database), but it is not necessary to have any Database Management System (DBMS) such as MySQL installed because the server has database management built-in (result structures are stored in flat files).

I have extended the server using a KDE-based IDE called KDevelop. KDE has many of the essential features that are found in Visual Studio, such as integrated debugging, auto-completion, color-coded syntax, code collapsing, and project management tools. The integrated debugger is a front-end for gdb, and it is quite excellent. It is certainly superior to using gdb from the command line (if you have ever done so then you know how cumbersome it is). I have used the KDevelop debugger extensively throughout development.

2.3.2 Market users and characteristics

I anticipate that end-users will primarily consist of college students living on campus (connected to a residence hall computer network). Since LiquidLan will be free and open source, I think the economic feasibility is undeniable. I expect for there to be high demand for LiquidLan once it has been deployed at several networks and has established a reputation for being a fast, easy, and effective file sharing solution for Local Area Networks. The competitive forces that exist are inferior solutions with far less functionality than LiquidLan (as discussed above).

The most important regulatory constraint that I must consider is copyright law such as the DMCA and recent precedent such as the one set last year by the Supreme Court's MGM v. Grokster ruling. I must be careful in my approach and make a concerted effort to respect and enforce DRM in LiquidLan. I am confident at this point that LiquidLan is completely non-infringing, but there is obviously some risk involved.

Customers (users) will require that LiquidLan have an intuitive user interface, solid performance, and the ability to search for and download files. LiquidLan provides all this and more, so I expect for it to be well-received when it is finished and released.

2.3.3 Environmental constraints

There are several human factors that will affect the success and acceptance of LiquidLan. For every network that it runs on, one user must operate the server and configure/manage that network's record on liquidlan.net. In addition, unless the operator has some kind of authority over the network in which the server is running on, the administration in charge must approve of the LiquidLan service. I have gone to great lengths to ensure that the LiquidLan system is reliable, efficient, non-infringing, and high quality, so I hope administrations will accept it.

2.3.4 System components

LiquidLan can be viewed as two components: an application client and a server. The client can be further broken down into networking, threading, and user interface components. The server can be further broken down into three components: the scanner (fetch), the search engine (search), and the daemon that continually monitors which hosts are alive and which are down (alived).

Another key component to LiquidLan is SMB/CIFS, the protocol that Windows Networks are based upon. However, LiquidLan runs on top of this layer, with Samba and the Windows API in between.

2.3.5 Software interfaces and libraries

The software interfaces and libraries for the client are the standard C++ libraries, Microsoft Foundation Classes (MFC), Win32 API, COM (for the integrated Web browser), the Windows Registry, and XML for communication with the server.

The software interfaces and libraries for the server are the standard C libraries, Unix/POSIX system call interface, smbclient (part of the Samba suite), and XML for communication with the client.

2.3.6 Communication interfaces

The communication interfaces for both the client and server consist of XML for message passing, and TCP/IP as the protocol suite for doing so. Both do so via an asynchronous (non-blocking) socket. Also, the client and server both require interfaces to the underlying Windows network (as previously discussed). The server also uses NetBIOS to resolve the NetBIOS name of each host it scans.

2.3.7 Hardware interfaces

The hardware interfaces required for using LiquidLan is a network link, and a computer with at least 64MB system memory and a 300 MHz processor. To use the client the computer must be an x86 architecture to run Windows.

2.3.8 System maintenance

The software maintenance life cycle and support will consist of improving and extending the system, fixing bugs, fixing security holes, etc. This will be administered through a Version Control system such as CVS. I may also use a bug-tracking solution such as Bugzilla.

2.4 Performance requirements

LiquidLan performs a lot of tasks that involve heavy network and disk I/O. Since network and disk access both tend to be system bottlenecks, it is extremely important for LiquidLan to have excellent performance. The client and server are both implemented in C or C++ and thus have good performance in general. Regarding the I/O, the file transfers are actually facilitated by Windows API calls, which results in highly efficient file transfers. LiquidLan can even enforce transfer quotas (determined by the server operator) that limit the number of concurrent transfers to reduce disk trashing and network congestion. LiquidLan provides other policies that can be adjusted to optimize performance for a given network. Therefore, LiquidLan can accommodate almost any user/network's performance requirements.

2.5 Resource requirements

The resource requirements of this project consist of time, money, and equipment. When LiquidLan is complete, hundreds of hours of time will have been invested to develop a fully-function system. The software used to develop it (Visual Studio) is also quite expensive. The equipment used to run the system includes a computer network with at least one host running a server. The remaining hosts can use the client to send search requests to the server and download files from other hosts.

2.6 Evaluation metrics

I may choose to use methods outlined in the ISO 9126 standard for evaluating the functionality, reliability, usability, efficiency, maintainability, and portability of the system. Since I will be using SourceForge to distribute LiquidLan, I may also use some of the tools that it provides (such as download statistics, bug reports, feedback, etc.). I can also benchmark the performance of my system, and compare it against the performance of similar software applications.

