Geoide PIV-17 : 3 Dimensional Surveilance Networks
Real-Time Distributed Computing

Patrick Diez
McGill University
Centre for Intelligent Machines
Montreal, Quebec
Email: technocratik@gmail.com

Sean Lawlor
McGill University
Centre for Intelligent Machines
Montreal, Quebec
Email: slawlor@slawlor.com

Frank P. Ferrie
McGill University
Centre for Intelligent Machines
Montreal, Quebec
Email: ferrie@cim.mcgill.ca

***Abstract*—A new application of a distributed hash table, notably Chord is demonstrated to allow O(log(n)) time distribution of data to n hosts over a unicast network such as the internet. The publish/subscribe paradigm is inherent in this application, and combined with Chord makes the basis of the ChoPS system. Along with results demonstrating the near-optimality of Chord when used for such distribution, a full system layout and specification is provided for a real implementation of ChoPS.**

***Keywords*-project; node; chord; chops; publish-subscribe;**

I. Introduction

As the internet continues to expand as a collaborative device, the constraint of unicast communication(in which data have a single destination) becomes of ever-increasing cost. As per RFC 2770 [1] , the IPv4 address space 232.0.0.0/8 is reserved for IP multicast (in which data have multiple destinations), however, as with IPv6, lack of support and availability of these addresses from consumer ISPs and internet hubs makes this feature unreasonable as a basis for an internet-based multicast networking. Instead, as is proposed here, the use of a distributed hash table, notably Chord, provides a means of establishing a virtual multicast network with commodity unicast internet connections that compares favorably against the naive process of unicasting to each client. Extending Chord, a Publish/Subscribe system and administration interface will be implemented to allow seamless collaboration between remote processes.

II. Chord and Publish/Subscribe

A. Background

For the purpose of this implementation, it is assumed that a given internet host's, *i*'s, ability to transmit and receive data is well-represented by two values: its transmission and receive costs, $c_{tx}(i)$ and $c_{rx}(i)$, which may be considered to have units of time, and encompass both the latency (delay incurred by routing) and transmission period (delay incurred by data rate, inversely proportional to bandwidth). The primary implication of this assumption is that routing costs from any host *i* to any other host *j* (i.e. the cost of data transmission aside from that incurred by the connections of *i* and *j* to their respective ISPs) is constant, and thus that the transmission cost from *i* to *j* is:

$$c_{tx/rx}(i,j) = c_{tx}(i) + c_{rx}(j) \tag{1}$$

The aforementioned naive process of unicasting to n clients therefore has a total cost of:

$$\begin{aligned} C_{naive}(i) = \sum_{j=1}^{n} c_{tx/rx}(i,j) &= n \times c_{tx}(i) + \sum_{j=1}^{n} c_{rx}(j) \\ &= O(n \times c_{tx}(i)) \end{aligned} \tag{2}$$

From equation 2, it is evident that the cost of multicasting from i is proportional to the transmission cost for i and the number of hosts in the virtual multicast network. In large-scale commercial applications, such as YouTube, the cost of this relationship is favoured over the cost of requiring that clients use custom software for multicast communications. However, acquisition of high-bandwidth links for finite-term and small-scale projects may be prohibitive, in which case efficient use of lower-bandwidth links is essential.

B. Chord

Chord is a distributed hash table (DHT) developed by MIT for the purpose of storing data across a large network of *n* hosts, in which lookups, stores, the addition of a new host, and the removal of a host require $O(log(n))$ time. The core of Chord is a consistent hash function, $h(k) = k'$, which maps host *keys* (called *identifiers*) and data *keys* to hashes. Hashes are organized into a ring of *2m* values, where *m* is the size (in bits) of a hash, and must be significantly larger that $log2(n)$ to avoid hash collisions. The successor of a key *k* is then the host (or *node*) $successor(h(k))$ such that $h(successor(h(k))) > h(k)$ for the smallest possible value of $h(successor(h(k)))$ if such an identifier exists, and the node with the smallest identifier hash otherwise. This establishes the *ring* of hash values, called the *Chord ring*, in which the successor of a key is the node with the next smallest identifier hash value, and the successor of the key(s) with the largest hash value(s) is the node with the

smallest identifier hash value. A key k is stored on node $successor(h(k))$ thus distributing the data stored in the hash table evenly amongst the nodes in the ring, assuming a sufficiently random hash function [2].

