SIPDHT: An Open Source Project for P2PSIP

Alex Fandrianto
Cisco Systems
08AlexF@students.harker.org

Abstract

SIPDHT fulfills P2PSIP's requirement of a distributed hash table arrangement. This article discusses the applications and improvements possible to the open source SIPDHT packages. Additionally, pointers to key code are provided. An extra section describes my work experience at Cisco Systems in the summer of 2007 as part of a 4 week internship before my senior year in high school.

1. Introduction

P2PSIP (Peer to Peer Session Initiated Protocol), the proposed model for serverless, computer to computer connections, faces the problem of how to store lookups and to arrange the peers in its network. By sharing the burden of data storage and removing the single point of failure, which is a central server, P2PSIP provides a fault tolerant and convenient system for message and file distribution. Thus, to provide an answer to these problems, SIPDHT (Session Initiated Protocol based Distributed Hash Tables), a project that has created a design for P2PSIP, takes the hash table of keys and values, normally stored in a central server, and distributes them among the nodes in the network. Coupled with the distributed hash tables is the arrangement of the networked peers.

SIPDHT originally followed the Chord algorithm when implementing its distributed hash table. However, with the new release in June 2007, it has switched from a ring arrangement of peers to a coordinate system arrangement called CAN (Content Addressable Network). When in 2 dimensions, this system is like a torus; the top wraps around to the bottom, as do the sides. Since this paper discusses the early second version of SIPDHT, its libraries are thus far not fully completed. The SIPDHT library forms the definitions of many structures. Peers hold a list of zones that they control and neighbor. These zones know two coordinates, the bottom left and top right one (for 2-D). Additionally, peers store their own address of record and their part of the hash table. Functions such as xpp_update, xpp_join, xpp_get, and the other basic sip operations are implemented here too. These functions connect the SIPDHT library to the other library, XPP. The Extensible Peer Protocol library provides the definitions of sessions and the protocol, which is then used by the Sofia Sip libraries. This last library is a user agent library that follows the IETF specifications for P2PSIP and is a separate open source project. It defines basic things like a url and takes care of the protocol for inviting, messaging, getting/putting, and updating. However, SIPDHT2, as it is called, is still quite functional and leaves a lot for the open source developers to work with.

2. SIPDHT's Implementation

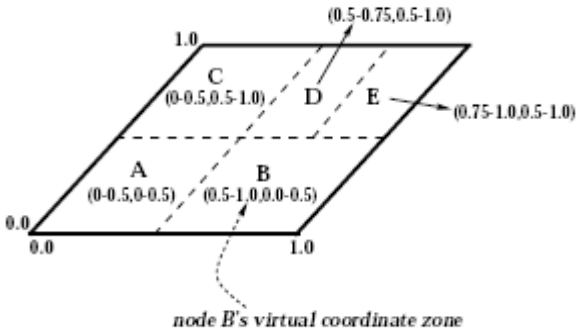

Figure 1: *Example 2-d space with 5 nodes*

[3]

As mentioned earlier, he SIPDHT libraries use a CAN (Content Addressable Network) implementation for its distributed hash table as described in "A Scalable Content Addressable Network" [3]. A virtual coordinate system for the peer to peer overlay is set up upon the first peer being created. That peer takes control of the whole overlay and stores the hash table entries in the form of a zone. Basically, two points are stored for a zone, in 2-D it is the bottom left and top right corner, where the lower left of the coordinate system is (0,0). When another node is created, it must know the IP address of a peer currently in the system. That node will be the bootstrap node and is able to set up a zone for the new peer. A JOIN and INVITE message is called via the application, and a peer is chosen to invite the new peer.

While invitation can be done at a random point in the coordinate system, the application can choose to make it more systematic. At any rate, the peer selected to invite the new peer will either split its zone or send an extra, if it has one. Zones are split in an alternating fashion (for example, vertically, then horizontally) to optimize the ability to merge zones and to maximize number of neighbors. When splitting, the one part of the new zone is kept by the peer, and the second is sent on to the new peer. Every old neighboring zone is told to update itself. Then the peer updates it's zone for its neighbor list as well as the new peers. Thus, the updates upon joining the system are very limited; they only affect neighboring peers. Figure 1 shows 5 peers sharing a 2-D coordinate system.

