Decentralized Evaluation of Regular Expressions for Capability Discovery in Peer-to-Peer Networks

B. Polot *M. Szengel* *R. Holz* *H. Niedermayer* *C. Grothoff*
Network Architectures and Services
Technische Universität München
Email: {polot,szengel,holz,heiko,grothoff}@net.in.tum.de

Abstract

We present a new approach for expressive distributed searches in Distributed Hash Tables (DHTs) based on regular expressions. Regular expressions are stored and evaluated in the network in a decentralized manner. The key idea of our approach is to convert regular expressions into finite automata and store the corresponding states and transitions in the DHT. This allows us to support a new way to express *service capabilities* in overlays. Offered capabilities are expressed as regular expressions. Offering peers are discovered by matching a given search string against the regular expressions that are stored in the DHT. Offering peers can merge their locally constructed Deterministic Finite Automaton (DFA) into the DHT using `PUT` operations. This yields a Nondeterministic Finite Automaton (NFA) in the DHT which can be evaluated by following through a series of `GET` operations. A particular advantage of our approach is that peers do not need to possess any knowledge about the current NFA as all merges are independent from each other. We have implemented our approach and evaluated the performance of the system using both emulation and simulation. The case we studied is Internet-scale routing using regular expressions to describe address ranges of Autonomous Systems and finding exit peers for a given IP address. Our results demonstrate the performance, scalability and flexibility of our approach. Our code is available for public download.

1 Introduction

DHT-based Peer-to-Peer (P2P) networks offer an efficient way for decentralized and distributed storage and retrieval of data objects. In traditional DHTs, indexing is typically based on hash-keys [3]. While DHTs with more expressive search capabilities exist, we are not aware of any existing decentralized and distributed search methods that would be capable of handling regular expressions with reasonable efficiency. This work presents a new approach for expressive distributed searches in P2P overlays based on regular expressions. General purpose regular expressions offer a potent method for participants to advertise services or capabilities. Our method enables peers to discover other peers with adequate capabilities in a fully decentralized and distributed fashion.

Use cases. There are a number of use cases where the distributed evaluation of regular expressions proves valuable. We present some typical representatives here. Our first use case are message queuing systems, e. g., enterprise message buses for service-oriented architectures, like [10]. Here, subscribers define topics of interest as regular expressions, such as `IBM/*ware/Results`. It is the responsibility of the message queuing system to deliver new messages to the correct subscribers. Our system allows to find subscribers by matching the topic of the message against the subscriber's regular expression, in a distributed fashion and without a central registry of subscribers. A similar use case would be distributed computation. Here, a scheduling node (or peer) must determine peers that are willing to contribute and offer certain platform properties, like enough RAM, CPU or bandwidth. Such peers can describe their properties in a regular expression; the scheduling peer would find them by matching its search string against this expression. Finally, the use case that we have implemented and chosen for our evaluation is a network-oriented one, in the spirit of P2P. We use the distributed evaluation of regular expressions to find such nodes in a P2P network that are willing to accept forwarded IP traffic and act as exit nodes to the Autonomous System they are located in. We use this to establish a kind of P2P-based VPN: Given destination IP address, the overlay is capable of determining the best exit node. A similar real-world application is the search for exit nodes in the Tor network [5]. Here, peers specify an exit policy with the set of acceptable TCP destination addresses. Today, Tor transmits the exit policies of all exit nodes to all other peers; our method would enable a more scaleable, decentralized lookup. Note that further properties, like acceptable bandwidth usage, could also be defined in the regular expression using our approach.

Principle. We represent the capabilities of peers in the system as data items, which we publish in the DHT under a regular expression. They can then be located by searching for strings that the regular expression matches. We distribute the matching operation (the evaluation of regular expressions) with the help of PUT and GET operations of a key-based DHT. The first step in publishing a regular expression is to locally convert it to an automaton, which consists of states and transitions. Then, we store (PUT) each state as well as the state's transitions under a certain key in the DHT. As multiple nodes will need to concurrently store states, we provide a mechanism to merge equivalent states and transitions. The result is an NFA stored in the DHT. Peers searching for a particular string are able to match a search string against the aggregated regular expressions that are stored in the DHT. This is achieved by issuing a series of GET requests, which essentially follow the transitions in the NFA, until an accepting state is reached.

Contributions. First, we present the design and proof of correctness for a new decentralized data structure based on eventual consistency for expressive distributed search operations using regular expressions. Specifically, we provide mathematical results that show that our mapping of states to the DHT produces the correct NFA, which recognizes exactly the correct language. Second, we provide a freely available implementation of the data structure and the associated network protocol. Finally, we present experimental results demonstrating the capabilities and limitations of the approach.

Outline. Section 2 provides background that may be helpful in understanding our design. In Section 3 we present our design and explain how operations can be conducted concurrently and independently by many peers in the network and still generate the correct result. Section 5 briefly presents our implementation and then discusses the results of our evaluation. We evaluated performance and scalability using both emulation and simulation. Related work is discussed in Section 6.

2 Background

This section summarizes fundamental concepts, terminology and related algorithms that are relevant to our approach.

2.1 Distributed Hash Tables (DHTs)

A DHT is a data structure that allows efficient storage and lookup of data in P2P networks. It provides hash table functionality for handling key-value pairs. Node identifiers and data item identifiers in a DHT use the same address space. Keys are typically values from a hash function, such as SHA1. Each node maintains connections to a number of other nodes and is responsible for storing certain key-value pairs, as well as routing requests to other nodes in the network. Most DHT designs incur a number of lookup steps that is logarithmic in the number of nodes in the DHT. Lookups are typically conducted for keys, i.e., hash values, which refer to exactly one data item or node. Some selected DHT designs provide more expressive search capabilities at the price of decreased lookup performance – we discuss some of these in Section 6.

With virtually all P2P systems, churn is a potential problem [18]. For our method, we assume that the DHT may loose key-value pairs due to churn or because values expire after some time. Naturally, the DHT is allowed to use replication and caching to alleviate this issue; the DHT simply does not have to guarantee that stored values persist. Our implementation addresses this issue by having the offerer periodically refresh all key-value pairs that it is storing in the DHT (by issuing another PUT). This method of performing keep-alive operations to maintain soft state is common practice in P2P networks. It also offers an implicit way to perform a *delete* operation — if a peer stops to refresh a key-value pair, it will effectively be deleted when the original value expires. Caching and possible replication, non-persistence and soft-state are typical semantics for most DHTs in practice [12], and our method is designed to operate in this context.