Minimally, to perform a lookup, each node i must store the IP address of successor(h(i)). Then, if a lookup is performed for key k, the node performing the lookup checks if it stores k. If it does not, it performs a remote procedure call and requests that successor(h(i)) lookup k. Given that such remote procedure calls may, in the worst case, require a traversal of the entire ring, this process is O(n) [2].

To mitigate this lookup cost, Chord implements a finger table on each node. Instead of storing only $successor(h(i))$, each node *i* stores up to *m* finger nodes' IP addresses in a table, where:

$$finger(i, j) = successor(h(i)+2^{m-j-1}), 0 \leq j < m \quad (3)$$

Clearly, depending on the value of *n*, not all fingers need to be stored. As shown in 3, the last finger that needs to be stored is that for which the condition:

$$finger(i, j) = successor(h(i)+2^{m-j-1} = successor(h(i)) \quad (4)$$

Again, for a sufficiently random distribution of keys, this reduces the size of the finger table from $O(m)$ to $O(log(n))$. Then, to perform a lookup for *k*, the node *i* performing the lookup first checks if it stores *k*. If it does not, if requests that $finger(i, j)$ performs the lookup, where *j* satisfies the condition:

$$h(finger(i, j)) > h(k) > h(finger(i, j + 1)) \quad (5)$$

In equation 5, the greater-than relationship is evaluated while taking into consideration the nature of the Chord ring. Since at each step of the lookup, the use of the finger table reduces the range of identifiers left to query by a factor of two, the worst-case time for a lookup is $O(log(n))$ [2].

Details regarding the processes involved in inserting and removing hosts from the ring are not covered here, as the rest of this section is dedicated to an analysis of a proposed modification for Chord allowing multicasting.

C. Multicasting

If a given node *i* were use the same lookup mechanism as Chord to transmit data to a node *j* instead of requesting it, the transmission time would be, logically, $O(log(n))$. Naively, a multicast from i to the entire Chord ring would require $O(nlog(n))$ time, an increase of $log(n)$ over the naive unicasting process presented in the introduction. However, during this multicast, $finger(i, j)$ would receive (redundantly) the same data to be multicast $2m-j-1$ times. Thus, optimally, transmitting the packet once to each $finger(i, j)$ would suffice. Each $finger(i, j)$ would then be required to transmit the packet to each $finger(j, j + k)$ for all $k > 0$, each $finger(j, j + k)$ to each $finger(j + k, j + k + l)$ for all $l > 0$, and so on, as shown in Figure 1.

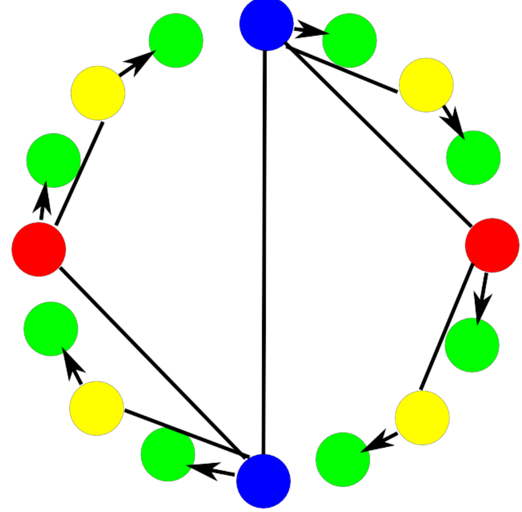

Figure 1. Chord Multicasting

Assuming all transmission times between nodes are identical, this process is $O(log(n))$ by virtue of the fact that at each timestep, the number of nodes having received the multicast data doubles. Additionally, this inherently provides the basis for the publish/subscribe system, as all hosts participating in a Chord ring will have to retransmit data multicast to the ring, irrespective of their need of it. In this sense, the Chord network itself acts as the broker (the host in publish/subscribe normally responsible for matching data producers with data recipients).

