Class-Chord: Efficient Messages to Classes of Nodes in Chord

Dan Fleck, Sharath Hiremagalore, Stephen Reese, and Liam McGhee
CARE Center
George Mason University
Email: {dfleck, shiremag, rreese, lmcghee2}@gmu.edu

Fengwei Zhang
Department of Computer Science
Wayne State University
Email: fengwei@wayne.edu

***Abstract*—Security Information Event Management (SIEM) systems are used to monitor large networks for malware infestations, DDoS attacks, and many other types of network intrusions. Typical SIEMs are centrally managed with information flowing in from across the enterprise. In this architecture, as the enterprise grows, the SIEM must also scale proportionally. In our research we are working to create a Peer-to-Peer distributed SIEM system to leverage the power of all the devices in the network for monitoring. The system scales naturally; as the enterprise grows, more devices come into the peer-to-peer network (P2P). The added devices increase the SIEM's processing power and storage ability. A P2P SIEM will drastically reduce upfront hardware costs and provide an increased processing power for advanced analytics. In this paper, we present Class-Chord which is a P2P network fabric designed to support a P2P SIEM. We have modified the well known Chord DHT to support efficient 1-n messaging that is required to enable SIEM administrators to query subsets of the network rather than flooding queries to all nodes. Class-Chord uses a modified Chord ID and a new message type that enables administrators to send messages to network subsets using complex class matching specifiers. We analyze theoretical models for the system and present experimental results from a live system deployed across 300 physical nodes. The results attest that Class-Chord is more efficient than traditional communication mechanisms used in SIEM systems.**

***Keywords*-Chord, Messages, Nodes.**

I. Overview

A. Motivation

Security Information Event Management (SIEM) systems are used to monitor large networks for malware infestations, DDoS attacks, and many other types of network intrusions. Typical SIEMs are centrally managed with information flowing in from across the enterprise. Information comes from all networked devices (end-user computers, routers, webservers, DNS servers, and others). In this architecture, as the enterprise grows, the SIEM must also scale proportionally. A recent report from HP notes their SIEM infrastructure includes 100s of archival databases and many instances of the SIEM manager. The upfront cost reported for a large enterprise SIEM is $3 to $5 million with additional yearly maintenance fees [1].

In our research we are working to create a Peer-to-Peer distributed SIEM system to leverage the power of all the devices in the network for monitoring. The system scales naturally; as the enterprise grows, more devices come into the peer-to-peer network (P2P). The added devices increase the SIEM's processing power and storage ability proportionally. A P2P SIEM will drastically reduce upfront hardware costs and provide a increased processing power for advanced analytics.

Among the several challenges inherent in the creation of a P2P SIEM, efficient message passing is of pivotal importance. According to [1] first level network monitoring personnel evaluate one to three thousand alerts per day. In the P2P SIEM system administrators will investigate network events by sending queries to individual nodes and groups of P2P nodes. To implement the SIEM we must create a peer-to-peer network architecture which supports a small number of querying nodes sending queries to individual nodes and large subsets of the monitored network. For example, a subset may be "all Windows machines" or "any server". These subsets need to be dynamic as security monitoring personnel may pose various questions based on the unfolding threat profile. Some example questions are: "is file XYZ present on your disk?" or "what processes are currently running?". The underlying message protocol must provide a flexible fabric to ask any questions as defined by the SIEM application layer. Creating this network architecture is the main focus of this work.

While motivated by the P2P SIEM problem, the solution we work towards is efficiently solving the 1-to-n messaging scenario where "n" is a dynamically defined subset of the network.

B. Contribution

We named the solution presented in this paper Class-Chord. Class-Chord is a modified version of the Chord Distributed Hashtable (DHT) network architecture [2]. Chord provides a robust P2P overlay which supports a simple flooding protocol (1-to-all messaging) and direct point-to-point messages (1-to-1 messaging). Class-Chord adds the ability for Chord to efficiently send messages to classes of nodes (1-to-n messaging). A class is a subset of nodes in the network that meet a class specification. In this paper, we present a detailed theoretical analysis of Class-Chord and provide experimental results from over 15,000 tests running on a Python implementation of the Class-Chord network

on over 300 nodes. The results show that Class-Chord is more time efficient than sending point-to-point messages or flooding messages to large subsets of nodes. We also compare our approach to publish/subscribe models [3], [4], [5], [6], and WAN-multicast message passing solutions [7], [8].

To summarize, the contributions of the presented research are:

- The Class-Chord algorithm which sends messages to classes of nodes within a Chord network more efficiently than either flooding or point-to-point approaches.
- A theoretical evaluation of the Class-Chord algorithm and an evaluation of our implementation
- A Python implementation of Class-Chord publicly available for use by future research teams for further experimentation

C. Paper Organization

The rest of the paper is organized as follows. In Section II we define the problem formally and the metrics to measure the results. Section III describes the Class-Chord approach in detail. In Section IV we present both theoretical results and actual results from our implementation. Section V describes related work regarding message passing systems as well as other related modifications to Chord. Finally, Section VII concludes the paper.

II. Problem Definition

To understand the Class-Chord system presented we first present a short introduction to the traditional Chord distributed hashtable [2].

A. Traditional Chord Messaging [2]

Chord is an overlay network that arranges nodes into a ring-based structure. Each node has a globally unique ID (GUID). In this paper, we refer to the GUID simply as the nodes' ID. The nodes are arranged into a ring ordered by ID. To create the ring each node maintains a pointer to its successor node which is the next node existing in the ring with an ID higher than the current node. The successor to the node with the highest ID is the node with the lowest ID which creates the "ring". This GUID is typically a hash of some attribute such as the MAC address. The full address space of the ring is typically much greater than the actual number of nodes present in the ring creating a sparse ring. For example, a typical node ID is a 32 bit number. During a node ID lookup, if the requested ID is found the node will be returned, or if not, the closest node succeeding the ID will be returned.