2.2 Finite Automata

Finite automata are a mathematical model of computation. They consist of states, input symbols and transitions.

Deterministic Finite Automata (DFA) A DFA $D = (Q, \Sigma, \delta, q_0, F)$ consists of a finite set of states Q, a finite set of input symbols Σ, a transition function $\delta(q,a) : Q \times \Sigma \rightarrow Q$ that takes a state q and an input symbol a and returns a state, a start state $q_0 \in Q$ and a set of accepting states F, where $F \subseteq Q$. A word is a string (sequence) of symbols. A DFA accepts a word if the sequence of symbols leads from the start state to an accepting state by following the transitions marked with the corresponding symbols. Note that a transition may be marked with a whole sequence of symbols, too. The language a DFA accepts is the set of all words that lead to an accepting state.

Non-deterministic Finite Automata An NFA $N = (Q, \Sigma, \delta, q_0, F)$ shares most properties with the DFA, but differs in the transition function δ. For an NFA, $\delta(q,a) : Q \times \Sigma \rightarrow \{q_i, \ldots, q_k\}$ is defined as a relation that takes a state and an input symbol and returns a set of states $\{q_i, \ldots, q_k\} \in \Sigma$. The non-determinism consists in the possibility that several transitions may exist that lead away from the same state but are marked with the same symbol. In the computational model, the NFA is allowed to 'guess' a transition leading to an accepting state. Any NFA can be converted to a DFA that accepts the same language.

2.3 Regular expressions and DFA/NFA

For every NFA or DFA that accepts a certain language L (a so-called regular language), one can give an equivalent regular expression that describes the same language (and vice versa). Regular expressions are typically evaluated by converting them to finite automata. The automata are then simulated on an input string (word) to decide if the string is in the language described by the regular expression. The steps to construct a DFA from a regular expression are parsing the regular expression, constructing an NFA from the parse tree and converting the NFA into an equivalent DFA. For every DFA, there exists an equivalent *minimal* DFA, which is a DFA that recognizes the same language with a minimum number of states. The minimal DFA is also unique for the language, except for the renaming of states. Converting an NFA to a DFA does generally not lead to a minimal DFA. Minimization requires removal of unreachable states and merging such states that are equivalent (non-distinguishable as defined in [17]).

2.4 Transitive Closure Algorithm (TCA)

The Transitive Closure Algorithm (TCA) [9] is a method to compute equivalent regular expressions from a DFA or NFA. We use the TCA as a key building block in our main algorithm. The TCA works by incrementally constructing a regular expression for each pair of states in a DFA. It has a worst-case space requirement of $O(|\Sigma| 4^n)$. Thus, simplifying the regular expression to reduce its size in each step in the TCA is important. This can be done by applying general algebraic rules to transform the abstract syntax tree of the regular expressions at each step of the iteration. For regular expressions, R, S and T, these are:

- Identity: $\emptyset | R = R$; $\varepsilon R = R \varepsilon = R$; $\emptyset R = R \emptyset = \emptyset$
- Associativity: $R|(S|T) = (R|S)|T$; $R(ST) = (RS)T$
- Commutativity: $R|S = S|R$
- Distributivity: $R(S|T) = RS|RT$; $(S|T)R = SR|TR$
- Idempotency: $R|R = R$

Using these rules, we simplify regular expressions in each iteration of the TCA. In our experiments (see Section 5), this has reduced space requirements to just a few kB per instance of the TCA.

3 Design

We now present the general ideas and the functionality of our design. Figure 1 shows a graphical illustration.

As shown in Figure 1, *Offerers* describe their services using regular expressions. For example, a peer that offers IP forwarding to a destination in its subnet 192.0.2.0/24 can do this with the regular expression "$\texttt{IPV4}-\texttt{C00002}.*$", where (following POSIX) "." (dot) stands for "any" character and "$*$" for "zero or more" of the preceeding element. Naturally, it is possible to extend the system to cover more complex requirements, such as allowed port numbers, software versions or other properties of the offerer. Similarly, representing the IP address in hexadecimal is a choice; other notations (such as binary or dotted decimal) would work just as well, with more or less complex regular expressions as a result.

The idea is to convert every regular expression to a DFA. This is done locally by the offerer. The resulting DFA is then published in the DHT. Specifically, each of the automaton's states is published under a certain key in the DHT. The entry for every state contains the outgoing transitions from this state, and the keys of the respective destination states. Finally, the identity of the offerer is stored under those keys that correspond to the accepting states in the automaton. As there are many offerers in the

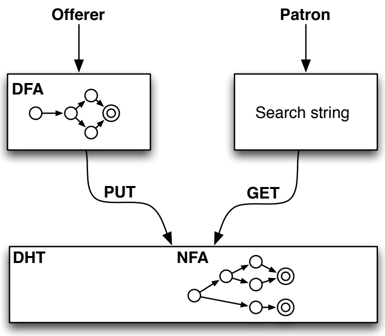


Figure 1: Schematic overview of the presented approach. An offerer creates a regular expression that describes its offered services, converts this into a DFA and stores it in the DHT. A patron searching for a particular string traverses the NFA stored in the DHT to find matching offerers.

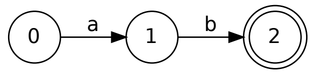
system wishing to store their DFA in the DHT, the store operations from all of the offerers need to somehow be merged by the DHT. As we will see, in the result the DHT will hold an NFA. We give an detailed description of why and how this happens in Section 3.1 and Section 3.2.

Patrons search for a particular service using a string that describes the *exact* service they are looking for. Following our example from above, this would, e. g., be `IPV4 – C00002EB`. In order to find a service that satisfies their search requirement, patrons follow the transitions of the NFA in the DHT to learn the identities of offerers with matching services. To this end, they compute the key corresponding to the starting state (see Section 3.4 for details) and issue a `GET` to retrieve that state's transitions from the DHT. From the replies, they learn a set of transitions and corresponding destination states. The patrons match the transitions against their search string to obtain the set of destination states (and the corresponding keys) which are a longer match for their search string. They then issue new `GET`s for these keys to retrieve the next transitions and destination states, which they then match against their search string. This process continues iteratively until the full search string is matched and an accepting state is reached. If the search string is exhausted without reaching an accepting state, there are no search results.

