Approximating the Number of Active Nodes Behind a NAT Device

Ali Tekeoglu, Nihat Altiparmak, Ali Şaman Tosun
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, 78249
Email: {tekeoglu,naltipar,tosun}@cs.utsa.edu

***Abstract*—Network Address Translation (NAT) is used for various reasons on the Internet and hides the IP address and number of nodes behind the NAT device. Although many applications benefit from the knowledge of number of active nodes behind a NAT device, existing schemes are limited. In this paper, we use TCP timestamp option to count the number of active nodes. Timestamp option includes current timestamp of the machine in the TCP packet. We propose an efficient scheme that counts the number of machines using clustering of timestamps. We use least-squares line fit of timestamp values and convex hulls to efficiently maintain the crucial information about existing clusters. Proposed scheme is online and requires minimal resources. We have investigated various aspects of the scheme to improve its performance. Using a developed tool to send packets, we have observed that the proposed scheme approximates the number of machines that send more than threshold number of packets well. Real experiments validate the proposed scheme.**

I. Introduction

NAT (Network Address Translation) [15] [14] [9] is the translation of an IP address used within one network to a different IP address known within another network. Although any machine that implements a NAT software can behave as a NAT device, it is generally implemented in routers. When a packet comes in to a NAT device, its "Destination IP" field is rewritten prior to forwarding it to a host behind NAT machine. The NAT software will keep track of this translation in a built-in table, and when the host sends a reply, it will translate back the other way.

Counting or approximating the number of active hosts behind a NAT device is important for many reasons. First of all, an ISP may want to know the number of computers connected to the Internet using a single IP address. Secondly, systems that need to grant certain privileges depending on the IP address, might need to know the exact number of clients using their services over that IP address. For instance, our starting point was monitoring the requests to streaming servers. A single client can overwhelm a streaming server by opening large number of RTSP connections. We can monitor the number of connections for each machine easily and place restrictions. However, nodes behind a NAT require special treatment since all of them appear to have the same IP address. Thus, the limits placed on NAT devices should be based on the number of active nodes behind a NAT device.

Partially Supported by US National Science Foundation (NSF) Grants CCF-0702728, CNS-0855247

II. Related Work

Recently, several papers on how to detect and approximately count the number of hosts behind a NAT device are published. In [2] and [7], IP header's ID field is used to differentiate hosts. This method works if the ID field is used as a simple counter as in Windows; however, most of the OSes do not implement the ID field as a counter. For example, Linux uses constant 0 and OpenBSD uses a random number for ID Field. Method presented in [6] relies on the TCP timestamp option. They present a simple way to compute the number of hosts behind a NAT, however their work does not include any experimental results nor do they evaluate the correctness of their methods. They try to find the 'number of increments per second' for each different machine and represent $timestamp$ as a linear function; $timestamp = numinbysec * sec + initialTimestamp$. The line equations found are kept and described as a different machine. Their method to find the $numinbysec$ is not robust and they rely on the line equation they formed using only the initial two packets; however, it is not realistic in a real network environment. In [11] authors presented a novel way to fingerprint devices using clock skews. They basically capture packets from the suspected machine and extract time dependent information from TCP Timestamp Options field and calculate unique clock skew for each device. They claim that one can use their clock skew detection methods to count the number of devices behind a NAT. However, their suggested technique works offline on packet traces and they did not provide neither implementation nor experimental results supporting their approach. Clock skew idea is expanded into a wireless setting in [8] to identify NAT devices in Wireless Sensor Networks. In [12] authors make use of IP-ID fields to detect subscribers using NAT devices in wireless networks. Another methodology is described in the following papers [17], [3], [5] and [4], in which authors utilized application layer information to discover NAT devices. They assume that instant messaging is one of the most popular softwares so they make use of instant messaging network packets to detect a NAT device. Machine learning techniques are used in [16]. They put the packets of suspected device through several analyzers. Each analyzer tries to determine

whether packets originate from users behind a NAT device. Analyzers are implemented making use of previously mentioned methods including IP-ID field, TCP Timestamps, Application Layer Information and Clock Skews. We present a solution to this problem using another approach which in turn requires less computational and spatial overhead than the other methods mentioned above. We support our idea with the implementation of the concept and provide experimental results for evaluation.