The overlay maintains itself through timer induced checks. Each zone at different points in time will have their timers run out. They will check their neighbors to see if they are still alive. If the neighbor's zone is unresponsive, the peer will initiate a TAKEOVER message. The neighboring peers will then decide amongst themselves who will take over the vacant zone. Sometimes, zones cannot be merged during the takeover and two distinct zones are kept in control of one peer. Sometimes, if many nodes fail, there may be problems in taking over the freed up zones. To prevent such data unavailability, there are improvements that increase fault tolerance, such as multiple dimensions or zone overloading.

3. Uses of SIPDHT

The SIPDHT library is very versatile though it cannot be used until its code is implemented at

the application level. Thus far, messages are limited to only text messaging. However, it will NOT involve changes in the protocol or the library to store and send sound files, image files, and so on. Instead, all that needs to be done is to change the application level implementation. Also, the supposed data stored in each zone does not really exist yet. In theory though, hashes are all SHA1, so a resource could still be found on the coordinate system through a hashing of the filename.

The csipdht2 application is a command line version of a peer. It can be networked with other peers on different or the same computer. Once, all have been invited into the overlay, they may send messages to each other and check how much of the zone they control. The gsipdht2 application is simply a graphical version of csipdht2. The difference here is that invitations are more difficult and are almost hidden from the user. The graphical user interface also supports the spawning of new peers, which are automatically invited into the overlay. Both of these applications supports the getting and putting of keys, uses a timer that counts down while the program runs, and allows messages to be sent between nodes.

The gsim-sipdht2 application was modified to allow for more uniform partitioning. Before, zones were split according to which peer was selected at the time. The selected peer would always split its zone in half or send an extra zone (if it had one) to the joining node. After the change, the simulator consults its running list of peers in the overlay and selects the peer with the most zone volume to split or send one of its zones. The result is the desired reduced variability in the number of connections and zone volume. Not much can be done here but to watch how nodes split and maintain connections with their neighbors in 2 dimensions.

4. Improvements to SIPDHT

Improvements to the current CAN design are implemented in the application level, but most require changing the library itself. Improvements in application implementations include a more even partitioning of nodes. As discussed later, it will increase the efficiency and equality of peers in the overlay. Another change would be multiple dimensions being used for the coordinate grid. It allows for extra neighboring zones, which provides for reduced path lengths and a very synergistic improvement. However, the downside of these improvements is either an increase in the data stored or an additional delay as extra functions are called.

These downsides also apply to improvements that can be made in the library code itself. These improvements are geared towards efficiency and fault tolerance. Multiple realities, or coordinate systems will allow a sort of teleportation towards the goal zone. During queries of a hashed filename, nodes would look at their zones in each reality before routing a message to the closest neighbor. To reduce node to node latencies, a timing system implemented for each node allows for round trip times between peers to be recorded. Then peers will forward messages to reduce the time spent through a greedy algorithm. Since this RTT strategy will reduce latency, the time between zone to zone jumps, overall, queries will go much faster. Fault tolerance can be increased by zone overloading; multiple peers may share pointers to the same zones, giving increased IP selection and backup of data. RTT can be used here to select the appropriate peer in the overloaded zones. Or instead of zone overloading, the overlay could multiply certain key value entries so that it is more likely that neighbors will have the desired value.

