Lessons for Autonomic Services from the Design of an Anonymous DoS Protection Overlay

David Ellis and Ian Wakeman

University of Sussex

Abstract. In this paper we report on the design and implementation of a Denial of Service protection overlay, and draw lessons for autonomous services. Our approach is novel in that each node is only aware of a subset of the other nodes within the overlay; the routing topology of the overlay is hidden from internal and external nodes and the overlay uses a distributed monitoring and trust system to detect misbehaving nodes. In meeting these design goals, we have had to move beyond the normal approaches to designing self-configuring and self-monitoring services, and we highlight these issues as being important for the design of future multi-organisation systems.

1 Introduction

Autonomic services attempt to provide services without human intervention. To this end, the services typically are designed so that the various distributed components work together to self-configure and self-regulate in the face of changing demands and resource availability. Nearly all such services follow the following paradigm for their design:

1. Decide on a desirable set of states to be maintained by the system
2. Find a set of local measurements and variables which can be used as indicators of the distance from the desired goal state. These can be solely recovered from local state, or can be found from other machines in the local vicinity.
3. Find a local action which moves the global state closer to the desired state
4. Repeat continuously.

The design problems within a single organisation are identifying the local and global states and actions which allow the service to be built, and which provide at least quasi-stability.

However, when the services are provided by distributed components owned by different organisations, then the design problem becomes much more complicated. Each organisation may be sensitive about revealing information to other organisations. We may therefore have to restrict ourselves to state which can be obtained from solely local sources. Worse, the system must be designed to deal with machines which misbehave, both accidentally and intentionally. We have to design the system to cope with denial of service attacks upon the information exchange and control mechanisms. The system will have to make judgements

upon whether information is valid, either by using consensus techniques, or by using policies set up by the system administrators.

In this paper we present the design and implementation of a Denial of Service Protection overlay, similar to SOS [1] and Mayday [2]. However, rather than assuming that each of the nodes within the overlay can be completely trusted, we have aimed to provide a design that can work with untrusted nodes. In this way, we believe the protection overlay can be used across multiple administrative domains and machines safely.

Our design has several novel features, applying design principles suitable for application to sutonomous services between untrusted components:

- Each internal node is only aware of a small subset of other nodes, and the full set of internal nodes is difficult to discover for any attacker.
- The internal routing topology of the overlay is hidden from each of the internal nodes, and from any of the external nodes. This makes it more difficult to launch DoS attacks against key nodes or links within the overlay. Performance is still comparable to other systems.
- The overlay uses a distributed monitoring and trust system to detect which nodes are misbehaving, and subsequently remove them from routing of traffic.

In the following sections, we present the design of the overlay; provide brief descriptions of the construction, routing, choking and trust maintenance protocols for the overlay as appropriate for autonomous services and present results from simulation within NS2 and experience on PlanetLab.

2 Design Overview

DoS protection overlays function through providing a larger set of ingress points for a service, thus providing increased inbound bandwidth, and allowing the actual machine providing service to reject all traffic apart from the known egress points from the overlay. In addition, the overlay should be able to detect the increased traffic from a DoS attack, and react so as to filter this traffic from within the overlay. The key question we have asked ourselves in this work is whether the overlay can be formed from nodes across multiple administrative domains, even to the extent of using the home machines of individual users? Since untrusted nodes can therefore join the overlay, along with machines which are easier to compromise by attackers, we want to design a protection overlay in which the nodes have very limited knowledge about the overlay structure, there is no central control, yet the system still auto-manages. Our basic requirements from the overlay are thus:

- The overlay should reform itself as nodes join and leave the overlay without disrupting connectivity for the end-point services.
- Routing must work without nodes having explicit knowledge of the topology.
- The overlay should identify DoS flows, and filter these flows at the entrance to the overlay.

– The overlay must monitor itself and react to misbehaving member nodes of the overlay.

In terms of the paradigm for autonomous services, each member of the overlay treats the rest of the overlay as a blackbox, making measurements without knowledge of the other members. If nodes may be untrustworthy, then each decision should be independent of other nodes.