In this paper, we use clustering of timestamps to count the number of machines behind a NAT device. The rest of the paper is organized as follows: In Section III we give background information about the TCP timestamp option. Proposed scheme is explained in Section IV and experimental results are given in Section V. We conclude our paper with Section VI.

III. Background

A. Timestamp Option Field on TCP Header

Timestamp is a TCP option defined in RFC1323 [10], which carries two four-byte timestamp fields; the Timestamp Value field (TSval) and the Timestamp Echo Reply field (TSecr). TSval contains the current value of the timestamp clock of the TCP stack sending the option. When TSecr is not valid, its value must be zero. TSecr echos a timestamp value in the TSval field of a packet that was received from a remote machine at the other end of the TCP connection. TSecr is only valid if the ACK bit is set in the TCP header. Timestamps are created to be used for two distinct mechanisms: round trip time measurement and protect against wrapped sequences. A TCP stack may send the Timestamp option (TSopt) in an initial <SYN> segment and may send a TSopt in other segments only if it received a TSopt in the initial <SYN> segment for the connection.

B. Timestamps in Different Operating Systems

Different operating systems increment their timestamp clock in different periods. Linux started to support timestamp with kernel 2.1.9 and the initial value of timestamp is set to 0 when the system boots up. TSval in Linux is incremented every 10 milliseconds. Windows supports the TCP timestamp option starting with Windows 2000. TSval in Windows is 0 during the 3-way handshake but after the syn/ack handshake is complete, TSval is incremented every 100ms starting from an initial random number. Solaris supports timestamp starting from Solaris 2.5 with a frequency of 100 ticks/second. MacOS and OpenBSD also implement this option in a similar way as the above operating systems.

Since timestamps are incremented regularly with respect to time, time versus timestamp plots look like linear lines as shown in Figure 1. Each linear line in this plot, actually, corresponds to a separate machine. Two different machines would have the exact same line only in the unlikely case that their timestamps start from the same value at the same time and their OSes would have the same increment frequency. Since measurement is done at a different node than the one

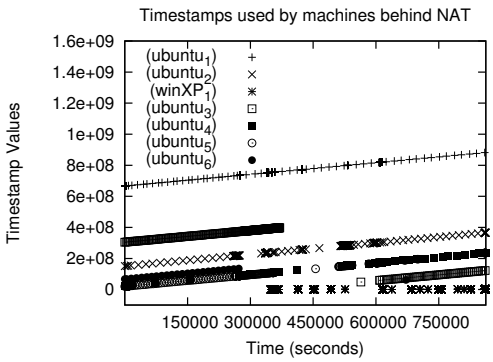


Fig. 1. Time behaviour of Timestamps

which generates the timestamp, there is some variation caused by network latency. We use these time vs timestamp graphs to count the number of distinct machines. Slope of Windows and Ubuntu machines are different as shown in the Figure 1 and a rebooted machine starts with a random timestamp value in the case of Windows or starts from 0 in case of a Linux machine.

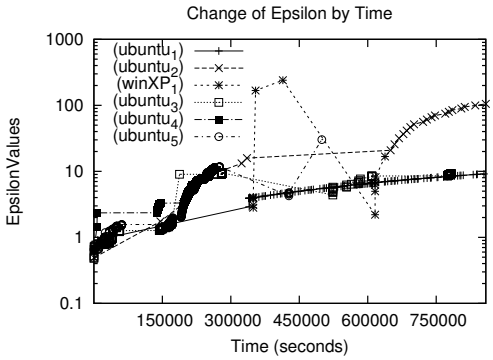


Fig. 2. Error from Line Fit

Due to delays and errors in measurements, time versus timestamp is not a linear line. Errors from least square line fit are shown in Figure 2 for the machines in Figure 1. Error or epsilon value, is defined as the maximum vertical distance of packets in the convex hull to the fitted line at the time a new packet is received. As shown in the figure Windows machine results in higher error than Ubuntu machines. Error values are proportional to the network congestion and they gradually increase during a long trace period, for instance the trace plotted in Figure 2 is taken over a week.

IV. Proposed Scheme

Our proposed scheme works by clustering timestamps of received packets into lines using least-squares line fit. It maintains convex hull of points to determine the quality of the clusters and to find the maximum distance to the lines. In order to improve performance we sort the lines according to time and reorganize them if the sort order changes. Last component of the approach is a merge operation that merges

clusters if they are very close to each other. We next describe these pieces in detail and finally we provide our algorithm.