5. Uniform Zone Partitioning

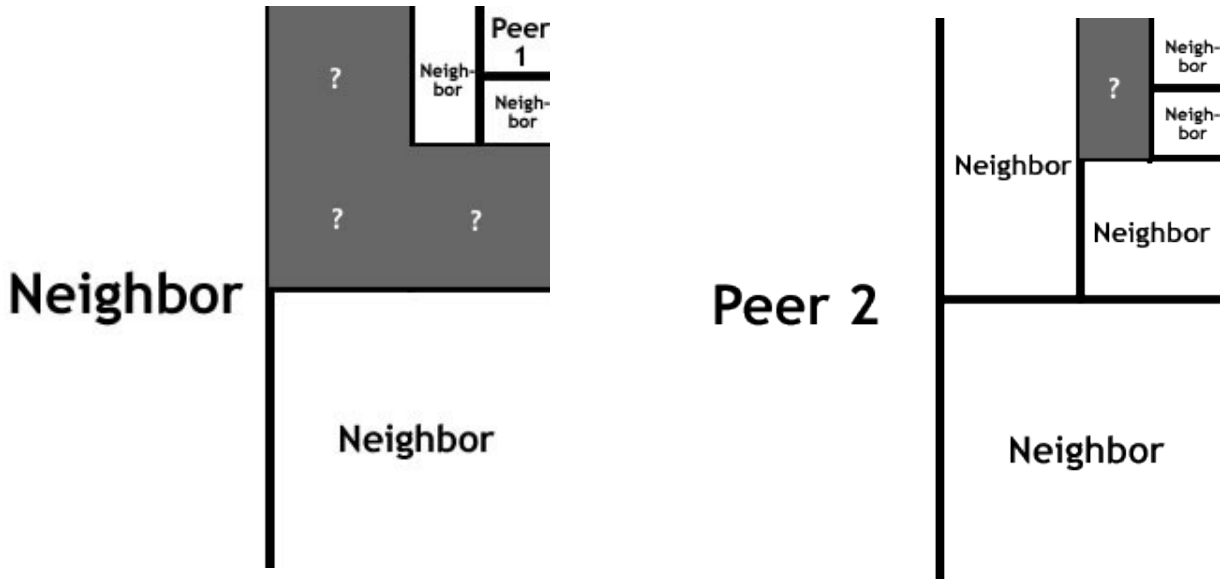

Figure 2: Result of Continual Splitting of Peer 1.
Left: Peer 1's zone and neighbors. Right: Peer 2's zone and neighbors

Without uniform partitioning, it is less likely that peers throughout the network will have equally sized zones (especially if there is a deterministic placement of peers based on IP address). For instance, if peers are always split at the top right most point of the coordinate system because there are many peers of the same IP address joining, there will be varying amount of neighbor zones for each peer that will always average to be much greater than 8 especially for non-hotspot areas. Similarly, zone volume will be unequal with larger zone volume corresponding to a larger number of neighbor connections. As Figure 2 shows, the mode number of connections is either 4 or 5 with this uncontrolled splitting, but in the larger, unsplit zones, they may have 12 or above connections since further splits in different zones almost always adds an extra neighbor to their lists. As can be seen in Figure 2, Peer 2 knows almost the whole coordinate system; clearly, most splits will add a neighbor to Peer 2 This causes unfair data storage, and an overall slower network since only a few nodes do all the work and on average, many nodes must be crossed to reach a desired zone.

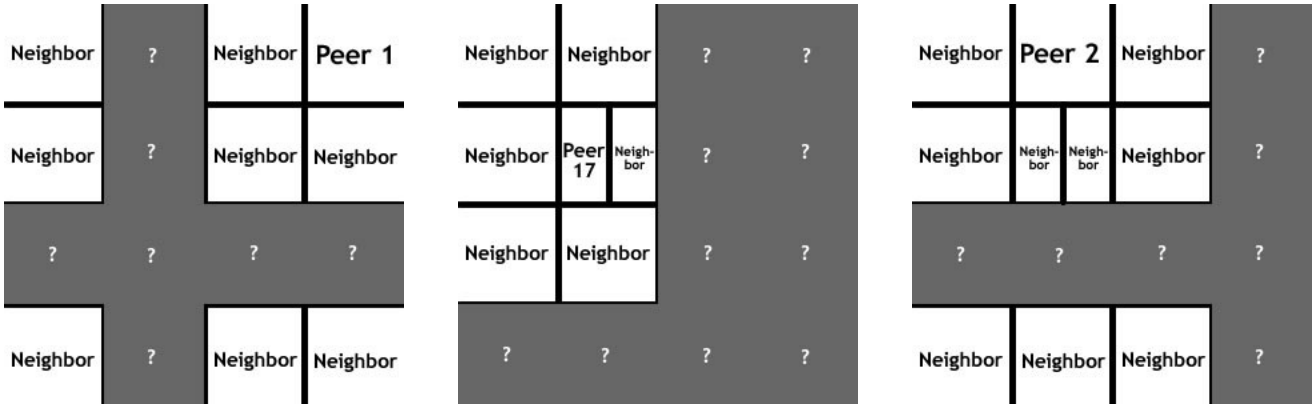

Figure 3: More Uniform Zone Partitioning in Action

For the cost of storing a list of peers in the application itself, connections can be made into an