A message could propagate around the ring simply by having each node send the lookup to their successor. In a SIEM system this is useful for sending flooding queries where each node should receive the query. However, to send a message to a specific node a more efficient node lookup is needed. To make a single node lookup more efficient Chord uses the concept of a "finger table". Each node maintains a finger table which stores a mapping from node IDs to node locations (i.e. IP addresses) which are responsible for the ID. Chord's finger table contains m entries where m is the number of bits in the ID. The i^{th} entry in the table is the node location responsible for ID $(n + 2^{i-1}) mod\ 2^m$. Using this approach, Chord node lookups are $O(logN)$.

The traditional Chord architecture provides a simple and efficient framework to send messages to all nodes or individual nodes in the network. This is attractive for a SIEM system which needs both capabilities. However, in a large organization a SIEM system operator may need to send query messages to subsets of the network. For example, to only routers or to all servers running Linux. Traditional Chord does not provide an efficient approach for these types of messages.

B. Class-Based Messaging

To create a peer-to-peer SIEM requires a network messaging architecture which supports messages directed to classes of nodes based on the nodes' attributes. We call these class-based messages. The system must support dynamic class specifications to let system operators target different classes of nodes based on attributes. These specifications must support attribute ranges, exact values, and lists of values. For example, a query may want to target all machines with the Windows operating systems or Linux patch level range "4.12-6.03". In the next section we formalize how to measure the efficiency of the messaging protocol we will use in the following sections to compare Class-Chord to a traditional Chord implementation.

Chord's ring topology enables a simple flooding message transmission in $O(N)$ time or messages by having each node forward the message to their successor. Each node caches its successor location so the message doesn't require any further information. We denote a cached lookup as a "short lookup" throughout this paper. Results reported in [2] show that an arbitrary node lookup can be done in $O(logN)$ time. We refer to these non-cached lookups as "long lookups" throughout this paper because they take significantly more time than constant time short lookups. Thus, a single message to another node is $O(logN)$. However, Chord provides no efficient approach to send a message to a group of nodes.

Our problem is: given a group of nodes in a class (C) within a Chord network, how do we efficiently send them a message? To measure the efficiency of our system we define a metric called waste (W). Waste has two components – the number of wasted messages plus the number of long node location lookups required to get the message to all nodes that match the message's class specifier. Wasted messages are messages received by a node where the message's class specifier do not match the node's class. After the message

reaches the node, checking the specification is a constant time operation O(1). The wasted messages will be discarded by the node.

A long lookup is a node address lookup which is not cached locally by the node. As previously mentioned, node lookups in Chord are $O(log_2(N))$. Waste (W) is computed using Equation 1 below where τ is a constant representing the time to process a message by the node.

$$W = (N_R - N_C)\tau + L * log_2(N)\tau \tag{1}$$

Where N_R is the number of nodes which received the message, N_C is the number of nodes that match the class specifier in the message and L is the total number of long lookups required. As an example, we compute this metric for a 1,000 node network with a node class specifier matching 100 nodes. Using a flooding query:

$$\begin{aligned} N_R &= 1000 \\ N_C &= 100 \\ L &= 0 \\ W &= (1000 - 100)\tau + 0 = 900\tau \end{aligned} \tag{2}$$

Using the same network parameters, sending individual query messages instead of a flooding message, results in the following waste:

$$\begin{aligned} N_R &= 100 \\ N_C &= 100 \\ L &= 100 \\ W &= (100 - 100)\tau + 100 * log_2(1000)\tau = 996\tau \end{aligned} \tag{3}$$

As shown, flooding requires many more wasted messages and individual messages require many extra long lookups. Later in this paper, we will show that Class-Chord reduces both long lookups and wasted messages.

III. Class-Chord Routing

Class-Chord adds meaning into Chord's node IDs to enable routing messages to groups of nodes that correspond to the characteristics of a query, through a simple numerical process. Thus, the Class-Chord node's ID is a class specification followed by unique bits. The class specification itself is broken down into sub-parts for each attribute of the node that is part of a class specifier. For example, a simple class specification could hold operating system type, device type, and patch level. Table I describes the example class attributes.

In the specific example shown in Table I, a node's class specification will be 13 bits. Given a unique ID of 115 bits, results in a node ID length of 128 bits. The specification of a Windows laptop at patch level 12 is shown in Figure 1 below.

To efficiently send a message to a class of nodes, we modify the Chord algorithm's lookup function to accept a class specification or a node ID as its destination. If the lookup function finds a class specification in the destination, it parses the destination and can determine the next appropriate node ID to lookup. A class specifier is similar to a class ID specification, however a major strength of our approach is the class specifier allows wildcards and ranges. In our implementation the class specifier can contain the value-types shown in Table II.

Table I
Sample Class Specification Values

Field	Num Bits	Values
OS Type	2	1: Windows 2: Linux 3: Mac OS X
Device Type	3	1: Desktop 2: Laptop 3: Switch 4: Printer 5: IP Phone
Patch Level	8	Numeric range 00-256

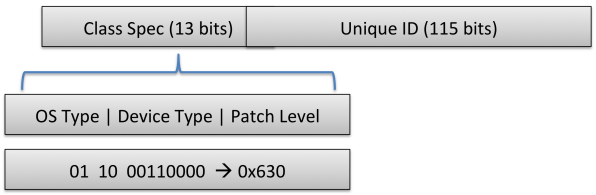

Figure 1. Sample class ID for a Windows laptop at patch level 12. Unique ID bits are a random 115 bit number not shown.