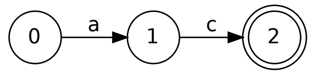
As patrons step through an NFA, the number of `GET` operations they need to perform is not bounded by the length of the search string. However, `GET` operations arising from the non-determinism can be performed in parallel (our NFA has no ε-transitions). Thus, total latency is still bounded by the length of the search string.

3.1 Designing a Mapping of States to Keys

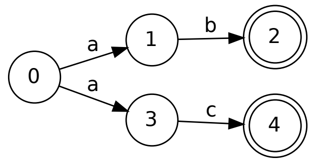
We will now illustrate the key considerations that guided us in designing our algorithm to map states to keys. The choice of such a mapping is a critical factor as it has a significant impact on both correctness and performance of our design. To illustrate this, consider a naïve (yet correct) design where the starting state is mapped to a single key, and each offerer then maps each state of his DFA to a new, random key. The initial state in the DHT would have to store (at least) n non-deterministic transitions for n offerers. To match search strings, patrons would have to perform a prohibitively high number of `GET` operations. Furthermore, the DHT node(s) responsible for the starting state's key would be under high load. To avoid such scenarios, it is important to minimize the number of non-deterministic edges we create in the DHT. In the following, we show informally how this can be achieved.



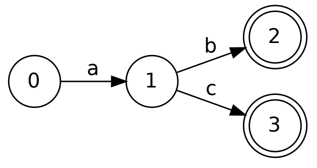
(a) DFA accepting *ab*



(b) DFA accepting *ac*



(c) Merged NFA accepting $ab|ac$



(d) Merged DFA accepting $ab|ac$

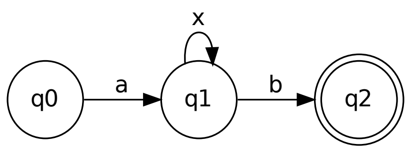
Figure 2: DFAs for *ab* (a) and *ac* (b) that should be merged in the DHT, and two possibilities for merging the corresponding DFAs. While both (c) and (d) are correct, DFA (d) will offer better performance.

Equivalent regular expressions

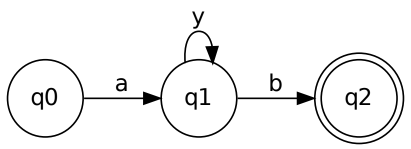
Regular expressions that are equivalent (e. g., $aa^*|b$ and $b|a^+$) describe the same regular language. Converting each into a DFA results in two isomorphic automata. Thus, the corresponding states in these automata should be mapped to the same keys.

Regular expressions with equivalent prefixes

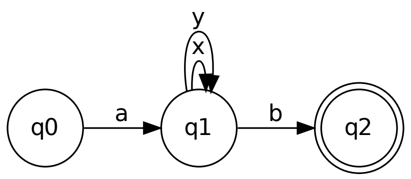
Regular expressions that share a prefix should, whenever possible, be merged into the DHT such that the resulting transitions are only deterministic. Consider the regular expressions *ab* and *ac* and the corresponding DFAs in Figure 2a and Figure 2b. When merging these DFAs into



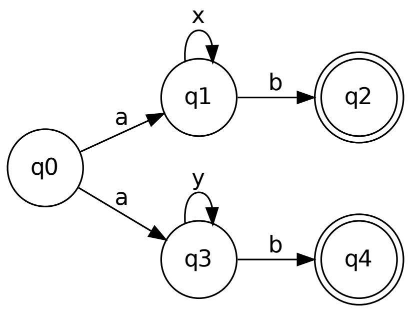
(a) DFA accepting $ax*b$



(b) DFA accepting $ay*b$



(c) Incorrectly combined DFA which would accept $a(x|y)*b$



(d) Correctly combined NFA

Figure 3: Example of an incorrect merge (c) – it accepts strings that are not in the union of the languages of the original DFAs. A correct combination of the two DFAs results in an NFA (d). Note that, as in the example from Figure 2, both original regular expressions started with the shared prefix a.

the DHT, there are two basic possibilities, shown in Figure 2c and Figure 2d. Both automata accept the same language, but Figure 2c is an NFA and Figure 2d is a DFA. Producing the DFA is desirable as the DHT has fewer transitions to store and patrons will need significantly fewer network operations to evaluate the DFA.

Concurrent operations

However, not all such structurally similar regular expressions can be safely merged to a DFA. Consider the regular expressions ax^*b and ay^*b and the corresponding DFAs in Figure 3a and Figure 3b. The merge illustrated in Figure 3c is incorrect, as the resulting DFA accepts strings that neither original regular expression would have matched (e.g., *axyxyb*). Our algorithm must not conduct such merges. Figure 3d shows a correctly merged automaton for the regular expressions from above. Here, merging the two DFAs yields an NFA. Note that a DFA for $ax^*b|ay^*b$ does exist. However, we prefer the NFA here. The reason is a problem we face in practice: two offerers generally do not know about each other and there is thus no way for them to efficiently cooperate. Furthermore, as both offerers may try to concurrently publish their regular expressions, attempts to lookup existing DFAs in the DHT cannot be guaranteed to succeed. Updating complete records in the DHT (as opposed to supplementing them) is also generally undesirable due to the need for locking and the possibility of a DHT performing distributed caching. Our concept is meant to allow concurrent operations while being agnostic to DHT implementations. Our notion of an *independent feasible merge* is the answer to these problems.

In the following, we formalize the problem and present our algorithms. We begin with the operation of correctly merging DFAs into the DHT. We then show how too many non-deterministic transitions and an overload of the starting state are avoided. In particular, we show that our solution is practically *implementable* in a distributed system with *concurrent operations*.

3.2 Formalisation of Problem

We give a formalisation of our problem statement in order to show that our algorithms yield correct results. We write the DFA that is to be merged as a 5-tupel $(Q',\Sigma',\delta',q'_0,F')$. The NFA that exists in the DHT is represented as (Q,Σ,δ,q_0,F); note that δ is a relation that takes a state and an input symbol and returns a set of states, $\delta(q,a) : Q\times\Sigma \to \mathscr{P}(Q)$, whereas δ' is a function $\delta'(q',a) : Q'\times\Sigma' \to Q'$.