Stoica et al. [2] focus on the cost to *lookups* of network latency, whose analysis cannot be assumed to be reflected in the process of multicasting. For the purpose of analyzing multicasting, we recast the multicasting process as a recursively defined cost function:

$$C_{multicast}(i, j) = c_{tx/rx}(i, finger(i, j)) + \begin{cases} 0 & \text{for finger(i,j)} \\ & \text{= finger(i,j+1)} \\ max(C_{multicast}(i, j + 1), & \\ C_{multicast}(finger(i, j), j + 1) & \text{otherwise} \end{cases} \quad (6)$$

where $C_{multicast}(i, j)$ is the cost of multicasting from node i to all nodes $finger(i, k)$ for kj. Assuming $c_{tx/rx}(i, j)$ is not constant, there exists at least one ordering of the nodes in the Chord ring that minimizes $C_{multicast}(0, 0)$, the Chord multicasting cost for node 0. Analysis of the directed graph of the Chord ring, in which vertices are nodes and edges are weighted with transmission

costs is difficult, and expected (though not proven) to be at least NP-hard by the following argument:

> Since nodes are only capable of multicasting after having received the data to multicast, maximizing the slope of the function of the number of transmitting nodes vs. time will minimize the multicast time, as the area under that function is constant (and equal to $n - 1$). In turn, maximizing this slope implies that the first nodes to transmit ought to have the least transmission costs to their fingers of any nodes in the ring. Conversely, the last nodes to transmit ought to have the greatest transmission cost. However, these nodes will be transmitting to their successor along the outside of the Chord ring. As maximizing the values along the outside of the Chord ring is equivalent to the travelling salesman problem, a known NP-hard problem [3], so is the problem of minimizing the Chord multicasting cost for a given Chord ring.

D. Results

Two sets of simulations were performed. For the purpose of simulation, the Chord ring was simplified to ignore the hash function, and instead consist only of evenly spaced nodes. This simplification is considered to have a negligible effect on the results, given that the number of nodes is small. A variation of an exponential distribution was used to generate random host transmit and reception costs, in which the likelihood of a host having a given cost is inversely proportional to that cost. Finally, the ratio of maximum cost to minimum cost is specified and provides a means of controlling the variance of the cost on the network.

The first simulations compared number of nodes to multicast cost. Maximum and minimum broadcast times represent worst-case and best-case, respectively. Results were taken from the average of 1000 simulations, each performed with different transmit and reception costs, and each consisting of n! broadcasts (one for each permutation).

As is seen in Figure 2 , there exists roughly a 53% time advantage in the optimal case over the average case. However, this factor remains relatively constant, that is, the growth of the worst case is proportional to the growth of the average case. In this sense, the performance of Chord is considered good, in that its computational complexity is (empirically) $O(log(n))$ in all cases.

From Figure 3, it becomes apparent that Chord performs best when the nodes in the Chord have similar bandwidths (or, more precisely, transmission and reception costs). These data suggest that Chord multicasting is susceptible to a “rotten apple spoiling the barrel, that is, that as the number of low-bandwidth computers increases, the overall network speed quickly drops.