2.1 Overlay Construction and Maintenance

```
Node allNodes, connectedNodes;

function maintainDegree
  if degree == 0 then
    ingress = pick(ingressSet);
    connect(ingress);
  elseif degree < MINDEGREE then
    pullInfo();
  elseif degree > MAXDEGREE then
    n = select(connectedNodes);
    pushInfo(n);

function pushInfo(n)
  nlist = select(allNodes);
  dropConnection(n);
  n.receivePush(nList);

function receivePush(nList)
  foreach n in nList
    allNodes.add(n);
  while(!connect(select(nList)));

function pullInfo
  n = select(allNodes);
  n.getInfo(this,parent(n));

function getInfo(n, referrer)
  if referrer.isValid(n) then
    nlist = select(allNodes);
    n.receivePush(nList);
```

Fig. 1. Gossip pseudo code for graph generation and maintenance

Overlay construction must work despite having limited information. We have based our overlay construction algorithm upon the Gossip protocols of Jelasity

et al [3]. We have found that such gossip protocols are an excellent match for the composition of autonomous services. They are fast, work well with limited knowledge, and work well to auto-regulate the configuration.

The key requirements of the topologies are:

- The graph should have a high degree of randomness, so that the topology cannot be inferred and extrapolated from a small subset of topological information.
- The graph should be fully-connected with respect to the egress nodes to the end-points.
- The graph should mirror the underlying network so as to reduce the problem of latency stretch.
- There should be a variety of disparate paths between ingress and egress nodes.
- There should be no single points of failure.
- Knowledge of the full topology should be hidden from nodes within the network.

To join the overlay, a node receives the list of public ingress nodes to the network. It then connects to one of the ingress nodes, and executes a pullInfo upon the node. From this node, it selects a node to connect to, and if the connection is successful, this becomes one of the node's outgoing links. The pseudo-code for this is shown in Figure 1.

MINDEGREE is set according to the expected size of the overlay network. We wish to ensure that the overlay graph is fully connected, so following the standard theory of random graphs [4], we need to ensure that for a graph of size N, the average degree of the graph k is $> ln(N)$. We therefore set MINDEGREE to be equal to $ln(N)$. Currently this is set statically, but it should be trivial to allow the degree to be set dynamically according to estimates of overlay size from the ingress nodes.

The public ingress nodes are aware of a large proportion of the nodes within the overlay. However, to have been selected as public ingress nodes, the nodes must have demonstrated themselves to be trustworthy as described below, and we accept the risk of discovering composition from suborning one of the public ingress nodes.

To prevent a node from crawling the network using a sequence of getInfo exchanges, we require each node to also pass across the address of a node which is connected to the queried node. This node can then be used to check the validity of the requester, and to reject the request if too many getInfo requests are being generated.

2.2 Route Learning

Since we wish to prevent dissemination of topology information, routes have to be learnt by experimentation. We therefore have devised a route learning technique based upon the use of probe packets sent out to the end-points within the overlay.

An *rlearn* packet is sent out down an outgoing link from an ingress node to probe for a route to a specified end-point. As the packet passes through nodes, these nodes record the passage of the *rlearn* packet. If the *rlearn* packet reaches an egress node responsible for the end-point, then it constructs an *rlearnResponse* packet which is returned along the return route. If a response is received, then we update the rtt statistics to that endpoint out of the corresponding outgoing link and the probability of using that outgoing link for that end-point is increased. We provide the pseudo-code representing the learning algorithm in Figure 2. Rtt statistics are collected and maintained using standard EWMA approaches, as

```
function rlearnSend(RlearnPkt p)
  nextNode = pickNode(outgoingNodes);
  rlearnEntry.from =p.src;
  rlearnEntry.timestamp = now;
  rlearnEntry.to = nextNode;
  rtable[p.mark] = rlearnEntry;
  send(p,next);

function rlearnReceive(RLearnRespPkt p)
  discardInvalidPackets();
  r = rtable[p.mark];
  rtt = now - r.timestamp;
  updateRtt(r.service,r.to,rtt);
```

Fig. 2. Pseudo-code for route learning

used in TCP. To compile the rtt statistics into the forwarding table probabilities, we calculate probabilities using the following equation

$$routingtable(i,s) = \frac{rttstore(i,s)}{\sum_{j=0}^{n} rttstore(j,s)}$$

These probabilities are used to bias the choice of outgoing link for a given end-point.