3. Design Specifications

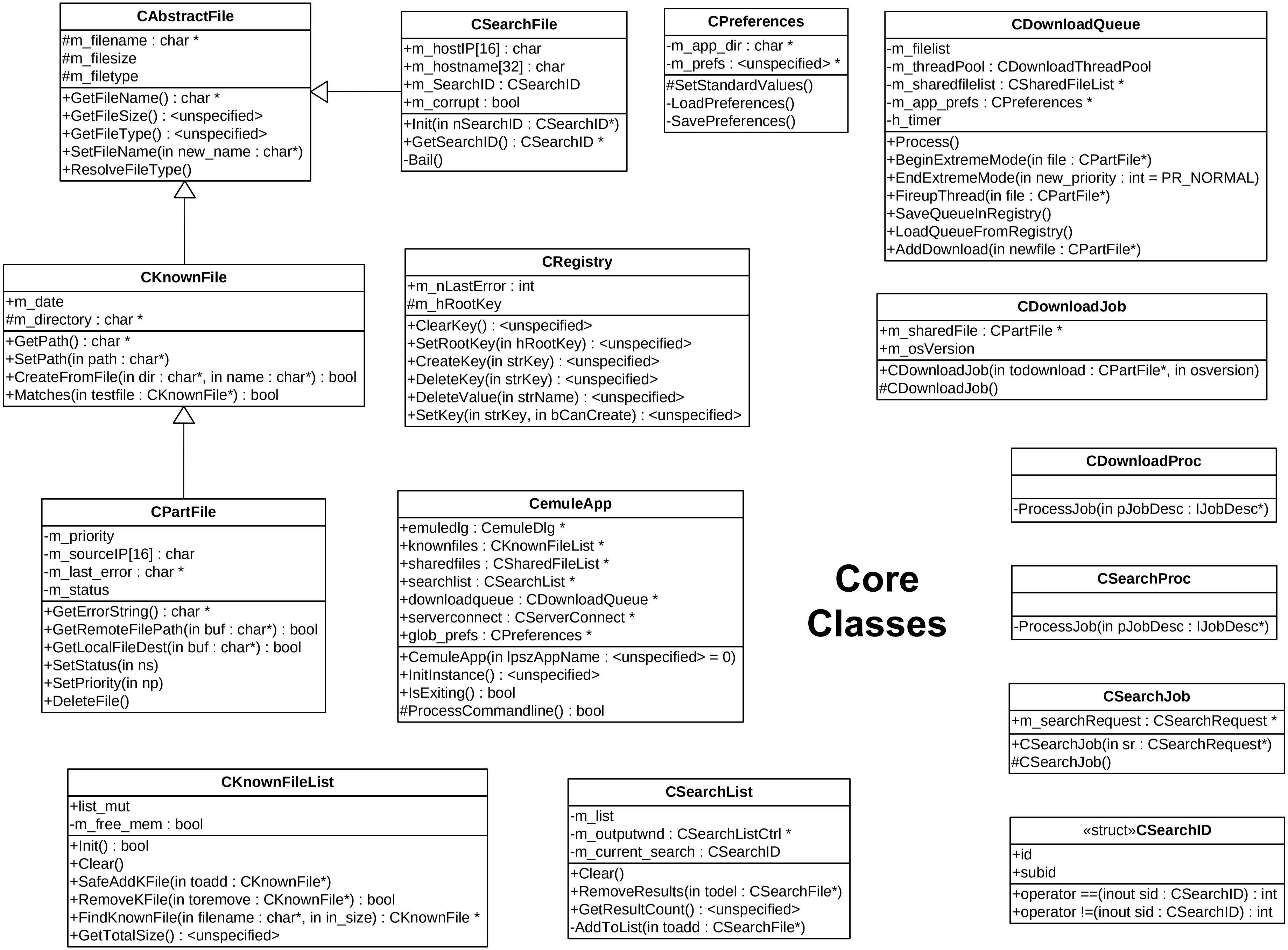
3.1 Software

The software design is largely influenced by objected oriented paradigms. The client application's framework is modeled after that of the eMule project (http://www.emule-project.net/). Thanks to tight integration between Microsoft Visio and Visual Studio, I was easily able to reverse engineer the client application's code base into a formal UML specification. To provide a graphical view of the software design, I modeled the specification into a UML diagram by dividing it into four logical subsystems: core logic, graphical user interface, threading, and networking. These diagrams are presented in the pages that follow.