It is concluded from these data that, without modifying the structure of Chord, any improvement to the time required to

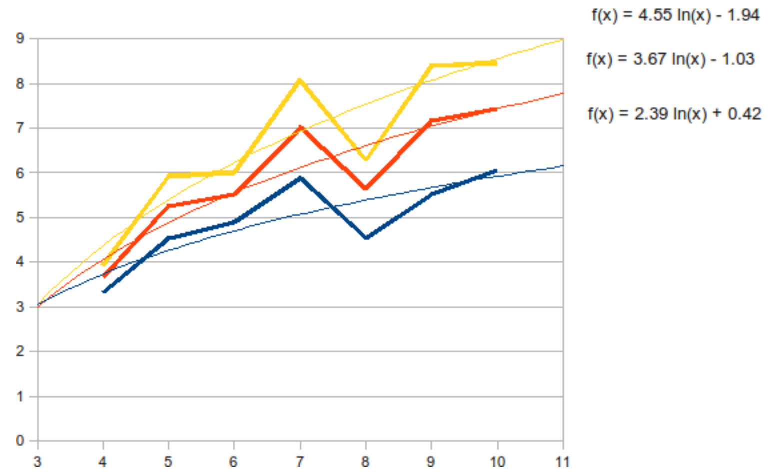

Figure 2. Graph of Multicast Cost vs. Number of Chord Nodes, with logarithimic regressions

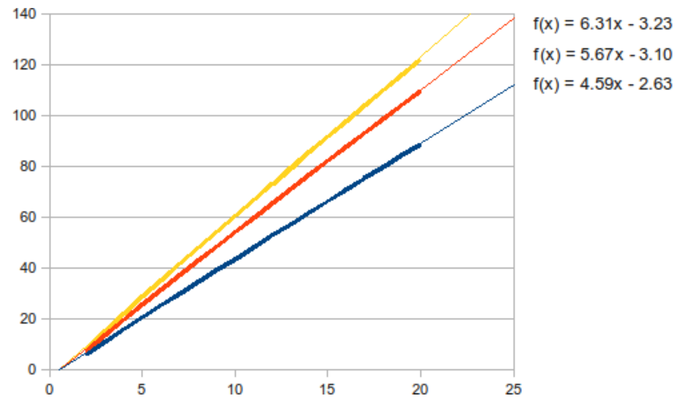

Figure 3. Graph of Multicast Cost vs. Bandwidth Ratio, with linear regressions

multicast from the unoptimized case will result in at most a constant factor of reduction, and that there is no difference in order of complexity between the unoptimized and optimized cases.

III. Implementation with ChoPS

A. ChoPS

ChoPS is the name given to an implementation of the Chord protocol with the Publish/Subscribe protocol as outlined in this paper. It defines an implementation which utilizes both theories to create a distributed real-time system which allows for indeterminate length data to be transmitted without knowledge of receivers (subscribers) nor the network transmission protocol. ChoPS will also allow for data transmission to be optimized, providing that each client within a ChoPS network (*projects*), maintains the ability to retransmit the data it has received to other computer within a project (*nodes*).

ChoPS allows for a simple front-end to large, complex data transmissions (I.e. *streams*), to a largely distributed

network through four features: Use of Chord for distributed, optimized data transmission, use of the Publish/Subscribe paradigm to allow for a loosely coupled network, a simple server which manages all connections, and a graphical user interface (GUI) to manage the entire system remotely.

B. Use of Chord and Publish/Subscribe in ChoPS

As previously outlined in Section II-B of this report, Chord allows for data stream transmission to be optimized when sending data to a large number of nodes. By utilizing the Chord mechanisms we can create projects within ChoPS. These projects allow for multiple nodes to communicate to each other whilst remaining separated from other projects. This guarantees that a node only contributes to stream transmission on the projects it is a member.

A ChoPS project is a Chord ring where the published stream within that ring is of a defined topic as outlined in Publish/Subscribe [4] . This design of building a Chord ring for each project allows for projects to be separated from each other, and guarantees that each node within a project will have its maximum bandwidth to (re)transmit the stream it is receiving to the other nodes in the project. The data transmitted across a ChoPS project is of indeterminate length and format. It therefore could be of infinite length, such as a security video stream, which never ends as long as the camera is active. Also a video format is only one of an infinite amount of data types which can be transmitted, meaning that ChoPS is truly a generalized data stream transmission system.