A. Timestamp Property

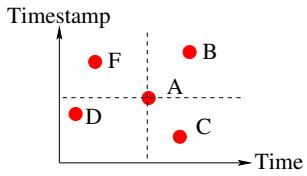


Fig. 3. Timestamp Property

Timestamp property requires that if two packets are from the same machine then the one with higher time value should have a larger timestamp. Consider Figure 3 for an example. In this figure packets A and B satistify the timestamp property. Packets D and A satisfy the property as well. However, neither A and C nor F and A satisfy the timestamp property. We make use of this property in proposed scheme when forming clusters, adding points to the clusters and merging clusters.

B. Least-squares Line Fit

In order to fit the data points ((X_i,Y_i) = (time, TSval)) into a straight line, we used linear least squares fitting method. The reason we are using this method is that it is calculated on the fly with a few fast operations. Using linear algebra, we can find the values of m and b, which gives us the equation of the best fitting line; $y = mx + b$.

Using this approach, we can update the best fitting line and threshold value on the fly while we capture new packets. Individual packet information need not be stored. The computation can be done in an efficient way using limited space. We maintain $\sum(X_i)^2$, $\sum X_i$,$\sum Y_i$, $\sum(X_i * Y_i)$ and i for each cluster. When new packets are received, these variables are updated based on the new packet (X_i, Y_i). This approach is computationally efficient since it just requires a few additional calculations to update the best fitting line and the threshold value.

C. Convex Hull

We used Convex Hull data structure to maintain boundary information about each cluster. Its use let us store minimal information. Instead of storing all points, convex hull contains the outer boundary points for each cluster. Convex hull of a set S is the smallest convex set that contains S.

Convex hull offers multiple advantages for our purpose. First, a small set of points are on the convex hull and they can easily be stored. In addition, using convex hull, maximum distance to the line can be computed easily since maximum distance from a data point to a line occurs at one of the points on the convex hull. Thus, to compute the maximum distance from a point set S to a dynamic line L, it suffices to store the convex hull $CH(S)$ of the point set S.

Only a small number of points end up being in the convex hull and this reduces the space requirement to compute the maximal distance. For the trace given in Figure 1 the number of points on the convex hull is given in Figure 4. Number of points on convex hull for Ubuntu machines is less than 10 and typically 3-6. For the single Windows machine we have the number of points on the convex hull goes as high as 22 and typically less than 15.

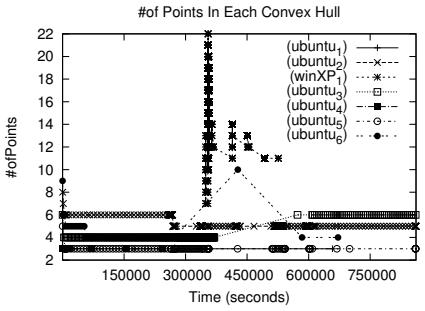


Fig. 4. Number of Points in Convex Hull

D. Line Sorting

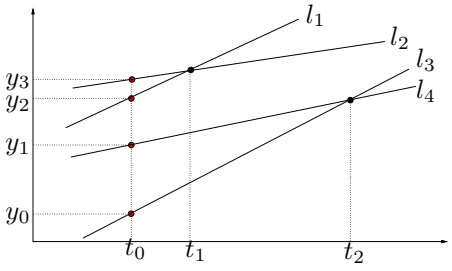


Fig. 5. Line Sorting

When a new packet is received, it needs to be compared with the existing clusters/lines in the system to find the best fitting line. This operation costs $O(N)$ if brute-force scheme is used. It is possible to reduce the complexity to $O(\log N)$ by sorting at a specific time. However, the sort order may change since lines have different slopes and can intersect each other. An example is given in Figure 5. At time t_0 the order of corresponding points are $line_3, line_4, line_1, line2$. At time t_1; $line_1$ and $line_2$ intersect and the order changes after this point. The way we handle this is based on maintaining the closest intersection point. Lines are sorted at current time value and closest intersection point of consecutive line intersections is maintained. Closest intersection point can be computed in $O(N)$ time if the lines are sorted and the sort order remains the same until the closest intersection point.