We can now define the notion of a *feasible merge*. This is a bijective relation that describes the desired merge operation of a DFA with an NFA. It has the desirable property that the resulting NFA accepts exactly the correct language:

Definition 1 (Feasible merge) *A bijection $m \in M' \to M$, with $M \subseteq Q$ and $M' \subseteq Q'$ of an NFA (Q,Σ,δ,q_0,F) with a DFA $(Q',\Sigma',\delta',q'_0,F')$ is a* feasible merge *if and only if $q_0 \in M$ and $q'_0 \in M'$, and the resulting merged NFA, namely the 5-tuple $(Q\cup(Q'\setminus M'),\Sigma\cup\Sigma',\hat{\delta},q_0,\hat{F})$, accepts exactly the union of the two original languages.*

Here, $\hat{\delta} :=$

$$\delta(q,a) \cup \begin{cases} \{\delta'(q,a)\} & q \in M' \wedge \delta'(q,a) \notin M' \\ \{m(\delta'(q,a))\} & q \in M' \wedge \delta'(q,a) \in M' \\ \{\delta'(q,a)\} & q \notin M' \wedge \delta'(q,a) \notin M' \\ \{m(\delta'(q,a))\} & q \notin M' \wedge \delta'(q,a) \in M' \end{cases}$$

and $\hat{F}$ *is the set of merged accept states*

$$\hat{F} := F \cup m(F' \cap M') \cup (F' \setminus M'). \tag{1}$$

The intuitive understanding of $\hat{\delta}$ is that this relation combines the two state machines, using the mapping function m to map the states $M' \subseteq Q'$ from the DFA to corresponding states $M \subseteq Q$ in the NFA. Edges that do not begin or end at nodes in M (or M') are simply preserved. Note that the requirement to include $q'_0 \in M'$ and $q_0 \in M$ results in 'linking' the original state machines via an initial empty word.

Note that the above definition implies that a feasible merge never leads to the creation of new states that were not already contained in either the original DFA or NFA. $\hat{\delta}$ only merges equivalent states, keeping states from the DFA or NFA that are not related via m.

We now proceed to address concurrent operation. As the CAP theorem [8] states, no distributed system can provide consistency, availability and partition tolerance at the same time. We assume availability and partition tolerance to be hard requirements for our application domain, and the underlying DHT to provide them. We thus sacrifice the global consistency property and require that feasible merges do not rely on globally consistent state in the DHT. In other words, it must be possible for many merge operations to be executed in parallel, without need for synchronisation between participants. We define an *independent feasible merge* as follows.

Definition 2 (Independent feasible merges)
Independent feasible merges are operations that compute feasible merges and permit the concurrent execution of other independent feasible merge operations, without knowledge or communication between the entities carrying out these operations. Upon completion of all independent merge operations, the resulting NFA must still only accept the union of the merged DFAs and the original NFA.

The above definitions state the properties that must be fulfilled by our algorithm that maps DFA states to DHT keys and merges a DFA into the NFA. Note that our feasible merge definition already implies that the NFA that results from independent merge operations is well-defined as the $\cup$ operation is associative and commutative.

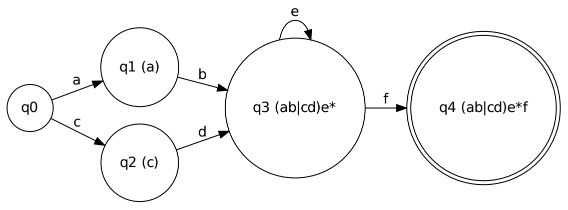


Figure 4: DFA graph for the regular expression $(ab|cd)e*f$ with partial regular expressions assigned to each state.

3.3 Algorithm to Compute Independent Feasible Merges

We now present our algorithm to compute the maximal feasible merge bijection (Algorithm 1), that is the largest (in number of mapped elements) bijection that represents a feasible merge. We then show that this algorithm can be carried out in parallel to provide the independence property.

The basic idea of our algorithm is to first derive the regular expression R_{0j} between the start state q_0 and every other state $q_j \in Q$, and then add a mapping for states with equal regular expressions to the bijection. We derive the regular expressions with the TCA. In order to be able to test them for equivalence, we need to preprocess the DFA to ensure deterministic processing by the TCA. Specifically, TCA relies on a numbering of the DFA states. As we will use the resulting regular expressions for unique state identification, we need to ensure that these expressions are canonical. Thus, the numbering of states also needs to be canonical. This is achieved by first sorting the outgoing transitions of each state in the DFA graph using the names of the corresponding labels and then numbering the states by performing a depth-first-search on the DFA graph.

Figure 4 illustrates an example DFA with (partial) regular expressions assigned to each state.

Starting with a canonical DFA and a function TCA for computing canonical regular expressions R_{ij} for each pair of states $q_i, q_j \in Q$, Algorithm 1 finds a maximal feasible merge that satisfies Definition 1.

Lemma 1 (Feasible merge) *Algorithm 1 computes a feasible merge.*

Proof A full proof can be found in [19]. The key ideas are the following. First, one shows that the algorithm computes a feasible merge. For a feasible merge, q_0 and q'_0 need to be in M and M' respectively. As q_0 and q'_0 are both start states, the corresponding regular expressions are in both cases the empty string, ε. Hence $R_{00} = R'_{00'}$ and thus the algorithm will include $m(q'_0) \mapsto q_0$ in the

Algorithm 1: Mapping DFA states to DHT keys

```
Algorithm: The algorithm starts with an empty
bijection $m \in M' \to M$ and computes the maximal
feasible merge bijections.
Input: NFA $N = (Q, \Sigma, \delta, q_0, F)$ and DFA
       $D = (Q', \Sigma', \delta', q'_0, F')$
Output: Maximal feasible merge bijection
        $m \in M' \to M$ with $M \subseteq Q$ and $M' \subseteq Q'$
1 $R_{ij} \leftarrow$ TCA($N$);
2 $R'_{ij} \leftarrow$ TCA($D$);
3 foreach $(q_j, q'_{j'}) \in Q \times Q'$ do
4     if $R_{0j} = R'_{0j'}$ then
5         $m \leftarrow m \cup \{q'_{j'} \mapsto q_j\}$;
6     end
7 end
```

bijection. Next, one shows that the merged NFA accepts exactly the union of the two original languages. To this end, one first shows that, if a word A is accepted by the NFA via a series of transitions (i.e. moving from state to state), then the union operator ensures that these transitions also exist in $\hat{\delta}$ (and thus the word is also accepted). For a word B, accepted by the DFA, via a sequence of transitions, one shows that the mapping that is conducted by a feasible merge results in a series of equivalent transitions in the merged NFA that also lead to an accepting state. Finally, one shows that no word that was not already accepted by either the NFA or the DFA is accepted in the merged NFA. This can be shown by assuming such a word exists and then showing this leads to a contradiction. □

