IN-2194 Project: Distributed Search

1 General Information

The goal of this project is to implement a Peer-to-Peer Search Engine. You should read up on `http://yacy.net/` for an existing design.

You must submit your solutions using Subversion by **August 15th 2011**. Late submissions will receive a score of zero unless you notify the instructor about a valid reason (such as the entire group having been abducted by pirates) **prior** to the deadline.

1.1 Subversion Access

In order to receive Subversion access, you must submit a request at

`https://projects.net.in.tum.de/projects/tum2194/register`.

In this request you must include the login name and desired password. Your login name should be your lastname followed by the last two digits of your student identification number.

Once your account(s) have been created and teams have been assigned, you can access the repository at

`https://projects.net.in.tum.de/svn-ext/tum2194/TEAMNAME/`.

You must submit a request for account and team creation by **Mai 15th 2011**. We want you to use Subversion for development, not just submission! Note that only the version of your code that is HEAD in the repository on August 15th 2011 at 23:59 will be graded.

1.2 Coding Style

You should read the GNU coding standards for a reasonable guideline for writing good C code. In general, a shorter and simpler implementation will receive higher marks than a complex and long implementation. Naturally, comments and testcases will not be counted against you when we evalute the size of the code (in fact, good testcases and comments may earn you style points).

Your implementation must not perform busy-waiting and must not use pthreads (!). You should also try to avoid (unnecessary) copying of data. You should stick to the GNUnet coding style as much as possible (if your implementation would be good enough to be merged without significant changes into the GNUnet code base itself, you will get full points).

2 The GNUnet Framework

Your implementation should work in the context of the GNUnet P2P framework. Specifically, you will be using the 0.9.x development branch which can be found at `https://gnunet.org/svn/gnunet/`. Some documentation is available online at `https://gnunet.org/`. The framework already contains a DHT, including a C API `src/include/gnunet_dht_service.h` that is used by other components to access the DHT and command line tools (`gnunet-dht-put` and `gnunet-dht-get`).

In GNUnet, each component is executed as a separate process. Your implementation will use the *core* service and/or the DHT to communicate with other peers. You can use the *peerinfo* service to bootstrap (discover peers initially).

Each of the services listed above provides a service access library (`libgnunetSERVICE.so`) to communicate with the service process that actually performs the operations. For this, the service process listens on a socket for requests from the service access library. Service processes are managed by the *arm* service.

For your own services, you need to write a service access library. For your implementation, you can choose to write the service process in C/C++ or in any other language of your choice. However, the various access libraries (to *core*, *dht*, *statistics* and others are currently only available in C. So if you choose a language other than C/C++, you may have to write a binding for those access libraries or re-write them entirely. Furthermore, if you write your service in C, you can use the `libgnunetutil` library to simplify your handling of the network processing.

Finally, you can ask questions not answered in the FAQ on the webpage in the `#gnunet` IRC channel on `irc.freenode.net`. We will answer non-FAQ questions in the public channel quickly. We will not answer questions about how to use GNUnet asked elsewhere.

3 Choices

Implementing a complete search engine would seem to be a task that is likely somewhat too big for a single team for a single course. Hence we decided to break up the implementation into components that can be independently implemented and tested. Teams are free to choose which component they want to work on; while we hope that each component will be implemented nicely by at least one team, the primary objective for each team should be to identify a component that they can actually complete.

We are providing a suggested API for each component. You are allowed to extend the API with additional functions if you feel this is necessary; however, in order to ensure that components can be combined in the end, all changes to existing functions require discussion and consent from all students present in the lecture.

3.1 Index: Local storage management

This component is about managing local data, including support for queries over collected results. The team should be familiar with databases such as sqLite, MySQL or PostGres, including accessing these databases using the respective C/C++ APIs. **Hard problem:** Peers should communicate with each other about which items which peers are storing to control replication. Part of this component is also implementing the shared library to access the storage service.

An in-memory storage implementation (but with the shared library for access to the service that provides in-memory storage via IPC) would be acceptable for a single-student team.

For testing, other teams are expected to implement a version of the database API that returns hard-coded sample data (and ignores storage requests).

3.2 Spider: HTTP access

This component is supposed to download sites using `libcurl`, parse HTML using `libtidy` to produce full-text for indexing and additional URLs for spidering. This component will need a mock of the `datasearch` API. URLs should be exchanged with other peers to distribute the load. Naturally, the same host should ideally only be spidered by a small set of peers (some redundancy might be good to address security and reliability concerns). **Hard Problems:** Open technical challenges include distributed prioritization and load balancing (which peer does which URLs in what order). You are also encouraged to support limiting bandwidth consumption by this component.

For testing, other teams are expected to populate their database with test data.

3.3 Crawling

This component is supposed to answer search queries by executing a distributed crawl over the individual local databases. **Hard Problems** include prioritization of requests, limiting the spread of the crawl and ranking of results. This component will need a mock implementation of the `datasearch` service. Performance is also more critical for this component compared to others. Part of this component is also implementing the shared library to access the crawl service.

3.4 User-Interface

This component has the least to do with the actual P2P application. Your user interface should provide a WWW-interface; this component will need a mock implementation of the `crawl` API. Your team should know HTML, JavaScript and enough C code to call the search API and generate the website using GNU libmicrohttpd (a simple library for embedding HTTP servers). **Hard Problem**: Search results from a distributed crawl are likely to arrive incrementally, and possibly the ranking of a result may change after it is initially obtained. A nice user-interface would need to dynamically update the page without ever changing the URL that the user is actively trying to visit.

For testing, other teams are expected to write a command-line tool to query the search engine.

4 Grading

You are expected to form teams of two to three students, but you can also work alone. Teams of four and more students are not permitted. Within each team, you are free to determine which team member is best for which task. Your overall grade will be determined primarily by the overall quality of the deliverable, but we will also discuss your contributions with you during the final demonstration and make sure that individual students are not penalized if some team members failed to contribute.

For the first three subprojects, the grading will be based on the following main points:

3 API implementation correct (library-only)

3 API testcases exhibit good coverage

4 Library-Service protocol documentation

8 Service implementation works on a single peer (no P2P)

5 Service testcases exhibit good coverage for local functions

4 P2P protocol documentation

4 Technical soundness of the design of the P2P protocol

6 Correctness of the implementation of the designed P2P protocol

5 Interesting solutions to the hard problems

5 P2P testcases exhibit good coverage

For the user-interface subproject, grading will be based on the following main points:

15 Correctness of the HTTP server implementation

5 HTTP server testcases exhibit good coverage of the server code

5 HTTP server code documentation

10 Correctness of HTTP site (search, displays results)

5 Responsiveness of the web-site to new results

10 Look & feel of the generated site (images, style, etc.)