When new clusters are formed, new lines are added to the existing set of lines. We need a mechanism to find the closest line in this dynamic setting. Above scheme can easily be extended to include new lines. Consider Figure 6 for an example. A new line shown with dots is added. We can find its position at time t_0 and add it in its proper location. We also need to update closest intersection time since it can be closer than the previous value as shown in the Figure 6. The only

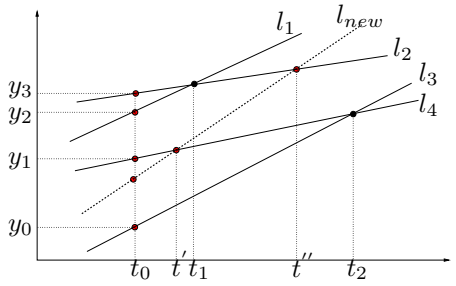


Fig. 6. Line Sorting with new line

intersections that need to be considered are the intersections of new line with the line above it and with the line below it. In this example, intersection of new line with line 4, which is above it, reduces the closest intersection point to time $t\prime$ from t_1.

Above discussion assumes that the lines are static once they are added. In reality, we update the lines as new points are added to the cluster. When a line is updated we need to find its intersection with the line above it and with the line below it and update the closest intersection point if necessary. This results in constant amount of work independent of the number of lines.

E. Merging Clusters

Based on the value of parameters used, sometimes packets for a given machine are split into multiple clusters/lines. Our approach is to detect these cases and merge the two clusters/lines. We use least square line fit for the set of points and maintain only the points on the convex hull for each cluster. When the clusters are merged, least square line fit and the convex hull needs to be updated to reflect the merge operation. To find the least squares line fit online we maintain $\sum(X_i)^2$, $\sum X_i$,$\sum Y_i$, $\sum(X_i * Y_i)$ and i for each cluster. By combining the values for two separate clusters we can find the least squares fit for the combined cluster. Constructing convex hull is easy as well. Convex hull of the union of two sets is equal to the convex hull of the union of the convex hulls of two subsets. These two properties make merge operation accurate as if all the points are maintained.

Before merging two clusters, we first check whether the two clusters logically belong to the same machine. We use *timestamp property* during the merge operation as well. Consider two clusters C_1 and C_2 and let H_i and L_i denote the high and low points of cluster i. If time of L_1 is larger than the time of H_2 then timestamp of L_1 should be larger than timestamp of H_2 as well. Similarly, If time of L_2 is larger than the time of H_1 then timestamp of L_2 should be larger than timestamp of H_1 as well. If these two properties are true then we call the clusters *mergable*.

Another condition for merge is the distance between the two clusters. In order to merge them they should be close enough to each other. We compute the center of gravity of each cluster and compute the distance to the best line fit of the other cluster. Minimum of these two distances is the metric we use to see how close the clusters are.

F. Algorithm

Proposed scheme uses an online algorithm to cluster the incoming timestamp values into lines. Detailed algorithm is given in Algorithm 1. Each cluster represents a line which represents a machine in the ideal case. A buffer of size N is used to store the packets. Initially, incoming packets are placed in the buffer and clustering is performed when the buffer reaches threshold number of packets for the first time. Once the clusters are created, a line L representing the points in that cluster is computed using least squares fit. For each incoming packet, binary search is used to find the closest line to it. If the distance between the packet and the closest line is $< \kappa$, the packet is inserted into that cluster and the line equation is updated. Lines are sorted according to current time and reorganized if the order changes. Packets that do not fit any of the clusters are kept in the buffer and the clustering process is repeated when the buffer reaches the dynamic threshold again. We have a minimum cluster size k and clusters with size less than the threshold k are destroyed and packets are inserted back into the buffer. This recovers from potential errors in clustering process.

Clustering process is explained in Algorithm 2. This algorithm is called whenever the buffer size reaches the dynamic buffer threshold value. Binary search is used to find the closest cluster and a metric ϵ is computed to check how good the fit is. If the point is a good fit based on the value of ϵ, and the *timestamp property* is satisfied, it is added to this closest cluster. Otherwise, it constitutes the first point of a new cluster.

The computation of distance from a packet to the closest cluster is based on the number of points in the cluster. If there is only one packet in the cluster, then the euclidean distance is used. If there are two or more points in the cluster, then the distance to the least squares fit line is computed.

When a new packet is added to the cluster, the value of the variables $\sum(X_i)^2$, $\sum X_i$,$\sum Y_i$, $\sum(X_i * Y_i)$ and i are updated and new least-squares line fit is computed. In addition, the convex hull of the cluster is updated if necessary.