Lemma 2 (Independent merge) *Algorithm 1 directly leads to an independent merge operation to compute a feasible merge.*

Proof Combined with Equation (1), the algorithm is turned into a decentralized independent merge operation as follows. An independent merge operation first computes canonical regular expressions R'_{0j} for each state $q'_j \in Q'$. Then, the operation hashes the string R'_{0j} to derive a key which then describes the location in the DHT for storing the union of all $\delta'(q'_j, a)$ (for all $a \in \Sigma'$). Note that prior to the DHT PUT operation, the operation needs no knowledge about the existing NFA in the DHT. The peer responsible for the key derived from hashing R'_{0j} can then perform the commutative and associative union operation on the δ'-mappings for the respective q'_j (and all $a \in \Sigma$), locally, and in any order. This merge operation is independent as the corresponding transformations to the NFA are all commutative and associative, and the steps can be executed without having a consistent global view of the NFA. □

Theorem 1 (Maximal feasible independent merge) *Algorithm 1 computes the maximal feasible merge that can be executed using an independent merge operation.*

Proof This proof works again by contradiction. With the previous lemmata, one shows that adding an additional element $\overline{q'}$ to M' does not constitute a feasible merge anymore. For details, we refer the reader to [19]. □

3.4 Decentralizing the Start State

The algorithm described so far exhibits a problem: all starting states correspond to the same regular expression, namely the empty string. Thus, they would be mapped to the same key in the DHT. The corresponding responsible nodes (hosts) would experience significant load: they would receive many GET and PUT operations, and the result sets they would need to return would also be rather large (in $O(|\Sigma|)$).

We thus distribute the starting states in the following way. For any string in the accepted language, we take the first k characters of the corresponding regular expressions. When merging its DFA, a peer inserts a synthetic start state under the hash of each of these prefixes into the DHT. The synthetic start states are then connected to the rest of the DFA as necessary. E.g., for $abc*def$, an offerer would choose the strings *abcc*, *abcd* and *abde* to determine the corresponding keys.

Instead of the naïve search, a patron peer hashes the first k characters of the string that should be matched, and looks up the information stored under this hash to find the entry point for the corresponding NFA.

This strategy trades load on peers hosting entry states for a potentially exponential increase in the number of synthetic entry states, i.e., offerers with highly generic offers may have to perform additional PUT operations. For an input alphabet Σ and k initial characters, the worst case would be a number of additional states in the order of $|\Sigma|^k$. Thus, k needs to be chosen with care and with respect to the Σ that is applicable for the respective application.

4 Hardening

So far we have assumed that the regular expressions and their associated capabilities are not private information, and that the peers providing the underlying DHT are honest. While this may be the case for certain types of services, it would be good to support private offers and to protect the values in the DHT against tampering by peers.

This can be done if we assume that each search term (and regular expression) starts with sufficient entropy to prevent enumeration attacks. For this section we will thus assume that for private services, search terms and regular expressions start with a random pre-shared key (PSK) of sufficient length L (say 128 bits) and that the prefix k (from Section 3.4) is larger or equal to k.

Let $(H(R), v)$ be any of the key-value pairs previously stored in the DHT in the clear, where R is some (partial) regular expression and v the respective encoding of the outgoing state transitions $\Sigma \mapsto \mathscr{P}(Q)$. Let $HKDF$ and $HKDF'$ be two hash-based key derivation functions [?] with different salts. Let G be the generator of an Elliptic curve, say Curve25519 and $n = |G|$ the group order. We can then define a private key $d = HKDF(R) \mod n$ and the corresponding public key $D = dG$ and use, say, EdDSA [?] for signing S_d using private key d. Given this setup, we can use a symmetric cipher E to store $\langle S_d(E_{HKDF'(H(R))}(v)), D\rangle$ under key $H(D)$.

Given the PSK, a peer performing a search s can start with its R_0-value using the first k bits from $\langle PSK, s\rangle$ and can then perform DHT lookups to incrementally obtain the encrypted blocks and perform the decryption to obtain a series of $H(R_i)$ values up until the accept state. The accept state may even store confidential access information with the value v which would never be disclosed to anyone who does not know the PSK. Furthermore, the signature scheme ensures that peers cannot tamper with the values in the DHT, and that only parties privy to the PSK can offer the respective service.

5 Implementation and Experimental Results

We implemented our approach for the GNUnet[1] Peer-to-Peer (P2P) framework. This allowed us to evaluate our approach using emulation and real GNUnet production code (as opposed to simulation code). It should be noted that GNUnet's default DHT is optimized with a view towards use in a route-constrained environment, such as it may be encountered where many peers are located behind a NAT or firewall, and there are relatively few peers that have near-global connectivity. The DHT deals with such problems by using a recursive routing strategy plus a short, randomized series of first steps in the network (after which search continues normally). A drawback of this is that latencies are slightly less predictable than in a conventional DHT. However, it also means that the results we obtained from emulation, especially latencies, generally represent upper bounds – conventional DHTs can be expected to be somewhat faster. We chose this particular DHT as it is best suited for our use case: P2P-based VPNs must deal with such route-constrained environments.

Metrics related to the structure of the DFA and NFA, such as the number of non-deterministic edges in the NFA, are independent of the underlying P2P framework. This allows us to determine these by simulation rather than emulation and obtain results that are valid for any DHT setup. Our experiment setup was the same for both emulation and simulation: In our experiments, regular expressions are used to find peers willing to forward traffic to a particular Autonomous System.

5.1 Integration into GNUnet and Experiment Design

Our implementation uses several GNUnet components. Fundamentally, states are stored in the GNUnet DHT [6]. Our experiments themselves are based on the Protocol Translation (PT) component. This is a GNUnet application which allows peers to provide network access for other peers, while translating between IP versions in the overlay if needed.