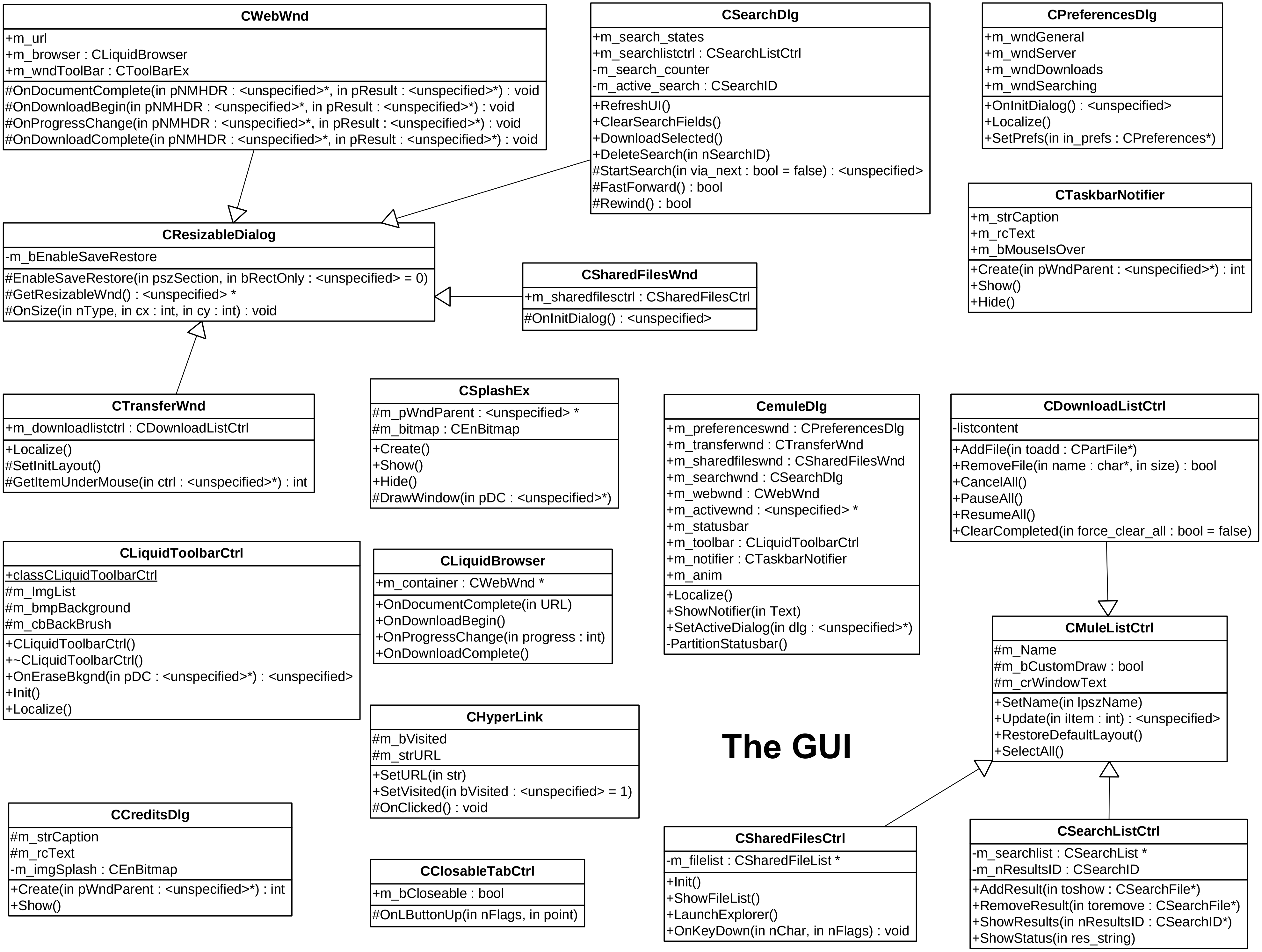
The server software is based on the Seek42 project (http://seek42.sourceforge.net/) and thus it has a very similar design. Since the code base is pure ANSI C, the project does not have an object oriented design, but it is quite modular nonetheless. Program entities and data structures are abstracted as much as possible.

The design process model I have used is very similar to the waterfall process model. I think there are many advantages to the design approach I took. For one, it is simple and efficient. My emphasis on a clean application framework has resulted in a very reusable and extendable code. I also thoroughly tested just about everything. The end result is high-quality software.