The frequency with which *rlearn* packets are generated is a tunable parameter which affects the rate at which routes are updated and how the network reacts to changes in the underlying topology. We set the frequency of *rlearn* generation based on the round trip times currently experienced by the node.

2.3 Choking and Pushback

Each node monitors the bandwidth used by each incoming link. In the autonomous systems paradigm, this is a local measurement which indicates how far the system is from its goal state. If any incoming link exceeds per-defined

thresholds, then the flow is throttled within that node, and a choke request is sent back through the incoming link, the controlling action with the control paradigm. When a choke message is received from an outgoing link, the receiving node reduces its outgoing bandwidth to the sending node. This may cause the receiving node to begin to drop packets, in which case a restriction is sent further upstream. As a side effect the probabilities associated with using that outgoing link will be reduced, as other links with more bandwidth will drop less packets. Nodes may implement per flow restrictions if they wish but the lower granularity is intended to fuel a simpler trust relationship model. Note how the measurement and control action allows nodes to make local policy decisions about what to accept and what to reject. Figure 3 shows the calculation of upstream virtual

```
function pushback(VirtualBandwidths, reduceAmount, avgDropped)
  let tbw = temporary array
  foreach upstream node
    VirtualBandwidths[node] =
      reduceAmount * avgDropped[node] / VirtualBandwidths[node]
    if VirtualBandwidths[node] < 0 then
      VirtualBandwidths[node] = 0
```

Fig. 3. Pseudo-code for pushback/choke

bandwidths. This algorithm is run periodically, to calculate any upstream bandwidth restrictions. Choke messages are sent upstream only if vbw has changed by some amount, we set this value 0.1. The amount the bandwidth is reduced by is set by another tunable parameter reduceAmount. If this amount is very high, it is equivalent to a complete block from an upstream node.

According to the rules above, there is no way for a node to re-establish its virtual bandwidth. So an additional thread must be willing to increase bandwidth if the upstream node is now operating within the set thresholds. This algorithm is presented below:

```
function creep(VirtualBandwidths, creepAmount, threshold, avgDropped)
  foreach upstream node
    if(avgDropped[node] < VirtualBandwidths[node] *threshold)
     VirtualBandwidths[node]*=creepAmount
```

Fig. 4. Pseudo-code for pushback/choke creep algorithm

Figure 4 shows the algorithm we used to creep the upstream bandwidths up after they had been reduced. The threshold is needed because we want to

allow links with some packet loss increase particularly if the virtual bandwidth is particularly low. How often creep is run is also a tunable parameter and must be carefully balanced against the virtual bandwidth calculations.

2.4 Distributed Monitoring and Trust

Our overlay is built with nodes belonging to multiple organisations and requiring distributed administration (MODA). Since the the different organisations do not necessarily trust one another, there is a need to provide evidential trust measures between the node and the organisations, which can be used to moderate the probabilities of routing through any given node. In particular, if a node is suborned by a malicious third party, we want the system to be self-monitoring to detect anomalous behaviours and isolate the suborned node from the overlay.

We take as our inspiration the Eigentrust work from Kamvar et al [5], in which the transitive trust relationships between nodes are used in the iterative calculation of the left eigenvalue of the normalised trust matrix. The resultant values show the relative levels of trust measure accorded to each node, and can be used to identify suspicious nodes.

In our system, we are trying to detect nodes which either inject traffic into the overlay, or drop traffic unnecessarily. To this end each node records the matrix of incoming (R) and outgoing traffic (S) from its neighbour nodes. These matrices are sent to a central trusted traffic monitor which calculates and monitors the nodes. The traffic monitor combines the received S and R matrices in the following fashion:

$$C = \sum_{i=0}^{N} (S'_i + R_i) + (S'_i + R'_i)$$

The global matrix C is normalised over the row totals, and the left hand eigenvalue is calculated by iterative multiplication of C. Note that the Ingress nodes and egress nodes ensure that the matrix is in general irreducible and aperiodic, and the left eigenvalues will converge. Nodes which inject or eject traffic emerge as having values close to one.

Our current implementation uses a centralised node to collect, calculate and monitor the traffic matrices. We are currently working on techniques to fully distribute the trust calculation, in line with the design principles of autonomous services.