PT is a particularly good choice for our experiments for two reasons. First, the service needs to discover suitable peers that are willing to act as Exits. Second, an Exit may not be able or willing to handle *all* incoming traffic, but, e.g., just particular IP versions for particular destinations due to policy or connectivity restrictions.

Figure 5 visualizes how PT works. The Exit daemon is one key component: it can be configured to share the local Internet connection. The VPN service is the counterpart to the Exit daemon; it provides peers with the ability to route Internet traffic arriving at a virtual network interface via the P2P network to a peer that is running an Exit. To communicate with peers that offer an appropriate Exit, Protocol Translation (PT) relies on the CADET service [?], which is responsible for routing messages to suitable peers. CADET needs to receive information about peers found using our implementation. An Exit can specify a *policy*, represented as a regular expression, with the protocols (e.g., IPv4, IPv6) and destination IP or networks it is willing to offer forwarding to. A patron looking for forwarding to a certain address will convert the desired target IP address to a suitable string and then ask the VPN to discover a matching Exit and create a CADET tunnel towards it. The idea behind this is that a peer may be willing to serve as an Exit, but only for traffic destined to its own Autonomous System (AS). In our experiments, we evaluated how effective and efficient a search for an IPv4 Exit node is when regular expressions are used to express this.

[1] `https://gnunet.org`

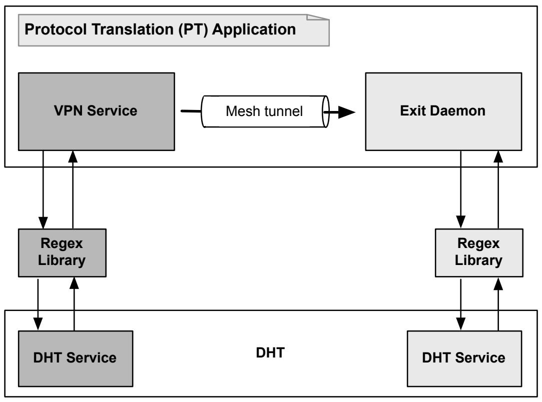


Figure 5: Protocol Translation (PT) application overview diagram for two peers, one acting as offerer (highlighted with light grey) that is running the Exit daemon and one acting as patron (highlighted with dark grey) that is running the VPN service and is trying to find a matching Exit.

5.2 Dataset for Experiments

We used the CAIDA *Routeviews Prefix to AS mappings Dataset (pfx2as)* to obtain mappings from IPv4 prefixes to AS numbers. This dataset is derived from the University of Oregon's Route Views Project [14] and updated daily; we used the data from 2012-09-11, which contains 440,448 lines with 40,696 unique AS numbers.

The maximum number of subnets for an AS in the data set is 2,773.

5.3 Results from Emulation

We first present results obtained by large-scale emulation. For the experiments, we used the GNUnet testbed [20] For the experiments, we used the GNUnet testbed [20] to deploy, control and observe up to 2,000 peers on a PC with an Intel Xeon W2530 processor at 2.8 Ghz and 24 GB RAM. The testbed runs the normal GNUnet code; however, special options were set to store DHT data in RAM instead of on disk and to artificially delay messages exchanged between peers.

5.3.1 Setup

In our setup, each AS ID corresponds to a peer responsible for announcing the appropriate regular expressions. The peers were distributed among all hosts and each peer was initially connected randomly to other peers in the testbed, until every peer had 40 connections on average. The DHT was not allowed to create additional routing table entries to keep the load predictable and under control. The DHT used, R^5N [6], supports this restricted routing scenario but performs better if new connections are allowed. While the DHT supports caching and replication, the effect of these features is limited due to the short duration of the PUT period and the fact that each lookup is for a different search string. Thus, in practice the system may perform better than what is shown in this emulation.

We conducted experiments with setups emulating $1,000$ and $2,000$ peers. Each peer out of a set of 3 was given a regular expression describing the range of IP addresses of its corresponding AS to store in the DHT. Each peer stored the regular expression 3 times, with a separation between operations selected at random between 0 and 1 seconds. Another peer, selected at random, would start the search of a string describing an IP address in the AS, after waiting for 2 seconds. This waiting period guarantees that the regular expression has been stored all 3 times in the DHT before the search starts. After the search succeeded or 90 seconds passed, whichever came first, a new peer started to store its regular expression in the DHT. This sequence was used to limit the number of peers simultaneously performing `GET` and `PUT` operations in the network.

5.3.2 Performance

Table 1 summarizes performance results from our experiments. As can be seen, our approach scales quite gracefully: bandwidth consumption per peer increases only slowly when doubling number of peers. Note that the searches use only `GET` operations – the majority of traffic is caused by storing the automata. The `PUT` operations are repeated 3 times and have to store all states and edges of the automata, while `GET` operations only traverse the relevant states for the search string. The traffic for bigger networks increases due to the nature of the DHT operating in a restricted route network. Given a random topology with a fixed number of connections per peer, a bigger network has a bigger diameter, and therefore the R^5N [6] `GET` and `PUT` request paths must be longer to guarantee reaching the appropriate peer for a given key. `RESULT` messages follow the route of `GET`s, therefore have the same tendency. Note that the results presented are for traffic data of individual peers, the standard deviation is thus a measure of the load distribution among the peers. As expected, the load is reasonably well balanced.

5.3.3 Search duration

We measured the time each peer needs to complete a search in different size networks. To achieve more realistic results, the peers run with a simulated 100ms network latency for each one-way peer-to-peer message, making a total round trip time between peers of 200ms. The results are illustrated in Figure 6, for runs with $1,000$ and

Table 1: Results from experiments for emulations with 1000 and 2000 peers. Numbers are on average per peer.

Bandwidth (kB)/peer		1,000	2,000
PUT:	Average	587	702
	Standard deviation	238	311
GET:	Average	67	82
	Standard deviation	27	36
RESULT:	Average	107	121
	Standard deviation	44	54

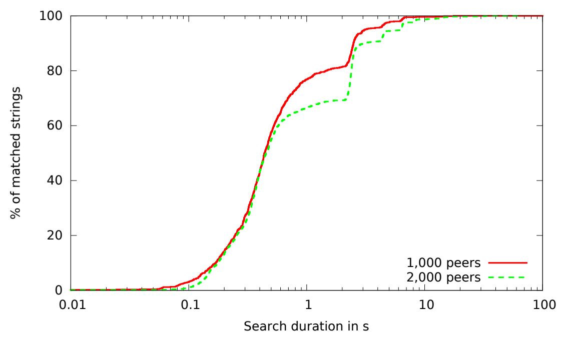


Figure 6: Search duration for runs with 1000, 2000 randomly connected peers. Note logarithmic scale in x-axis.