3.1 Software Specification (UML)

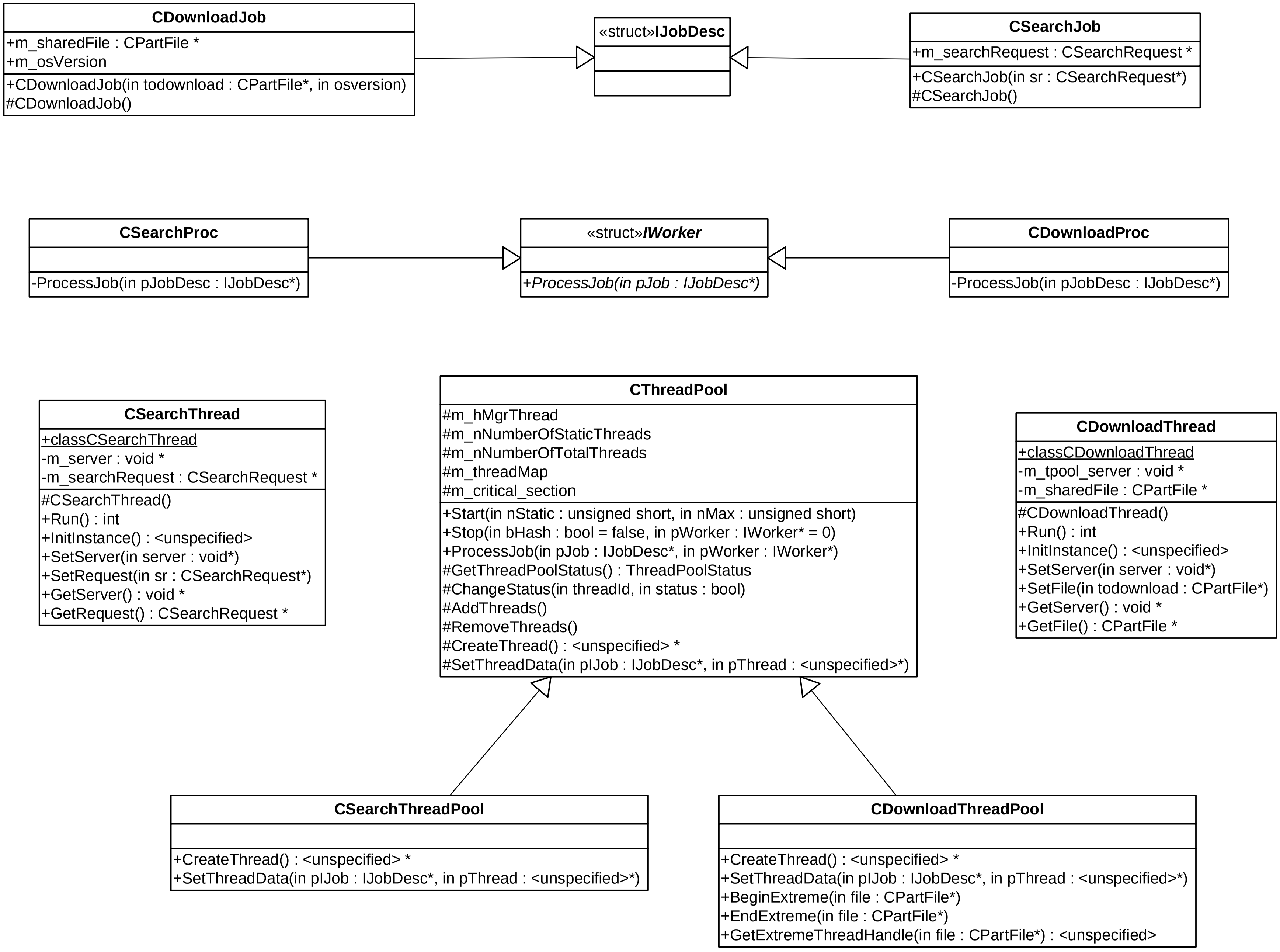

CAbstractFile
#m_filename : char *
#m_filesize
#m_filetype
+GetFileName() : char *
+GetFileSize() : <unspecified>
+GetFileType() : <unspecified>
+SetFileName(in new_name : char*)
+ResolveFileType()
CSearchFile
+m_hostIP[16] : char
+m_hostname[32] : char
+m_SearchID : CSearchID
+m_corrupt : bool
+Init(in nSearchID : CSearchID*)
+GetSearchID() : CSearchID *
-Bail()
CPreferences
-m_app_dir : char *
-m_prefs : <unspecified> *
#SetStandardValues()
-LoadPreferences()
-SavePreferences()
CDownloadQueue
-m_filelist
-m_threadPool : CDownloadThreadPool
-m_sharedfilelist : CSharedFileList *
-m_app_prefs : CPreferences *
-h_timer
+Process()
+BeginExtremeMode(in file : CPartFile*)
+EndExtremeMode(in new_priority : int = PR_NORMAL)
+FireupThread(in file : CPartFile*)
+SaveQueueInRegistry()
+LoadQueueFromRegistry()
+AddDownload(in newfile : CPartFile*)
CKnownFile
+m_date
#m_directory : char *
+GetPath() : char *
+SetPath(in path : char*)
+CreateFromFile(in dir : char*, in name : char*) : bool
+Matches(in testfile : CKnownFile*) : bool
CRegistry
+m_nLastError : int
#m_hRootKey
+ClearKey() : <unspecified>
+SetRootKey(in hRootKey) : <unspecified>
+CreateKey(in strKey) : <unspecified>
+DeleteKey(in strKey) : <unspecified>
+DeleteValue(in strName) : <unspecified>
+SetKey(in strKey, in bCanCreate) : <unspecified>
CDownloadJob
+m_sharedFile : CPartFile *
+m_osVersion
+CDownloadJob(in todownload : CPartFile*, in osversion)
#CDownloadJob()
CDownloadProc
-ProcessJob(in pJobDesc : IJobDesc*)
CPartFile
-m_priority
-m_sourceIP[16] : char
-m_last_error : char *
-m_status
+GetErrorString() : char *
+GetRemoteFilePath(in buf : char*) : bool
+GetLocalFileDest(in buf : char*) : bool
+SetStatus(in ns)
+SetPriority(in np)
+DeleteFile()
CemuleApp
+emuledlg : CemuleDlg *
+knownfiles : CKnownFileList *
+sharedfiles : CSharedFileList *
+searchlist : CSearchList *
+downloadqueue : CDownloadQueue *
+serverconnect : CServerConnect *
+glob_prefs : CPreferences *
+CemuleApp(in lpszAppName : <unspecified> = 0)
+InitInstance() : <unspecified>
+IsExiting() : bool
#ProcessCommandline() : bool
Core
Classes
CSearchProc
-ProcessJob(in pJobDesc : IJobDesc*)
CSearchJob
+m_searchRequest : CSearchRequest *
+CSearchJob(in sr : CSearchRequest*)
#CSearchJob()
CKnownFileList
+list_mut
-m_free_mem : bool
+Init() : bool
+Clear()
+SafeAddKFile(in toadd : CKnownFile*)
+RemoveKFile(in toremove : CKnownFile*) : bool
+FindKnownFile(in filename : char*, in in_size) : CKnownFile *
+GetTotalSize() : <unspecified>
CSearchList
-m_list
-m_outputwnd : CSearchListCtrl *
-m_current_search : CSearchID
+Clear()
+RemoveResults(in todel : CSearchFile*)
+GetResultCount() : <unspecified>
-AddToList(in toadd : CSearchFile*)
«struct»CSearchID
+id
+subid
+operator ==(inout sid : CSearchID) : int
+operator !=(inout sid : CSearchID) : int

