Delft University of Technology
Parallel and Distributed Systems Report Series

Dispersy Bundle Synchronization

Niels Zeilemaker, Boudewijn Schoon, Johan Pouwelse
niels@zeilmaker.nl

Completed January 2013.

Report number PDS-2013-002

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Group
Department of Software and Computer Technology
Faculty Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.ewi.tudelft.nl

Information about Parallel and Distributed Systems Section:
http://www.pds.ewi.tudelft.nl/

Abstract

In this technical report we present Dispersy, a fully decentralized system for data bundle synchronization, capable of running in challenged network environments. Key features of Dispersy are stateless synchronization using Bloomfilters, decentralized NAT traversal, and data bundle selection algorithms that allow the system to scale over 100,000 bundles in the presence of high churn and high-load scenario's.

The versatility and good performance of Dispersy is shown by comparing it to Cassandra using the Yahoo! Cloud Serving Benchmark (YCSB). We have modified YCSB as to include nodes randomly joining and leaving, a kind of behaviour that is very typical in challenged network environments. Furthermore, we integrated Dispersy in the BitTorrent client Tribler and show that it is performing very well in various real-time challenged network scenario's (3G and WIFI).

Contents

List of Figures

List of Tables

1 Introduction

In this paper we introduce Dispersy, middleware for data dissemination in challenged networks. Dispersy requires only minimal assumptions about the reliability of communication and network components. Our middleware is fully decentralized. It does not require any server infrastructure and can run on systems consisting of a large lumber of *nodes*. Each node runs the same algorithm and performs the same tasks. All nodes are equally important, resulting in increased robustness. Dispersy offers distributed system developers both one-to-many and many-to-many data dissemination capabilities. Data is forwarded between nodes. All injected data will eventually reach all nodes, overcoming challenging network conditions.

Dispersy is designed as a building block for implementing fully decentralized versions of, for instance Facebook, Wikipedia, Twitter, or Youtube. These Web 2.0 applications often require on a direct Internet connection to their central servers. In contrast, our building block does not depend on continuous connectivity to offer reliable data dissemination.

Networks characterized by challenges, such as intermittent connectivity, large delays and absence of end-to-end paths are called *challenged networks*. These can be found in a broad spectrum of fields. For instance, vehicular ad hoc networks (VANETs) in which connection windows are very small [10], and delay tolerant networks (DTNs) in which connections are sometimes only possible when the network utilization is low [13]. Peer-to-peer networks are particularly challenging due to nodes lifetimes which can be measured in minutes (churn), NAT-firewall constrained Internet connections, and frequent interaction with potentially malicious nodes. Smartphones pose another challenge due to their limited processing capability and battery lifetime.

In the past, scientists targeted each of these fields individually. Dispersy distinguishes itself from this previous work by unifying previous solutions into a single data dissemination middleware solution. We combine the robustness of VANETs with the scalability of P2P, allowing it to scale from one to one million nodes. An example of this combination is depicted in Figure 1. Shown is a combination of stationary and mobile devices, and wireless and stable Internet connectivity.

Different from prior work is the continuous state of partial synchronization of Dispersy nodes. Periodically, nodes advertise their locally available data in a single message. A node replies to an incoming advertisement, by sending missing data from the advertisement. Repeated random interaction between nodes in the network result in quick spreading of data [7]. Within Dispersy, we call a unit of data a *bundle*. A bundle is a data unit of variable length, that is application dependent. Each node locally stores a collection of bundles. Depending on the application scenario, it is possible to personalize this bundle collection for each node. We define bundle delay as the time between one node injecting and all nodes receiving a bundle. Due to the challenged network, bundles are likely arrive out-of-order at a node and may suffer from significant delays. Bundle interdependencies should therefore be avoided.
Key contributions of this paper are:

1. One-to-many and many-to-many data dissemination with a minimal amount of complexity and state information.
2. Facilitation of repeated random interactions between nodes, resilient to churn, node failures, and NATs.
3. Bundle advertisement mechanism facilitating scaling to over 100,000 bundles and millions of nodes.
4. Experimental validation and real-world deployment.

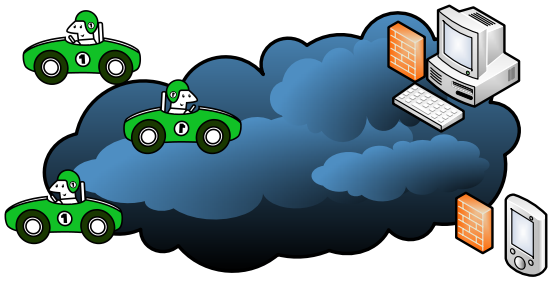

Figure 1: Data dissemination in a challenged network.

2 Background and Related Work

More than 25 years ago, Demers et al. [7] outlined methods for creating consistency across many nodes linked together using a large heterogeneous, slightly unreliable and slowly changing network. Paving the way for even more challenged networks, Demers introduced the concept by stating that nodes can only be consistent after all updating activity has stopped. However, if we assume a reasonable update rate, most information at any given node should be up-to-date.

Demers et al. describe three different methods of distributing updates; *direct mail*, in which an update is sent from the initial node to all other nodes, *anti-entropy*, in which a node chooses another node at random and resolves the differences between the two, and finally, *rumor mongering*, in which a node after receiving a new update (a hot rumor), will keep forwarding this update until it goes cold (due to receiving it from many other nodes).

Demers et al. considered the anti-entropy update method as too expensive, as it involves sending a complete copy of the database over the network in order to be able to compare the two copies. As an optimization, Demers suggested that nodes could maintain check-sums of their database and compare those beforehand. If no difference is found, then the expensive operation of sending a complete copy of the database can be prevented.

VANETs, vehicular ad hoc networks are quickly becoming commercially relevant, because of recent advances in inter-vehicular communications and their associated costs. In many respects VANETs can be considered as a P2P system as they are not restricted by power requirements, suffer from excessive churn, and have to be able to scale across millions of nodes without the use of central components. In VANETs churn is caused by the movement of vehicles, which causes them to have a very small connection window wherein communication between two nodes is possible.