If no packet has been received from a previously detected machine for a certain amount of time, it is marked as inactive and removed from the set of active nodes. Convex hull always includes the last packet sent from a machine and time value of last packet is used to detect whether the cluster is active.

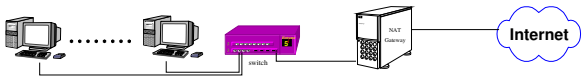


Fig. 7. Network Setup

V. Experimental Results

In this section, we provide experimental results for the proposed scheme using the system in Figure 7. We tested the system in two different ways; first using synthetic packet sending tool running on a single machine behind NAT, second using real traffic generated by the machines behind NAT.

Algorithm 1 Detect number of machines behind NAT device

```
// k; min number of points to cluster
// ε_i; max error to be accepted for closeness to line i
// κ; error constant
// clusterThreshold; Initially equals to Buffer Size, N
malloc buffer[N]
clusters ← ∅
lines ← ∅
while newPacketCaptured() do
    fitsToLine ← false
    if newPacket.Time > nextIntersection.Time then
        lines ← reorganizeLines(lines, newPacket.Time)
        nextIntersection ← foundNextIntersection(lines)
    closestLine ← binarySearchClosestLine(packet)
    dist ← distance(packet, closestLine)
    if dist < κ then
        updateLineEquation(packet, closestLine)
        fitsToLine ← true
    if fitsToLine ≡ false then
        buffer.append(newPacket)
    if buffer.size() == clusterThreshold then
        for all packetInBuffer p=1 to N do
            closestLine ← binarySearchClosestLine(packet)
            if distance(p, closestLine ) < κ then
                updateLineEquation(p, closestLine)
                removePacketFromBuffer(p)
        clusters ← clusterBuffer(buffer[N])
        buffer.clear()
        for all clusters i=1 to K do
            if clusters.get(i).numberOfElements < k then
                buffer.append(clusters.get(i).allPoints())
                clusters.remove(i)
        clusterThreshold ← updateClusterThreshold(
        buffer.size() )
        newLines ← lineFit(clusters)
        newLines ← sort(newLines, newPacket.Time)
        lines ← combineSortedLines(newLines, lines)
        lines ← mergeLines(lines)
        lines ← checkInactiveLines(lines)
    NumMachines ← lines
```

Algorithm 2 clusterBuffer($buffer[]$)

```
clusters ← ∅
for all packets in buffer i = 1 to N do
    if clusters.size() > 0 then
        closestCluster ← binarySrchClosestClstr(buffer[i])
        dist ← distanceToCluster(closestCluster, buffer[i])
        numOfPoints ← closestCluster.size()
        if numOfPoints == 1 then
            ε ← threshold × κ
        else
            ε ← κ
        if dist < ε then
            closestCluster.appendToCluster(buffer[i])
        else
            clusters.append( new Cluster(buffer[i]) )
    else
        clusters.append( new Cluster(buffer[i]) )
return clusters
```

A. Capture Filter

Our goal is to develop a light-weight high performance tool to count the number of nodes behind the NAT device. Proper specification of packet filter is essential for high throughput packet processing. A packet filter is a kernel facility to classify packets on the line and forward the packets selected by the filter to user applications without traversing the network stack.

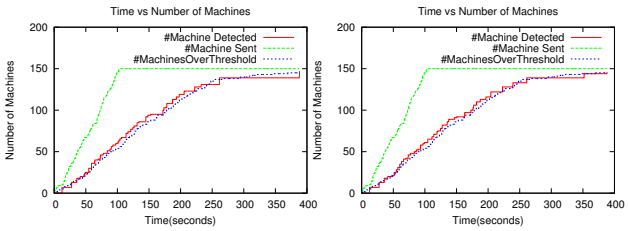
For sending and capturing packets, we used Jpcap [1] library. Jpcap uses pcap (libpcap/WinPcap) filter language as a capture filter and there is $setFilter(stringfilterExpression, booleanoptimize)$ method in jpcap library to create, compile and activate a filter from a filter expression. Utilizing this function, we set an efficient kernel level filter for selecting timestamped packets.

B. Synthetic Tool Experiments

We first wanted to test our scheme implementing a packet sending tool that can simulate the timestamp behavior of several different machines. Although this tool runs on one of the machines behind NAT device in Figure 7, it is able to generate outbound network traffic of hundreds of machines by sending crafted packets.