The GUI

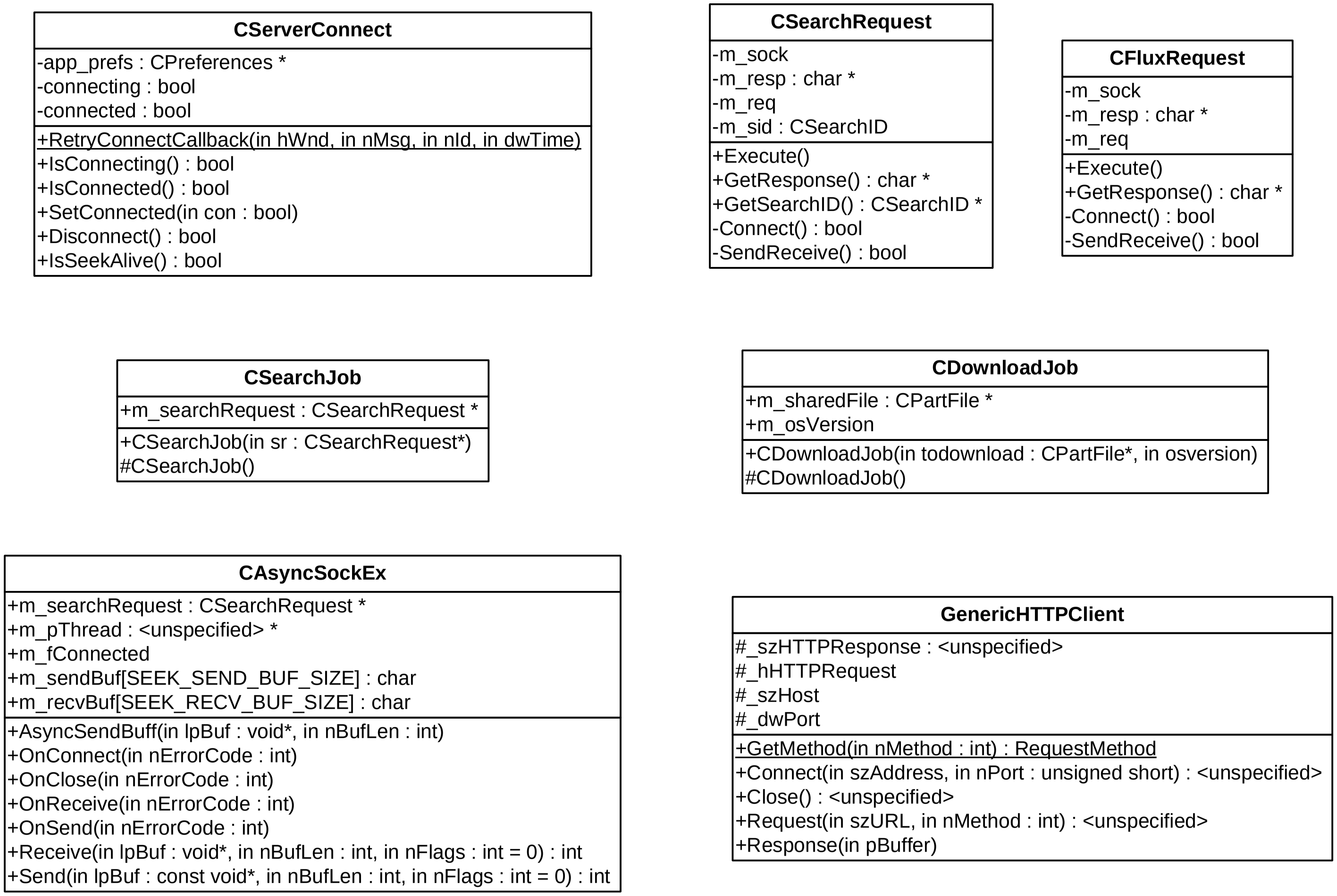


CWebWnd
+m_url
+m_browser : CLiquidBrowser
+m_wndToolBar : CToolBarEx
#OnDocumentComplete(in pNMHDR : <unspecified>*, in pResult : <unspecified>*) : void
#OnDownloadBegin(in pNMHDR : <unspecified>*, in pResult : <unspecified>*) : void
#OnProgressChange(in pNMHDR : <unspecified>*, in pResult : <unspecified>*) : void
#OnDownloadComplete(in pNMHDR : <unspecified>*, in pResult : <unspecified>*) : void
CSearchDlg
+m_search_states
+m_searchlistctrl : CSearchListCtrl
-m_search_counter
-m_active_search : CSearchID
+RefreshUI()
+ClearSearchFields()
+DownloadSelected()
+DeleteSearch(in nSearchID)
#StartSearch(in via_next : bool = false) : <unspecified>
#FastForward() : bool
#Rewind() : bool
CPreferencesDlg
+m_wndGeneral
+m_wndServer
+m_wndDownloads
+m_wndSearching
+OnInitDialog() : <unspecified>
+Localize()
+SetPrefs(in in_prefs : CPreferences*)
CResizableDialog
-m_bEnableSaveRestore
#EnableSaveRestore(in pszSection, in bRectOnly : <unspecified> = 0)
#GetResizableWnd() : <unspecified> *
#OnSize(in nType, in cx : int, in cy : int) : void
CSharedFilesWnd
+m_sharedfilesctrl : CSharedFilesCtrl
#OnInitDialog() : <unspecified>
CTaskbarNotifier
+m_strCaption
+m_rcText
+m_bMouseIsOver
+Create(in pWndParent : <unspecified>*) : int
+Show()
+Hide()
CTransferWnd
+m_downloadlistctrl : CDownloadListCtrl
+Localize()
#SetInitLayout()
#GetItemUnderMouse(in ctrl : <unspecified>*) : int
CSplashEx
#m_pWndParent : <unspecified> *
#m_bitmap : CEnBitmap
+Create()
+Show()
+Hide()
#DrawWindow(in pDC : <unspecified>*)
CemuleDlg
+m_preferenceswnd : CPreferencesDlg
+m_transferwnd : CTransferWnd
+m_sharedfileswnd : CSharedFilesWnd
+m_searchwnd : CSearchDlg
+m_webwnd : CWebWnd
+m_activewnd : <unspecified> *
+m_statusbar
+m_toolbar : CLiquidToolbarCtrl
+m_notifier : CTaskbarNotifier
+m_anim
+Localize()
+ShowNotifier(in Text)
+SetActiveDialog(in dlg : <unspecified>*)
-PartitionStatusbar()
CDownloadListCtrl
-listcontent
+AddFile(in toadd : CPartFile*)
+RemoveFile(in name : char*, in size) : bool
+CancelAll()
+PauseAll()
+ResumeAll()
+ClearCompleted(in force_clear_all : bool = false)
CLiquidToolbarCtrl
+classCLiquidToolbarCtrl
#m_ImgList
#m_bmpBackground
#m_cbBackBrush
+CLiquidToolbarCtrl()
+~CLiquidToolbarCtrl()
+OnEraseBkgnd(in pDC : <unspecified>*) : <unspecified>
+Init()
+Localize()
CLiquidBrowser
+m_container : CWebWnd *
+OnDocumentComplete(in URL)
+OnDownloadBegin()
+OnProgressChange(in progress : int)
+OnDownloadComplete()
CMuleListCtrl
#m_Name
#m_bCustomDraw : bool
#m_crWindowText
+SetName(in lpszName)