Table II
Class Destination Specification Value Types

Type	Description	Example value
Exact	A simple numeric value, cannot be changed.	03
Range	A numeric range, where any value in the range is accepted.	04-06
Multiple	A list of multiple exact value (comma separated).	3,6,8

Any value-type can be used in the class-specifier by implementing the following two functions for the value-type:

$boolean \leftarrow$ IsValid$(specifier, currentValue)$
$long \leftarrow$ GetNextValue$(specifier, currentValue)$

Once parsed, the algorithm mathematically determines the next hop that is valid in the class.

A. Computing the next hop

The key efficiency improvement in Class-Chord is the ability to skip over nodes which are not in the specified class when routing the message. This is done by parsing the class specification and using it in conjunction with the current node's ID to determine the next feasible ID (i.e. the

next hop). The next feasible ID may not be assigned to a real node because in a sparse network most available node IDs are un-used. However, in Chord this is still efficient because a node lookup results in the node present in the network with that ID or the next higher ID present in the network. Thus, the next feasible ID is simply the next ID greater than the current ID that is a member of the class specification.

To compute the next hop we present the pseudocode snippet in Algorithm 1. The full working code for Class-Chord is being made available at https://github.com/danfleck/Class-Chord.

The pseudocode in Algorithm 1 first checks if the next ID (computed by adding 1 to the current node's ID) matches the class specifier in the message. This is the simplest and most common case. When this happens, the node just forwards the message to its successor. This is a constant time operation. Once the message reaches a node whose successor does not meet the class specification's criteria, then the node must lookup the next valid class, which is not simply the node's current ID plus one.

To compute the next node which should receive the message the class specification is parsed into "atoms". Each atom is a single item in the class specification with a valid set of values. An example atom could be patch level (e.g. "3-7"). In our implementation atoms can be ranges, exact values, or multiple values. The code then looks at each atom to find the first invalid atom. Once found, all the atoms to the right of it are set to their minimum valid value. The invalid atom is then incremented to its next valid value. If that value is greater than the maximum valid value, we must "carry" the increment to the next atom to the left and increment it. Otherwise, if there is no carry required, all remaining atoms to the left of the invalid atom remain their same value. Examples showing the next hop are given in Table III.

Table III
NEXT HOP EXAMPLES (WITH 4 CLASS ATOMS AND A 4 DIGIT GUID)

Class Spec	Sample Node ID	Next Hop ID
11-12 ! 22 ! 33 ! 44	112233449999	122233440000
11-12 ! 22 ! 33 ! 44	112233445555	112233445556
11,30,99 ! 22 ! 33 ! 44	112233449999	302233440000
11-12 ! 22 ! 33,99 ! 44	112233449999	112299440000

While we have implemented several atom-types (e.g. exact, range, and multiple values) many other atom-types could be implemented as needed. Each atom simply needs to support the following methods:

- $boolean \leftarrow isValid(atomSpec, currentAtom)$ returns True if currentAtom meets the specification, False otherwise.
- $long \leftarrow getMinValue(atomSpec)$ returns the minimum valid value for this atom
- $long, boolean \leftarrow getNextId(atomSpec, currentAtom)$ returns the next ID and a boolean set to True if carrying the one, False otherwise. Carrying the one happens when the next valid ID would otherwise be greater than the maximum allowed by the atom.

Finally, *getNextHop* must handle the remaining case when there are no more valid class IDs. Because Chord is a ring structure, the next ID is then the minimum class ID with the minimum GUID. The code shown in Algorithm 1 does this naturally through the "carry" mechanism.

While most nodes not satisfying the class specification are skipped using this approach, there are times when nodes outside the class get the message. These are wasted messages. A wasted message happens when a message is sent to a destination ID in the class, but a node with the destination ID does not exist, and the next existing node in the Chord ring does not match the message's class specifier. For example, consider the simplified Chord ring shown in Figure 2. Node A is sending a class message to class 1 (CID=1). *getNextHop* will return an ID of 111 (class 1 and GUID 11). Because no nodes in the network are actually present with that ID, the message goes to the next higher ID present in the network. The message gets sent to node B, which is a wasted message because B is not part of class 1. Node B then uses *getNextHop* which sends to node D, the first successor node within class 1. From this example, we can see that some messages are still wasted; however this only happens at class boundaries on the ring. In practice, there will be a much smaller number of wasted messages than correctly sent messages as shown in the results section.

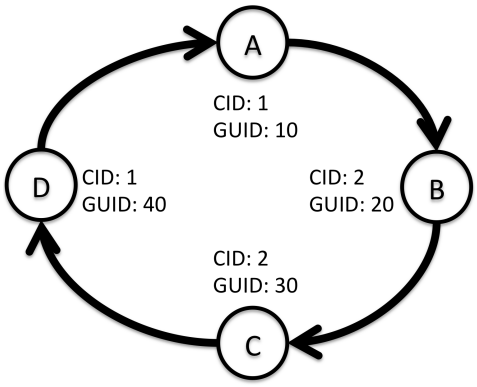

Figure 2. Sample network shown with simplified class IDs (CID) and GUID split apart.

IV. RESULTS

In this section we present results that quantify the number of wasted messages based on network parameters. We present both theoretical results and experimental results computed using our Python implementation.