C. ChoPS Server

The ChoPS Server is implemented as a system daemon, which opens port(s) on a node, each of which can accept connections. The server implements a connection listener which can spawn multiple concurrent connections from multiple administrative clients and/or project nodes. Each connection has its own packet listener, which will simply sleep until a packet is received, and then will pass the packet to a centralized packet handler which processes the packet information and sends a response accordingly. Each client connection, whether from an administrator or simply a stream transmission through a project, will get its own connection information, which the server knows about and can process independently from the other client connections. The server however, as currently implemented is a single packet handler with multiple clients connecting to it, each being processed in series, and therefore must be and is thread-safe.

Each node within the ChoPS network runs a local instance of the ChoPS server. The server initially sleeps, only listening for administrative connections until an administrator tells the node to connect to a project and start (re)transmission of a stream. Each node stores all the projects it is connected to, as well as information about each project which is required to participate in stream transmission within the project (finger-table, successor, etc).

D. Administrative Server Connections

To administer a node's local ChoPS server instance, an administrator must log into the node with the node's local administrative username and password. This information is stored locally on the node, in a secure hash. Once connected, all data transmissions are through console commands as typical with a standard Unix server process. When the server receives a new console command, it processes the command and sends a response.

Only commands which do not affect a ChoPS project as a whole are accessible as an local administrator. Meaning that only commands which affect the local server instance, and will not affect nodes other than itself are accessible. However a local instance of the project settings can be made, and because the changes don't propagate, they will not be pushed to the rest of the nodes within the modified project.

From this authenticated mode, projects can also be added or removed from a node. If removed, a notification is sent to all members of the project notifying the other member nodes that the original node has dropped from the project and the finger tables as well as hashes need to be updated. Adding a project works in a similar manner. One node within the new project needs to be made aware that a new node wishes to join the current project. Once that node is notified of a new addition, all the finger tables and hashes of each node within the project are updated per a update request from the original connected node, not the joining node.

E. Administrative Project Connections

To administer an entire ChoPS Project, an administrator must log into a single member node of the project with the project's administrative username and password. Once authenticated, all commands which allow for changes to an entire ChoPS project are available to the administrator.

Project administration can be viewed as a series of changes that will effect an entire ChoPS project. They include the ability to stop a project and remove it completely, effectively shutting down all member nodes' access to that project's data, as well as changing how data is routed through the project. Once a change has been made on the node to which the administrator is connected, the change in immediately pushed to all subsequent nodes within the project so the change takes place on all nodes.

As similar to an administrative server connection, project administration cannot affect any local node's instance, except in relation to the project for which it is currently authenticated.

F. Project Monitoring

A user can also log into a ChoPS project through a more restricted account to simply get data about a project.

They cannot change any aspects of the running project, including disconnecting nodes. They cannot change aspects about any single system as well. This mode is designed for non-authorized users to still be able to gather data about the network, and if necessary notify the administrator of a change that needs to take place.

G. Server Data Processing

Once a node is a member of a project, and the project has been notified that a new member node has been added, then the node starts receiving the data which is being published across that project. The node now has two abilities, it can either drop the data and simply participate in retransmission, or it can subscribe to the data stream and begin processing it. Once the data has been processed, the node also has the option to publish the data into a new ChoPS project for other nodes to connect to and process/view/transmit.

If the node is receiving the data and passing it into the userspace (the computer system area where a users' programs run in user-editable memory), the node will create a local socket to which the stream is passed, and a system user can then connect to the stream an process the incoming data as it is received. For example the user could connect to the local socket, process the data using a tool such as Matlab^{TM}, and then send the output to another stream which can either be viewed locally or passed back to the ChoPS server to become another publication in a new ChoPS project. However even if a user is processing data, if the node is not a data endpoint in the Chord process, the node will still retransmit the data to the nodes listed in its finger table. All nodes must participate in the ChoPS data transmission protocol regardless of whether they are reading and/or storing the stream locally.