+Update(in iItem : int) : <unspecified>
+RestoreDefaultLayout()
+SelectAll()
CHyperLink
#m_bVisited
#m_strURL
+SetURL(in str)
+SetVisited(in bVisited : <unspecified> = 1)
#OnClicked() : void
CCreditsDlg
#m_strCaption
#m_rcText
-m_imgSplash : CEnBitmap
+Create(in pWndParent : <unspecified>*) : int
+Show()
CClosableTabCtrl
+m_bCloseable : bool
#OnLButtonUp(in nFlags, in point)
CSharedFilesCtrl
-m_filelist : CSharedFileList *
+Init()
+ShowFileList()
+LaunchExplorer()
+OnKeyDown(in nChar, in nFlags) : void
CSearchListCtrl
-m_searchlist : CSearchList *
-m_nResultsID : CSearchID
+AddResult(in toshow : CSearchFile*)
+RemoveResult(in toremove : CSearchFile*)
+ShowResults(in nResultsID : CSearchID*)
+ShowStatus(in res_string)

Threading


CDownloadJob
+m_sharedFile : CPartFile *
+m_osVersion
+CDownloadJob(in todownload : CPartFile*, in osversion)
#CDownloadJob()
«struct»IJobDesc
CSearchJob
+m_searchRequest : CSearchRequest *
+CSearchJob(in sr : CSearchRequest*)
#CSearchJob()
CSearchProc
-ProcessJob(in pJobDesc : IJobDesc*)
«struct»IWorker
+ProcessJob(in pJob : IJobDesc*)
CDownloadProc
-ProcessJob(in pJobDesc : IJobDesc*)
CSearchThread
+classCSearchThread
-m_server : void *
-m_searchRequest : CSearchRequest *
#CSearchThread()
+Run() : int
+InitInstance() : <unspecified>
+SetServer(in server : void*)
+SetRequest(in sr : CSearchRequest*)
+GetServer() : void *
+GetRequest() : CSearchRequest *
CThreadPool
#m_hMgrThread
#m_nNumberOfStaticThreads
#m_nNumberOfTotalThreads
#m_threadMap
#m_critical_section
+Start(in nStatic : unsigned short, in nMax : unsigned short)
+Stop(in bHash : bool = false, in pWorker : IWorker* = 0)
+ProcessJob(in pJob : IJobDesc*, in pWorker : IWorker*)
#GetThreadPoolStatus() : ThreadPoolStatus
#ChangeStatus(in threadId, in status : bool)
#AddThreads()
#RemoveThreads()
#CreateThread() : <unspecified> *
#SetThreadData(in pIJob : IJobDesc*, in pThread : <unspecified>*)
CDownloadThread
+classCDownloadThread
-m_tpool_server : void *
-m_sharedFile : CPartFile *
#CDownloadThread()
+Run() : int
+InitInstance() : <unspecified>
+SetServer(in server : void*)
+SetFile(in todownload : CPartFile*)
+GetServer() : void *
+GetFile() : CPartFile *
CSearchThreadPool
+CreateThread() : <unspecified> *
+SetThreadData(in pIJob : IJobDesc*, in pThread : <unspecified>*)
CDownloadThreadPool
+CreateThread() : <unspecified> *
+SetThreadData(in pIJob : IJobDesc*, in pThread : <unspecified>*)
+BeginExtreme(in file : CPartFile*)
+EndExtreme(in file : CPartFile*)
+GetExtremeThreadHandle(in file : CPartFile*) : <unspecified>