3 Simulation Results

We have developed our algorithms within a homegrown simulator, and then have used NS2 to verify our simulation results. In the following simulations, we use topologies of 800 nodes, generated from the Georgia Institute of Technology's topology generator [6]. The overlay has 5 ingress nodes, 40 internal nodes, and 5 egress points to the 2 end-points. There are 10 clients randomly selected which

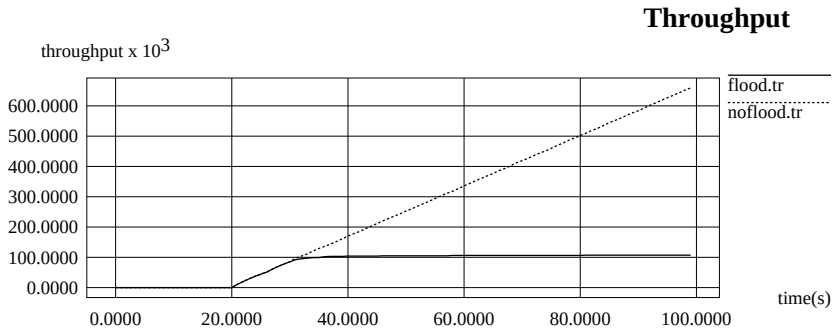

Fig. 5. The effect of a DDOS attack against the network when no pushback is used

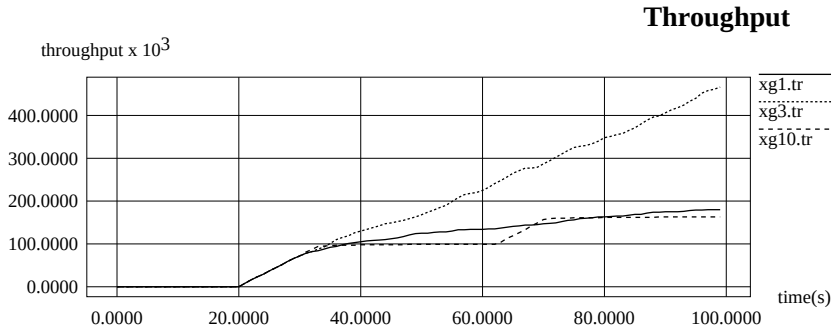

Fig. 6. Using pushback with various creep settings to mitigate a DDOS attack

connect to a random ingress point and continually send traffic to the end-point using TCP. For reasons of space, we show only how the total throughput is protected by the overlay. For more complete results from simulation, see [7]. In the following graphs, the throughput is the total bytes received, whilst the RTTs are measured in milliseconds.

Figure 5 shows shows how the overlay deals with a DoS attack from one ingress node in the absence of pushback and choking. When a flood occurs, the bytes received are flattened, and an effective DOS attack has been accomplished. When there is no flood about 600×10^3 bytes get transferred. The graph in figure 6 shows the same scenario but with pushback/choke enabled. We used three settings indicating how quickly the algorithm attempts to "creep" (see section 2.3). The values were 1 second (xg1.tr); 3 seconds (xg3.tr) and 10 seconds (xg10.tr). We can see that when we creep every three seconds we restore the clients throughput to two thirds of what it was when no attack occurred. You can also see that creeping too frequently or not frequently enough hinders the recovery process. This is because creeping too frequently will restore the flooders bandwidth too quickly; and creeping too infrequently will not allow legitimate clients to recover from pushback requests.

4 Implementation on PlanetLab

Many systems perform well within simulation, but fail when deployed on the Internet. By applying the the principles of autonomous services design, we believe that our system will be robust to the various problems that are created within the Internet. We have therefore deployed our code within PlanetLab as a Perl implementation. To pass between nodes, the end-points are included as the Satnet IP packet option, and we use standard NAT translation techniques to transfer the packets within the overlay. The Satnet IP option [8] was used originally to pass stream identifiers through nodes which didn't recognize streams. We have re-used this option to carry the end-point identifiers between nodes within the overlay. Although it has been reported that IP options are not carried through many of the Internet routers, the connections between many of the PlanetLab nodes do allow IP options through, and we have successfully rate limited traffic at low bandwidths. Obviously, we have not been able to run full DoS style attacks within the infra-structure, but the results so far have been encouraging.