2,000 peers. The majority of strings is found after a relatively short time: 50% of the strings in the 1,000 peer network are found after 441ms, which is a little more than two round trips. On the upper end, 95% of results in the 2,000 peer network are found after 6.3s. We find these results quite encouraging, given the high potential expressivity of our search mechanism and the unfavorable conditions (latency, topology) for the DHT operation.

We also found that some strings are found very soon. There are two explanations. First, some strings are searched by the same peer that already holds a matching regular expression. Second, sometimes a direct neighbour has a matching regular expression (i.e., the network is well-meshed). There are some strings that take a very long time to match, likely due to the DHT using recursive randomized routing, which has higher latency and larger variance in latency compared to other DHTs.

5.4 Results from Simulation

A key parameter in our design is the maximum length of path compression, which is a trade-off between state explosion and the number of lookups. We now present an evaluation of the effects of path compression. We obtained these results by simulation as the effects are independent of network structure. To this end, we replaced GNUnet's VPN and Exit services with a profiler and the DHT with a database.

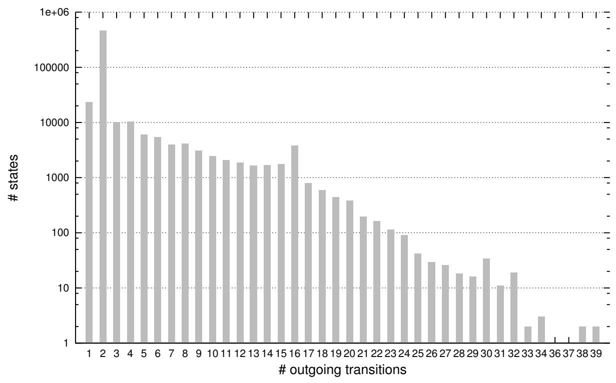


Figure 7: Number of states in the merged NFA by outdegree.

5.4.1 Number of Edges

The number of transitions stored at a given key is an important parameter for performance. Many transitions at a key mean a larger result set, a higher load on the responsible peer, and potentially a higher degree of non-determinism. Figure 7 visualizes the out-degree distribution for our experiment. It shows the number of states that have a certain number of outgoing transitions in the NFA. Note the logarithmic *y*-axis (indicating the number of states). The ratio of edges per state behaves quite gracefully, with most nodes having a low degree and no nodes having an excessively high degree. This is expected, especially given our use of a hexadecimal alphabet for the interesting part of the regular expression.

5.4.2 Non-Determinism in the DHT

As our automaton is an NFA, the degree of non-determinism (non-deterministic edges per state) directly affects the performance of the DHT, in particular searches, as it translates into more necessary iterative lookups. We thus evaluated this important metric.

Figure 8 shows that, for the AS data set, the maximum number of non-deterministic edges at any state in the merged NFA is only two. Furthermore, the overall fraction of non-deterministic edges is quite low. These values are very encouraging and show that our approach will show good performance for the AS use case. However, we do have to add a word of caution: the values do not necessarily generalize to other use cases. One can expect some applications to exhibit a higher degree of non-determinism, depending on the structure of the regular expressions they need to use. We have done other experiments where the maximum number of non-deterministic edges was three [19]. Systematically de-

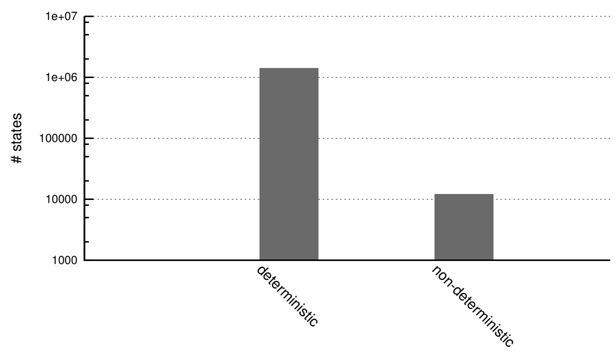


Figure 8: Characterization of states of the merged NFA from the CAIDA dataset. Non-deterministic states have exactly 2 transitions with the same label.

signing rules for writing regular expressions that, when independently merged using our method, would minimize non-determinism, is left for future research.

6 Related Work

The presented method is a specialized design and implementation of a Convergent Replicated Data Type (CRDT) [16]. In general, CRDTs address the fundamental issue of the CAP theorem [8] using eventual consistency [21]. In contrast to the generic CRDT graph representations described in [16], our method does not require explicit delete operations (the DHT simply discards entries after a timeout) or garbage collection. Furthermore, assuming the DHT itself is Byzantine fault-tolerant, our decentralized data structure should also operate correctly even in the presence of malicious participants. Existing CRDTs generally operate under the stronger assumption of non-byzantine behaviour [16].

Our work differs from most of the related work on searching in DHTs in two respects. First, and to the best of our knowledge, the use of general regular expressions is a unique feature of our concept. Second, many related concepts [7, 15] assume that the offerer publishes a string and that the patron searches for a range or set of strings. Despite these fundamental differences, some concepts do appear in related work.

Garcés-Erice et al. [7] propose a system to discover objects stored in a DHT. Objects are stored under a combination of all their keywords; individual keywords are mapped to keyword combinations. The authors expect the application to traverse the graph starting with keywords provided by the user, followed by an iterative, interactive refinement process using search refinement suggestions generated automatically. The notion of iterative traversal is also a part of our concept.

PastryStrings [1] creates a publish-subscribe system using a tree-structure on top of a DHT. It supports a variety of queries, like range and comparison queries for numerical values, and prefix, suffix, and containment matching for strings. Subscribers express interest in a specific value. For each starting character of a string, PastryStrings builds a tree with all of the words that have the character as prefix. The publisher will search the trees to find all nodes interested in its data. Prefix Hash Trees (PHTs) [15] add range query support to arbitrary DHTs. PHTs add a trie structure to normal DHT storage; the trie structure links neighboring items in the item name space. The idea is similar to PastryStrings; however, intermediate nodes are located at the hash of their prefix and can thus be reached without contacting the root node. Compared to our system, PastryStrings and PHT are limited in the expressiveness of queries (we support regular expressions).