A. Theoretical Results

Recall the measurement we are interested in is the waste (W) computed using Equation 1. To compute W we need

Algorithm 1 Pseudocode for getNextHop

```
function GETNEXTHOP(currentNodeID, classSpec)
    if ISVALID(currentNodeID+1,classSpec) then
        return currentNodeID + 1
    end if
    currentClassID, currentGUID ← SPLIT(currentNodeID)      # We need to update the current
    newGUID ← 0                                             # ClassID so we set the GUID
    newClassID ← ""                                         # portion to zeros.
    atoms ← GETCLASSATOMSFROMSPEC(classSpec)
    firstInvalid ← FINDFIRSTINVALIDATOMFROMLEFT()
    if firstInvalid is None then
        firstInvalid ← lastAtomIndex                        # Still need to increment something
    end if
    for atom in atoms do                                    # Now the atoms to the left of
        if atom.index == firstInvalid then                  # firstInvalid should be okay, the
            newAtomVal, carry ← INCREMENTID(atom, currentClassID)   # ones to the right need to be set
            newClassID ← newAtomVal + newClassID            # to min values and the invalid one
        else if atom.index > firstInvalid then              # incremented.
            newClassID ← MINVALUE(atom) + newClassID
        else
            while carry do
                newAtomVal, carry ← INCREMENTID(atom, currentClassID)
                newClassID ← newAtomVal + newClassID
            end while
        end if
    end for
    return newClassID + newGUID
end function
```

to compute the number of nodes which actually receive the message (N_R), the number of nodes in the class (N_C), and the number of long lookups (L).

A long lookup occurs in Class-Chord when a node receives the message and the *getNextHop* function does not simply increment by one. (The simple increment by one case uses the nodes' current successor which doesn't require a lookup.) The long lookup only occurs when a node must send to a node outside their current class, or in the one case where the node's ID is the maximum ID (i.e. all nines in base 10). The number of nodes with the maximum ID is at most one, and thus negligible for these computations. Thus, a node does a long lookup when it receives a message destined for a class other than its own. This means that the lookups only occur when a message is "wasted". Thus $L = N_R - N_C$ transforming the waste equation into Equation 4 below for Class-Chord.

$$W_{CC} = (N_R - N_C)(1 + log_2(N))\tau \tag{4}$$

To compute the number of nodes that receive the message (N_R), we must compute the number of messages which go to nodes outside the class specification. This number depends on the specific class specification and where the atom is in the class specification. For example, if the class specification is "OS | TYPE" then all the nodes with the same OS are grouped together. Therefore, a message to class "WINDOWS | ANY " would proceed from the beginning of the block of Windows machines and end after the last machine, thereby wasting only one message. However, using the same network but sending a message to class "ANY | LAPTOP" would generate a wasted message after every different operating system value. The message from the numerically highest Windows | Laptop would proceed to the Windows | Desktop network segment. From there, the node would recognize it's not of the correct class (i.e. not a laptop), perform a long lookup, and send the message directly to the first Linux | Laptop. In this way, the number of wasted messages is equal to the number of non-consecutive different classes which meet the message's class specification.

To simplify the calculation, we denote a class atom as A and the number of valid values within the class as V. Thus, V_I is the number of valid values in the I^{th} class atom A_I for a given class specification. For example, in Figure 3 the OS class atom is at $I = 0$ and the device type is $I = 1$. The number of non-consecutive different classes meeting the class specification is thus given by Equation 5. Where S is defined as the atom with the highest index (outermost ring)

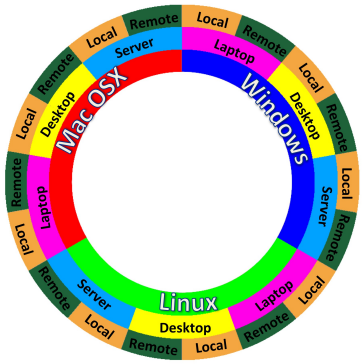

Figure 3. Sample class layout for a network with class specification "OS | Device Type | User classification".

that *is restricted* by the class specification. Every atom above S is not restricted and can be treated as a block. For example, in Figure 3 the class specification "WINDOWS | LAPTOP | ANY" has only one wasted lookup and similarly the class specification "WINDOWS | ANY | ANY" also has one wasted lookup, although many more members of the class. Equation 5 computes the number of non-consecutive classes that meet the class specification. Substituting Equation 5 into Equation 4 yields our final waste equation 6 for Class-Chord. The V_I term in Equation 6 varies with the class specification and layout of the atoms in the Chord ring. This is the desired property: to ensure the number of wasted messages varies based on the class and atom layout. Flooding messages' waste doesn't vary with the class specification and point-to-point's waste doesn't vary with the organization of the nodes in the ring. Results shown in the next section validate this analysis experimentally.

$$N = \prod_{i=0}^{S} V_i \tag{5}$$

$$W_{CC} = (1 + log_2 N)\tau \prod_{i=0}^{S} V_i \tag{6}$$

B. Measured Results

1) Test Setup: To test the actual number of messages and long lookups we implemented the system described in Python. We then deployed 314 nodes spread across four servers. Each node runs as an independent process and reports measurements back to a central monitoring node. We configured this network with the three class atoms shown in Figure 3 (i.e. OS | Device Type | User classification). Class attributes were distributed evenly among all the nodes yielding over 100 of each machine type, device type, and user classification.

During the test a random class specification was created. Then, three separate tests were run. The first test sent the message using the Class-Chord messaging algorithm presented, the second test used Chord's flooding message protocol. The third test, labeled "PtP", sent individual messages from the source node to each target node matching the class specifier. Using these tests, we can compare the ability of traditional Chord to Class-Chord for a variety of class specifications which each result in different numbers of matching target nodes. For some class specifications hundreds of nodes meet the criteria, while for others very few nodes meet the criteria. Using this approach over 15,000 tests were run.

2) Results: The X-axis of Figure 4 is the number of nodes within the network that match the chosen class specifier. The Y-axis shows the number of messages received by the nodes that matched the class specifier. The figure shows that all nodes receive the message as expected. In some cases, the class message is received twice by a single node. This happens when the message travels around the ring and is finally stopped at the node which originated the message.

Figure 5 shows the number of messages a node received that did not match its class specification. These wasted messages unnecessarily use bandwidth and CPU of the nodes that received them. As expected the figure shows the flooding approach wastes many messages. Point-to-point wastes no messages because an individual message is sent directly to each node as needed. Class-Chord has slightly more wasted messages when compared to point-to-point for different class specifiers. However, Class-Chord has significantly fewer wasted messages compared to the flooding approach.