For the following results, we measured the performance of the overlay over a 24 hour period on PlanetLab. We used 80 nodes within the overlay, with 5 ingress nodes and 5 egress nodes. Every 15 seconds, we measured the round trip time from the ingress nodes to the egress nodes, both through the overlay, and directly using a ping measurment, and the number of hops. As can be seen, the average latency stretch is very respectable given the problems of context switching on Planetlab nodes, and the system reacts well to the vagaries of PlanetLab connectivity.

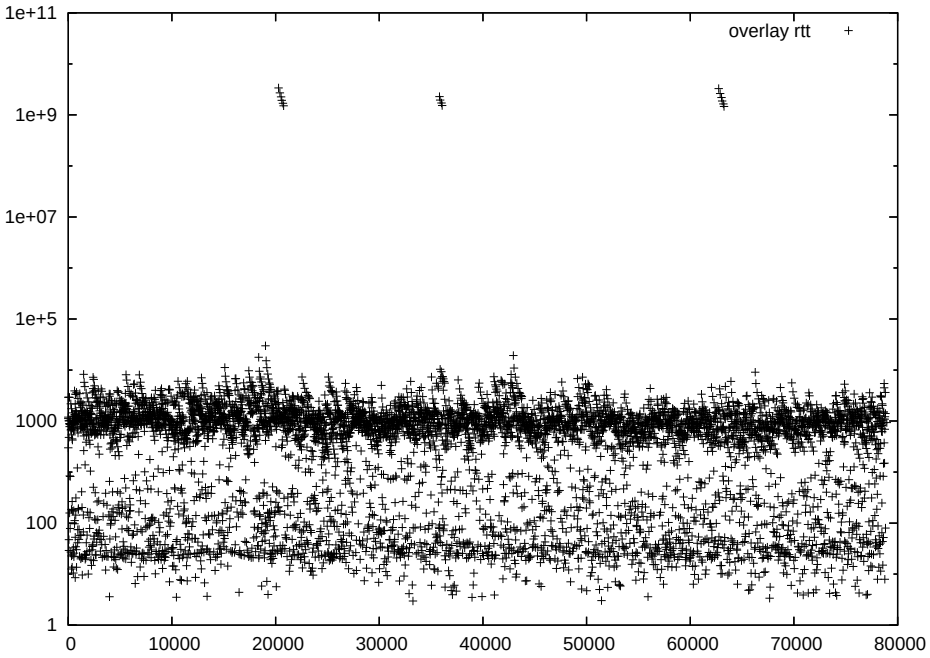

Fig. 7. Round trip time for overlay measurements in milliseconds

5 Conclusion

We have presented the design and implementation of an anonymous DoS protection overlay network. Our simulations and experience on PlanetLab show that the approach can effectively reduce the effect of a DoS attack. If nodes can be offered an incentive for participation, such as the use of micro-payments, then such schemes as ours may find a niche for collaborative protection of small to medium scale web sites and services.

We currently do not attempt to maintain packet ordering within a flow. If we were to attempt to pin a route to a flow as soon as we identified a flow, then entire flows may be sent into routing black holes within the overlay, which would be unacceptable. Instead, we are investigating maintaining flow state, and pinning a route to a flow once an acknowledgment has returned.

We have show that the careful design of the overall system to use black box measurements and local actions which can be controlled by local policy can lead to autonomous services between untrusted nodes. We believe that these principles can help build robust services in the future.