Lee et al.[14] envisioned a vehicular sensing platform able to provide proactive urban monitoring services by using sensors found in vehicles to detect events from urban streets, e.g. recognizing license plates, etc. Their MobEyes middleware runs inside a vehicle and creates summaries of detected events. Those summaries are then pushed by the creator for a period of time to all one-hop neighbors, but are harvested in parallel using a Bloomfilter. By building a Bloomfilter of the already-harvested still-valid summary packets, all other nodes within the broadcasting range can detect the missing summaries. Lee et al. are using a Bloomfilters which are 1024 Bytes in size combined with a periodically changing hash function to reduce the number of false positives.

Introduced as Method 2 by Bloom et al. [3], a *Bloomfilter* have become popular due to being a very space efficient method for membership testing. Bloomfilters use a hash area consisting of N bits, which are initially set to 0. For each item that needs to be "stored" in the Bloomfilter, K distinct addresses are generated using a hash function. The bits addressed in the Bloomfilter are set to 1. Upon checking if an item is part of a Bloomfilter, the same addresses are generated,

but now the bits addressed are checked to be 1. If not all bits are 1, then this item is not in the Bloomfilter. When all bits are set to 1, this item might be part of the Bloomfilter or (due to the compression of the Bloomfilter) could be a false positive.

3 System Overview

We propose a bundle synchronization middleware which in contrast to current solutions can run in a challenged network, and additionally supports large networks containing more than 100,000 bundles. Our solution extends the anti-entropy approach of Demers et al. [7], but uses compact hash representations (Bloomfilters) to advertise locally available bundles. This allows two nodes to compare their datasets without sending a complete copy. Figure 2 gives an overview of both synchronizing and receiving bundles.

Our synchronization technique consists of five steps:

1. Select a node from our candidate list;
2. Select a range of bundles for synchronization;
3. Create a Bloomfilter by hashing the selected bundles;
4. Send the Bloomfilter to the selected node;
5. Pause for a fixed interval and goto step 1.

Our synchronization technique extends the harvesting approach of Lee et al. [14], in order to allow for networks which synchronize many bundles. By selecting only a range of bundles to be synchronized, we can keep the false positive rate low without having to increase the size of the Bloomfilter. In Section 5 we will show that this solution allows Dispersy nodes to synchronize over 100,000 bundles in a single network. In contrast, Lee et al. use a Bloomfilter of a fixed size without selecting bundles. This prevents them from constructing networks which try to synchronize many bundles.

4 System Components

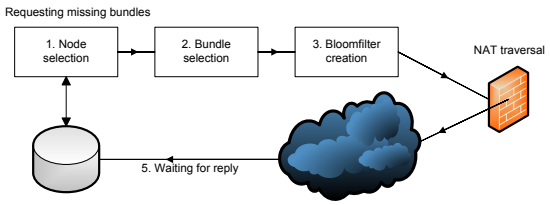

Figure 2: The five step Dispersy synchronization technique.

In order to support different types of challenged networks we designed Dispersy using modular components. This section introduces these components one by one, starting with the network construction.

4.1 Network Construction

In Dispersy all nodes communicate in one or more *overlays*. An overlay, is an additional network built on top of e.g. the Internet. To construct an overlay, one node generates a public/private keypair. The public key is used as the identifier of the overlay, which has to be known to all nodes attempting to join. As Dispersy is designed to be run in a challenged network, all communication between nodes uses UDP. UDP was selected over TCP, as it allows for easier NAT-firewall puncturing.

For each overlay a Dispersy node maintains a *candidate list*, this is a list of active connections within that overlay. When connecting to an overlay, the candidate list is initially empty requiring a node to initiate the node discovery algorithm.

4.2 Node Discovery

There are several mechanisms available to discover nodes; if we assume an Internet connection then the most basic solution is to use trackers. Trackers are known to all nodes and must be connectable, i.e. they must not be behind a NAT-firewall. Trackers maintain lists of nodes for several overlays, and return these upon request. After a node has populated his candidate list with initial nodes, he can use those to attempt new connections. A connection attempt is called a *step*. At a fixed interval, Dispersy will attempt a new step.

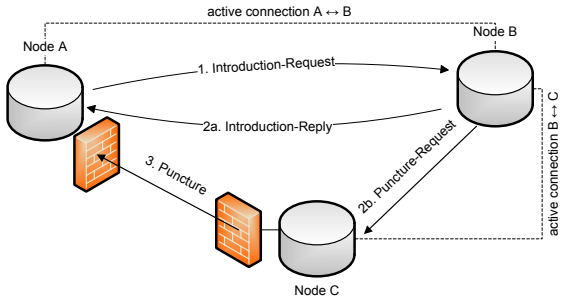

Figure 3: A step of our integrated discovery and NAT-puncturing mechanism.

Figure 3 gives an overview of a single step taken by Node A. Additionally shown is the fully integrated NAT-puncture mechanism, which will be explained in Section 4.4. In every step, a node will first select a node from his candidate list (Section 4.3). Nodes present in the candidate list are assumed to be connectable. In this example Node A selected Node B. After selecting a Node B, Node A will send an *introduction-request* to Node B. An introduction-request message is a request to the receiver to introduce this node to another node. Optionally, the introduction-request message may contain a Bloomfilter advertising locally available bundles. Upon receiving an introduction-request message, Node B will select a node from its candidate list (Node C) and will send an *introduction-reply* message containing the ip/port of this node (Node C). After receiving the introduction-reply from Node B, Node A will add the address of Node C to its candidate list.

4.3 Robust Node Selection

In each step, we have to select a node to send the introduction-request message to. However, selecting a node is a non-trivial task as *attacker nodes* could be present in the candidate list.

Manipulating the candidate lists of nodes such that they mostly contain attacker nodes is often referred to as an *eclipse attack*. After "eclipsing" a node, an attacker node has complete control over a node as the attacker is controlling all data received by its victim. Candidate lists can be easily manipulated, by simply contacting a node which multiple colluding attacker nodes.

Furthermore, after accidentally selecting an attacker node in an introduction-request, this attacker can easily direct us to his (colluding) friends by introducing only them in the introduction-reply message. For more information regarding eclipse attacks we refer to the paper by Singh et al. [17].

In order to create a less predictable, more robust node selection algorithm a Dispersy node will divide his candidate list into three categories:

I) Trusted nodes;

II) Nodes we have successfully contacted in the past;

III) Nodes who have contacted us in the past; either through

  a) Nodes that have sent an introduction-request; or

  b) Nodes that have been introduced.