Figure 6 shows the number of long lookups that were completed to send the message through the network. Recall, a long lookup is one that requires the node to ask additional nodes for assistance when routing the message. This is part of the Chord protocol when a node does not know the address associated with the given destination ID. These lookups are costly, and one of our goals is to minimize them. As shown in Figure 6, the point-to-point protocol is wasteful because it must do a long lookup for almost every destination node. Flooding does very few long lookups because the message is blindly forwarded to the successor of the current node. This is what makes flooding efficient in lookups, but also increases the number of wasted messages seen previously. Class-Chord succeeds in having very few long lookups across the range of class specifiers as shown in Figure 6.

In Figures 7 and 8 we compare the amount of time it takes for a message to reach all destination nodes which match the message's class specifier. As shown in Figure 7 the point-to-point approach can take up to 27 seconds depending on the number of matching destination nodes. This is very inefficient when sending a message to large

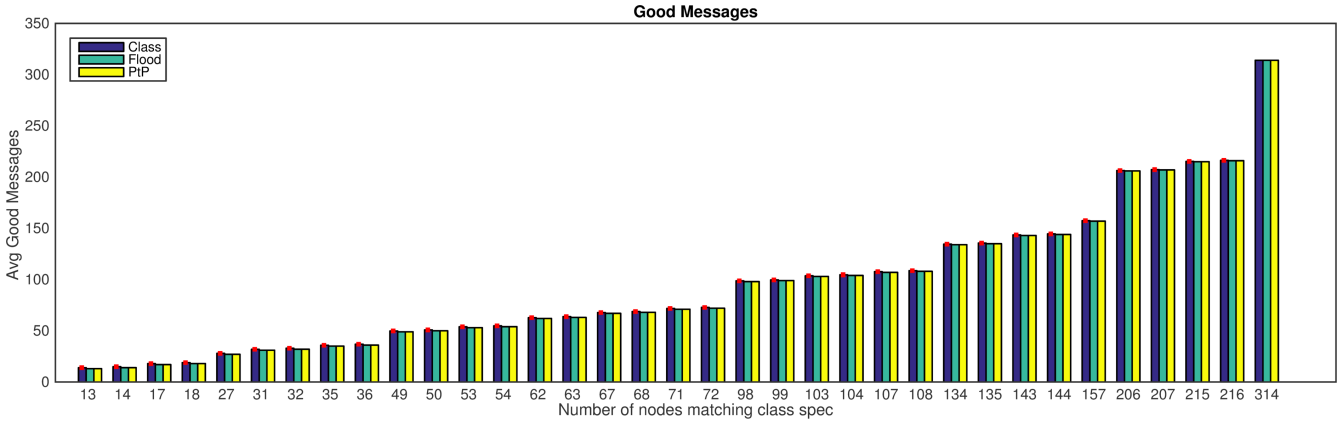

Figure 4. Good messages are ones where the message's class specifier matches the node's class. This chart provides a simple sanity check that as the number of matching nodes increases in the X dimension, the number of messages received correctly increases for each method. 95% confidence interval bars shown in red for class chord. Flooding and point-to-point have no variance in good messages.

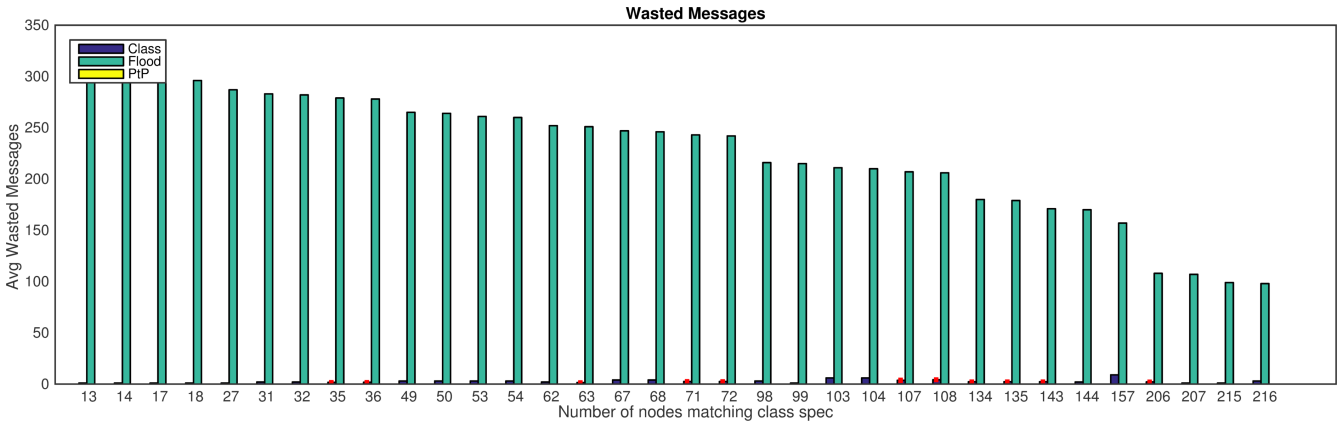

Figure 5. Wasted messages do not match the node's class. This chart shows that flooding messages which get sent to all nodes frequently waste messages while the point-to-point method only sends directly to matching nodes with zero waste thus no PtP bars are visible. Class-Chord has very few wasted messages, only slightly above zero. 95% confidence interval bars shown in red for data which had variance.