average of 8 by splitting the largest peer when inviting a new peer. As Figure 3 (left) shows, the average is 8 connections because each corner and each touching side makes for a neighbor. When splitting, the zone split and the new zone will have 6 neighbors (or 2 less) (Figure 3, middle). The zones above and below the split will have 1 more neighbor, making an equal split situation far superior to the current SIPDHT model (Figure 3, right). This situation fairly splits the amount of data that must be stored; all peers are treated equally. At the same time, having 8 connections between each node optimizes the time it takes to reach a desired zone (and hence, node) for information. Also, there are various extra ways to improve the latency of zone to zone hops because there are on average, 3 other nodes that can be chosen for the hop and 3 different IP's that may be similar to the current node's IP compared to 2, for the unequally split zones.

To split zones equally, the list of peers is consulted. The peer with most zone volume is selected to split or send one of its zones. This can apply to only neighboring zones, or can be extended further to ask for the neighbor's neighbors. The amount of recursion may be increased for a more even spread in the amount of connections stored at the cost of longer waiting time; increasing from O(1), about 8 connections checked, to O(n), all connections checked. There may actually be no way to prevent repeat checks if only neighbor peers are checked recursively. Perhaps CHORD would be superior to CAN in that searches can simply go around the whole ring, half the ring, a quarter, or less. At any rate, such splitting improves the equality and fairness of the CAN system.

6. Pointers to SIPDHT Code

Since it may be difficult to browse through the code for specific implementations, a bit of guidance may be found here. One may find the implementation of the distributed hash table in xpp.c and peer.c in the libsipdht directory. Invite, join, and the function for peer creation can be found there. For the main header files, look in libsipdht/include. There, structures such as sipdht_peer_t and czone_t are defined. Some important things to note are that peers do not have pointers to other peer structures, only the zones themselves. Communication occurs through the address of record instead and through xpp sessions. To have peer to peer access, the application itself must keep its own running list of peers in the overlay. While this fact may severely limit the application level implementations, it is important for privacy and security; malicious applications would otherwise be able to disrupt the function of other peers.

7. My Experiences with C, Linux, and SIPDHT

Upon my arrival at Cisco Systems, I was not quite sure what to expect. Well, I was expecting to be programming simple algorithms in C, a programming language I had just read about myself a few weeks before. While I was reading on how to program in C, I was quite lost. Who really cares about longs and shorts? It's not like you'll ever need to use a short or a long specifically. Just use ints, right? I was in for quite a shock. Not only for the C programming aspect but also for what lay in store for me.

As is usual for me, I was quite shy on my first day, spending the first hours installing the C compiler I was given. When it finally installed, I wrote a few short programs. On the second day, I was told to look up P2PSIP and was impressed (but mostly intimidated) by the idea of working with other open source programmers on the future in computer technology. I found the SIPDHT libraries and extracted them. But there was nothing to run, no executable file. And then I was told to use a Linux machine....

At first, Linux seemed like a ridiculously complicated and useless operating system. Who wouldn't want to use Windows? Windows can do everything that Linux does and then play GAMES. All you can do here is GUESS what the terminal commands are. In fact, it was such a daunting shift

that it took days for me to willingly use the Linux machine I was assigned. When I was told to install a virtual Linux machine on my computer, I was extremely reluctant to agree. Fortunately, my objections to Linux were quite mistaken. Linux replaces Windows with the versatility of an open source system. It is continually improved and adored by open source programmers. Games are quite insignificant when compared to the masses of available open source software being developed. The future of computer technology is only available to Linux.

SIPDHT, similarly, was difficult to get used to. After finding my first open source library, I wasn't really sure where to start. I had simply arrived here by using Google and searching for possible implementations of P2PSIP. And when added to my Linux troubles, I was not quite sure what to do with it. I noticed a bunch of .c files and .h files. Then I noted that they weren't compiled yet. I tried to compile on my laptop with the new C compiler, and I watched as it failed miserably. If it could not be installed on my computer, I wondered how SIPDHT would ever be installed on Linux. There were no directions for this sort of thing. Moving to the Linux machine, I was able to find the cc command. From the vague generalizations in the web, I managed to enter the command on the Linux machine. No such luck, the same errors popped up. Going back to the website, I noticed that there were specific compiler instructions. ./configure, make, and make install. Luckily, my new attempt produced the error messages that were much clearer. Apparently the Sofia-sip libraries had to be installed first.