Whenever a node is discovered or selected, it is placed in one of the categories above. Nodes that have replied to an introduction-request message to are put into Category II, while the node they introduce is put in Category IIIb. Nodes that have send us an introduction-request are placed in Category IIIa. The trusted nodes category is populated by a list of predefined nodes, i.e. trackers. A node can only be part of one category, this causes a node which was introduced to us to move from Category IIIb to Category II after a successful connection attempt.

Before selecting a node from its candidate list, a node will first choose a category from which to select the node from. The category is determined by predefined probabilities. The trusted node category has a probability of 1%. The remaining 99% is divided between Category II and III. However, as Category III consists of two subcategories both of these subcategories get 24.75%. After choosing a category, a node will selected the node which we had the least recent interaction with. We select the "oldest" node due to NAT-timeouts, which is further explained in Section 4.4.

If an attacker node tries to perform an eclipse attack as described above, then the size of Category III will increase. However, increasing the size of this category only has a limited effect on the selection probability of this attacker node. As the selection probability is primarily defined by the category probabilities. Furthermore, as a node can only be part of one category, an attacker node is not able to put a single colluding attacker node into more than one category in order to increase its selection probability.

Dividing the nodes into categories, thus has a dampening effect on a possible attack. However, if an attacker has access to a lot of resources he can still eclipse a node. This is why Dispersy uses trusted nodes.

Trusted nodes

Trusted nodes (i.e. a tracker) help clean the candidate list of a node whenever its manipulated by attacker nodes. On average, a trusted node is contacted once every 100 steps. Contacting a trusted node will completely reset the candidate list, thereby removing any attacker nodes which could be present.

In a small experiment we have evaluated how many attacker nodes are required to pollute Category II within 100 steps. We consider a node to be eclipsed when Category II consists of only attacker nodes.

Averaging over 100,000 runs, an attacker should contact a node (send an introduction-request) with 1250 nodes within the 100 steps. These nodes should preferably be located on different machines as multiple nodes on one machine could be easily identifiable. Any attack with less than 1250 nodes will probably never result in a successful eclipse attack. An attack with more than 1250 nodes, will allow an attacker to eclipse a node. However, every 100 steps, the trusted node resets the manipulated candidate list of a node and allow it to start fresh. Furthermore, as a node is expected to receive roughly 1 introduction-request in each step he can easily detect that "something" is going on when he suddenly starts receiving more than 12.

Trusted nodes, such as trackers, are much less susceptible to attacks as they are contacted by a constant stream of honest nodes. To perform a successful attack on a tracker, attackers should ensure that there are more attacker nodes than honest nodes contacting it. As P2P networks have shown to be able to accommodate more than 4 million nodes concurrently [18], the probability of this happening is low. A tracker will then be contacted by 40,000 honest nodes in every step, requiring an attacker to at least do the same using different machines for every step.

4.4 NAT puncturing

Since up to 64% of all nodes connected to the Internet are behind a NAT [9], we have integrated each step with a distributed UDP NAT puncturing mechanism. As shown in Figure 3, upon receiving an introduction-request Node B will additionally send a *puncture-request* message to Node C. Subsequently, Node C will send a *puncture* message to Node A, punching a hole in its NAT and making sure that Node A is able to connect to him.

Node C will only puncture his NAT-firewall, the puncture message itself will be blocked by the NAT-firewall of Node A. However, if Node A selects Node C in a next step, then the NAT-firewall of Node A will be punctured by the outbound introduction-request message, allowing Node A and Node C to successfully communicate.

The NAT traversal approach as described is a variant of the approach as described by Halkes et al. [9], the modification being that we do not specify the node to whom we should be introduced to beforehand as we do not know whom we want to connect to in the next step.

As NAT-firewalls will close inactive connections after a certain timeout, we remove candidates from the candidate list for which the probability of the NAT-firewall having closed the port is high. If the NAT-firewall closes the port, any message sent to this node will never arrive. Using the measured timeouts specified by Halkes et al. [9], we invalidate nodes to which we have sent or received an introduction-request from after 55s, introduced nodes are removed after 25s. As described in the node selection algorithm, a node will select the node from a category which it had the least recent interaction with. This is due to the NAT-timeouts, and prevents more than one node being removed from a single category. Not selecting the "oldest" node causes one node from a category to be removed by the NAT-timeouts, and another one by selecting it.

4.5 Synchronization

In Dispersy, nodes synchronize bundles by advertising their locally available bundles using Bloomfilters. The Bloomfilters contain the bundles which a sending node has received previously, allowing the receiving node to check for missing bundles. By piggybacking on the introduction mechanism, we prevent an additional message to be sent.

Since the 1970's, Bloomfilters have become popular due to being a very space efficient method for membership testing. An overview of the use of Bloomfilters in different network applications can be found in [4]. Moreover, as can be read in Section 2, Bloomfilters are also used in the harvesting approach of Lee et al. [14].

Similar to Lee et al [14], Dispersy uses multiple hash function configurations to prevent false positives. False positives could cause some of the bundles to never be synchronized to a node. However, in contrast to the solution as used by MobEyes, there is no predefined set of configurations, instead we include a randomly chosen salt character every time we send a Bloomfilter. By initializing the hash functions with this random salt, we achieve a similar reduction in the false positive rate.

Furthermore, because we use UDP in the NAT puncturing mechanism, and are piggybacking the synchronization on these messages, we limit the size of the Bloomfilter such that an introduction-request message will not exceed the MTU of the link. If a UDP packet consists of multiple fragments, it will have a higher probability of being lost, as the loss of a single fragment will result in the loss of the complete packet. IP fragmentation is caused by a router in the path having a lower MTU (maximum transmission unit) than the packet size. A typical MTU size used on the Internet is 1500 Bytes, for 802.11 it is 2304 Bytes [5].

Lee et al [14] limit the size of their Bloomfilters to 1024 Bytes. This causes their approach to allow for less than 1000 bundles to be synchronized if we assume a false positive rate of 1%, or less than 2000 if we assume a false positive rate of 10%. Moreover, there is no limit implemented in MobEyes resulting in very high false positive rate for larger datasets. This causes their solution to only be applicable to VANETs which have few bundles to be synchronized or have a low rate of creating new bundles.

In contrast, we fix the false positive rate to predefined constant. This causes the Bloomfilters to have a fixed capacity, as we are also limiting the number of bits available (the MTU). If a node has more bundles than the Bloomfilter capacity, then we have to select a subset of bundles to be included in the Bloomfilter.