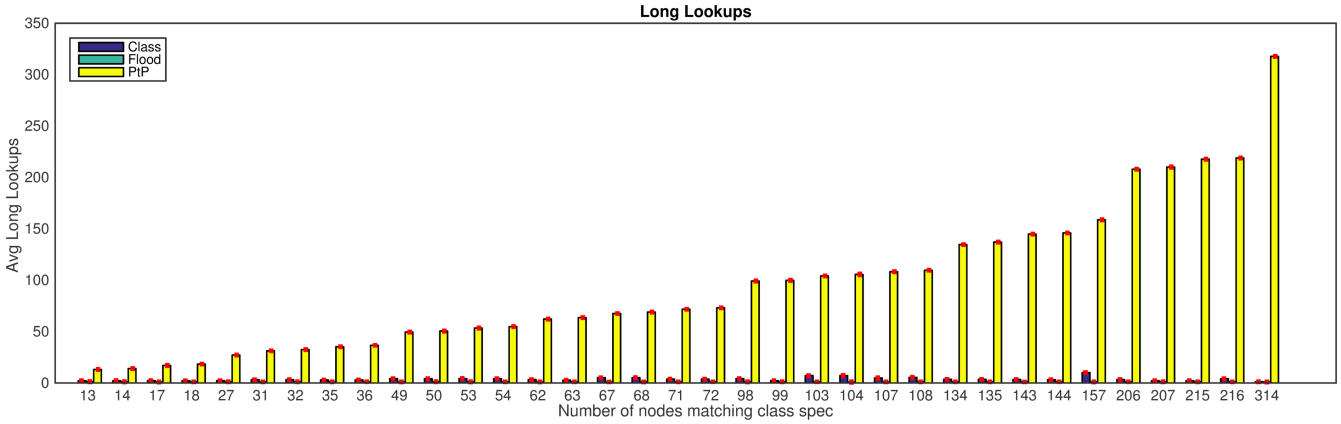

Figure 6. Number of long lookups for each method for varying numbers of matching nodes within the network. Flooding messages have zero long lookups and thus aren't visible in the chart. The point-to-point method requires many long lookups. Class-Chord has slightly more than zero long lookups. 95% confidence interval bars in red show minimal variance among the methods.

numbers of nodes when compared to flooding or Class-Chord. Due to differences in overall scale, Figure 8 compares the Class-Chord method to traditional flooding and Figure 7 shows times solely for the point-to-point method. As expected, there is little variation in the time for flooding messages to reach all nodes. The time is not dependent on the number of matching destination nodes. However, the Class-Chord message's timing profile shows that messages that have fewer matching nodes within the class are more efficient than messages which must travel to more matching destination nodes. This is a desirable property enabling very large networks to send messages to subsets efficiently. In our testing, flooding is more efficient only when the class specifier matches all nodes in the network (314 nodes in our example). For this message, the flooding message is more efficient because it does no checking or validation. In all other cases, Class-Chord is more efficient as shown in the results.

Overall, our test results are consistent with the theoretical models and show that Class-Chord is more efficient than either point-to-point or flooding when sending to sub-groups of nodes within a Chord ring. The efficiency is gained because Class-Chord has many fewer wasted messages than the flooding approach and many fewer long lookups than the point-to-point method.

V. Related Work

A. Message Passing Systems

Message passing systems are used in a variety of contexts. Chord [2], CAN [9], Pastry [10] and Tapestry [11] are all Distributed Hashtable (DHT) implementations used to create a networked hashtable. In our work we use the Chord structure solely for message passing. In 2002 Kademlia [12] was proposed as another DHT solution which was then modified for use in BitTorrent [13] for file sharing. BitTorrent's protocol was then adapted for secure message delivery as BitMessage [14]. Google's Thialfi [15] is a massively scalable message passing service used to keep cached client objects synchronized with changes on a server. PIER [16] is one of many distributed database systems built on peer to peer (P2P) network ideas. While all of these systems are used for different purposes, they all pass messages to nodes and share similar challenges of scalability, network efficiency, storage efficiency and security. Class-Chord helps address network and storage efficiency in derivatives of the Chord protocol.

Some early DHTs are Chord [2], CAN [9], Pastry [10] and Tapestry [11]. Each of these enables direct point-to-point messages and they can perform flooding. However, none of them natively support sending a message to classes of nodes. The Class-Chord method relies on having a simple one dimensional ID space. Using an automated approach similar to [17] the Class-Chord approach could be applied to Pastry. Additionally, future research could manually modify the approach and apply it to the multi-dimensional coordinate spaces of the other common DHTs. Kademlia and its derivatives, BitTorrent and BitMessage, have efficient flooding protocols, but do not natively support messages to sub-groups.

Another common message passing paradigm is publish subscribe (or pub/sub). An overview of various pub/sub approaches is given in [18]. In these models a node indicates interest in a topic or subject. A publisher then uses this information to send messages to interested nodes. Early approaches used group-based communication where a subscriber would indicate their interest in a group and publishers would store that information. Events would then be published to particular groups. Examples include [3], [4], [5], and [6]. These methods are inflexible and require publishers to maintain information about all subscribers. Class-Chord requires no information to be maintained explicitly for the sub-groups and supports dynamic "groups" created by the message's class specification.

Flexible content-based approaches have been proposed more recently in [19], [20], and [21]. In these systems the subscriber provides a matching predicate for events they are interested in. The message dispatcher then uses these predicates to match events being dispatched to subscribers. Similar to Class-Chord this enables sending messages to the flexible groups based on the message content. This does not require pre-formed groups, but does require the publisher or another entity to maintain the predicate information for the nodes which is not required in Class-Chord which uses the node ID structure to obviate that requirement. Additionally, efficiently matching predicates to nodes is a research problem discussed in [22].

Thialfi [15] is a recent caching solution from Google which models the problem as one of keeping versioned objects current throughout the network. The basis of Thialfi is a publish/subscribe type architecture with a heavy emphasis on fault-tolerance and scalability. The system sends the latest object version numbers to subscribers, but relies on the subscribers to then update their objects when possible.