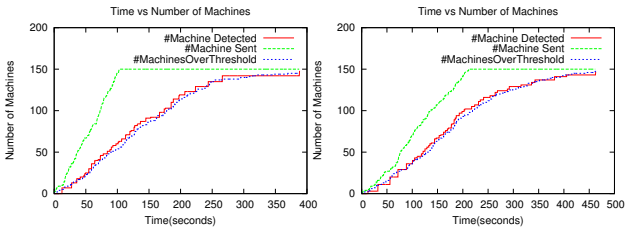
1) Packet Sending Tool: In order to test our proposed system, we created a tool to send artificially timestamped packets with different source IP addresses to a specified destination IP address. For packet creation, we used Jpcap library. There are two IP lists in our testing system. The first IP list, $ChoosenList$, includes the source IPs which are used to send a packet before. The second IP list, $AvailableList$, has source IPs that has never been used to send a packet. In the initial phase, the $AvailableList$ includes specified number of source IPs but the $ChoosenList$ is empty. For each source IP in the $AvailableList$, a random timestamp value ranging from 1 to 10^9 is assigned. Once an IP is chosen from the $AvailableList$, that IP is extracted from $AvailableList$ and put into the $ChoosenList$. We use a variable p which is the probability of sending a packet from a chosen IP picked up from the $ChoosenList$. Finally, we send the packets from chosen source IP to the specified destination IP with the chosen value of burst length. For each source IP address, there is a counter to keep track of the number of packets sent from that specific source IP address. The reason for counter is that it is difficult to detect machines that sent only a few packets. Therefore, we defined a threshold value t and each source IP address sends at least t packets. When the $AvailableList$ becomes empty and all the source IPs in the $ChosenList$ used to send at least t packets, test program terminates.

2) Number of Machines Detected: We plotted actual number of machines sending packets (#Machine Sent), number of



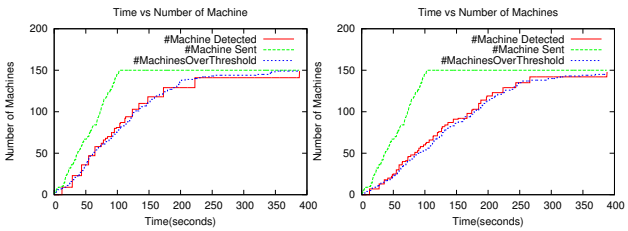
(a) Linux=%30, Windows=%70 (b) Linux=%70, Windows=%30

Fig. 8. Effect of Linux/Windows Ratio, for P=0.6



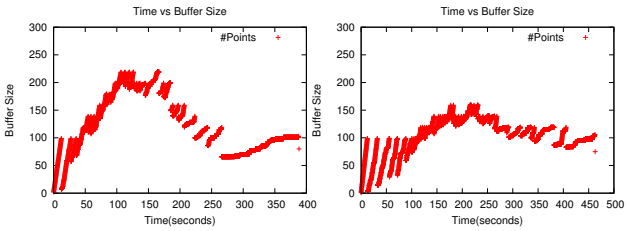
(a) Probability=0.6 (b) Probability=0.8

Fig. 9. Effect of P, for Linux/Windows Ratio=%50



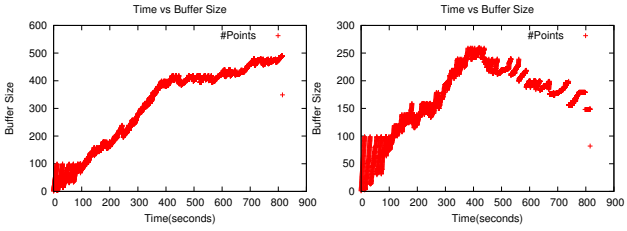
(a) Minimum Cluster Size=3 (b) Minimum Cluster Size=5