Subset selection

Subset selection is based on the global-time property of each bundle. The *global-time* property is an implementation of Lamport's logical clock [12], and is included in all bundles. The global-time is the highest global-time received by a node + 1 when creating a bundle; it will not be unique, but will indicate a partial ordering of bundles in the overlay. Furthermore, we expect the distribution of global-times in the overlay to be roughly uniform. By specifying a range of global-times a node can define the subset of bundles which are included in the Bloomfilter.

We implemented two heuristics which define the subset of bundles that should be included in the Bloomfilter. Depending on the state of the overlay we choose one accordingly. The state is determined by looking at the number of bundles received after sending the previous synchronization requests.

If a node has received many new bundles, then the *modulo heuristic* is used as we assume that a node is lagging behind in synchronization. The modulo heuristic first counts the number of locally available bundles, then divides this by the capacity of the Bloomfilter to compute a modulo value, e.g. if we have 10.000 bundles, and a Bloomfilter capacity of 1000 we use a modulo of 10 in our range selection. Next we combine the modulo value with a random offset such that we can select every n^{th} bundle starting at offset X, e.g. using a modulo of 10 and an offset of 1 a node will select all bundles with a global-time equal to 1, 11, 21, etc. The benefit of this approach is that it resembles a linear download in performance, but does not require any state. This results in many missing bundles being retrieved for each Bloomfilter sent, when a node is catching up. However, when a node is almost synchronized (missing only the newest bundles) it could take up to a modulo number of steps before a new bundle is received, which for larger overlays can introduce high latency.

To improve the latency when a node is almost synchronized, we implemented the *pivot heuris-*

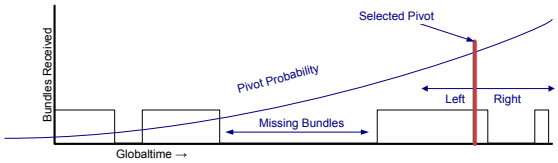

Figure 4: Bundle subset selection for Bloomfilter advertising.

tic. This heuristic (as shown in Figure 4) first chooses a pivot according to an exponential distribution between 0 and the maximum (locally known) global-time. By using this pivot, we select up to the Bloomfilter capacity bundles both to the left and right of it. Then finally, we compare these two ranges and select the one that has the largest difference in global-time. The range with the largest difference contains the least amount of bundles, thus is a likely candidate for any missing bundles. By choosing a pivot using an exponential probability, newer bundles are synchronized more often resulting in a reduction in latency.

Because the type of heuristic is chosen depending on the state of a node, new nodes will use the modulo heuristic to efficiently receive most of the available bundles, and after receiving those switch to the pivot heuristic to fetch the newer bundles with a lower latency.

Upon receiving a Bloomfilter, a node will check if it has some missing bundles to send. Using the subset as defined by the range of global-times, the modulo, and offset values, a node is able to define a local subset of bundles and test those against the Bloomfilter.

4.6 Synchronization performance

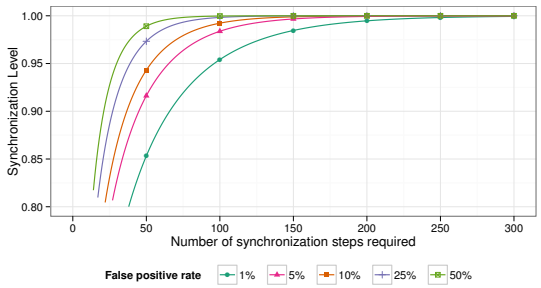

Figure 5: Synchronization performance for various false positive rates.

In the following paragraphs we give a performance model for the synchronization process. Considering the case of a new node joining an overlay, two factors influence the number of synchronization steps required to be fully synchronized in Dispersy, the size of the MTU, and the false positive rate of the Bloomfilter. If we assume a typical Internet MTU (1500 bytes), and a false positive rate (P) of 1% then a Bloomfilter can accommodate 1250 bundles (referred to as C, the Bloomfilter capacity as described by [1])

$$C = \text{MTU} \times 8 \times \frac{(\ln 2)^2}{|\ln P|}$$

However, if a node has a *synchronization level* (S) of 90% (a node having 90% of all bundles), then his Bloomfilter is actually describing a range which spans 1388 bundles and a receiving node

could thus detect up to 138 missing bundles. The false positive rate reduces this number to 137 ($138 * 0.99$), which is now referred to as the detectable range R

$$R = \left(\frac{1}{S} \times C\right) \times (1 - P)$$

Finally, the synchronization level of the receiving node determines how many bundles are returned. Generalizing over this results in the following equation which allows us to calculate the synchronization level S in the next cycle, where N defines the total amount of bundles in an overlay:

$$S_{i+1} = S_i + \frac{R \times \sum_{j \neq i} S_i^j}{N}$$

Increasing the capacity of a Bloomfilter will increase the detectable range R, the probability that missing bundles are detected, and thus the synchronization speed. As we cannot increase the MTU between two nodes, increasing the capacity of a Bloomfilter can only be achieved by incurring a higher false positive rate. However, increasing the capacity will additionally cause the creation of the Bloomfilter, and testing for missing bundles to be more computational intensive, as more bundles need to be hashed.

Figure 5 shows synchronization steps required using different false positive rates. As can be seen increasing the false positive rate has a positive effect on the synchronization speed. However, as described above a higher false positive rate causes the Bloomfilters to be more computational intensive. Therefore, we use a 10% false positive rate as compromise between computational costs and synchronization speed. This results in a Bloomfilter capacity of 2500 bundles, and a decrease of 125 steps compared to a 1% false positive rate.

Finally, there is one additional limitation implemented in Dispersy which is influencing the synchronization speed. This is a limit on the maximum number of returned missing bundles. By limiting this, a node is able to control the amount bandwidth used by the synchronization process. If bandwidth is not a problem, then this limit can be removed allowing for a quicker synchronization.

5 Evaluation

In the following section, we will evaluate the performance of Dispersy in different scenarios. Unless stated otherwise the experiments are run on our DAS-4 supercomputer[1].

In our first experiment, we investigate the randomness of our node selection algorithm. This experiment is conducted in a non-challenged network, i.e. all nodes are online and connectable at all times. We leave out the challenged network, to get a better understanding of the optimal performance of Dispersy.