Cubit [4] tackles the problem of finding approximate matches for keywords. In Cubit, nodes have keywords as identifiers and the overlay structure is based on edit distances between those keywords. Mappings are stored on nodes with a small edit distance to the keyword. Approximate searches are then conducted by querying all nodes with identifiers within a certain edit distance to the query. Keywords also play a central role in the Distributed Pattern Matching System (DPMS) [2]. Here, each keyword represents a bit in the identifier space. Objects are associated with a set of keywords, and an object is identified with the combination of the object's keywords. Pattern matching is enabled by using n-grams as keywords. The overlay structure of DPMS is a hierarchical lattice; searches traverse the hierarchy. While these approaches support approximate matching, they need specialized network structures. Our approach can be used with any DHT.

Cooperative Information Sharing System (CISS) [13] enables range query support for existing DHTs by replacing the hash function with a locality preserving function. This achieves a high level of clustering of similar objects. However, creating an appropriate clustering function is non-trivial; the choice impacts the DHT's load-balancing properties.

Karnstedt et al. [11] present an approach to store and select entries of a relational database in a DHT. Each entry has an identifier, and the complete entry is stored under this identifier. In addition to that, all of its columns are stored individually in the DHT under combinations of their identifier, their value, and their field name with a link back to the entry identifier. The system is able to run on generic DHTs. However, it only supports a subset of SQL and suffers from significant storage overhead.

7 Summary, Conclusion and Outlook

We have described our mechanism for searches based on regular expressions in DHTs. Using our approach, offerers can describe services in regular expressions, and patrons can find them by searching for strings that match the regular expression. To the best of our knowledge, there are no search mechanisms yet that would offer equivalent expressiveness.

We have evaluated our mechanism using both emulation and simulation. Our use case was VPN-like functionality with searches for Exit nodes to certain networks. We have found that our design scales well with the number of peers and searches in the network. Lookups require only a low number of traversed nodes, and an acceptable number of iterated edges. The majority of searches terminates after a comparatively short time. The expected total network latency is linear in the length of the search string, which should be suitable for applications that can tolerate moderate latency.

Still, there remains room for improvement, We have tested our design for only one use case, and it will be highly interesting to determine its performance for other use cases, e. g., distributed computation. Furthermore, our cluster allowed us to emulate only a relatively small number of peers. With access to a new cluster, we hope to be able provide more insight for much larger networks soon.

On the whole, however, we find our results very encouraging and the mechanism certainly usable, especially when considering the high expressiveness that it supports. Our mechanism is integrated in GNUnet and thus available for public review.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG) under ENP GR 3688/1-1. We thank David Barksdale for insightful discussions and Sree Harsha Totakura for help with running the experiments.

References

[1] Ioannis Aekaterinidis and Peter Triantafillou. PastryStrings: a comprehensive content-based publish/subscribe DHT network. In *Proc. 26th IEEE Int. Conf. on Distributed Computing Systems (ICDCS '06), Lisboa, Portugal*, page 23, 2006.

[2] Reaz Ahmed and Raouf Boutaba. Distributed pattern matching: A key to flexible and efficient P2P search. In *Proc. 10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006)*, pages 198–208.

[3] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Commun. ACM*, 46(2):43–48, February 2003.

[4] Aleksandrs Slivkins Bernard Wong and Emin Gün Sirer. Approximate matching for Peer-to-Peer overlays with Cubit. Technical report, Cornell University, Computing and Information Science, 2008.

[5] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proc. 13th USENIX Security Symposium*, August 2004.

[6] Nathan S. Evans and Christian Grothoff. R^{5}N: Randomized recursive routing for restricted-route networks. In *5th Int. Conf. on Network and System Security (NSS 2011)*, Milan, Italy, September 2011.

[7] L. Garcés-Erice, P. A. Felber, E. W. Biersack, G. Urvoy-Keller, and K. W. Ross. Data Indexing in Peer-to-Peer DHT Networks. In *Proc. 24th Int. Conf. on Distributed Computing Systems (ICDCS'04), Tokyo, Japan*, pages 200–208, 2004.

[8] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *SIGACT News*, 33(2):51–59, June 2002.

[9] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, 2007.

[10] International Business Machines Corporation. *WebSphere MQ: Publish/Subscribe User's Guide 7.0*, January 2009.

[11] Marcel Karnstedt, Kai-Uwe Sattler, Manfred Hauswirth, and Roman Schmidt. Similarity queries on structured data in structured overlays. In *Proc. 22nd Int. Conf. on Data Engineering Workshops (ICDEW '06)*, page 32, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Brad Karp, Sylvia Ratnasamy, Sean Rhea, and Scott Shenker. Spurring adoption of dhts with openhash, a public dht service. In GeoffreyM. Voelker and Scott Shenker, editors, *Peer-to-Peer Systems III*, volume 3279 of *Lecture Notes in Computer Science*, pages 195–205. Springer Berlin Heidelberg, 2005.

[13] Jinwon Lee, Hyonik Lee, Seungwoo Kang, Su Myeon Kim, and Junehwa Song. CISS: An efficient object clustering framework for DHT-based Peer-to-Peer applications. *Computer Networks*, 51(4):1072–1094, March 2007.

[14] David Meyer. University of Oregon Route Views Archive Project . `http://www.routeviews.org/`. [Last retrieved: January 2013].

[15] Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Range queries over DHTs. Technical Report IRB-TR-03-009, Intel Research, 2003.

[16] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. (7506), 01/2011 2011.

[17] Y.N. Singh. *Mathematical Foundation Of Computer Science*. New Age International (P) Limited, 2006.

[18] Daniel Stutzbach, , and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 189–202, New York, NY, USA, 2006. ACM.

[19] Maximilian Szengel. Decentralized evaluation of regular expressions for capability discovery in Peer-to-Peer networks. Master's thesis, Technische Universität München, November 2012.

[20] Sree Harsha Totakura. Large scale distributed evaluation of peer-to-peer protocols. Masters, Technische Universitaet Muenchen, Garching bei Muenchen, 06/2013 2013.

[21] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008.