In addition to publish/subscribe models built upon P2P overlay networks, we can compare Class-Chord to multicast solutions. Traditional multicast is efficient but requires nodes to be on the same subnet and explicitly join multicast groups. Some researchers have also built upon and augmented IP multicast that solves these problems. One example is [7] in which the authors modify IP multicast to support a channel with exactly one publisher and many subscribers. This approach enables scalability improvements over traditional IP multicast. A different modification of IP multicast is exemplified in [8]. The authors add two protocols on top of IP multicast to enable inter-domain multicast routing to create a scalable multicast solution.

A different approach was taken in [23]. The authors develop a new P2P overlay architecture for message passing.

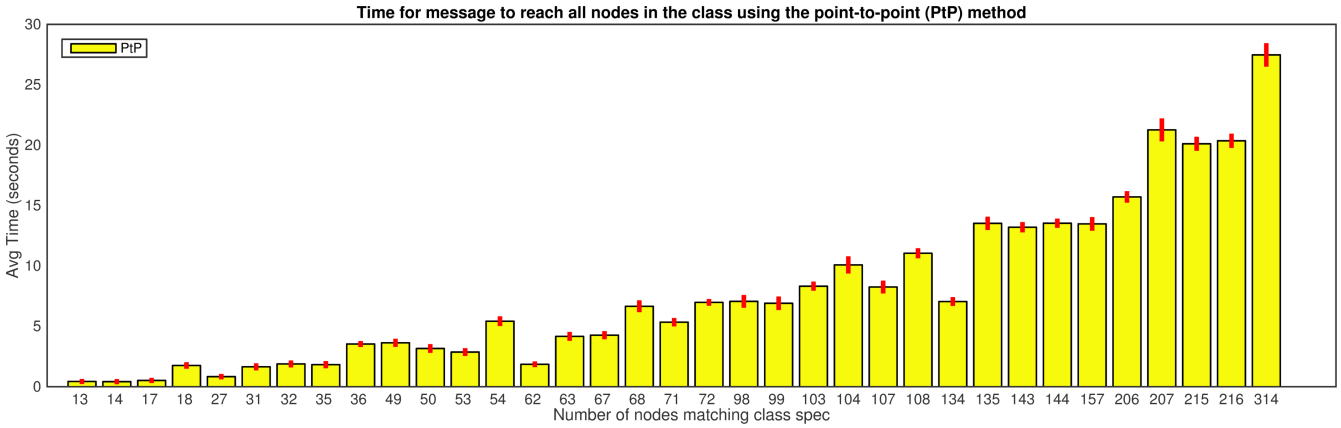

Figure 7. Average time for the message to reach all nodes matching the class specifier for point-to-point messaging. 95% confidence interval shown in red.

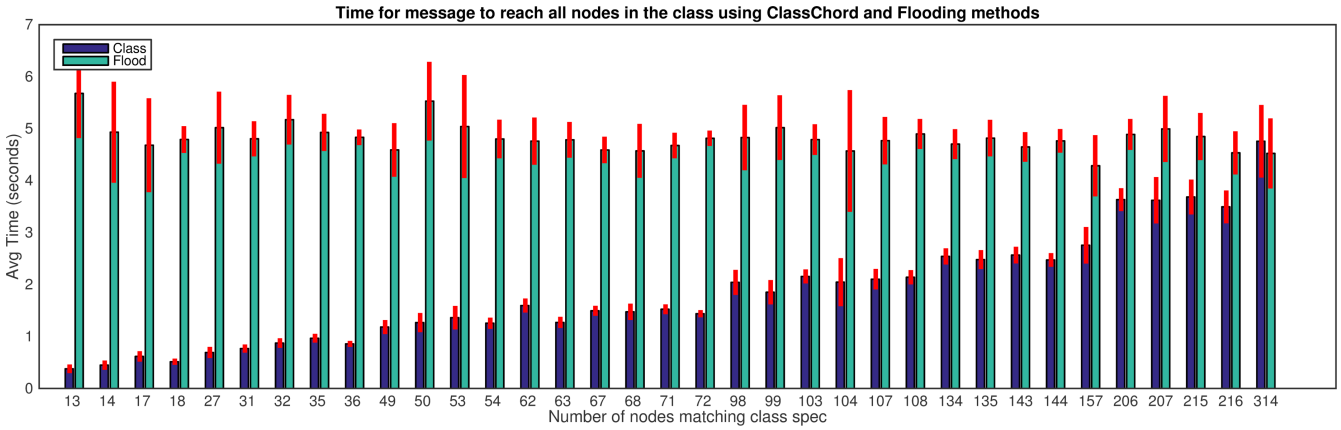

Figure 8. Average time for the message to reach all nodes matching the class specifier for the flooding and class messaging methods. 95% confidence interval shown in red.

The nodes are organized to place nodes with similar subscription interests near each other in the overlay network. The nodes then forward messages to all neighbors when an incoming message matches their subscription preference. This is done by creating a Containment Hierarchy Tree which is used to limit how far the message needs to spread to ensure it reaches all interested subscribers.

B. Modifications to Chord

Other authors have also modified Chord in a variety of ways to support message passing. Most similar to our approach is [24]. The authors create a publish/subscribe system using Chord. The approach uses nodes to store subscriptions which are then matched to events. Once matched, the event is sent to each matching node. It is unclear in the paper how the matching messages are sent to the nodes, but using typical Chord protocols either flooding or point-to-point messages would be used resulting in a less efficient implementation than Class-Chord.

In [24] the authors modify Chord to support scalable publish/subscribe messages. To route the message they add all keys which match the subscription into the message that enables nodes to effectively route the message without overly burdening a single node. Class-Chord reduces the burden similarly, but does so by modifying the node ID to include the class attributes, and thus does not require the matching IDs to be included within the message.