In our node selection algorithm, the 49,5% probability of choosing Category II will cause a node to recontact other nodes with a high probability. However, as recontacting is a critical part of our security mechanism we want to determine how much the resulting overlay differs from random uniform. To determine this we have run two experiments using the approaches as defined by Jelasity et al. [11].

First, using the "Diehard Battery of Tests of Randomness"[2] we checked whether our node selection algorithm provides a random node. By generating two random integers we emulate the discovery of two nodes. One integer representing an incoming connection, the other the

[1] http://www.cs.vu.nl/das4/
[2] http://www.stat.fsu.edu/pub/diehard/

	Random overlay	Our overlay
Average node degree	11	10.9319
Average clustering coefficient	0.0100299	0.0461267
Average path length	3.139084	3.456935
Average diameter	3.14038	3.1835
Maximum diameter	6	6

Table 1: Comparing a random overlay to our overlay

introduced node. Next, we add these 'nodes' to their categories and remove any invalidated nodes due to the NAT timeouts. Finally, we select a category and node from our candidate list using the probabilities as described in Section 4.3. After selecting 10 million integers we process them using the "Diehard Battery of Tests of Randomness". As expected, recontacting nodes from Category II causes most randomness tests from the battery to fail.

Random overlay comparison

Next, we compare the overlay constructed by our node selection algorithm to a random overlay by looking at the node degree, path length, clustering coefficient, and diameter. We deployed 1000 nodes on the DAS-4 and collected 20 snapshots of the candidate lists of all nodes after the overlay was stabilized. We consider the overlay to be *stabilized* when all nodes have filled the categories of their candidate list to their expected size. From each snapshot, we constructed a directed graph using only the nodes from Category II. The maximum diameter is the largest shortest path between two nodes encountered in the 20 snapshots. To show the differences between our and a random overlay, we compute the same properties using a random overlay build with the theoretical limit of 11 neighbors per node (Category II will at most contain 11 nodes (we set the step interval to 5 seconds in this experiment).

Node selection

Table 1 shows the results. As expected, the average node degree is slightly lower due to a minimal amount of timeouts. Clustering in our overlay is caused by the introduction of nodes, resulting in overlap between the neighbors of two nodes, hence increasing the clustering coefficient. However, the increase in clustering coefficient does not result in a big impact on the path length.

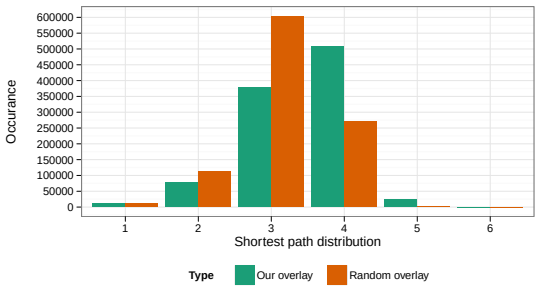

Figure 6: Comparing the distribution of shortest paths between a random and our overlay.

Nodes	Type	Remarks	Success
1,2,3	APDM	New port for every dest ip:port	43%
4	ADM	New port for every dest ip	75%
5,6,7,8,9	EIM,APDF	Accepts if packet was send to dest ip:port	81%
10,11,12	EIM,ADF	Accepts if packet was send to dest ip	100%
13,14,15	EIM,EIF	Accepts if opened before	100%
16,17	No NAT		100%

Table 2: NAT-experiment performed with 17 nodes connected by 15 different NAT-firewalls.

Furthermore, as shown in Figure 6 the distribution of shortest paths clearly differs between the random overlay and the overlay constructed by our node selection algorithm. A random overlay contains many more links which are 2 or 3 hops. This can be explained by having a lower clustering coefficient, and is additionally shown in Table 1 in the average path length.

As described by Jelasity et al. [11], the propagation speed of an overlay largely depends on the overlay diameter. As can be seen in Table 1, the average diameter is very similar between the two overlays. Furthermore, the maximum diameter is equal, which leads us to believe that only in rare cases our overlay propagation speed will be lower compared to a random overlay.

NAT traversal

For this experiment we obtained 15 different NAT-firewalls from various vendors and used them to connect 17 Dispersy nodes to the Internet (2 nodes had a direct connection, the others were connected to a single NAT-firewall which was connected to the Internet).

By recording the number of connections attempts and timeouts, we can determine the success rate. A node will consider a request to be successful if it gets a reply within 5 seconds. If not, then this request is considered to be timed out. To allow the NAT-firewalls to close inactive ports, we only allow a node to attempt a connection every 5 minutes. A short description on how the different types of NAT-firewalls behave can be read in Table 2, for more information we refer to the BEHAVE workgroup [2]. We used the NAT identification script by Andreas et al. [15] to detect which types of NAT-firewalls we obtained.

Additionally shown in Table 2 are the success rates for incoming connections. In general we achieve a 77% success rate for connections made between nodes. However, some combinations of NATs do not work. APDM nodes 1, 2, 3 are unable to connect to each other and to APDF nodes 5, 6, 8, and 9. Furthermore, nodes 5, 6, 8, and 9 are unable to connect to nodes 1, 2, and 3. This is expected, as a APDM NAT will open a port for every new node and a APDF NAT will only accept an incoming packet if this node has previously sent a packet to this specific ip/port combination. The new port opened by the APDM NAT cannot be known beforehand thus the APDF NAT will block any packet trying to pass. Hence, if both the APDM NAT and the APDF NAT are implemented correctly no connection should be possible.

In Figure 7, we show the success-rate between the NAT-firewalls in more detail (Table 2 can be used to convert the node-id into the NAT-types). From the figure we can observe some particularities in the implementation of some NAT-routers. The NAT-router of node 14 has severe difficulties in connecting to other nodes (even those without a NAT-router). Looking at the raw output of the identification script, we can see that this router closes inactive connections after 0 seconds. If this is correct, then our walker approach should not work as the puncture and introduction-request messages are send at least 5 seconds apart. This causes the NAT-router of

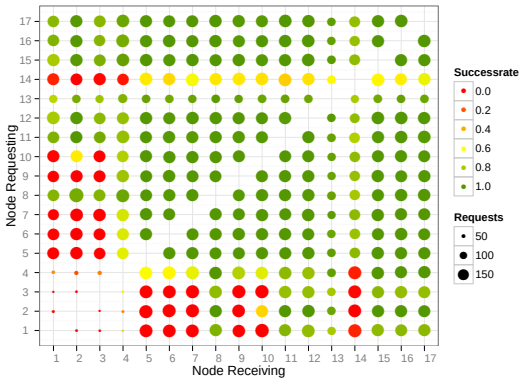

Figure 7: Success-rate of connections between nodes in the NAT traversal experiment.

node 14 to close the inactive port before receiving the introduction-request message.