References

1. A. Keromytis, V. Misra, and D. Rubenstein. Sos: Secure overlay services. In *SIGCOMM*, Pittsburgh, PA, August 2002.

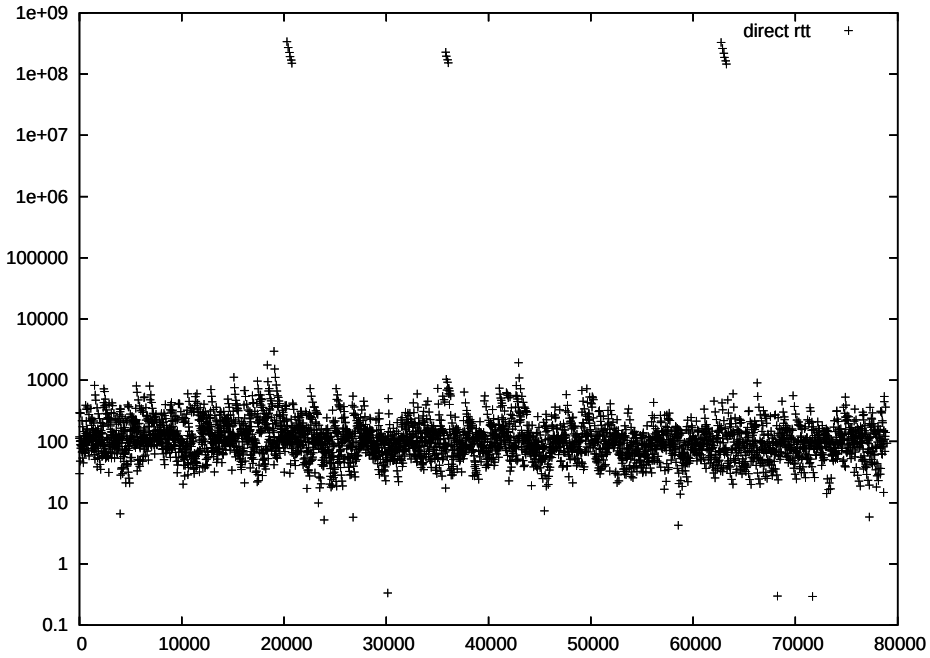

Fig. 8. Round trip time for direct measurements in milliseconds

2. David G. Andersen. Mayday: Distributed filtering for internet services. In *4th Usenix Symposium on Internet Technologies and Systems*, Seattle, Wa, March 2003.
3. Mrk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, August 2005.
4. R. Albert and A.-L. Barabsi. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74:47–97, 2002.
5. Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the Twelfth International World Wide Web Conference*, 2003.
6. Ellen W. Zegura, Ken Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE Infocom*, San Francisco, Ca., 1996.
7. David Ellis and Ian Wakeman. Design and implementation of an anonymous dos protection overlay. In *Submitted for publication*, 2006.
8. J.B Postel. Internet protocol. Technical Report RFC791, IETF, September 1981.

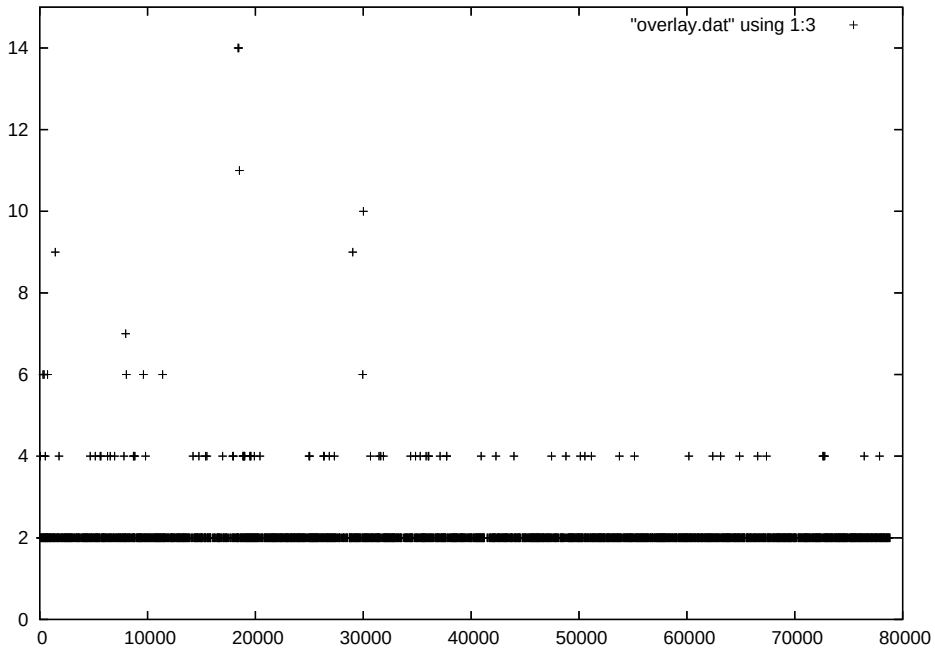

Fig. 9. Number of hops across the overlay