H. ChoPS Administration GUI

The ChoPS administration graphical user interface is built so that it is a remote administration module to the ChoPS system. To run the administration GUI, one does not need to have a local instance of the ChoPS server running. The administration GUI is built around the console command system, which means that the GUI will send commands to any system it is connected through as console ASCII text, and receive responses the same way. This allows an user to connect to a ChoPS server simply through a telnet [5] session without the need for the GUI if they so wish.

I. ChoPS Administration Console

The ChoPS administration GUI can open a series of consoles, which allow an administrator to override the default GUI interface, and send system commands directly to the node. Each console can support only one connection to a server, however one can have multiple consoles open, each with its own connection.

J. ChoPS Administration GUI Interfaces

The ChoPS administration GUI also has graphical representation to any command that can be passed through the console. This allows for two modes of administration, as shown as well with the ChoPS server in sections III-D and III-E, only certain aspects can be changed depending on how the administrator is logged into the Node.

The administrative GUI interfaces are designed for each of use, with features to aid the user such as tool-tips and labels describing behavior of functions. The goal of the ChoPS administration GUI is to allow any user, without knowledge of the underlying system, to start a publication of a stream and allow the stream to be transmitted effectively. Ideally it would even reach the point where no documentation would need to be read to use the system.

IV. Conclusion

There are many options to distributed computation, however many current systems do not take into account the possibility of large, possibly infinite, data streams being transmitted in a multicast nature to many end hosts. Also the current implementation of the IPv4 internet backbone doesn't allow for a simple multicasting protocol to be implemented. To build a multicast network across a large, distributed collection of hosts there currently exists two main options. They are to spend large sums of money, and setup a dedicated routing network which would allow the multicast protocol, or to tunnel all traffic over a dedicated channel, and increase network overhead. By utilizing the Chord DHT model, coupled with a Publish/Subscribe protocol, ChoPS can get around these problems by providing a stable environment for these data transmissions.

The features outlined with the ChoPS administrative graphical user interface allow for basic users to start right away with the ChoPS system and begin processing their data. By utilizing Chord's proven stability, ChoPS also guarantees that data transmission will be stable throughout a project's life. ChoPS also utilizes practices which are standard with Unix based processes, which have been shown to have superior security and stability. For example, all commands to the ChoPS server are outlined as ASCII text, allowing any system administrator to login without the GUI and control any aspect of the system.

Finally due to ChoPS generic nature, the system is not limited to video stream transmission, as will mostly be utilized by the Geoide PIV-17 project, the parent project to ChoPS. ChoPS can be applied to a data stream of any type and nature with an infinite number of participants, when properly applied.

Acknowledgment

Thanks to Professor Frank P. Ferrie of *McGill University* and Professor James H. Elder of *York University* for supporting this system in *Geoide PIV-17*.

References

[1] D. M. of Cisco Systems and P. L. of Sprint, "Rfc 2770 - glop addressing in 233/8," Network Working Group, Tech. Rep., 2000, http://www.rfc-editor.org/rfc/rfc2770.txt.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," MIT Laboratory for Computer Science, Tech. Rep., 2001, http://pdos.lcs.mit.edu/chord/.

[3] R. M. Karp, *Complexity of Computer Computations - Reducibility Among Combinatorial Problems*. New York, New York: Plenum, 1972, pp. 85–103.

[4] T. J. K. Birman, "Exploiting virtual synchrony in distributed systems," in *Proceedings of the eleventh ACM Symposium on Operating systems principles*, 1987, pp. 123–138.

[5] M. McKenzie, "Rfc 495 - telnet protocol specification," Network Working Group, Tech. Rep., 1973, ftp://ftp.rfc-editor.org/in-notes/rfc495.txt.