Churn resilience

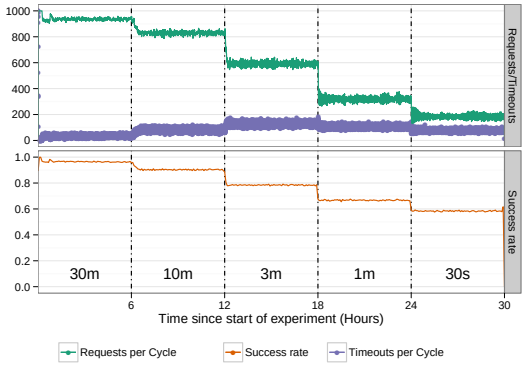

Figure 8: Churn-experiment performed with 1000 nodes while lowering the average session-times from 30 minutes to 30 seconds.

In this experiment will show the ability of Dispersy to cope with different levels of churn. In an experiment very much similar to that conducted by Rhea et al. [16], we change the average session-time of the nodes during the experiment. The *session-time* denotes the duration of online time of any given node in the overlay. Rhea et al. state in their paper that these average session-times can range from 50% of the nodes having a session-time of less than 60 minutes to 50% of the nodes having a session-time of less than 1 minute. In a VANETs, average session-times can be as low as 50% of all connections having a session-time lower than 7 seconds have been reported by Huang et al. [10]. However, these figures are based on the GPS-data from 4000 taxis in the Shanghai downtown area, a larger deployment could increase the session-times as more vehicles in transmission range are equiped with sensors.

In our experiment, a node chooses a random session-time between 50% less and 50% more than the average session-time, i.e. if the average session-time is 30 minutes, the nodes will choose a random value between 15 and 45 minutes. When the session-time expires and a node

goes offline, it waits at least 2 minutes to make sure that is completely forgotten by all nodes in the overlay before joining it again. We evaluated 5 different average session-times using 6 hour intervals, ranging from 30 minutes to 30 seconds.

As before, we deployed 1000 nodes to the DAS-4 supercomputer. By lowering the average session-times of the nodes, less nodes are online at the same time. Additionally, this will cause a higher number of "offline" nodes being present in the candidate list of a node. We expect that a lower average session-time will thus result in more timeouts i.e. a lower success rate. Furthermore, a lower average session-time will additionally cause less requests per cycle to be made as less nodes are online at the same time.

As can be seen in the Figure 8, Dispersy is coping well with the decrease in average session-times. For an average session-time of 30 seconds, Dispersy is still able to synchronize successfully for more than 50% of the attempts. In such an overlay, nodes would on average successfully synchronize 3 times within 30 seconds, allowing these nodes to obtain many missing bundles before going offline again. In contrast, the results from Rhea et al. [16] showed that Chord was only able to cope with average session-times of 1.4 minutes, and incurred very high latency while trying (up to 5 seconds). Pastry however, did not manage to cope with average session-times lower than 23 minutes.

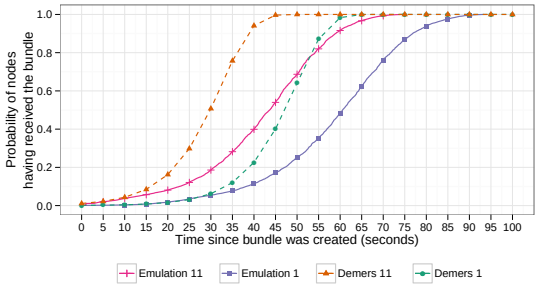

Figure 9: Differences in propagation speed in an overlay consisting of 1000 nodes. Demers 1 and 11 show the theoretical performance of a random overlay based on epidemic theory, Emulation 1 and 11 show the actual performance of our overlay.

Propagation speed

In next experiment, we focus on propagation speed and bandwidth consumption. This experiment is conducted in a non-challenged network, i.e. all nodes are online and connectable at all times. However, we are now going to actually synchronize some bundles. In this scenario we used a bundle which contained; the public-key of the sender, a random 20 Byte string, and a signature authenticating the body of the message (the random string).

The theoretical propagation speed of a bundle is defined by Demers et al. [7]. They state that as a basic result of epidemic theory, anti-entropy will eventually infect the entire population. Moreover, Demers et al. state that if implemented on top of a random node selection, the probability of a node not having received a bundle after i synchronization cycles is:

$$P_{i+1} = (P_i)^2$$

Similar to Demers et al. we want to measure the average speed by which the entire popular is infected (has received the bundle). In each round, we let a random node create a bundle. By recording the time at which this bundle is received by other nodes we can determine the

	Average	Worst Case
The time required for an update to propagate to all nodes	74.34 s	103 s
The network traffic generated in propagating a single update to all nodes	29.98 KBytes	37.38 KBytes

Table 3: Time/Bandwidth used to propagate a message using an initial push to 10 nodes.

propagation speed. We conduct two experiments, one in which the random node, after creating a bundle, immediately pushes it to 10 nodes from its candidate list. While in the other experiment, the random node does not push it to any other nodes, and thus solely relies on the synchronization algorithm (for more information we refer back to Section 4.5). During the experiments, the nodes created over 100 bundles. Between each round, nodes wait for 2 minutes such that they round do not interfere with each other.

Figure 9 shows the measured and theoretical curves. The initial probability for the theoretical curves depend on the push configuration, e.g. for a push to 10 nodes it is 989/1000 (the initial node + the 10 other nodes the bundle is pushed to). Similar to the previous experiment we have set the step interval at 5 second, and used this to plot the theoretical curves.

As can be seen in the figure, Dispersy is performing significantly less than the theoretical bounds as defined by Demers et al. The drop in performance is most likely caused by the higher clustering coefficient due to recontacting nodes. As this is one of our essential security features, allowing a node to be able to withstand an eclipse attack, this is a compromise between performance and security. Furthermore, as Dispersy can synchronize more than one bundle per request, the performance as shown here is the worst case.