Fig. 10. Effect of Minimum Cluster Size



(a) Probability=0.6 (b) Probability=0.8

Fig. 11. Effect of P on Buffer Size, for Linux/Windows Ratio=%50



(a) Kappa=10 (b) Kappa=50

Fig. 12. Effect of Kappa on Buffer Size, with Machine Size=255

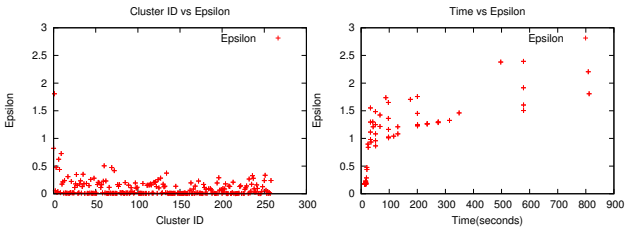
machines sending at least threshold number of packets (#MachinesOverThreshold) and the number of machines detected by proposed scheme (#Machine Detected). Figure 8 shows results for $p = 0.6$. As shown in the figure, proposed scheme closely approximates the number of machines if each machine sends at least threshold number of packets, which was set to 5. We also looked at the effect of p on the results. New machines are introduced at a slower rate for large values of p. The results are given in Figure 9. Detected machine count is closer to the actual count when p is higher. Number of machines detected for different minimum cluster size given in Figure 10. When a higher value is used for minimum cluster size, the accuracy is slightly better. Experiments show that the value of κ does not have a large impact on the number of machines detected However, the number of machines are overestimated when κ is set too low.

3) Buffer Size: We investigated the effect of various parameters on the number of packets in the buffer over time. The effect of p is given in Figure 11. When machines are introduced at a slower rate, the size of the buffer is smaller since clusters are more likely to have more than threshold packets necessary to form a cluster. As shown in the Figure 12, when lower value is used for κ, packets that are further away from the cluster end up in the buffer increasing the number of packets in the buffer. These packets remain in the buffer at the end of experiment since these outlier packets do not fit any cluster. When minimum cluster size is larger, packets stay in the buffer longer increasing the buffer size.

4) Impact of Epsilon: We investigated how the value of ϵ (distance from most distant node of convex hull to the line) changes for each cluster and over time. The results are given in Figure 13. Figure 13(a) shows the values for $\kappa = 10$. The largest ϵ observed in a cluster is 1.8. Majority of the values are less than 0.5 indicating that the clusters are compact. The change of ϵ over time for a single cluster is given in Figure 13(b). The values of ϵ changes over time and it can both increase and decrease when nodes are added.

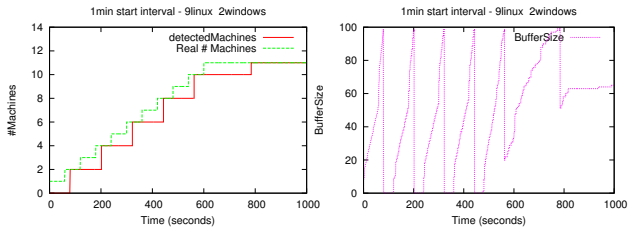
C. Experiment on a Real Network Environment

We also verify that our implementation of proposed scheme works on a real network environment. For this purpose a gateway machine running Debian/Linux was configured as a NAT device and 2 Windows, 9 Linux machines are connected to this NAT gateway over a switch to access web over it. This setting is depicted in Figure 7. Setup included Windows and Linux machines to test how our program handles different



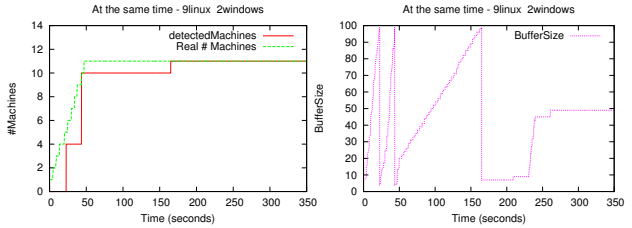
(a) Epsilon of each cluster at the end (b) Change of Epsilon for a line

Fig. 13. Impact of Epsilon



(a) Number of Machines (b) Buffer Size

Fig. 14. One minute interval between each connection



(a) Number of Machines (b) Buffer Size

Fig. 15. All connections established at the same time

operating systems. Due to lack of equipment we could not extend the experiment with more machines.

Two different experiments are conducted on this network setup. In the first experiment shown in Figure 14, connections from the machines behind NAT gateway are established with one minute intervals. After about 11 minutes, all of the machines are sending packets over NAT. On the second experiment, machines established their connection with a remote machine at the same time. With these experiments, scaling of the implementation was tested and as could be seen in the Figure 15 program was able to handle the connections quite well even if all the machines generate traffic at the same time.