In [25] the authors enable multi-attribute range queries by mapping attributes into one dimension using space filling curves. They use range queries to find content stored in the DHT. Their approach could be modified to set node IDs using the one dimensional values and create a new messaging type to support a class messaging scheme. They currently separate nodes by ID into sub-rings which would support static classes, but not dynamically created classes.

Another common approach using Chord is to modify the node IDs to be near to each other in some dimension. One example is Geo-Chord [26] which arranges nodes along the ring to minimize physical distance. This approach allows the ring to quickly send messages overall and along the single dimension could be used to create classes. For example, a

class of "all United States" would be quickly addressable. In all approaches similar to this, only a single static class is supported. Class-Chord supports multiple dynamic classes which can be based off of any number of attributes.

VI. Future Work

Class-Chord's efficiency requires changing the node IDs to incorporate information about the node's class. While this increases efficiency when sending messages, it adds costs when changing the information. For example, if a node's class specifier encodes its patch version, every patch update will cause the node to generate a new ID and re-join the overlay network. This is likely an infrequent event, and one already handled well using Chord's join and leave mechanisms as described in [2]. A larger challenge is when new class attributes are added globally to the network. For example, if the network administrators decided to start designating nodes by the amount of hard drive storage they possess. In these cases all nodes will need new IDs to incorporate the new information. In future research, we will investigate ways to re-generate IDs efficiently. For our SIEM example, the attributes change over days or weeks and don't pose a problem. However other problem domains may require more frequent global attribute modifications.

VII. Conclusion

In this paper we presented Class-Chord. A modified version of the Chord algorithm used to send messages to classes of nodes within the network. Class messages are routed through the network using a class specifier which enables any class attribute combination. Class specifier attributes can include exact values, ranges, and lists of valid values. We analyzed both the theoretical performance and provided results from a thorough test using our implementation. The results show that Class-Chord outperforms using the native Chord network's flooding or point-to-point messaging protocols.

References

[1] S. Bhatt, P. K. Manadhata, and L. Zomlot, "The operational role of security information and event management systems," *Security & Privacy, IEEE*, vol. 12, no. 5, pp. 35–41, 2014.

[2] I. Stoica, R. Morris, D. Karger *et al.*, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[3] K. P. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM*, 1993.

[4] S. Mishra, L. L. Peterson, and R. D. Schlichting, "Consul: A communication substrate for fault-tolerant distributed programs," *Distributed Systems Engineering*, vol. 1, 1993.

[5] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen, "The information bus: an architecture for extensible distributed systems," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5. ACM, 1994.

[6] D. Powell, "Group communication," *Communications of the ACM*, vol. 39, no. 4, pp. 50–53, 1996.

[7] H. W. Holbrook and D. R. Cheriton, "Ip multicast channels: Express support for large-scale single-source applications," in *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4. ACM, 1999.

[8] S. Kumar, P. Radoslavov, D. Thaler *et al.*, "The masc/bgmp architecture for inter-domain multicast routing," in *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4. ACM, 1998.

[9] S. Ratnasamy, P. Francis, M. Handley *et al.*, *A scalable content-addressable network*. ACM, 2001, vol. 31, no. 4.

[10] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware 2001*. Springer, 2001, pp. 329–350.

[11] B. Y. Zhao, L. Huang, J. Stribling *et al.*, "Tapestry: A resilient global-scale overlay for service deployment," *Selected Areas in Communications, IEEE Journal on*, vol. 22, no. 1, pp. 41–53, 2004.

[12] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[13] B. Cohen, "The bittorrent protocol specification," 2008.

[14] J. Warren, "Bitmessage: A peer-to-peer message authentication and delivery system," *bitmessage.org/bitmessage.pdf*, 2012.

[15] A. Adya, G. Cooper, D. Myers, and M. Piatek, "Thialfi: a client notification service for internet-scale applications," in *ACM Symposium on Operating Systems Principles*, 2011.

[16] R. Huebsch, B. Chun, J. M. Hellerstein *et al.*, "The architecture of pier: an internet-scale query processor," *Departmental Papers (CIS)*, p. 305, 2005.

[17] S. A. Baset, H. G. Schulzrinne, and E. Shim, "A common protocol for implementing various dht algorithms," 2006.

[18] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[19] G. Banavar, T. Chandra, B. Mukherjee *et al.*, "An efficient multicast protocol for content-based publish-subscribe systems," in *Distributed Computing Systems*. IEEE, 1999, pp. 262–272.

[20] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, 2001.

[21] B. Segall and D. Arnold, "Elvin has left the building: A publish/subscribe notification service with quenching," *Proceedings of the I997 Australian UNLX Users Group (A UUGÂŠI997)*, pp. 243–255, 1997.

[22] M. K. Aguilera, R. E. Strom, D. C. Sturman *et al.*, "Matching events in a content-based subscription system," in *ACM Principles of distributed computing*. ACM, 1999.

[23] R. Chand and P. Felber, "Semantic peer-to-peer overlays for publish/subscribe networks," in *Euro-Par 2005 Parallel Processing*. Springer, 2005, pp. 1194–1204.

[24] P. Triantafillou and I. Aekaterinidis, "Content-based publish-subscribe over structured p2p networks," in *Third international workshop on distributed event-based systems (DEBS)*. Citeseer, 2004, pp. 104–109.

[25] A. Sen, A. Islam, and M. Uddin, "Marques: Distributed multi-attribute range query solution using space filling curve on dths," in *Networking Systems and Security (NSysS)*, Jan 2015, pp. 1–9.

[26] A. Pethalakshmi and C. Jeyabharathi, "Geo-chord: Geographical location based chord protocol in grid computing," *International Journal of Computer Applications*, vol. 94, 2014.