Networking



CServerConnect

-app_prefs : CPreferences *
-connecting : bool
-connected : bool

+RetryConnectCallback(in hWnd, in nMsg, in nId, in dwTime)
+IsConnecting() : bool
+IsConnected() : bool
+SetConnected(in con : bool)
+Disconnect() : bool
+IsSeekAlive() : bool

CSearchRequest

-m_sock
-m_resp : char *
-m_req
-m_sid : CSearchID

+Execute()
+GetResponse() : char *
+GetSearchID() : CSearchID *
-Connect() : bool
-SendReceive() : bool

CFluxRequest

-m_sock
-m_resp : char *
-m_req

+Execute()
+GetResponse() : char *
-Connect() : bool
-SendReceive() : bool

CSearchJob

+m_searchRequest : CSearchRequest *

+CSearchJob(in sr : CSearchRequest*)
#CSearchJob()

CDownloadJob

+m_sharedFile : CPartFile *
+m_osVersion

+CDownloadJob(in todownload : CPartFile*, in osversion)
#CDownloadJob()

CAsyncSockEx

+m_searchRequest : CSearchRequest *
+m_pThread : <unspecified> *
+m_fConnected
+m_sendBuf[SEEK_SEND_BUF_SIZE] : char
+m_recvBuf[SEEK_RECV_BUF_SIZE] : char

+AsyncSendBuff(in lpBuf : void*, in nBufLen : int)
+OnConnect(in nErrorCode : int)
+OnClose(in nErrorCode : int)
+OnReceive(in nErrorCode : int)
+OnSend(in nErrorCode : int)
+Receive(in lpBuf : void*, in nBufLen : int, in nFlags : int = 0) : int
+Send(in lpBuf : const void*, in nBufLen : int, in nFlags : int = 0) : int

GenericHTTPClient

#_szHTTPResponse : <unspecified>
#_hHTTPRequest
#_szHost
#_dwPort

+GetMethod(in nMethod : int) : RequestMethod
+Connect(in szAddress, in nPort : unsigned short) : <unspecified>
+Close() : <unspecified>
+Request(in szURL, in nMethod : int) : <unspecified>
+Response(in pBuffer)

3.2 Data Requirements

Since LiquidLan is not a scientific application or data-processing system, it does not require any substantial data collection. However, if the file shares are viewed as data, then such data must be available for the system to be useful. For example, if every node on the network is behind a firewall then LiquidLan will not be able to do its job.

3.3 Hardware

The hardware requirements for using the LiquidLan client include a computer with at least 64MB system memory, a 300 MHz x86 processor (for running Windows), and a network interface card. The server requires hardware capable of running Linux or BSD. For non-trivial networks, the server should have at least 128 MB memory, a 500 MHz CPU, and a network interface compatible with the underlying network.

3.4 Testing Methods

I will undoubtedly continue to use many of the testing and debugging tools included in Microsoft Visual Studio. I may also choose to use methods outlined in the ISO 9126 standard for evaluating the reliability, efficiency, and robustness of LiquidLan.

3.5 Scheduling Diagrams

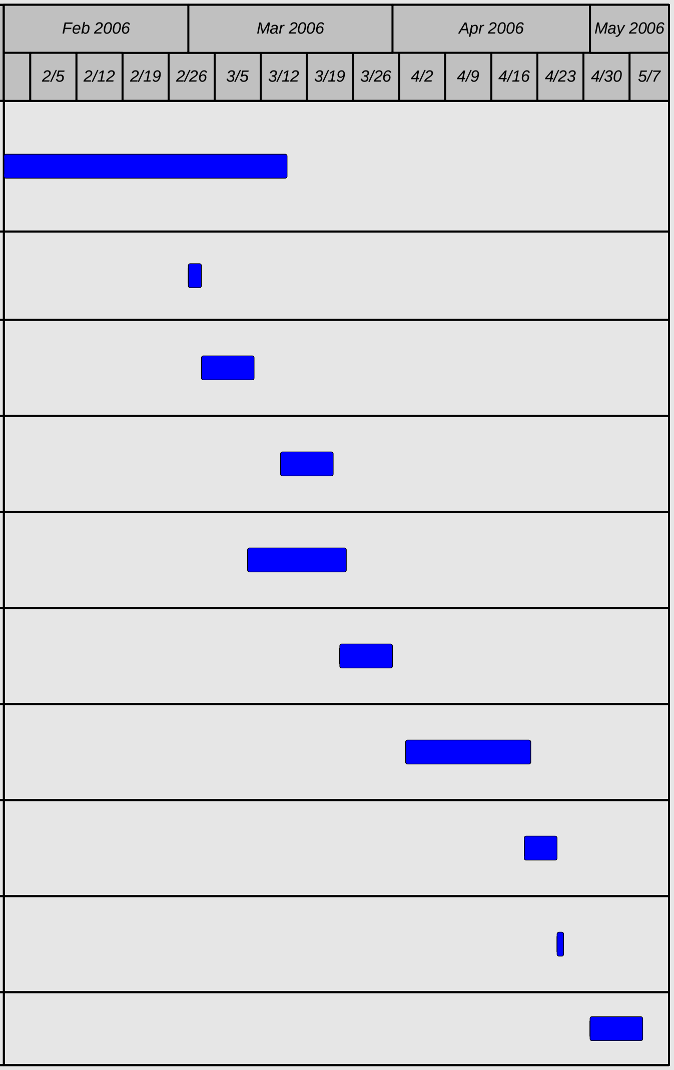
I have compiled a Gantt chart (see next page) to show an approximation of the schedule that was followed during design and development of LiquidLan.