Bandwidth consumption

Using the data from the previous experiment, we can compute the bandwidth consumption required to propagate a single bundle. A single node will for each step send one introduction-request, one introduction-response, one puncture-request, and one puncture-message. The combined size of those messages is 1746 Bytes, if we assume a 1500 Bytes MTU. Including the typical IP and UDP header-sizes, $4 \times (20 + 8)$, results in a total of 1858 Bytes send per node in each step. A similar amount of data is expected to be received by a node in every step. Depending on the step interval a node will thus consume a 1858 Bytes every X seconds, e.g. at a 5 second step interval the overhead would thus be less than 0.4 KBytes/s.

We calculate the total amount of traffic generated to propagate a single bundle as follows:

$$|\text{nodes}| \times (\text{bundle size} + 1858 \text{ Bytes} \times \text{steps required})$$

Table 3 shows the average and worst case numbers. As Dispersy is optimized to allow for multiple bundles to be synchronized per step, the overhead for the single bundle case is considerable. However, the overhead will not change depending on the number of new bundles in the overlay, as the overhead is caused by a node taking a step, and not by the number of bundles created. Furthermore, as most of the overhead is caused the Bloomfilter, we lower the frequency of which a Bloomfilter is included when an overlay does not have many updates. Not sending a Bloomfilter will result in 1432 bytes saved per step, a reduction of more than 77%.

High workloads

In our next experiment we will test Dispersy in a much larger scale. Using the Yahoo Cloud Serving Benchmark we will compare the performance of Dispersy to Cassandra in a challenged environment. Cassandra is a popular key-value store developed by Facebook [8]. To compare it to Dispersy we implemented a basic key-value store which is able to perform three actions; reading, inserting, and updating a key. In this experiment, each bundle contained the public key of the sender, a dictionary containing key/value pairs, and a signature authenticating the body.

As Cassandra has no NAT-traversal, an experiment involving NAT-firewalls would be impossible. However, by exposing Cassandra to a moderate average session-time, we can see how Cassandra is able to handle this aspect of a challenged network. A value of 3 minutes is used, as a lower average session-time caused Cassandra to completely fail.

The experiment is conducted as follows. First we start all nodes, allow the overlay to stabilize, and then start the YCSB-insert scenario (Scenario A). This scenario inserts 2,500,000 key/value pairs in the key-value store. After inserting all keys, we let the overlay settle for 30 minutes and then continue with the read/update scenario (Scenario B). This scenario will perform 2,500,000 operations consisting of 50% reading and 50% updating of keys. Both scenarios are explained in-depth by Cooper et al. [6].

We deployed the Dispersy nodes on our supercomputer, and use one additional coordinator node to send all commands issued by the YCSB test. This coordinator is not part of the synchronization process, and thus will not store any data. Commands are send from the coordinator to one single node in the cluster, but are load-balanced using our node selection algorithm. Additionally, since YCSB was written in Java and Dispersy in Python, we used Pyro[3] to expose the read, insert, and update methods to YCSB. Pyro uses a separate socket per thread to allow an RPC-like exposure of Python methods to Java. YCSB was configured to use 120 threads in parallel.

Next we repeat this experiment, but now using Cassandra instead of Dispersy. Similar to Dispersy, we deploy Cassandra to the nodes in our cluster and use one additional coordinator instance to serve the cluster the commands issued by YCSB. We used predefined identifiers to perfectly balance the key ranges, and configured Cassandra to replicate to all nodes. Cassandra was configured to replicate to all nodes, because using less replicas resulted in very low performance. When performing inserts/updates, the coordinator node will wait until at least one of the nodes in the replicating range is online and acknowledges the operation. By introducing churn this is not always the case, resulting in very high latency.

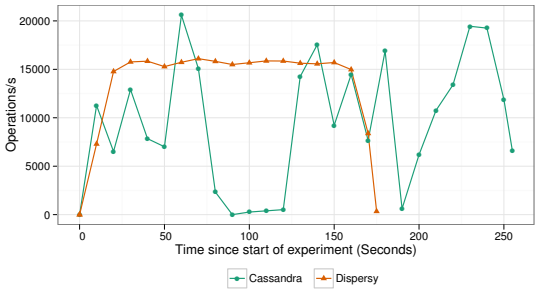

Figure 10: YCSB benchmarking results, comparing the performance of Cassandra to Dispersy when reading 2,500,000 keys from the key-value store.

[3] http://packages.python.org/Pyro4/

Figure 10 shows the operations/s during the read and update scenario. Comparing Cassandra to Dispersy, we can see that Dispersy has a more constant, and overall better performance, while Cassandra achieves the best peak performance, 20,593 operations/s.

However, Cassandra shows severe latency problems even though we opted for a 3 minute average session-time. Furthermore, due to using 10 replicas the coordinating node will store large amounts of data (in excess of 15 GBytes). This, we believe, is caused by the hinted handoff mechanism which according to the latest documentation[4] is handled by the coordinator node since version 1.0. Hinted handoff is used when the primary node of a given keyrange is not available, and indicates that this update still needs to be delivered.

On average Cassandra is able to achieve 9803 operations/s, while Dispersy is able to achieve 14285 operations/s. In this challenged environment, Dispersy is thus able to read/update 2,500,000 keys within 170 seconds.

Internet deployment

In order to evaluate the performance of Dispersy in the wild, we have integrated it into Tribler. Tribler is an enhanced BitTorrent client, which besides downloading .torrents, provides remote search, video-on-demand, live steaming, and channels. Initiated in 2005, Tribler is the product of a research project which has been funded by multiple European research grants, and has been downloaded over 1 Million times. Moreover, it has roughly 3000 daily users.

Using Dispersy we implemented a feature called channels in Tribler. Channels are created by users and can be compared to lists of liked .torrents. If another user of Tribler likes a channel, he can casts a vote. Using the votes, Tribler determines the popularity of channels, and presents users with the most popular channels. A vote is published as a bundle to a Dispersy overlay, and in this overlay Dispersy spreads all votes to all Tribler users (full-replication). A vote-bundle includes the public key of the voter, an identifier of the channel the user likes or dislikes, the actual vote (-1, or +1), and a signature. In total, users have voted for a channel over 100,000 times. Beside channels, Tribler implements many other BitTorrent improvements, for more information on Tribler we refer to [19, 20].

