IN-2194 Project: Distributed Hash Table

1 General Information

For the course project, you are to implement a Distributed Hash Table (DHT) for GNUnet. The instructor will (randomly) create teams of four students for this project. We expect that teams will find a reasonable way to assign tasks such as coding, documentation, evaluation and testing among the members. We will ask the team if all members contributed equally to the project and consider the response when assigning individual grades.

There are six different DHTs that can be implemented. Some designs are more complex than others. Teams that choose simpler designs will be held to higher standards as far as correctness, documentation, evaluation and testing are concerned.

You must submit your solutions using Subversion by **July 15th 2010**. 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 2010**. 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 July 15th 2010 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 threads (!). 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 is good enough to be merged without significant changes into the GNUnet code base itself, you will get full points).

1.3 Testing

You should consider the following issues when testing your code:

- Cover the entire external API
- Local results
- Remote results
- Insufficient local space
- Insufficient bandwidth
- Peers joining (before and after API operations)
- Peers leaving (before and after API operations)

2 The GNUnet Framework

Your DHT 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://ng.gnunet.org/svn/gnunet/`. Some documentation is available online at `https://ng.gnunet.org/`. The framework already contains a basic harness for DHT implementations, specifically a C API `src/include/gnunet_dht_service.h` that is used by other components to access the DHT, command line tools (`gnunet-dht-put` and `gnunet-dht-get`) and a starting point for the implementation of the DHT service (`gnunet-dht-service.c`).

In GNUnet, each component is executed as a separate process. Your DHT will use the *core* service to communicate with other peers. You can use the *peerinfo* service to bootstrap (discover peers initially) your DHT (in conjunction with the *hostlist* daemon) . GNUnet uses HELLO messages to exchange information about peer's public keys and their network addresses. You can obtain your own HELLO from the *transport* service. Once you receive HELLOs from other peers, they can be given to the *transport* service for validation. After HELLOs have been validated, they will become available via the *peerinfo* service. Finally, you might find the *datacache* service useful to store and retrieve data that your node is supposed to keep locally.

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 DHT, a service access library is already provided. You "only" have to implement the service process itself. 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*, *peerinfo*, *transport* and *datacache*) 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 the DHT service process 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

Each team should give an order of preference among the following DHT designs. We will then assign each team a DHT to implement based on their preferences (and with the goal of avoiding all teams implementing the same design).

- CAN, as in: "A scalable content-addressable network."
- Pastry, as in: "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems."
- Chord, as in: "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications"
- Kademlia, as in: "Kademlia: A Peer to peer information system Based on the XOR Metric"
- RSG, as in: "The rainbow skip graph: a fault-tolerant constant-degree distributed data structure"
- Freenet, as in: "Distributed Routing in Small-World Networks"

Note that for most of these designs, the further papers have been published that added various extensions or made improvements to the original design. While you are free to implement such improvements, this is not expected. In any case, your documentation should state clearly what the final implement design is (with references to the original works as appropriate).

4 Grading

You are not free to form teams as you like. Instead, teams of four students will be assigned at random — just as in the real world. Furthermore, each team member should be declared responsible for one area.

Specifically, the areas are:

- Coding (DHT implementation, coding style, appropriate use of APIs, correctness)
- Testing (correctness evaluation, code coverage, unit testing)
- Documentation (most importantly, high-level view of the implementation and how it differs (or matches) the description from the research paper
- Performance evaluation (latency, bandwidth consumption, success rates, load balancing, handling of churn)

Teams of three students (should they exist) will be allowed to omit the performance evaluation part. 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 score for the area you were responsible for, and in smaller part by the overall quality of the deliverable. 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 fairly contribute.