3.5 Scheduling Diagram

LiquidLan Development

Gantt Chart

ID	Task Name	Start	Finish	Duration
1	Define the problem and research it. Draft the problem statement, and requirements analysis, and design specification.	2/1/2006	3/15/2006	6.2w
2	Build server, install Linux and development tools.	3/1/2006	3/2/2006	.4w
3	Re-familiarize myself with client and server source codes and frameworks.	3/3/2006	3/10/2006	1.2w
4	Prepare the initial class presentation.	3/15/2006	3/22/2006	1.2w
5	Finish extending the server and update Samba patches (move to 3.x)	3/10/2006	3/24/2006	2.2w
6	Finish development of DLCP admin interface and request processing.	3/24/2006	3/31/2006	1.2w
7	Finish implementing the client and add DLCP support. support.	4/3/2006	4/21/2006	3w
8	Test system thoroughly. Fix bugs. Make refinements. Make it production-ready!	4/21/2006	4/25/2006	.6w
9	Present and demonstrate final project (for committee).	4/26/2006	4/26/2006	.2w
10	Write the final report.	5/1/2006	5/8/2006	1.2w



4. Performance, Testing, and Evaluation

LiquidLan performs a lot of tasks that involve heavy network and disk I/O. Since network and disk access both tend to be system bottlenecks, it is extremely important for LiquidLan to have outstanding overall performance. Due to LiquidLan being implemented in C/C++ and LANs typically having high-speed/low-latency data links, this has not surprisingly been the case. File transfers are facilitated through the Windows API, which result in highly efficient file transfers. LiquidLan can even enforce transfer quotas (determined by the server operator) that limit the number of concurrent transfers to reduce disk trashing and network congestion. LiquidLan has other policies that can be adjusted to optimize performance for a given network.

For testing and evaluation, I will continue to use many of the testing and debugging tools that ship with Microsoft Visual Studio 2005. I may choose to use methods outlined in the ISO 9126 standard as well for evaluating the functionality, reliability, usability, efficiency, maintainability, and portability of the system. Since I will be using SourceForge to distribute LiquidLan, I can also use the tools they provide (such as download statistics, bug tracking, feedback, etc.). Finally, I can benchmark the performance of my system, and compare it against the performance of similar applications.

5. Summary and Conclusions

I am very pleased with how LiquidLan turned out. Windows/Samba networks have been around for many, many years, but have always lacked a way to be efficiently searched. The main purpose of LiquidLan is to provide a fast search mechanism and feature-rich download management tools. I wanted to create a powerful file sharing system for Windows Networks, and I believe I have succeeded in doing so.

I am proud of the software engineering skills I utilized while developing LiquidLan. In the final week I tried diligently to crash LiquidLan, running as many as 100 concurrent transfers at high throughput. No matter how much I tortured the client, it would not crash! I am very proud of this fact since parallel programming is regarded as a difficult task.

6. Future Work

LiquidLan is currently a working prototype only, so there is still a lot left to do. On top of the *To-Do* list is adding the ability for the client to programmatically create public, read-only file shares for users. This will require traversing the user's file system for media to share, asking the user which directories are OK to share, and then using an API to share the files out with the proper security settings.

Another important feature that is yet to be implemented is opt-in support for the server. This would tightly integrate with the client so that the user could opt-in through the client's user interface, and even force the server to re-index the client machine's shares.

In the future I also want to add an auto-update mechanism, IPv6 support, MBCS support (TCHAR conversion), improved Digital Rights Management (DRM), and last but not least the option to use an un-patched *smbclient*.

References

1. Jun, S., Ahamad, M., Incentives in BitTorrent induce free riding, *ACM SIGCOMM workshop on Economics of peer-to-peer systems*, 2005, pp. 116-121

2. Gummadi, K., Gummadi, R., Gribble, Ratnasamy, Shenker, Stoica, The impact of DHT routing geometry on resilience and proximity, *ACM conference on Applications, technologies, architectures, and protocols for computer communications*, 2003, pp. 381-394.

3. Kusumoto, T., Kunichika, Y., Katto, J., Okubo, S., Tree-based application layer multicast using proactive route maintenance and its implementation, *ACM SIGCOMM workshop on Advances in peer-to-peer multimedia streaming*, 2005, pp. 49-58.

4. Tari, Z., McKinlay, M., Malhotra, M., An XML-based conversational protocol for Web services, *Proceedings of the 2003 ACM symposium on Applied computing*, March 2003, pp. 1179-1184

5. Hamra, A.A., Felber, P.A., Design choices for content distribution in P2P networks, *ACM SIGCOMM Computer Communication Review*, October 2005, pp. 29-40

6. Wikipedia, 2006, http://en.wikipedia.org/wiki/Distributed_hash_table