With the Sofia-sip libraries installed, I was at an impasse. SIPDHT simply could not find the installed library file. For a few days, I moved the .la file around the Linux machine into different directories. Finally, after my experience with Linux increased, I realized that the Library PATH file might not have been set to any directory at all! After a few moments, I found the directions to set the PATH and was able to install SIPDHT by the end of that day.

Linux would not let me go so easily of course; the next step was to actually execute the programs. But the executable files were not in the directory I had installed. Even after a lengthy search, I could not find anything. I concluded that there had to be some sort of terminal command to run them. But what was it? There was no help anywhere to be found. I'd looked on the SIPDHT website itself, and there were only vague directions that actually applied only to the previous, Chord version of SIPDHT. In the end, the README that came with each application gave a sort of description of how to run the applications.

It was depressingly obvious to me that even after a few days no progress had been made on my assigned project, to work on P2PSIP code. In fact, I wasn't even sure what these applications did or how they worked. At first, I decided that the gsipdht, the graphical sipdht peer application was the most useful. I could actually understand and see most of the node's functionality. After a few hours, I was able to have quite a nice 3 node chat with myself. Quite excited, I printed out the code for each application.

Scanning code is no fun at all. To tell you the truth, after a day of it, I was quite bored. There were so many structures, and I had not a clue of what they were. That is, until I identified what where the message sending part was. From there, I was able to work myself forward, and with the aid of my computer's new compiler, look up the definitions of each structure. That second day, I had a sudden realization; C was in fact quite like Java. Classes were analogous to structures. Objects calling methods were simply functions being passed a pointer to a structure. And booleans were only ints that were either zero or non-zero. Discovering the connection between Java and the original programming language gave me quite a morale boost, spurring me to search deeper, understanding the structures and functions with ease.

The problem with coming into a new field is that one does not understand all the vocabulary and terms in it. I've tried my best so far do define everything, but I must have come up short

somewhere. In most technical papers describing P2PSIP and SIPDHT, most were not friendly. I had to bear with it, and work my way up. I entered reading and left with general ideas of what was meant by P2PSIP. But then I found the paper describing CAN. It was easy to read and understand; I swiftly understood how the distributed hash table worked, how the zones fit in, and what the peers were saying to each other.

Last week, I decided to write changes to SIPDHT based on the suggested improvements that the CAN article described. Uniform zone partitioning was what I selected. However, as simple as it might have seemed, I was not prepared for the shocking revelation that peers did not have pointers to other peers, and that the zones of peers did not have access to those who owned them. “Why?!” I wondered helplessly. Then I attempted to simply split a zone without the consent of its owner. Well, it worked for most of the peers, according to the graphic simulator for SIPDHT. Unfortunately, the node whose zone was split did not realize the difference; I had to have access to its hashtable, which was only reachable if I had a pointer to the peer itself. I had almost lost hope for the algorithm and alternate algorithm I had written, for each strategy had failed me. In truth, I had been led in this direction by the sight of get_peer_f, a pointer to an undefined function that probably would have gotten a pointer to a peer when given an address of record. Turning back to the CAN paper, I noticed that it was meant to be implemented at the application level. Immediately, I turned to the graphical simulator and edited its functionality accordingly. Now, it uniformly splits the zones.

What I have learned these 3 and a half weeks is to look; there is always an answer.

8. Acknowledgments

I would like to thank my dad for bringing me to Cisco and John Lai for giving me my C compiler, the virtual Linux machine, and mentoring.

9. References

1. SIPDHT2. http://sipdht.sourceforge.net/sipdht2/index.html . To download: http://sourceforge.net/project/showfiles.php?group_id=166130
2. Sofia-SIP. http://wiki.opensource.nokia.com/projects/Sofia-SIP To download: http://sourceforge.net/project/showfiles.php?group_id=143636
3. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In Proceedings of ACM SIGCOMM 2001. Available at http://www.sigcomm.org/sigcomm2001/p13-ratnasamy.pdf
4. Singh, Kundan. P2P-SIP. 2006. 13 July 2007. http://www1.cs.columbia.edu/~kns10/research/p2p-sip/