When we look closer in Figure 14, program detects the number of machines precisely when its buffer gets filled up since only at that moment captured timestamps in the buffer are clustered. After all the machines are introduced, number of unmatched packets in the buffer stabilizes, because captured timestamps are found to belong to an already detected machine represented as a cluster.

Figure 15, shows another scenario with a more frequent machine introduction rate, in which program detects the number of machines precisely after it clusters the buffer. In Figure 15(b) the number of packets in the buffer increases slowly between 50-150 second interval since there is only one undetected machine during that time. When the buffer fills up, program clusters the packets and detects the last machine. In both figures, packets that are not close enough to any machine's line equation remain in the buffer for a certain amount of time, then they are discarded completely. Reducing the buffer size will enhance the spontaneity of the program, while overloading it with more frequent clustering operations.

VI. Conclusion

Many applications benefit from accurate count of active nodes behind a NAT device. However, existing schemes are limited. In this paper, we use TCP timestamp option to count the number of active nodes. Timestamp option includes current timestamp of the machine in the TCP packet. We propose an efficient scheme that counts the number of machines approximately using clustering of timestamps. We use least-squares line fit of timestamp values and convex hulls to maintain limited information about existing clusters. Proposed scheme is online and requires minimal resources. We have investigated various aspects of the scheme to improve its performance and by combining methods from different areas of computer science we developed a novel approach. Using a developed tool to send packets we have observed that the proposed scheme approximates the number of machines that send more than threshold number of packets well. Proposed scheme further tested on a real environment of networked computers. Future work includes improving the performance and accuracy of the tool further.

References

[1] A java library for capturing and sending network packets. URL: http://netresearch.ics.uci.edu/kfujii/Jpcap/doc/index.html.

[2] S. M. Bellovin. A technique for counting natted hosts. In *Internet Measurement Workshop*, pages 267–272, 2002.

[3] J. Bi, M. Zhang, and L. Zhao. Security enhancement by detecting network address translation based on instant messaging. In *Emerging Directions in Embedded and Ubiquitous Computing*, pages 962–971, 2006.

[4] J. Bi, M. Zhang, and L. Zhao. Application presence information based source address transiton detection for edge network security and management. In *International Journal of Computer Science and Network Security*, page 147, 2007.

[5] J. Bi, L. Zhao, and M. Zhang. Application presence fingerprinting for nat-aware router. In *Knowledge-Based Intelligent Information and Engineering Systems*, pages 678–685, 2006.

[6] E. Bursztein. Time has something to tell us about network address translation. In *Nordic Workshop on Secure IT Systems*, Nov. 2007.

[7] M.I. Cohen. Source attribution for network address translated forensic captures. *Digital Investigation*, 5(3-4):138 – 145, 2009.

[8] Chih-Yuan Wang Hsuan-Yu Huang Ding-Jie Huang, Wei-Chung Teng and Joseph M. Hellerstein. Clock skew based node identification in wireless sensor networks. In *IEEE GLOBECOM 2008*, 2008.

[9] K. Egevang and P. Francis. RFC1631: The ip network address translator (nat), 1994.

[10] V. Jacobson, R. Braden, and D. Borman. RFC1323: Tcp extensions for high performance.

[11] Tadayoshi Kohno, Andre Broido, and K. C. Claffy. Remote physical device fingerprinting. In *In IEEE Symposium on Security and Privacy*, pages 211–225. IEEE Computer Society, 2005.

[12] Li Erran Li, Aiyou Chen, Tian Bu, and Scott Miller. Detecting subscribers using nat devices in wireless data networks. *Bell Lab. Tech. J.*, 14(2):223–233, 2009.

[13] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. The 1999 darpa off-line intrusion detection evaluation. *Comput. Netw.*, 34(4):579–595, 2000.

[14] P. Srisuresh and K. Egevang. RFC3022: Traditional ip network address translator (traditional nat), 2001.

[15] P. Srisuresh and M. Holdrege. RFC2663: Ip network address translator (nat) terminology and considerations, 1999.

[16] O. Zakin, M. Levi, Y. Elovici, L. Rockach, N. Shafrir, G. Sinter, and O. Pen. Identifying computers hidden behind a nat using machine learning techniques. In *The 6th European Conference on Information Warfare and Security*, pages 335–340, 2007.

[17] L. Zhao, M. Zhang, J. Bi, and J. Wu. Detecting private address space based on application layer information. In *The First IEEE Workshop on Adaptive Policy-based Management in Network Management and Control*, April 2006.