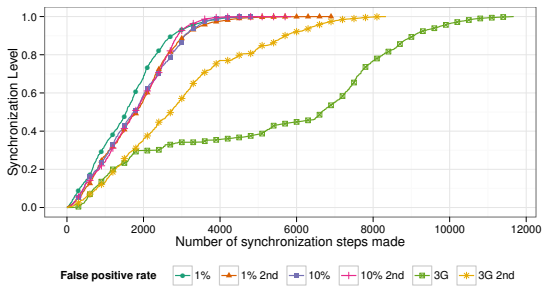

Figure 11: Results from actual deployment. Dispersy synchronizing 100,000 bundles using only Internet deployed clients.

By modifying one client we have generated traces on the synchronization speed of Dispersy in this overlay. We're measuring how long it takes to synchronize all 100,000 vote-bundles, using only nodes which are online in this Internet deployed overlay. Figure 11 shows the results of three different synchronization configurations. We have made two synchronization runs with a Bloomfilter with a false positive rate of 1%, two times with a false positive rate of 10%, and

[4] http://wiki.apache.org/cassandra/HintedHandoff

finally two times using a 3G cellphone network instead of a WIFI connection and a Bloomfilter false positive rate of 1%. The cellphone runs were created by tethering a cell-phone to a laptop to connect to the Internet. Tribler is configured to limit the number of returned bundles, therefore a node can only return up to 50 KBytes of bundles per synchronization request.

The performance difference in using different false positive rates as shown in Section 4.6 can be seen in the wild as well. Using a false positive rate of 10% causes a nice improvement in the number of synchronization steps required. If we consider the endgame of the synchronization only (the steps not being limited by the default return limit of 50 KBytes), then using a 1% false positive rate requires a 160 steps more than using a 10% false positive rate (646 vs 486 steps).

From the cell-phone experiments we can see that the performance is much lower, compared to the WIFI connections. There are a couple of different explanations for this behaviour, first cell-phone providers use carrier-grade NAT-firewalls. These are usually APDM routers, which are (as can be seen in Section 4.4) the most difficult routers to puncture and establish a successful connection with. Furthermore, as Dispersy is using UDP, cell-phone providers can simply drop packets when congestion occurs. As the use of mobile Internet is currently exploding, the probability of network congestion occurring, and thus the provider dropping packets is real.

To synchronize all 100,000 bundles/votes Dispersy connected to 882 distinct nodes, and consumed roughly 30 MBytes of bandwidth per run.

6 Conclusion

In this paper we have presented Dispersy, a distributed bundle synchronization platform capable of running in a challenged environment. By using Bloomfilters we can provide a stateless synchronization algorithm, which can scale to over 100,000 bundles. Furthermore, our random walk inspired node discovery algorithm, is able to achieve a connection success rate of 77% in a deployment using 15 different NAT-routers. The node selection algorithm has shown to be capable of handling excessive churn (average session-times of 30s), while still maintaining a success rate larger than 50%. Using the popular YCSB benchmark, we have shown that Dispersy is able to provide better performance than Cassandra during a challenged high-load experiment. Finally, during actual deployment, Dispersy has shown to efficiently synchronize more than 100,000 bundles.

7 Acknowledgments

This work was partially supported by the European Community's Seventh Framework Programme through the P2P-Next[5] and QLectives[6] projects (grant no. 216217, 231200).

References

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, March 2007. 11

[2] C. Audet, F., Jennings. Network Address Translation (NAT) Behavioral Require- ments for Unicast UDP. RFC 4787 (Best Current Practice), 2007. 14

[5] http://p2p-next.eu
[6] http://qlectives.eu/

[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970. 5

[4] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002. 9

[5] P. Congdon, B. Aboba, A. Smith, G. Zorn, and J. Roese. Rfc 3580, ieee 802.1x remote authentication dial in user service, 2003. 10

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM. 18

[7] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM. 4, 5, 6, 16

[8] Dietrich Featherston. Cassandra : Principles and Application. 2010. 18

[9] Gertjan Halkes and Johan Pouwelse. Udp nat and firewall puncturing in the wild. In *Proceedings of the 10th international IFIP TC 6 conference on Networking - Volume Part II*, NETWORKING'11, pages 1–12, Berlin, Heidelberg, 2011. Springer-Verlag. 9

[10] Hong-Yu Huang, Pei-En Luo, Minglu Li, Da Li, Xu Li, Wei Shu, and Min-You Wu. Performance evaluation of suvnet with real-time traffic data. *Vehicular Technology, IEEE Transactions on*, 56(6):3381 –3396, nov. 2007. 4, 15

[11] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), August 2007. 12, 14

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. 10

[13] Nikolaos Laoutaris, Georgios Smaragdakis, Pablo Rodriguez, and Ravi Sundaram. Delay tolerant bulk data transfers on the internet. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, pages 229–238, New York, NY, USA, 2009. ACM. 4

[14] Uichin Lee, Eugenio Magistretti, Mario Gerla, Paolo Bellavista, and Antonio Corradi. Dissemination and harvesting of urban data using vehicular sensing platforms. Technical report, 2007. 5, 6, 9, 10

[15] A. Muller, N. Evans, C. Grothoff, and S. Kamkar. Autonomous nat traversal. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pages 1 –4, aug. 2010. 14

[16] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht. In *In Proceedings of the USENIX Annual Technical Conference*, 2004. 15, 16

[17] Atul Singh, Miguel Castro, Peter Druschel, and Antony Rowstron. Defending against eclipse attacks on overlay networks. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, EW 11, New York, NY, USA, 2004. ACM. 8

[18] Daniel Stutzbach, Shanyu Zhao, and Reza Rejaie. Characterizing files in the modern gnutella network. *Multimedia Systems*, 13:35–50, 2007. 10.1007/s00530-007-0079-8. 9

[19] Niels Zeilemaker, Mihai Capotă, Arno Bakker, and Johan Pouwelse. Tribler: P2p media search and sharing. In *Proceedings of the 19th ACM international conference on Multimedia*, MM '11, pages 739–742, New York, NY, USA, 2011. ACM. 19

[20] Niels Zeilemaker and Johan Pouwelse. Open source column: Tribler: P2p search, share and stream. *SIGMultimedia Rec.*, 4(1):20–24, March 2012. 19