Name: Ling, Tzu Hsuan (Sam)
Purdue ID: ling21

CHORD Distributed Hash Table

Introduction

The CHORD distributed hash table provides an efficient lookup API for applications utilizing the library. CHORD allows dynamic joins of servers for easier scalability and efficient lookup by consistent hashing. Furthermore, it provides feature, the finger table, that further enhances the efficiency of lookup operation.

I chose to implement a simplified version of CHORD from scratch, attempting to simulate many of the functions described in the paper by Stocia et al.[1], and implemented it as library source that C++ programs can link and import. In this report, I will describe briefly on the design decisions and examples as included in the source files. I will then attach a brief feedback on my opinion towards this algorithm.

[1] Referenced Paper - http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf

Name: Ling, Tzu Hsuan (Sam)
Purdue ID: ling21

Design Decision and Implementations

This section briefly describes the design decision and implementation of the simplified CHORD service. In this report, the word application is used to refer to the program that includes, links, and utilizes the CHORD library, and the word CHORD or Chord is used to refer to the library itself.

Parts Not Implemented

The source code was written in C++, attempting to implement the CHORD service as described in the paper. Due to time constraints, although it had been thought through during the code planning phase, the fault tolerant part and voluntary leaving nodes handling were not implemented.

Failure handling, if were implemented, would be in the form of extended list of successors tracking instead of the immediate succeeding node. Upon successor failure, the affected node will wait for a grace period until determining the death of the node. It will then select the next node from the list of successors as immediate successor and refresh the successor list to add another server. The list size should be configurable.

In addition to failure handling, adding heartbeat monitoring of the successor and predecessor was also planned. It was, however, not implemented in the end because this would not be useful if there would be no failure handling.

Notification System

The program is composed of one main thread that handle messages exchange and logistics, accompanied by an assistant thread that provides notification functions to the application.

Notification system (`include/ServiceNotification.hpp`) is currently composed only of new node join event. Normally, this would also include events where nodes crash or leave voluntarily, in order to update the application part so that it properly transfers files that the successor was responsible of. This notification component, ServiceNotification class, is an abstract base class allowing other classes to extend and implement notification handlings.

It provides a function for the application to listen for new notification, and another function that lets application retrieve and remove the notification (`hasNotification()` and `popNotification()`). The main Chord class extends this class publicly, provides a casting declaration to `popNotification()`, which returns a `ChordNotification` class object.

The `ChordNotification` class implements `Notification` class from the same header file and adds in members and functions.

Main Event Loop

In the main thread, an event loop is used to wait for new messages from the network channel. The thread is an implementation over pthread library, by the `ThreadFactory.hpp` class that encapsulates many useful functions into object-oriented format. This allows class instances to run thread over its member function without needing to extract the event loop out of the class and run it standalone. The event loop, will need to be implemented over `threadWorker()` method, and the class will need to extend the `ThreadFactory` class.

`ThreadFactory` provides basic functions for threading - `startThread()`, `detachThread()`, and `waitExit()`. These functions correspond to `pthread_create()`, `pthread_detach()`, and `pthread_join()` respectively.

Network Configuration and Properties

Chord will be running on a port configurable by the application, but this port must be also used for all other Chord nodes that will be deployed on the same network. On the other hand, application port can be arbitrary, and application will need to specify it when constructing a new instance of Chord class.

The service uses UDP with only necessary messages being made reliable. Most of the mission critical messages are acknowledge separated and timed, with retransmission on time out. A few exceptions are the periodically exchanged messages, such as `FingerQuery` and `StabilizeRequest`. These messages are not given reliability enhancement because they are exchanged every few seconds, which makes it redundant and may potentially hamper the performance of the system.

Message Handling

Under `include/MessageTypes.hpp` file, there is a list of messages structures being used to transmit across the network. Most of the message type have corresponding Request-Response pair, as well as their own type identifier.

Note that, although specified in the type identifier, there are no corresponding message structure for `MTYPE_FINGER_QUERY/RESPONSE` (used for updating finger tables) and `JOIN_SUCCESSOR_QUERY` (used for join operation). This is because these two types of message functions extremely similar to the `SuccessorQuery` and `SuccessorResponse` type, thus they use the same structure with different message type member. The `MessageHandler` class (`src/MessageHandler.cpp` and `include/MessageHandler.hpp`) are implemented to add in support functions for different type of messages. For example, serializing and unserializing messages for sending and receiving, different message type creator, and general message property retriever (`getType()` and `getSize()`). This makes the maintenance of the code rather simple, and the `MessageHandler` class can used as an abstract layer.

Miscellaneous

The service makes uses of simple utility functions (implemented as static functions in `include/Utils.hpp`) like `getTimeInUSeconds()` (gets time in microseconds since UTC, with larger seconds truncated because this function is used for micro timing), `getHostname()` (wraps the gethostname system to return a char array with the local or remote hostname specified by IP address), etc. The file also contains simple split function used in the sample app to split command and arguments from stdin in the application's event loop.

Error handling is provided by `ChordError` class (`include/ChordError.hpp`), extended by Chord main class and provides error numbering system. The implementation avoids using system error numbers (Linux `errno`) to avoid confusion over service error and system error. `ChordError` provides an error number to error string converter to print on screen, as well as a set of self-defining error variable names. It can be easily understood in code.

Name: Ling, Tzu Hsuan (Sam)
Purdue ID: ling21

`SampleApp.cpp`, since the only purpose is to provide an example on integrating this simplified Chord service, is not part of the sources and can be tampered with. It demonstrates how most of the Chord function can be used within an application.

Thoughts on CHORD

I think the structuring of Chord is very interesting. It forms the nodes into logical rings, albeit not uncommon among distributed algorithms and networking topologies, the way each node interacts with one another while keeping only “minimally necessary” information in the memory, as well as the use of finger table, is quite inspiring.

The way it uses of consistent hashing is also new to me, which brought me many issues during the implementation. For example, the 160-bit output from SHA-1 cannot be easily converted to C/C++ primitives; I had to use GNU Multiple Precision Arithmetic Library (GMP) to calculate modulo over a 32-bit bit size. At some point of time I was seriously considering moving to Python for its ability on handling large numbers (I still stayed with C++ due to time and my familiarity with this language over Python).

Overall, I enjoyed doing this project. Some of its concepts may seem convoluted in the beginning (e.g. how finger table is indexed), yet the level and amount of thoughts I poured into this project inspired me of many other thoughts on the distributed system architectures in association with what we have learnt in class.